Table des matières

Table des figures

INTRODUCTION GÉNÉRALE

---

1. *Contexte*

La science d'aujourd'hui couvre des problèmes de plus en plus complexes et est poussée par des technologies de plus en plus puissantes. Elle est aussi bien basée sur le calcul, l'analyse de données et la collaboration que sur les efforts des théoriciens et expérimentalistes. L'avancée technologique en matière de puissance de calcul, de stockage de données et de communication ne cesse de croître, en effet, les ressources classiques n'arrivent pas à satisfaire les exigences des scientifiques en matière de calcul et de temps de réponse requis par les applications modernes. Dans de nombreux domaines de recherche, les besoins en matière de puissance de calcul croissent plus rapidement que la puissance des machines disponibles. Paradoxalement, les ressources de calcul sont souvent sous exploitées, si l'on se place à un niveau global. En effet, de la simple station de travail jusqu'à la machine parallèle de grande taille, il est rare qu'une machine soit utilisée à son potentiel maximum en permanence. De cette constatation vient l'idée de grille de calcul. Les grilles de calcul sont constituées de nombreuses ressources informatiques hétérogènes, géographiquement éloignées, et mises en réseau. De plus, ces plates-formes sont dans la plupart du temps construites en utilisant des processeurs ou des machines existantes, ce qui génère une hétérogènèité de fait. Construite autour de l'Internet et le World Wide Web, la grille est une infrastructure offrant des mécanismes sécurisés, évolutifs, et de haute performance pour la découverte et la négociation d'accès à des ressources distantes. Elle permet de rendre possible les collaborations entre scientifiques dans le but de partager des ressources à une échelle sans précédent, et à des groupes géographiquement distribués de travailler ensemble, ce qui n'était guère possible auparavant [127].

Les technologies de grille offrent aux utilisateurs l'accès à une multitude de services et de puissances de calcul pour traiter des données ou en générer de nouvelles. Ces perspectives ont donné un nouvel essor à la modélisation des applications scientifiques dans de nombreux domaines tels que la bio-informatique [115] la physique [104], l'astronomie [38] et bien d'autres domaines, contenant généralement de nombreuses tâches contraintes par des relations de précédence. Ces applications sont souvent exprimées sous forme de workflows scientifiques comportant une série d'étapes de traitements (tâches) liées, où chaque étape prend des données en entrée, effectue un traitement, et produit certaines données en sortie qui seront transmises aux étapes de traitement suivantes. Ces applications peuvent être modélisées par des graphes orientés acycliques, DAG (Direct Acyclic Graph), où les sommets dans le DAG représentent les tâches et les arcs représentent les dépendances des flux de données. Ces workflows impliquent souvent de longues simulations, des calculs à grande échelle et la manipulation massive des données. Ils requièrent, pour leurs exécutions, des puissances de calcul élevées et la disponibilité de grandes infrastructures informatiques.

En raison de l'importance des applications de workflows, plusieurs projets de recherche ont été menés pour concevoir des systèmes de gestion de workflows et des algorithmes d'ordonnancement appropriés. Les systèmes de gestion de workflows peuvent être considérés comme un type de service facilitant l'automatisation des applications de e-business et e-science sur la grille et le cloud. Ils sont utilisés pour gérer ces applications de façon transparente en masquant l'orchestration et les détails d'intégration spécifiques lors de l'exécution des tâches sur les ressources distribuées de grille ou de cloud. En vue d'ordonnancer efficacement les tâches de ces applications sur les environnements de grille, les systèmes de gestion de workflows requièrent des stratégies d'ordonnancement plus élaborées pour répondre aux exigences de QoS (makespan, coût et autres), ainsi qu'aux relations de précédence entre les tâches du workflow. L'étude des stratégies d'ordonnancement de workflows devient un enjeu important dans les grilles de calcul.

2. *Problématique et motivations*

Les workflows sont généralement composés de milliers de tâches, avec des interdépendances. Exécuter un workflow sur une grille de manière efficace se confronte à un pro-

blème d'ordonnancement. Ce problème est d'autant plus difficile lorsqu'il y a plusieurs facteurs à prendre en compte, à savoir : les différentes contraintes et exigences de QoS des utilisateurs par exemple : (i) le makespan ; (ii) l'hétérogénéité ; la dynamicité ; (iii) la fiabilité. Cependant, le problème d'ordonnancement de workflow est vu comme un problème d'optimisation combinatoire, où il est impossible de trouver la solution globale optimale en utilisant des algorithmes ou des règles simples. Il est bien connu comme un problème NP-complet [101] et dépend de la taille du problème à résoudre.

L'ordonnancement de workflows tolérant aux fautes dans les grilles est largement étudié dans de nombreux travaux [71, 87, 133]. Dans tous les cas, les algorithmes d'ordonnancement doivent :

- Respecter la contrainte de précédence entre les différentes tâches qui composent le workflow ;
- Prendre en compte plusieurs métriques de QoS, telles que : le makespan, le coût, la fiabilité, la disponibilité.

La mise en œuvre de la tolérance aux fautes à travers l'utilisation de la réplication dans les grilles soulève plusieurs problématiques. Certaines de ces problématiques sont spécifiques aux grilles, tandis que d'autres sont plus générales et peuvent être rencontrées lors de l'utilisation de la réplication dans n'importe quel type de système distribué.

- La première problématique que nous soulevons est relative au grand nombre de ressources consommées en utilisant la réplication comme moyen pour la tolérance aux fautes.
- La seconde concernerait l'évaluation de fiabilité de l'algorithme d'ordonnancement tolérant aux fautes.

3. *Contribution*

Trouver un algorithme d'ordonnancement pour un workflow afin que la fiabilité du système soit optimisée, que la contrainte de temps soit satisfaite, et que la consommation de ressources soit réduite. Ce problème serait pris en charge dans notre recherche à travers l'utilisation d'une stratégie dynamique d'ordonnancement de workflow tolérante aux fautes nommée DFWS (Dynamic Fault Tolerant Workflow Scheduling Policy). Cette stratégie combine l'ordonnancement de workflow avec la réplication des tâches, de sorte que le workflow est exécuté de manière efficace et fiable en utilisant la réplication passive

et dynamique des tâches avec moins de consommation de ressources et tout en respectant les contraintes de précédences qui existent entre les tâches.

Nos contributions consistent en la proposition d'un modèle, pour l'ordonnancement de tâches tolérant aux fautes et dont les propriétés sont :

- il est constitué de deux niveaux, quelle que soit la complexité topologique d'une grille ;
- il supporte l'hétérogénèité, la dynamicité et le passage à l'échelle ;
- il facilite l'évaluation de fiabilité et du temps d'exécution ;
- il prend en compte la dépendance de données et la tolérance aux fautes.

Sur la base de ce modèle nous développerons [83, 84] :

- Une stratégie d'ordonnancement, à quatre phases :
  (a) calcul des priorités des tâches ;
  (b) tri des tâches en fonction de leurs priorités ;
  (c) clustering des tâches ;
  (d) assignement des tâches aux ressources qui minimisent leur temps de fin d'exécution.
- Une stratégie de tolérance aux fautes basée sur la réplication passive et dynamique, cette stratégie minimise la fiabilté et la consommation des ressources ;
- Une métrique d'évaluation de fiabilité. Cette métrique étant basée sur la théorie des graphes.

4. *Organisation de la thèse*

Les travaux que nous avons menés dans le cadre de la problématique d'ordonnancement de workflow tolérant aux fautes s'orientent autour de quatre chapitres, une introduction générale et une conclusion.

- Le chapitre 1, traite le problème de la tolérance aux fautes dans les grilles de calcul. Nous mettons, en particulier, l'accent sur la sûreté de fonctionnement, ses attributs et ses méthodes. Ensuite, nous mettons en évidence les différentes fautes possibles dans les grilles de calcul et les moyens pour les détecter. Enfin, nous détaillerons les différentes techniques de tolérance aux fautes dans les grilles de calcul.

– Dans le chapitre 2 nous ferons un tour d'horizon sur les workflows scientifiques, puis les algorithmes d'ordonnancement de workflow, et les gestionnaires d'exécution de workflow dans les grilles de calcul, et nous finirons par l'exposition des algorithmes d'ordonnancement de workflow tolérant aux fautes ;
– Le chapitre 3, présentera en détail le modèle hiérarchique que nous proposons pour représenter une grille de calcul. Il décrira par la suite la stratégie d'ordonnancement de workflow tolérante aux fautes développée sur ce dernier. Cette stratégie combine une méthode d'ordonnancement de tâches et une autre pour la tolérance aux fautes ;
– Le chapitre 4, prendra en charge une série d'expérimentations et d'évaluations de notre approche, nous tenterons de montrer que l'approche proposée permet d'atteindre les objectifs que nous nous sommes fixés. La conclusion récapitule à la fois la problématique que nous avons traitée dans cette thèse, et les résultats obtenus. Nous proposerons ensuite un certain nombre de pistes en vue d'éventuelles recherches autour de la réflexion sur le problème de la tolérance aux fautes dans les grilles de calcul.

Chapitre 1

Tolérance aux fautes dans les grilles de calcul

Sommaire

1.1 Introduction

L'objectif de ce chapitre est de présenter le vocabulaire relatif à la "sûreté de fonctionnement" et d'introduire la tolérance aux fautes comme un moyen de sûreté de fonctionnement. Nous y présentons les principales techniques utilisées pour réaliser la tolérance aux fautes dans les systèmes répartis [7]. Avec l'émergence de la technologie des grilles de calcul, la tolérance aux fautes est devenue une de ses importantes propriétés : la fiabilité de ses ressources ne pouvant être totalement garantie. Les spécialistes cherchent à développer des modèles de tolérance aux fautes adaptés aux grilles de calcul et des techniques de tolérance aux fautes pour les applications de grille [125].

1.2 Sûreté de fonctionnement

La sûreté de fonctionnement des systèmes informatiques est définie dans [99] comme étant :

"*La propriété permettant aux utilisateurs d'un système de placer une confiance justifiée dans le service qu'il leur délivre. Mettre en œuvre la sûreté de fonctionnement d'un système correspond à lutter contre les défaillances du système.*"

Dans une autre définition [161], la sûreté de fonctionnement est considèrée comme étant :

"*L'aptitude d'une entité à satisfaire à une ou plusieurs fonctions requises dans les conditions données.*"

Pour bien comprendre les tenants et les aboutissants de la sûreté de fonctionnement, nous présenterons dans ce qui suit ses entraves, ses attributs, et ses moyens.

1.2.1 Entraves à la sûreté de fonctionnement

Les fautes, les erreurs et les défaillances sont les causes et les conséquences de la non sûreté de fonctionnement [7] :

- ***Faute*** : c'est toute cause (événement, action, circonstance) pouvant provoquer une erreur. La faute dans un système informatique représente soit un défaut d'un composant physique, soit un défaut d'un composant logiciel de ce système. Elle pourrait être créée

de manière intentionnelle ou accidentelle, à cause de phénomènes physiques ou d'imperfections humaines. Durant l'exécution du système, la faute reste dormante jusqu'à ce qu'un événement intentionnel ou accidentel provoque son activation [5, 95] ;

- ***Erreur*** : l'activation d'une faute durant l'exploitation du système peut se manifester par la présence d'un état interne erroné dans ce système, ce qui entraînerait un résultat incorrect ou imprécis par rapport à celui attendu. Cet état pourrait rester longtemps non détecté (latence de la faute) [47], mais pourrait tout aussi conduire à court ou à long terme à une défaillance. Une défaillance du système survient lorsque le service délivré diffère du service spécifié ;
- ***Défaillance*** : elle survient lorsque le service délivré par le système ne correspond plus à sa spécification [95]. Les attributs de la sûreté de fonctionnement d'un système mettent plus ou moins l'accent sur les propriétés que doit vérifier la sûreté de fonctionnement du système. Ces attributs permettent d'évaluer la qualité du service fourni par un système.

Comme l'illustre la « Figure 1.1 », la défaillance survient parce que le système se caractérise par un comportement erroné : une erreur est la partie de l'état du système qui est susceptible d'entraîner une défaillance. La cause adjugée ou supposée de l'erreur est une faute. Une erreur est donc la manifestation d'une faute dans le système, et une défaillance est l'effet d'une erreur sur le service. Ceci conduit à la chaîne fondamentale présentée dans la « Figure 1.1 ».

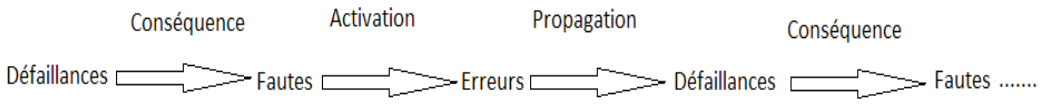

FIGURE 1.1 – Relation entre faute, erreur et défaillance [98]

1.2.2 Attributs de la sûreté de fonctionnement

Les attributs de la sûreté de fonctionnement d'un système mettent plus ou moins l'accent sur les propriétés que doit vérifier la sûreté de fonctionnement du système [5, 78]. Ces attributs permettent d'évaluer la qualité du service fourni par un système. Parmi ces propriétés, nous trouvons :

La Disponibilité : Probabilité pour qu'un système soit disponible à un instant donné t, cette probabilité, notée A(t), peut être estimée à travers la formule suivante :

$$A(t) = \frac{temps \quad en \quad fonctionnement}{temps \quad total} \tag{1.1}$$

La Fiabilité : Probabilité pour qu'un système soit continûment en fonctionnement sur une période donnée (entre 0 et t). Pour quantifier la fiabilité d'un système, il est courant de se référer au temps moyen avant la première panne, noté MTTF(Minimum Time To failure). Dans ce cas si Rel(t) représente la fiabilité du système durant la durée t, alors l'équation suivante définit le MTTF du système :

$$MTTF = \int Rel(t) \qquad dt \tag{1.2}$$

La Sûreté : est une propriété qui respecte la non occurrence de défaillance catastrophique, similaire à la fiabilité mais par rapport aux conséquences catastrophiques causées par les fautes ;

La Sécurité-confidentialité : Cette propriété concerne l'occurrence des accès non autorisés ou l'acquisition non autorisée d'informations. Cette propriété évalue la capacité du système à fonctionner en dépit de fautes intentionnelles et d'intrusions illégales ;

L'Intégrité : L'intégrité d'un système définit son aptitude à assurer des altérations approuvées des données ;

La Maintenabilité : Définit l'aptitude aux réparations et aux évolutions. C'est la probabilité pour qu'un système en panne à l'instant 0 soit réparé à l'instant t. La sûreté de fonctionnement est obtenue par l'utilisation de méthodes et de techniques permettant de fournir à un système l'aptitude à délivrer un service qui soit conforme à sa spécification et d'accorder une certaine confiance à cette aptitude.

1.2.3 Moyens d'assurer la sûreté de fonctionnement

Les moyens utilisés pour assurer la sûreté de fonctionnement sont définis par des méthodes et des approches [82]. Les approches connues sont :

- ***Prévention des fautes*** : cette méthode vise à empêcher l'occurrence ou l'apparition de fautes par le développement des systèmes informatiques de manière à éviter l'in-

troduction de fautes de conception ou de fabrication et à empêcher que des fautes ne surviennent en phase opérationnelle ;

- ***Tolérance aux fautes*** : elle consiste à délivrer un service correct en dépit de l'occurrence de fautes [7, 78, 95]. Le degré de tolérance aux fautes se mesure par la capacité du système à délivrer son service même en présence de fautes ;
- ***Elimination des fautes*** : cette méthode consiste à réduire le nombre et la sévérité des fautes dans le but de les éliminer du système ;
- ***Prévision des fautes*** : elle consiste à estimer le nombre de fautes (physiques, de conception ou malveillantes) courantes et futures ainsi que leurs conséquences. La sûreté de fonctionnement peut être illustrée par le schéma de la « Figure1.2 ».

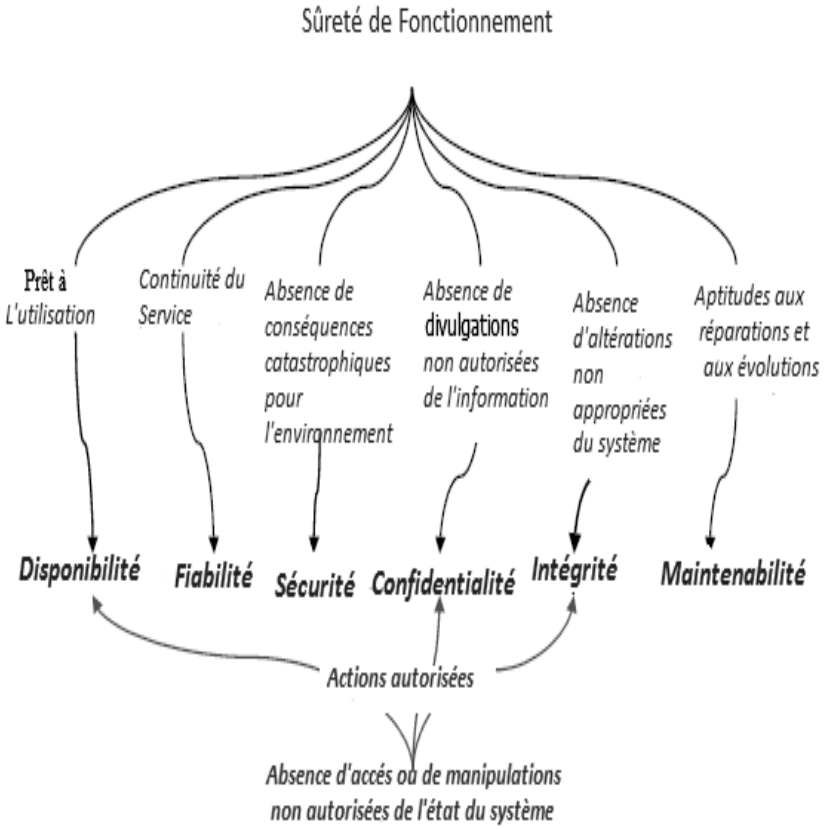

FIGURE 1.2 – Arbre de la sûreté de fonctionnement [98]

1.3 Classe des fautes

La capacité d'identification du modèle de pannes joue un rôle très important si l'on veut réaliser la tolérance aux fautes. En effet, les conséquences et le traitement d'une faute diffèrent selon son modèle. Les fautes peuvent être classées selon plusieurs critères : leur degré de gravité, leur degré de permanence et leur nature [7].

Selon leur degré de gravité

- ***Fautes d'état*** : le changement des variables d'un élément peut être la conséquence de perturbations dues à l'environnement, des attaques ou simplement des défaillances du matériel ou du logiciel utilisé. Il est par exemple possible que des variables prennent des valeurs qu'elles ne sont pas censées prendre lors d'une exécution normale du système ;
- ***Fautes de code*** : le changement arbitraire du code d'un élément résulte la plupart du temps d'une attaque, mais certains types moins graves peuvent correspondre à des bogues ou à une difficulté à supporter la charge d'un élément du système ;
- ***Fautes franches ou de type crash*** : à un point donné de l'exécution, un élément cesse définitivement d'être actif et n'effectue plus aucune action ;
- ***Fautes d'omission*** : à divers instants de l'exécution, un élément peut omettre de communiquer avec les autres éléments du système, soit en émission, soit en réception. Le composant cesse momentanément son activité puis la reprend ;
- ***Fautes byzantines*** : elles correspondent simplement à un type arbitraire de fautes, et sont donc les fautes les plus malicieuses et donc les plus complexes à tolérer. Un composant présentant ce type de faute agit de manière complètement imprévisible pour l'observateur extérieur.

Selon leur degré de permanence

- ***Faute transitoire*** : elle se produit de manière isolée ;
- ***Faute intermittente*** : elle se produit aléatoirement à plusieurs reprises ;
- ***Faute permanente*** : elle persiste dès son apparition jusqu'à ce qu'elle soit réparée.

Selon la nature de la faute

- ***Faute intentionnelle*** : elle est créée avec une intention qui pourrait s'avérer malicieuse ;
- ***Faute accidentelle*** : elle se produit de manière accidentelle.

1.4 Tolérance aux fautes dans les systèmes répartis

La tolérance aux fautes dans les systèmes répartis est un domaine de recherche qui a été et qui demeure très largement étudié [60]. Les travaux dans ce domaine se différencient principalement selon trois critères : le type de fautes prises en compte (fautes de l'opérateur, fautes logicielles ou fautes matérielles), la technique de détection de fautes utilisée et l'approche de tolérance aux fautes proposée.

1.4.1 Comment faire face aux fautes ?

Plusieurs phases successives, non obligatoirement toutes présentes, font partie d'un processus de tolérance aux fautes [60] :

- ***Détection*** : Découvrir l'existence d'une faute (état incorrect) ou d'une défaillance (comportement incorrect) ;
- ***Localisation*** : Identifier le point précis (dans l'espace et le temps) où l'erreur (ou la défaillance) est apparue ;
- ***Isolation*** : Confiner l'erreur pour éviter sa propagation à d'autres parties du système ;
- ***Réparation*** : Remettre le système en état de fournir un service correct. Le composant défectueux est identifié et le système fonctionne comme si les composants défectueux ne sont pas utilisés ou sont utilisés d'une façon telle que la faute ne cause guère de défaillance.

1.4.2 Mise en œuvre de détecteurs de défaillances

Les détecteurs de fautes sont des éléments centraux dans les systèmes répartis tolérants aux fautes. La capacité d'un détecteur de fautes de fonctionner de manière complète et efficace, en présence d'une messagerie non fiable ainsi que des composants assujettis à une forte occurrence de fautes, peut avoir un impact majeur sur la performance de ces systèmes. "La complétude" est la garantie que la défaillance d'un membre du système soit finalement détectée par tous les autres membres. "L'efficacité" signifie que les défaillances sont détectées rapidement et avec une précision acceptable. Le premier travail pour répondre à ces deux propriétés fut pris en charge par Chandra et Toueg. Ces auteurs [31] ont démontré l'impossibilité pour tout algorithme de détection de fautes d'atteindre à la fois la complétude et l'efficacité dans un système non fiable et asynchrone. Cette impossibilité résulte de la difficulté inhérente au fait

de déterminer si un processus à distance est réellement défaillant ou si ses transmissions sont simplement retardées. Il est donc impossible de mettre en œuvre un service de détection de fautes fiable sans faire plus d'hypothèses sur le système. Il existe deux grandes techniques pour la mise en œuvre de détecteurs de défaillances : celle basée sur des échanges périodiques de messages de vie, et celle basée sur des allers-retours (ping/pong).

Messages "Ping/Pong"

De manière périodique ou à la demande, les nœuds envoient un message "Ping" à tous les autres nœuds ou à une partie d'entre eux. Cette technique donne lieu à une détection plus ciblée : elle permet de ne surveiller qu'un sous-ensemble de nœuds. En revanche, pour obtenir autant d'informations qu'avec la technique d'échange de messages de vie, deux fois plus de messages s'avèrent nécessaires [110].

Echanges de messages de vie (heartbeats)

Chaque nœud envoie périodiquement un message de vie à tous les autres et attend donc, à chaque période, un message de vie de chacun d'entre eux. Lorsqu'un nœud ne reçoit pas de message de vie d'un autre nœud, il le considère comme défaillant. De nombreux projets de recherche se sont concentrés sur la fiabilité de la suspicion de défaillance, en prenant en compte la variation de latence dans l'arrivée des messages de vie d'un nœud particulier [110].

1.4.3 Techniques de tolérance aux fautes dans les systèmes répartis

La majorité des techniques de tolérance aux fautes est basée sur une duplication spatiale, qui consiste à affecter un job à plusieurs nœuds. Une duplication temporelle de l'exécution des jobs qui consiste à capter des états d'exécution de ces jobs pour les faire revenir en cas de défaillance ou de duplication informationnelle (redondance de données, codes, signatures)[78].

Tolérance aux fautes par duplication

Trois approches fondamentales du problème de la tolérance aux fautes, basées sur la duplication [68], sont proposées dans la littérature : l'approche masquante à base de duplication active, l'approche recouvrante à base de duplication passive et l'approche semi-active à base de duplication passive et active.

– **Duplication active** : La duplication active consiste à mettre en œuvre un ensemble de répliques jouant le même rôle dans un modèle client/serveurs, ainsi les serveurs qui reçoivent toutes les requêtes, les traitent (en mettant à jour leurs états internes) et envoient une réponse au client, ce dernier choisit une de ces réponses, la « Figure 1.3 » indique la duplication active ;

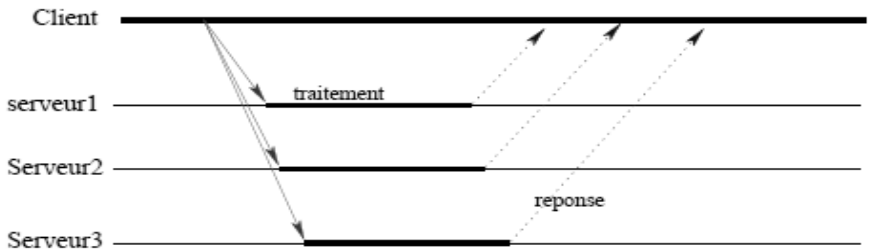

FIGURE 1.3 – Duplication active [154]

– **Duplication passive** : Comme le montre la « Figure1.4 », la duplication passive désigne un élément du groupe d'objets comme primaire, les autres sont appelés serveurs secondaires. Dans un modèle client/serveur, le client envoie sa requête uniquement au serveur primaire qui exécute le traitement. Une fois le calcul fini, le serveur primaire met à jour les secondaires et envoie la réponse au client. Si le serveur primaire n'est plus fonctionnel alors l'un des secondaires est promu et devient primaire.

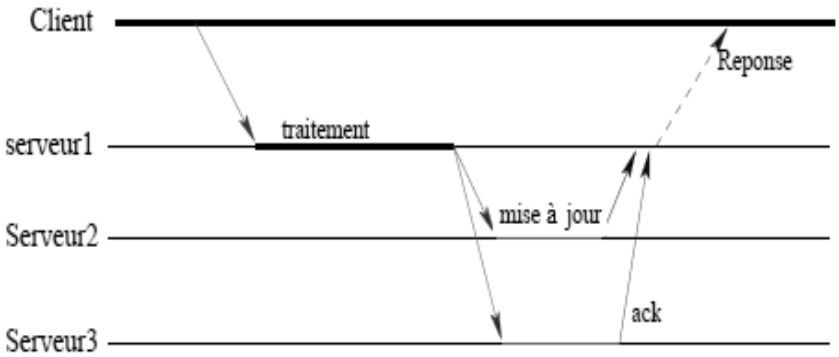

FIGURE 1.4 – Duplication passive [154]

– **La duplication semi-active** : Dans le cas de duplication semi-active, les mises à jour ne sont envoyées qu'à travers une seule copie. Cette dernière met à jour l'ensemble du groupe avant de retourner un acquittement au nœud initiateur de la mise à jour. Les

garanties offertes ici sont les mêmes que pour ce qui est de la duplication active mais il suffit de connaître la copie primaire du groupe pour pouvoir envoyer une mise à jour. C'est en revanche la technique qui induit le plus de latence lors des mises à jour (4 phases de communication contre 2 pour les autres types de duplication).

Tolérance aux fautes par rollback-recovery

Le Rollback-Recovery (RR) est une technique pour rendre un système distribué plus fiable et tolérant aux fautes. Deux approches sont essentiellement utilisées pour implémenter un système de RR [146]. Ces deux techniques sont basées sur les points de reprise (checkpoint) ou sur les fichiers logs. A la suite d'une faute, le rollbackrecovery restaure l'état du système associé au plus récent ensemble consistant de points de reprise, qui est appelé ligne de recouvrement [146]. Le rollback basé sur les fichiers logs (ou fichiers de trace) utilise le fait que l'exécution d'un processus peut être modélisée comme étant une séquence d'intervalles d'états déterministes, où chacun commence par l'exécution d'un événement indéterministe [107]. Un tel événement pouvant être la réception d'un message d'un autre processus ou un événement interne au processus même, comme par exemple les événements probabilistes [114].

1.5 Tolérance aux fautes dans les grilles de calcul

La prochaine génération scientifique nécessite une puissance de calcul et de stockage qui ne peuvent être fournies par une seule institution. De plus, un accès facile à des données distribuées est nécessaire pour améliorer le partage des résultats par les communautés scientifiques réparties autour du monde. La solution proposée à ces défis est de permettre aux différentes institutions, travaillant dans le même domaine scientifique, de mettre leurs ressources de calcul, de stockage et de données en commun afin d'atteindre les performances requises [127].
La grille est un type de système parallèle et distribué qui permet le partage, la sélection et l'agrégation de services de ressources hétérogènes répartis sur de multiple domaines administratifs en fonction de leur disponibilité, capacité, performance, coût. Comme les performances du réseau avaient bien dépassé la puissance de calcul et la capacité de stockage, ce nouveau paradigme a évolué pour permettre le partage et l'utilisation coordonnée des ressources géographiquement distribuées.

1.5.1 Définition et avantages de la grille

En 1998, Ian Foster et Karl Kesselman [49] ont posé le paradigme de la grille informatique dans leur ouvrage : The Grid Blueprint for a New Computing Infrastructure.

"*Une grille informatique ou grid est une infrastructure virtuelle constituée d'un ensemble coordonné de ressources informatiques potentiellement partagées, distribuées, hétérogènes, externalisées et sans administration centralisée.* "

Une grille est en effet une infrastructure, c'est-à-dire, des équipements techniques d'ordre matériels et logiciels. Cette infrastructure est qualifiée de virtuelle car les relations entre les entités qui la composent n'existent pas matériellement mais numériquement. Une grille garantit des qualités de service non-triviales, c'est-à-dire, qu'elle se distingue des autres infrastructures dans son aptitude à répondre adéquatement à des exigences (accessibilité, disponibilité, fiabilité, ect.) compte tenu de la puissance de calcul ou de stockage qu'elle représente.
Une grille se compose de ressources informatiques : tout élément qui permet l'exécution d'une tâche ou le stockage d'une donnée numérique. Cette définition inclut bien sûr les ordinateurs personnels, mais également les téléphones portables, les calculatrices et tout objet qui présente un composant informatique. Ces ressources sont potentiellement qualifiées de :

1. ***Partagées*** : elles sont mises à la disposition des différents consommateurs de la grille et éventuellement pour différents usages applicatifs ;
2. ***Distribuées*** : elles sont situées dans des lieux géographiques différents ;
3. ***Hétérogènes*** : elles sont de toutes natures, les ordinateurs peuvent différer par exemple, par leur système d'exploitation ;
4. ***Coordonnées*** : les ressources sont arrangées, mises en relation et politisées en vue d'une fin, d'un objectif. Cette tâche est souvent remplie par un ou plusieurs ordonnanceurs ;
5. ***Externalisées*** : les ressources sont accessibles à la demande chez un fournisseur externe ;
6. ***Non-contrôlées*** : les ressources ne sont pas contrôlées par une unité commune. Contrairement à un cluster, les ressources sont hors de la portée d'un moniteur de contrôle.

1.5.2 Architecture de la grille

L'architecture de la grille est souvent décrite en termes de « couches » [49], chacune fournit une fonction spécifique. En général, les couches les plus élevées sont concentrées sur l'utilisateur, tandis que les couches inférieures sont concentrées sur des ordinateurs et des réseaux (matériel-centraux). La « Figure1.5 » indique les couches de la grille.

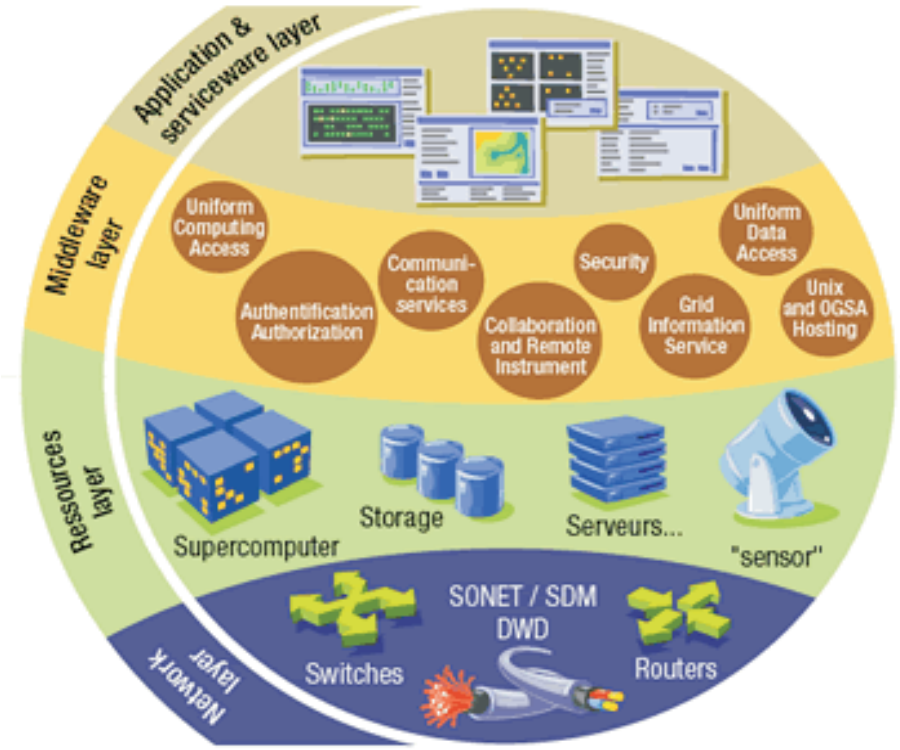

FIGURE 1.5 – Architècture de la grille [11]

À la base de tout, la couche du fond est **la couche réseau** « Network layer », qui assure la connectivité pour les ressources dans la grille, et sur laquelle se trouve **la couche de ressources** « Resources layer ». Cette couche est composée des ressources réelles qui font partie de la grille, tels que des ordinateurs, des systèmes de stockage, des catalogues de données électroniques, ou d'autres instruments, qui peuvent être reliés directement au réseau. **La couche de logiciel personnalisé** « Middelware layer » fournit les outils qui permettent les divers éléments (serveurs, stockage, réseaux, etc.) pour participer à un environnement unifié de grille. La couche de logiciel personnalisé peut être considérée comme le cerveau de la grille (réunissant les divers éléments). La couche la plus élevée de la structure est **la couche application** « Application and Serviceware layer », qui inclut l'ensemble des

différentes applications d'utilisateur (science, technologie, affaires, financières). C'est la couche que les utilisateurs de la grille « voient ».

1.5.3 Topologies de grilles

On peut répertorier les grilles d'un point de vue topologique en trois types par ordre croissant d'étendue géographique et de complexité : IntraGrilles (Intragrids), ExtraGrilles (Extragrids) et InterGrilles (Intergrids).
La « Figure 1.6 » indique les topologies de grilles [142].

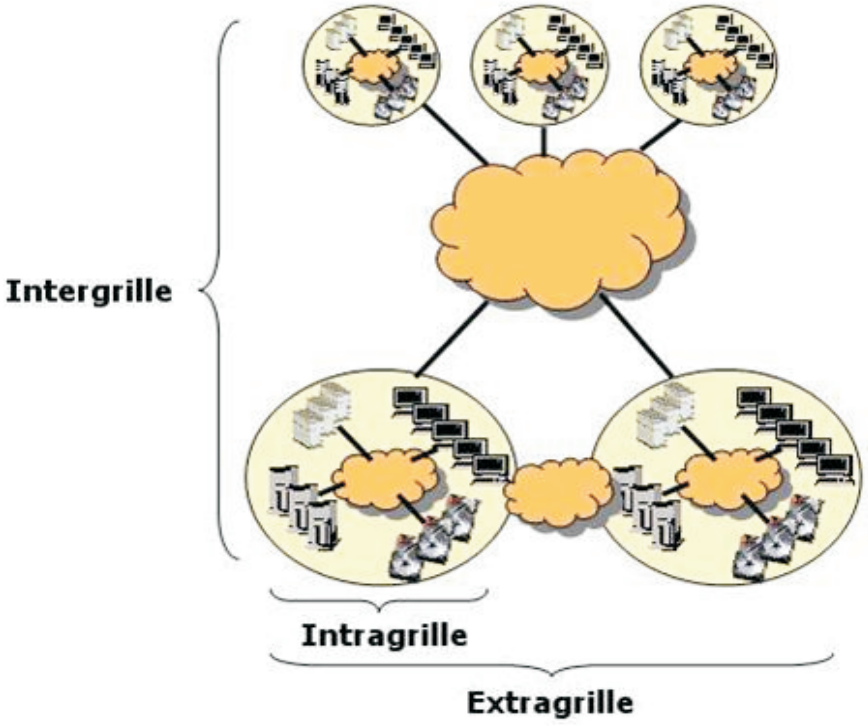

FIGURE 1.6 – Topologies de grilles de calcul [11]

1. **Intra grille** (en analogie avec Intranet) : la plus simple des grilles est l'intra grille, composée d'un ensemble relativement simple de ressources et de services et appartenant à une organisation unique. Les principales caractéristiques d'une telle grille sont la présence d'un réseau d'interconnexion performant et haut débit, d'un domaine de sécurité unique et maîtrisé par les administrateurs de l'organisation et d'un ensemble relativement statique et homogène de ressources.

2. **Extra grille** (en analogie avec Extra net) : une extra grille étend le modèle en agrégeant plusieurs intra grilles. Les principales caractéristiques d'une telle grille sont la présence d'un réseau d'interconnexion hétérogène haut et bas débit (LAN/WAN), de plusieurs domaines de sécurité distincts, et d'un ensemble plus ou moins dynamique de ressources.
3. **Inter grille** (en analogie avec Internet) : une inter grille consiste à agréger les grilles de multiple organisations, en une seule grille. Les principales caractéristiques d'une telle grille sont la présence d'un réseau d'interconnexion très hétérogène haut et bas débit (LAN/WAN), de plusieurs domaines de sécurité distincts et ayant parfois des politiques de sécurité différentes et même contradictoires, et d'un ensemble très dynamique de ressources. Les inter grilles seront souvent mises en œuvre lors de grands projets industriels (conception d'un avion par un consortium aéronautique par exemple) ou scientifiques (modélisation de protéines) où plusieurs organisations seront amenées à participer.

1.5.4 Domaines d'application des grilles de calcul

De par les potentialités qu'elle offre, une grille est un excellent moyen de partage des données et de puissance de calcul. Ce principe de mise en commun des ressources est utilisé dans une multitude de projets industriels et scientifiques. Les domaines d'utilisation des grilles étant très diversifiés [11].

Calcul distribué

Les applications de calcul distribué sont évidemment d'excellentes candidates pour être exploitées sur une grille. Elles bénéficient ainsi d'un nombre beaucoup plus important de ressources de calcul leur permettant de résoudre des problèmes qui leur étaient auparavant inaccessibles.

Calcul à haut débit

Contrairement aux calculs distribués, les applications de calcul à haut débit nécessitent la résolution d'un très grand nombre de calculs de manière indépendante et parfois coordonnée. Les ordinateurs, lorsqu'ils sont inutilisés, peuvent ainsi contribuer à agréger une puissance de calcul pour résoudre des applications très exigeantes en matière de puissance de calcul. Le projet SETI@home est un exemple d'infrastructure qui a permet d'agréger une puissance de

calcul très importante uniquement à partir de périodes d'inactivité des processeurs de simples ordinateurs de type PC.

Calcul à la demande

Ce type d'applications nécessite de la puissance de calcul pour des besoins ponctuels. Plutôt que d'acheter des ressources qui ne seront utilisées qu'occasionnellement ou pour un objectif bien précis, il serait plus judicieux de pouvoir utiliser, en cas de besoin, les ressources d'autres utilisateurs notamment pendant la période de leur inactivité. Pour rendre cette nouvelle forme d'utilisation des ressources possible, il est primordial de développer des systèmes performants et sécurisés d'utilisation des ressources, même dans le cas ou elles sont payantes.

Traitement massif de données

Dans ce type d'applications, le but est d'extraire de nouvelles informations à partir de grandes bases de données géographiquement distribuées (fouilles de données, bio-informatique, météorologie, etc.). Généralement, ces types de traitement sont également de grands consommateurs de puissance de calcul et de bande passante. Les principaux problèmes à résoudre, pour ce type d'application, sont l'acheminement et la configuration de données très importantes à travers les éléments d'une grille.

Travail collaboratif

Les grilles de calcul permettent non seulement de regrouper (virtuellement) des ressources informatiques hétérogènes, mais aussi de fédérer des milliers de personnes réparties sur des entreprises et institutions différentes, et ce dans le but d'un travail collaboratif. Dans ce contexte, les grilles, de par leurs caractéristiques, constituent une véritable plate-forme de travail collaboratif à moindre coût.

1.5.5 Projets sur les grilles de calcul

Nous présentons dans cette section quelques exemples de simulateurs et de systèmes de gestion de grilles :

Condor

C'est un système de gestion de ressources conçu pour supporter un calcul à haut débit, en allouant les requêtes des applications aux ressources oisives du réseau. La principale fonction de Condor est d'exploiter les machines oisives. Cependant, il peut être configuré pour partager des ressources, il supporte également les applications séquentielles et parallèles. Dans ce système, les tâches soumises par les utilisateurs sont mises dans une file d'attente et ordonnancées, de manière transparente pour l'utilisateur, aux machines disponibles qui sont gérées dans un Condor pool. Une tâche distante ne peut s'exécuter sur une machine, que si celle-ci est oisive. Autrement dit, la machine n'a aucune tâche locale. Une fois que la machine dispose d'une tâche locale, la tâche distante qui s'exécutait sera interrompue et migrée vers une autre machine oisive [8].

Legion

C'est un système à base d'objets qui permet de connecter des réseaux, stations de travail, supercalculateurs, et d'autres ressources de calcul dans un système pouvant renfermer différentes architectures, systèmes d'exploitation, et sites géographiques. Dans Legion, toutes les composantes matérielles et logicielles sont représentées par des objets ; chaque objet est géré par sa propre classe qui peut activer ou désactiver un objet, créer de nouvelles instances et les ordonnancer pour être exécutées. Un utilisateur peut redéfinir des classes en ajoutant ou supprimant des fonctionnalités pour répondre à ses besoins. Legion [9] définit une API pour l'interaction des objets, non pas un langage de programmation ni un protocole de communication.

Globus

C'est un projet de recherche fournissant les capacités et les services de bases (sécurité, allocation et gestion de ressources, réservation de ressources, communications, etc.) requis pour la construction de grilles de calcul. Il offre une infrastructure logicielle permettant aux applications de manipuler les ressources hétérogènes et distribuées comme s'il s'agissait d'une seule machine virtuelle. Globus est un support pour de nombreux langages, modèles de programmation, et applications [9]. Il offre un ensemble de services parmi lesquels, on peut citer :

– ***Grid Security Infrastructure (GSI)*** : qui offre des outils pour l'authentification et

la sécurité des transactions dans la grille (certificat d'autorité, cryptographie,ect.) ;

– ***Metacomputing Directory Service (MDS)*** : un annuaire de ressources qui enregistre les machines permettant la réalisation de la grille ;

– ***GridFTP*** : version spécialisée de FTP pour les grilles ;

– **Globus Resource Allocation Manager (GRAM)** [51] : qui permet l'allocation des ressources et la gestion des processus ;

– ***Global Access to Secondary Storage (GASS)*** : permettant l'accès à distance aux données via des interfaces séquentielles et parallèles ;

– ***Global Advanced Resource Reservation and Allocation (GARA)*** : en mesure de réserver des resources.

1.5.6 Défis des grilles de calcul

Il y a de nombreux défis qui doivent être abordés afin de construire un environnement de grille de calcul. La liste suivante montre les principales exigences d'une grille qui doivent être satisfaites [59, 159] :

– **Services d'information** : concerne des informations sur les ressources disponibles sur la grille. Il comprend l'ensemble des ressources disponibles, les spécifications matérielles ainsi que les informations dynamiques comme la charge de travail. Ces informations doivent être automatiquement maintenues en état et à jour ;

– **L'accès uniforme aux ressources** : toutes les ressources du même genre (éléments de calcul, éléments de stockage) doivent être accessibles d'une manière uniforme, et peu importe sur quelles technologies ou normes elles sont basées ;

– **Sécurité** : Les mécanismes de sécurité sont nécessaires afin de permettre aux administrateurs du système d'appliquer les règles d'accès pour toutes les ressources mises à disposition sur la grille. Ce point est fortement lié à la notion d'organisation virtuelle ;

– **Ordonnancement des tâches** : les travaux soumis par les utilisateurs doivent être placés de manière efficace ;

– **Accès aux données** : les utilisateurs de grille devrait être en mesure d'accéder à des données distribuées de manière uniforme ;

– **La réplication des données** : la grille devrait permettre la création automatique des répliques de fichiers dans l'ordre de rapprocher les données à l'utilisateur ou à l'équipement informatique qui les traitera. C'est aussi un moyen d'augmenter la tolérance aux fautes.

1.5.7 Fiabilité des grilles de calcul

Plus le nombre de nœuds est important, plus la probabilité d'occurrence d'une faute est grande [72]. Par exemple, Considérons un système composé de n processeurs. Par définition la fiabilité R(t) de ce systéme est la probabilité qu'il soit opérationel pour tout instant $t_i \epsilon$ [0, t]. Dans [143], les auteurs montrent que $R(t) = e^{\lambda t_n}$ où λ est une constante représentant l'intensité de défaillance sur un processeur [98]. La constante est l'inverse du temps moyen entre deux fautes, notée (MTBF) du système, λ=1/MTBF [143]. Si nous considérons un MTBF= 2000 jours pour chaque processeur et le temps d'exécution varie d'un jour à 30 jours. La probabilité de défaillance augmente considérablement en fonction du nombre de processeurs utilisé dans le système et de leur temps d'exécution. La probabilité que le système ne soit pas opérationel est la non fiabilité du système sur l'intervalle de temps [0, t] ; elle s'écrit :

$$F(t) = 1 - R(t) \tag{1.3}$$

Ainsi, comme la « Figure 1.7 » l'illustre, la probabilité de défaillance des applications parallèles réparties augmente considérablement en fonction de la durée d'exécution et du nombre de processeurs utilisés.

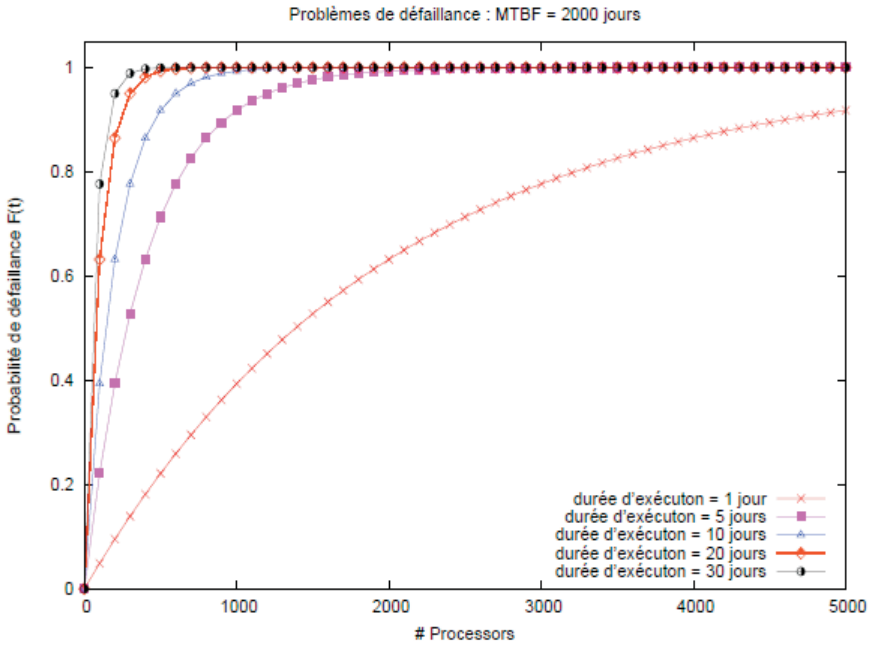

FIGURE 1.7 – Probabilité de défaillance dans les environnements de grille [78]

1.5.8 Classe de fautes pour les grilles de calcul

Avec la croissance de la fonctionnalité, la complexité et le passage à l'échelle des grilles de calcul, les défaillances de ressources sont inévitables. Pour les applications scientifiques, les défaillances peuvent entraîner une dégradation fréquente des performances, ou dans le pire des cas, l'arrêt prématuré de l'exécution ou la corruption de données et leurs pertes. Pour les applications commerciales, les défaillances peuvent entraîner la violation des accords au niveau du service, et causer une perte dévastatrice de clients et de revenu [90, 102, 109]. De manière générale, les défaillances dans une grille de calcul peuvent être :

Des Fautes franches

Le grand nombre de nœuds présents dans une grille de calcul implique une éventualité non négligeable de fautes (défaillances franches) de ces nœuds. Ces fautes sont nettement envisageables.

Des Fautes par omission

La présence de certains équipements réseau tels que les routeurs, notamment entre les différentes institutions composant une grille de calcul, rend possibles certaines défaillances par omission. Lorsque le réseau est congestionné, les routeurs ignorent, en effet, volontairement des messages pour limiter la congestion. Des pertes de messages peuvent également avoir lieu lorsqu'un nœud reçoit des paquets réseau plus vite qu'il ne peut les traiter.

Des Fautes des middlewares

Un middleware permet la communication entre les clients et les serveurs ayant des structures et des implémentations différentes. Il permet l'échange d'informations dans tous les cas et pour toutes les architectures. Enfin, le middleware doit fournir un moyen aux clients pour trouver leurs serveurs, aux serveurs pour trouver leurs clients et en général de trouver n'importe quel objet atteignable [139, 162]. Les principaux types de défaillances sont liés à l'environnement de configuration. Selon certaines études dans [91], le manque de contrôle des ressources sur la grille est la principale source des fautes de configuration

1.5.9 Détection de fautes dans les grilles de calcul

S'appuyant sur les travaux antérieurs sur la détection des fautes et des défaillances dans les systèmes répartis [72, 98, 157], les chercheurs ont étudié les méthodes scalables de détection de fautes pour les environnements de grille. Des travaux [46, 58] ont également porté sur les techniques de reconnaissance des différents types de fautes, particulièrement les fautes difficiles à détecter, qui devraient se produire entre les ressources hétérogènes dans des conditions dynamiques, y compris l'isolation des fautes et les méthodes de diagnostic.

1.5.10 Problèmes des méthodes de détection de fautes dans les systèmes répartis

Les méthodes de détection de fautes développées pour les systèmes répartis ne conviennent pas généralement aux systèmes de grille à grande échelle, dynamiques et hétérogènes et ceci est dut à :

– Les protocoles disponibles de surveillance du réseau et des outils, tels que ceux basés sur le protocole SNMP (Simple Network Management Protocol) s'appuient sur une connaissance approfondie de la structure du réseau. Ces informations sont susceptibles d'être toujours disponibles dans des environnements de grilles ;

– Pour la détection des défaillances se produisant dans les systèmes distribués asynchrones, quand les fonctions de gestion (y compris les détecteurs de défaillance) sont décentralisées et peuvent elles-mêmes tomber en panne. Ces conditions sont susceptibles d'être rencontrées dans les systèmes de grille à grande échelle ;

– Le manque de capacités de détection des fautes dans les environnements scalables. Les auteurs dans [92, 169] ont conduit une enquête sur 19 systèmes de surveillance des grilles basés sur l'architecture GGF de monitoring de grille [74], qui prescrit les exigences pour une fonction de surveillance afin de supporter la détection de fautes. L'enquête a révélé que la plupart de ces systèmes ont été conçus sans mécanismes de détection de fautes, et n'ont pas la possibilité de passer à l'échelle dans leurs fonctions de surveillance. En outre, ces lacunes ont motivé la recherche sur les détecteurs de fautes dans les grilles de calcul.

1.5.11 Méthodes de détection de fautes dans les grilles de calcul

Les premiers travaux sur un détecteur distribué de fautes pour le middleware Globus abordent les issues de complétude et de précision [48, 120] en permettant aux détecteurs de défaillances, potentiellement peu fiables, de communiquer l'information sur la probabilité de défaillance d'une ressource. L'approche a également été conçue pour améliorer la flexibilité, l'efficacité et la scalabilité par le découplage du monitoring, la détection et les fonctions de notification. Dans [157], un détecteur de fautes a été proposé pour accomplir la scalabilité en organisant les ressources de la grille en groupes de vie sur la base de la topologie du réseau. Les auteurs dans [74] ont proposé un système de détection de fautes scalables et auto-organisé basé sur les travaux antérieurs des protocoles de l'appartenance à un groupe et de détecteurs de fautes dans les systèmes répartis. Dans cette approche [72], chaque processus est surveillé par un petit groupe (4 ou 5) processus choisis au hasard sur des nœuds distants. Le processus de monitoring établit une connexion au processus surveillé et transmet périodiquement des messages de vie. Il en résulte la création d'un sous-réseau virtuel de surveillance dans une grille, qui peut être constitué de ressources hétérogènes. Dans [74], les ressources affectées à une application sont placées dans des domaines distincts où ils émettent des messages de vie aux moniteurs des domaines, qui sont organisés de façon hiérarchique. Nous avions classifié les approches de détection de fautes scalables comme suit :

– ***Approche hiérarchique*** : Les auteurs dans [16] proposent une version hiérarchique de détecteurs de défaillances utilisant des échanges de messages de vie [15, 114]. L'ensemble des nœuds est partitionné en sous-ensembles et les échanges de messages de vie de type : tous-vers-tous, sont restreints à chaque sous-ensemble. Chaque sous-ensemble choisit un représentant et les différents représentants des sous-ensembles s'échangent des messages de vie. Cette approche permet d'obtenir des détecteurs de défaillances adaptés aux systèmes à grande échelle comme les grilles.

– ***Approche basée sur un anneau logique*** : L'utilisation d'un anneau logique déterminant les schémas d'échanges de messages de vie et de propagation de l'information est proposée par les auteurs dans [56]. De même que pour l'approche hiérarchique, cela permet de limiter le nombre de messages de vie émis sur le réseau et d'offrir le passage à l'échelle.

– ***Approche probabiliste et apprentissage automatique*** : Une autre approche permettant de concevoir des détecteurs de défaillances pour les systèmes à grande échelle est l'utilisation de techniques probabilistes [89, 138, 137]. Dans SWIM [37], les nœuds envoient

des messages de type "ping", périodiquement à un ensemble aléatoire de nœuds. De plus, les messages "ping" et "pong" contiennent les informations sur les nœuds suspects permettant ainsi de propager les détections.

– ***Approches basées sur des modèles mathématiques*** : Ces modèles [116] ont émergé comme des approches éminentes pour le diagnostic des fautes des systèmes à temps discret ou continu. Ces approches [17, 141] sont basées sur des modèles mathématiques du processus contrôlé, de sorte que les résidus peuvent être calculés en prenant la différence entre les valeurs estimées des variables de sortie du système et les valeurs mesurées. Les résidus sont ensuite comparés à des seuils appropriés pour la détection des fautes afin de fournir une décision assurant le bon fonctionnement du système.

– ***Approches basées sur la fouille de données*** : L'exploitation des techniques de la fouille de données pour la détection des fautes dans les systèmes à large échelle a fait l'objet de plusieurs travaux de recherche [52]. Dans [33] les auteurs présentent une approche basée sur l'apprentissage supervisé des arbres de décision. Le système nécessite des exemples étiquetés de défaillance et une connaissance du domaine. Dans [117] les auteurs explorent la détection de fautes des machines virtuelles en utilisant des arbres de décision. Le système supervisé, nécessite un apprentissage sur des exemples étiquetés et des compteurs sélectionnés manuellement. Les auteurs dans [132] comparent trois approches de prédiction des fautes : à base de règles, réseau bayésien et d'analyse de séries chronologiques. Ils ont appliqué avec succès leurs méthodes sur un cluster de 350 nœuds pour une période d'une année. Leurs méthodes qui furent supervisées, s'appuient en outre sur la connaissance substantielle du système surveillé. Dans [35] les auteurs induisent une classification par les réseaux bayésiens basée sur un arbre augmenté (Tree-augmented Bayesian network). Bien que cette approche ne nécessite pas de connaissances de domaine autre que d'un ensemble d'apprentissage étiqueté.

– ***Les protocoles Gossip*** [55] : Ces protocoles sont basés sur du gossiping, inspirés du service de détection de fautes proposé par les auteurs dans [158]. Le principe du gossiping est que chaque élément du système réparti à surveiller est accompagné d'un détecteur local. Ces détecteurs locaux s'échangent de manière périodique, l'état de vitalité de ressource qu'ils ont en la responsabilité, mais aussi leurs connaissances les plus récentes de l'état de vitalité concernant les ressources distantes.

1.5.12 Méthodes de tolérance aux fautes dans les grilles de calcul

Beaucoup de travaux sur la tolérance aux fautes se focalisent sur les grilles de données en utilisant des techniques de réplication [90, 102]. Pour assurer la tolérance aux fautes dans les grilles de calcul, on utilise généralement les mêmes protocoles pour les clusters avec quelques modifications dans certains cas [40, 53, 151]. Les méthodes de tolérance aux fautes dans les grilles de calcul peuvent être classer en :

Checkpoint et recouvrement

Comme dans d'autres systèmes [75, 80], cette méthode est largement utilisée dans les réseaux. Toutefois, si le hôte tombe en panne, le processus peut être transféré vers un environnement d'exécution différent. La migration d'un processus, qui est incapable de poursuivre sur son processeur d'origine, vers un nouveau processeur est connue sous le nom de basculement (failover) [39, 164].

– ***Points de reprise et de récupération dans les systèmes de grille*** : La technique de checkpoint et les méthodes de migration de processus ont été largement utilisées dans des environnements de calcul à haute performance [41, 69]. De nombreuses grilles composées d'une fédération de clusters, exploitent le checkpoint et les techniques de migration pour un ensemble de processus parallèles qui sont fondées sur les méthodes utilisées dans le calcul à haute performance [67]. Les premiers efforts dans l'utilisation des points de reprise et de migration de processus dans les réseaux à grande échelle ont été signalés dans le middleware Légion [112] et Cactus [64, 97]. Les auteurs dans [129] ont proposé un framework pour la tolérance aux fautes se composant de deux entités logicielles, nommées : Service de Tolérance aux Fautes (FTS), et client de Tolérance aux Fautes (FTC). Le FTC se trouve sur chaque machine physique dans le Cloud et communique avec un FTS, qui peut donc communiquer avec plusieurs FTC. Ils ont défini l'ensemble des modules composant le FTC qui sont nécessaires à l'implémentation de la Tolérance aux Fautes, par exemple : un module de Checkpointing, un module de prédiction des pannes, ou encore, un module d'analyse des communications des VMs. Du côté du FTS, ils ont également défini un ensemble de modules assurant, entre autres, la détection des pannes, le rétablissement du système, ou encore, le placement des VMs et des points de sauvegarde. Les auteurs dans [130] ont évoqué la possible indisponibilité du point de sauvegarde au moment de la panne d'une VM. Pour

répondre à ce problème, ils ont proposé une approche de Checkpointing multi-zones dans laquelle plusieurs points de sauvegarde sont créés pour chaque VM . Pour minimiser la probabilité de perte des points de sauvegarde. Ces auteurs ont défini plusieurs zones de pannes. Chaque zone de pannes contient alors un point de sauvegarde qui est mis à jour selon une fréquence qui peut différer d'une zone à une autre. Les auteurs dans [131] se sont penchés sur le calcul de l'intervalle de Checkpointing . Cet intervalle définit la fréquence avec laquelle des points de sauvegarde sont créés et transférés vers des mémoires secondaires pour chaque VM. Lorsque cet intervalle est grand, peu de points de sauvegarde sont créés. Ainsi, les VMs sont plus vulnérables aux pannes. D'un autre côté, lorsque cet intervalle est petit, beaucoup de points de sauvegarde sont créés et le Checkpointing a un impact plus important sur les performances. Plusieurs travaux ont entrepris d'élaborer un compromis pour fixer cet intervalle. Dans cette contribution [131], les auteurs ont abordé ce problème avec deux différences par rapport aux travaux existants. Premièrement, ils ont supposé que la taille des points de sauvegarde est non pas fixe, mais relative au temps d'exécution des VMs entre deux points de sauvegarde, ce qui est plus réaliste. Et deuxièmement, ils ont considéré la possible présence de techniques de prévision de pannes. Ces techniques sont complémentaires au Checkpointing et permettent de réduire son impact sur les performances sans avoir une incidence négative sur la tolérance aux fautes des VMs.

– ***Méthodes de recherche en checkpointing*** : Trois stratégies de checkpointing ont été considérées pour les processus concurrents dans [170]. Des points de reprises coordonnés ont été implémentés dans MPI, en LAM/MPI [134]. Les auteurs dans [165] ont proposé une version tolérante aux fautes de MPI, MPICH-GF, pour les grilles qui utilisent les points de reprise coordonnés avec blocage. D'autres extensions de MPI utilisant les points de reprise coordonnés ont été proposées pour les environnements de grille [19] et en particulier dans [80], où la performance a été démontrée dans des conditions limitées de scalabilité. Récemment, les auteurs dans [136] présentent le Fault Tolerant Messaging Interface (FMI), pour une tolérance aux fautes rapide et transparente. FMI fournit des message-passing sémantique similaires à MPI, mais les applications écrites avec FMI peuvent fonctionner malgré l'occurrence d'une défaillance. Les auteurs dans [128] ont proposé deux approches. La première est non-coordonnée et permet de réduire la latence réseau en créant des points de sauvegarde forcés lorsque c'est nécessaire. Pour ce faire, ces auteurs ont défini, entre autres, un délai maximum de rétention des messages de communication. La seconde approche

est coordonnée et consiste à regrouper les VMs en clusters virtuels, puis à coordonner le processus de Checkpointing et de rétablissement dans chaque cluster. Les communications des VMs vers d'autres clusters ont toujours besoin d'être retenues mais ce n'est plus le cas pour les communications dans le même cluster. Pour maximiser le gain en performance, ils ont établi une fonction de coût et donné une heuristique pour que le groupement des VMs dans des clusters se fasse de façon optimum.

– ***Migration de processus dans des environnements hétérogènes*** : Des exemples de travaux préliminaires qui précèdent l'émergence de réseaux sont décrits [122, 168], Pour les systèmes de grille, le problème a été étudié dans [155]. Dans [45] les auteurs ont proposé une méthode pour le transfert de processus concurrents d'une application parallèle dans un environnement de grille ayant des architectures de processeurs différents, ainsi que différentes implémentations de MPI. D'autres travaux de recherche relatifs aux points de reprise dans les environnements de grille comprennent l'élaboration de méthodes pour stocker les données des points de reprise dans des annuaires distribués, ou pour sélectionner des intervalles de point de reprise optimaux [23, 126].

Réplication des ressources d'une grille

Les approches de réplications sont basées sur :

– ***Sélection des répliques et méthodes de placement*** : Les auteurs dans [163] ont proposé un système de gestion des répliques qui alloue dynamiquement les ressources répliquées sur la base des demandes des utilisateurs.

Le système de recherche SRIRAM [160], pour la réplication automatique des ressources informatiques dans les grilles et d'autres environnements répartis, a été conçu pour améliorer la disponibilité des ressources et la tolérance aux fautes. D'autres systèmes de réplication ont été proposés, ainsi dans [156] l'auteur a proposé une approche de réplication de services où les répliques sont situées les unes à proximité des autres pour former des îles dans le réseau d'une grille. Une approche de réplication et d'ordonnancement distribuée et évolutive, appelée DistReSS, a été présentée par les auteurs dans [30]. Dans le cadre du projet E-Demand [150], il a été proposé une méthode de réplication qui détecte les fautes de calcul dans un workflow de grille composé de plusieurs tâches. Les auteurs dans [57] ont développé un mécanisme pour MPI qui utilise une réplication de ressources visant à accroître la tolérance aux fautes des calculs parallèles. Par ailleurs, d'autres méthodes pour la ré-

plication des calculs sur les ressources ont été proposées par les auteurs dans [3, 21]. Ces méthodes ont été vérifiées expérimentalement dans des conditions de passage à l'échelle. Les auteurs dans [106] avaient étudié une approche simple où une application toute entière, est répliquée. Fournissant ainsi un modèle théorique d'un schéma d'exécution avec réplication lorsque la distribution des fautes suit une loi exponentielle. Ils proposèrent des algorithmes de détermination des dates de sauvegarde quand la distribution des fautes suivait une loi quelconque.

– ***Synchronisation des répliques*** : Peu de travaux semblent avoir été faits en ce qui concerne la proposition de méthodes efficaces et scalables pour la synchronisation des répliques. Une méthode, basée sur le placement de répliques sélectif, a été proposée par les auteurs dans [156, 170]. Dans [96], les chercheurs ont proposé une version optimisée de l'algorithme Paxos pour la synchronisation des répliques dans des environnements répartis et avaient ainsi démontré l'efficacité de leur approche dans des conditions locales et à large échelle. Lors de travaux antérieurs [171], une approche plus traditionnelle de copie primaire a été utilisée pour étudier la réplication des services de grille a été implémentée en utilisant OGSI et Globus Toolkit [48]. Dans [171], il a été constaté que cette stratégie pourrait être facilement implémentée aboutissant de la sorte à une plus grande disponibilité de services dans des environnements locaux.

Les applications de type "Diviser pour régner"

"Diviser pour régner" est un paradigme populaire et efficace pour développer des applications parallèles de la grille. Le paradigme "Diviser pour régner" est une généralisation du paradigme maître/esclave recommandé par le Global Grid Forum, en tant que représentation efficace pour le développement des applications de grille. Les applications "Diviser pour régner" fonctionnent en divisant un problème de manière récursive en sous-problèmes. La subdivision récursive continue ainsi jusqu'à ce que le sous-problème devienne trivial. Après la résolution des sous-problèmes, les résultats sont combinés de façon récursive jusqu'à ce que la solution finale soit trouvée [65, 145].

1.6 Conclusion

Dans ce chapitre, nous avons présenté la notion de sûreté de fonctionnement d'un système informatique. Nous avons vu que la tolérance aux fautes n'est qu'un moyen pour assurer la sûreté de fonctionnement, nous avons fait un survol des différentes techniques de détection et de tolérance aux fautes. Les politiques de tolérance aux fautes comme la réplication et le recouvrement donnent des résultats très satisfaisants dans les systèmes répartis ordinaires. Le passage à l'échelle de ces systèmes, qui s'est accentué par l'apparition des clusters et plus tard par les grilles de calcul, a créé des contraintes nouvelles, comme la dynamicité et l'hétérogénéité. Cela a conduit les chercheurs à faire des choix stratégiques, entre la mise à jour des anciennes techniques de tolérance aux fautes et à les adapter aux nouveaux systèmes ou le développement de nouvelles techniques répondant convenablement aux nouvelles contraintes. Le chapitre suivant présente une idée sur les workflows scientifiques et les algorithmes d'ordonnancement de workflows, ainsi que les algorithmes d'ordonnancement de workflow tolérants aux fautes.

Chapitre 2

Ordonnancement de workflow dans les grilles de calcul

Sommaire

2.1 Introduction

Depuis quelques années déjà, les technologies de grille offrent aux utilisateurs l'accès à une multitude de services et de puissances de calcul pour traiter des données ou en générer de nouvelles. Ces perspectives ont donné un nouvel essor à la modélisation des applications et des services qui s'appuie sur l'utilisation des workflows « flux de travail » afin d'organiser l'utilisation des applications ou des services disponibles sur une grille informatique [18]. Dans ce chapitre, nous nous proposons d'étudier le problème d'ordonnancement de workflow.

Tout d'abord, nous commençons par une présentation des workflows scientifiques, et la modélisation en graphe des applications. Nous exposons dans un deuxième temps les différentes heuristiques utilisées classiquement pour résoudre le problème d'ordonnancement d'un graphe de tâches sur un ensemble de processeurs hétérogènes. Nous exposons ensuite d'une manière particulière un tour d'horizon des gestionnaires d'exécution de workflow dans les grilles de calcul et nous conclurons par l'exposition des différentes heuristiques d'ordonnancement de workflow tolérantes aux fautes.

2.2 Définition d'un workflow

Les workflows sont utilisés dans un nombre important de domaines. Il existe les workflows de gestion de processus métier [1], c'est à dire l'organisation et la formalisation de plusieurs tâches pour décrire une activité. Ils se nomment aussi organigrammes de processus ou logigrammes. Ainsi, dans une entreprise, les processus métier se décrivent par la dépendance entre les activités :

- création d'un produit : conception, réalisation, commercialisation, etc. ;
- gestion de documents : acquisition, classement, stockage, diffusion, etc. ;
- achat de produits : commande, paiement, livraison, etc. ;
- prise de commandes : commande, livraison, paiement, etc. ;

Cette schématisation de l'activité permet de rationaliser et d'identifier les différentes étapes. Le traitement de l'information, et donc l'informatique, relève d'un workflow. Le groupe workflow management coalition [1] propose la définition suivante :

1. Business Process Management.

"La gestion des workflows est l'automatisation des processus métiers ou «workflows», pendant laquelle les documents, l'information ou les tâches sont transmises d'un acteur à un autre suivant des règles et des procédures établies."

Cette rationalisation n'est pas le propre de l'informatique, elle se retrouve dans beaucoup de domaines. La modélisation est intimement liée à la notion d'ordonnancement des activités. En effet, une fois les processus définis et schématisés par un graphe, un ordre de traitement est établi : il est naturellement déterminé par le parcours du graphe suivant les arêtes.
Une des premières méthodes de suivi et d'organisation dans la gestion de projets est la méthode PERT [2]. Introduite dans les années cinquante dans la marine américaine, elle a permis de gérer la fabrication des missiles à ogive nucléaire Polaris. Cette technique consiste à mettre sous la forme d'un graphe les dépendances entre les activités d'un projet.

2.3 Les workflows scientifiques

En informatique, et plus particulièrement dans le calcul scientifique, une formalisation en graphe est aussi employée pour la modélisation d'applications venant de domaines aussi variés que les neurosciences, l'imagerie médicale, la physique des molécules, l'astronomie, ou encore la biologie cellulaire et moléculaire. Nous désignons ces applications par des workflows scientifiques. Elles sont comme leurs homologues des processus métiers d'une entreprise ou la recette de cuisine d'un grand chef, une succession d'étapes qui permet l'étude d'un problème scientifique. Ce formalisme apporte plusieurs avancées pour l'utilisateur, il permet d'exprimer un mécanisme complexe d'analyse par un graphique simple, flexible et dynamique. Il décrit les interactions et la logique scientifique entre les différentes étapes. Lorsque ces workflows mettent en action une succession d'applications informatiques, ils expriment aussi le parallélisme intrinsèque de l'exécution des tâches qui seront exécutées par une ressource informatique. Aussi, dans ce contexte, des domaines comme la bio-informatique s'organisent-ils autour d'un portail internet collaboratif d'échange et de publication de workflows scientifiques. Il est important de noter qu'à ce stade un workflow peut avoir des étapes itératives, ou des branches conditionnelles, comme il peut avoir un langage de workflow s'apparente à un

2. l'acronyme de : Program (ou Project) Evaluation and Review Technique.

langage de programmation. Le génie logiciel a poussé ce formalisme en introduisant l'UML [3] qui permet de décrire graphiquement les données et les traitements effectués. Ce langage fournit au concepteur de logiciel un moyen d'exprimer toutes les dépendances, toutes les données et tous les traitements effectués dans un logiciel. Pas moins de 13 types de diagrammes différents existent pour conceptualiser un logiciel. Cependant, ce formalisme bien que reconnu et standardisé par l'OMG [4], il reste dans le domaine du génie logiciel et de l'ingénierie. Il n'a pas encore sa place dans les domaines des sciences utilisant les workflows scientifiques même s'ils ont aussi une vocation visuelle de déroulement d'étapes, de méthodologie et d'algorithme de traitement [62].

2.4 Modélisation des applications

Une multitude d'applications informatiques se modélisent sous la forme d'un graphe orienté. La résolution d'un système linéaire $Ax = \mathrm{b}$ est un cas d'école. A est alors une matrice triangulaire inférieure inversible de taille $n \times n$, et b un vecteur à n composantes. Il existe encore un nombre d'applications d'imagerie travaillant sur une image et utilisant plusieurs algorithmes (programmes) différents pour analyser, filtrer et décomposer l'image. Le même genre de décomposition est appliqué pour les workflows scientifiques. Nous verrons par la suite la modélisation, sous la forme de graphe de tâches. Cependant, il est important de faire une distinction entre les différents types de workflows que nous retrouvons dans la littérature. Tristan Glatard propose dans sa thèse une classification basée sur le contenu du langage d'expression d'un workflow [61] :

– **Fonctionnelle** : la description de l'application est de haut niveau. Tous les types d'opérateurs de composition sont disponibles : boucles, conditions, etc. Les données en entrée des programmes ne sont pas forcément connues et, par exemple, le nombre d'itérations d'une boucle peut dépendre du résultat d'un programme. Tous les langages de programmation tombent dans cette catégorie ;

– **Services** : comme pour la classe précédente la description est de haut niveau, néanmoins une information supplémentaire est ajoutée permettant, ainsi, de définir la ressource qui exécutera le programme ;

3. Langage de Modélisation Unifié
4. Object Management Group

– **Tâches** : la description de l'application étant concrète, toutes les itérations et les conditions sont connues, il n'y a pas de boucles ni de conditions, l'ensemble des opérations est par conséquent connu ;

– **Exécutable** : comme pour la classe précédente, toutes les tâches sont connues, la ressource qui exécutera la tâche étant tout aussi désignée.

– **Modèle formel** : Il permet de faire des analyses théoriques sur la description et le déroulement de l'application.

Nous représentons les 4 classes d'expression des workflows dans la « Figure 2.1 ».

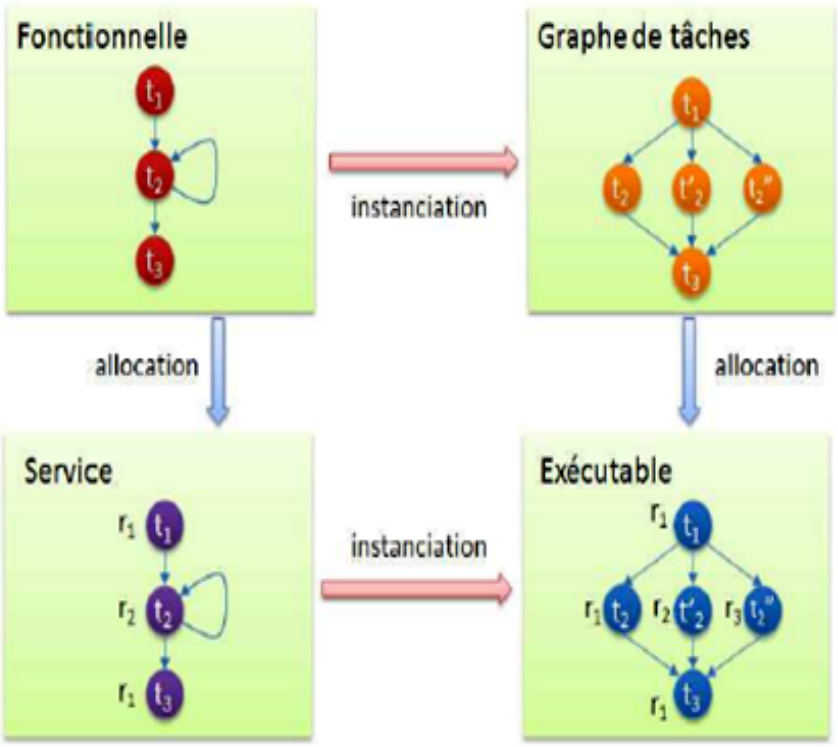

FIGURE 2.1 – Classification des workflows [18]

Il existe des opérations permettant de passer d'une classe à l'autre. Ainsi nous appelons instanciation l'opération qui consiste à évaluer chaque boucle, l'opération de contrôle, les valeurs de retour, etc., afin de connaître le nombre de tâches et les données transférées. L'opération de spécification des ressources est nommée allocation. Un exemple de passage entre les différentes classes de workflows est donné dans la « Figure 2.1 ».

Ici, la modélisation sera restreinte à des graphes acycliques dirigés qui peuvent être appelés workflows concrets, ou aussi DAG (Directed Acyclic Graph), ou simplement workflow par abus

de langage. Soit un graphe valué G(T,D,w,c) ou :

– l'ensemble T des sommets représente les tâches ;

– l'ensemble D des arêtes représente les dépendances entre les tâches ;

– la fonction de coût/poids d'un sommet w : $T \rightarrow R$, représente la quantité de travail nécessaire pour la tâche considérée ;

– La fonction de coût/poids associée à une arête c : $E \rightarrow R$, représente le volume des données transférées entre deux tâches.

L'existence d'un lien (d_{ij}) entre deux tâches (t_i) et (t_j) traduit l'ordre partiel qui existe entre elles, c'est-à-dire que l'exécution de la tâche (t_j) ne peut commencer avant que l'exécution de (t_i) ne soit terminée et que les données ne soient transmises. Une tâche sans aucune arête entrante est appelée tâche d'entrée (t_1), une tâche sans aucune arête sortante est appelée tâche de sortie (t_3).

Cette modélisation met en évidence le parallélisme d'une application, ainsi dans l'exemple de la « Figure 2.1 » on voit qu'un maximum de 3 tâches (t_2, $t_2^{'}$, $t_2^{''}$) peut être exécuté en parallèle après l'exécution de t_1.

2.5 Modélisation des ressources

Il existe plusieurs manières de définir les ressources d'une grille informatique. De façon générale, celles-ci sont aussi modélisées à l'aide d'un graphe dirigé valué, ou les sommets représentent les ressources de calcul et les arêtes le lien entre deux ressources.

2.5.1 Définition de topologie

Soit un ensemble de ressources R $\{r_u,\ 1 \leq u \leq |R|\}$ et un ensemble C de connexions entre deux ressources r_u et r_v, $c_{u,v}$ la capacité de ce lien, la forme du graphe $\wp$ = (R,C) représente la topologie de la plate-forme. Il existe une quantité pléthorique de topologies : des chaînes linéaires, des bus de communication, des anneaux, des étoiles, des cliques, des arbres, des hypercubes, des mailles ou filets, et des combinaisons mixtes des différentes topologies énoncées.

2.5.2 Définition des liens

Une fois la topologie de la plate-forme définie, il faudrait définir ses liens. Ils sont caractérisés par une fonction de coût associée au transfert d'une donnée. Cette fonction peut être plus ou moins élaborée suivant le degré de réalisme de la modélisation. Elle dépend du protocole qui est utilisé dans le réseau interconnectant les machines. Cette fonction de coût est le plus souvent basée sur la notion de bande passante qui dénote la vitesse de transfert d'une donnée et la notion de latence qui modélise le temps d'initialisation d'une communication. Nous définissons la manière de communiquer selon trois types de modèles de transfert :

- mono-port : une machine ne peut communiquer qu'avec une seule autre machine ;
- multi-port : une machine peut communiquer avec plusieurs machines à la fois ;
- multi-port borné : une machine peut communiquer avec plusieurs machines à la fois, la somme de toutes les communications ne peut pas excéder la bande passante totale de sortie ou d'entrée.

Les modèles de transferts peuvent être asymétriques ou symétriques, c'est-à-dire qu'une machine peut ne pas avoir les mêmes caractéristiques selon que l'on considère les communications entrantes ou sortantes (exemple : protocole ADSL). Les liens d'une plate-forme peuvent être aussi tous homogènes, ou différents (hétérogènes) suivant chaque lien considéré [42, 44].

2.5.3 Définition des ressources de calcul

Il convient de caractériser les sommets du graphe de la plate-forme, c'est-à-dire les ressources de calcul, par une fonction de coût. Celle-ci se base sur la vitesse de calcul des processeurs, il serait également possible de la baser sur la quantité de mémoire. Comme pour les liens de communication, on définit aussi un modèle d'exécution d'une tâche :

- mono tâche (une seule tâche à la fois)
- partagé : plusieurs tâches en même temps ;
- préemptif : une tâche peut être commencée, puis arrêtée pour être remplacée par une autre puis reprise de nouveau ;
- non-préemptif : une fois la tâche commencée on ne pourra plus l'arrêter sans avoir à la recommencer depuis le début. Les ressources qui composent la plate-forme sont qualifiées :
- homogènes : toutes les ressources sont identiques, ayant toutes les mêmes caractéristiques ;
- hétérogènes uniformes : les ressources ont des caractéristiques différentes ; mais il existe

une relation entre les propriétés des ressources ;

– hétérogènes non liées : les ressources ont des caractéristiques totalement différentes dépendant de la nature de tâche attribuée.

Etant donné qu'il existe plusieurs caractéristiques pour une machine de calcul (vitesse de calcul, mémoire, etc.), il est possible d'imaginer, par exemple, des combinaisons dans l'homogénèité et l'hétérogénéité des vitesses et de la mémoire [73, 88, 103].

2.6 Ordonnancement et allocation de ressources

L'ordonnancement d'un graphe de tâches sur une plate-forme de grille pose la question de l'attribution d'une date de début et d'une allocation de ressources pour l'exécution des tâches qui composent le graphe. Cet ordonnancement doit respecter les contraintes de précédence définies par les liens dans le DAG. La difficulté dans le choix de la machine et dans l'ordonnancement des tâches réside dans la prise en compte à la fois des tâches à exécuter et des transferts de données entre les tâches.

Le problème de décision associé au problème d'ordonnancement d'un graphe de tâches sur un ensemble fini de machines est NP-complet [50]. En termes simples, la théorie de la complexité pour les problèmes informatiques définit la difficulté de trouver une solution qui satisferait les contraintes pour l'objectif fixé. En particulier, la classe NP est comme la classe des problèmes dont on peut vérifier les solutions en temps raisonnable (polynomial en la taille des données). Parmi ceux-là, on distingue les problèmes NP-complet, dont il est impossible de trouver la ou les solutions exactes autrement qu'en les énumérant et en les vérifiant toutes. Une manière, plus informelle pour définir la classe NP : c'est la classe des problèmes pour lesquels les seuls algorithmes connus de résolution sont ceux ayant une complexité exponentielle en la taille des données.

Il existe un formalisme de classification des problèmes dans le domaine de l'ordonnancement des problèmes d'atelier (en anglais job-shop, flow-shop, Open-shop . . .). L'ordonnancement d'atelier correspond au domaine qui s'intéresse aux problèmes d'optimisation des usines de confection de pièces ou de construction. Cette classification, issue des ouvrages de référence sur la théorie de l'ordonnancement [36], et de l'article [66], introduit une notation à trois paramètres ($\alpha \mid \beta \mid \gamma$). Ils décrivent les instances du problème d'ordonnancement.

– α décrit les ressources du problème ;

– β décrit les contraintes et les caractéristiques du système ;
– γ décrit l'objectif fixé dans les critères d'optimisation.

Nous renvoyons le lecteur intéressé au livre de Peter Brucker [20] et au site internet de l'auteur qui référence les problèmes étudiés et les résultats sur la complexité de leur résolution.
En utilisant la notation ($\alpha \mid \beta \mid \gamma$), le problème d'ordonnancement d'un graphe de tâches sur une plate-forme grille hétérogène se note par $(Q_m \ |prec, c|\ \mathrm{C}_{max})ou(\mathrm{R}_m \ |prec, c|\ C_{max})$ suivant les applications et les ordinateurs. Dans ces deux notations prec exprime la contrainte de précédence, c exprime les contraintes de communication entre les tâches, C_{max} correspond au critère d'optimisation recherché, ici, la minimisation du temps de terminaison de l'application (makespan). Enfin Q_m et R_m représentent respectivement un ensemble de **m** machines uniformément hétérogènes et un ensemble de **m** machines dont les temps d'exécution dépendent de la nature des tâches exécutées et de la machine sélectionnée [100, 119].

2.7 Catégories d'ordonnancement

Pour résoudre un problème d'ordonnancement NP-complet, des heuristiques (ou intuitions) sont utilisées, elles permettent de s'approcher d'une solution acceptable en faisant certains choix. Il existe un nombre pléthorique d'heuristiques développées dans le domaine de l'ordonnancement de graphe de tâches incluant les méthodes de branch and bound (séparation et évaluation), programmation dynamique, algorithme génétique. Il existe deux grands types d'algorithmes d'odonnancement :

2.7.1 Ordonnancement statique

Les algorithmes d'ordonnancement de charges dit statiques, que l'on peut définir à la compilation (réalisés avant l'exécution de l'application). Les stratégies statiques utilisent des approches multiples pour calculer à priori, une distribution optimale des tâches parmi les différents processeurs Elles se fondent essentiellement sur une connaissance préalable de différents paramètres associés aux tâches (comme les coûts de calcul et de communication). Une bonne modélisation et analyse statique ne permettent d'obtenir une bonne solution qu'à condition de disposer de ces informations préalables (chose qui n'est pas toujours possible) et d'avoir un système assez stable pour que la non utilisation de l'état dynamique du système reste une démarche réaliste [94]. Il couvre :

– heuristique de liste ;
– heuristique de clustering de tâches ;
– heuristique de duplication de tâches ;
– méta-heuristique.

Les heuristiques de liste

Les heuristiques de liste maintiennent une liste de tâches triées suivant une priorité. Ces heuristiques sont toutes basées sur la succession des deux étapes suivantes :

– Prendre une tâche parmi celles qui sont prêtes et/ou pas encore allouées à une machine. Une tâche devient prête lorsque toutes les tâches parentes sont terminées et, que les données nécessaires aux calculs sont disponibles.

– Sélectionner une machine pour exécuter la tâche et allouer celle-ci à la machine.

La première étape est aussi nommée phase de prioritisation car elle permet de faire un choix lorsqu'il existe plusieurs tâches disponibles. La priorité attribuée aux tâches permet de donner un ordre d'exécution. Cet ordre est déjà induit par l'ordre partiel défini par la relation de précédence qui se retrouve avec une traversée topologique du graphe de tâches. Les valeurs fréquemment utilisées pour construire une métrique de priorité sont :

– **t-level** : qui correspond à la longueur d'un plus grand chemin depuis le nœud d'entrée jusqu'à la tâche considérée (t_i). La longueur du chemin étant définie par la somme des valuations des tâches et des arêtes tout en excluant la valeur de la tâche (t_i). Parfois, cette valeur est appelée ASAP [5] car elle correspond à la date de commencement au plus tôt de la tâche.

– **b-level** : qui correspond à la longueur d'un plus grand chemin depuis la tâche (t_i) jusqu'au nœud de sortie.

– **s-level** : qui correspond comme pour le b-level à la longueur d'un plus grand chemin jusqu'au nœud de sortie sans compter la valuation des arêtes.

– **ALAP** [6] : qui correspond à la différence entre la longueur d'un chemin critique et la valeur du b-level du nœud considéré. C'est aussi la date « au plus tard » de la tâche. La deuxième étape est la phase d'allocation. Il faut choisir le processeur qui exécutera la tâche sélectionnée à l'étape précédente. Là encore, les critères de choix sont nombreux.

5. As Soon As Possible
6. As Late As Possible

Les principaux répertoriés [152] sont min-min, suferage, max-min, mct [7], srpt [8], lrpt [9]. À ce stade, il faut allouer une tâche sur une plate-forme hétérogène, compte tenu des choix faits auparavant. Les heuristiques de liste sont, la plupart du temps, utilisées car une garantie sur la qualité de l'ordonnancement fut démontrée. En effet, dans le cas des processeurs homogènes, et sans considérer les communications entre les tâches du DAG, les travaux de Graham montrent que toute heuristique de liste obtient un ordonnancement dont la durée totale d'exécution est, au pire, à cinquante pour cent de l'ordonnancement optimal (que l'on ne connaît pas). On parle de rapport de compétitivité, il est exactement égal à 2 -($\frac{1}{p}$) avec p le nombre de processeurs homogènes disponibles. Dans le cas d'hétérogénéité, en considérant les temps de communication, cette garantie ne tient plus, mais l'on obtient néanmoins de bonnes performances, en effet, plusieurs articles l'ont vérifié par simulation [24]. Une multitude d'heuristiques de liste a été proposée, pour en citer les principales : Heterogeneous Earliest Finish Time (HEFT)[148, 149], Critical Path on Processor (CPOP)[148, 149], Levelized Min-Time(LMT)[77], Generalized Dynamic Level(GDL)[105], SDC[174].
La « Figure 2.2 » indique un exemple d'un DAG.

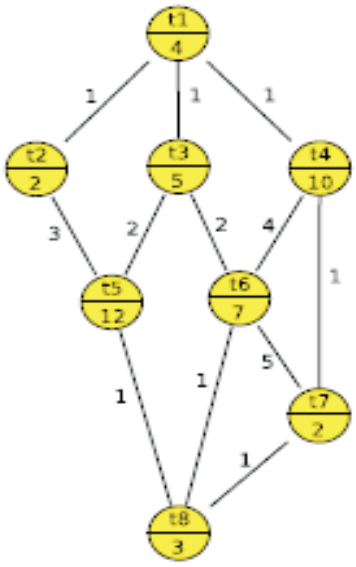

FIGURE 2.2 – Exemple de DAG [18]

7. Minimum Completion Time
8. Shortest Remaining Processing Time
9. Longest Remaining Processing Time

Clustering de tâches

Les heuristiques de groupement de tâches (clustering algorithms) reposent sur l'idée de regrouper des tâches ensemble dans le but de supprimer les communications trop coûteuses. Les algorithmes fonctionnent en deux phases :

– la première phase groupe les tâches du DAG. Le groupement est effectué par une succession d'étapes de raffinement sans retour en arrière : une fois les tâches groupées, le groupement d'une étape précédente n'est pas remis en cause. Une méthode classique consisterait à rassembler deux tâches qui s'échangent une grande quantité de données. Cette phase permet d'obtenir un nouveau DAG dont les nœuds représentent cette fois un groupement de tâches, la « Figure 2.3 » indique le groupement de tâches selon la première phase.

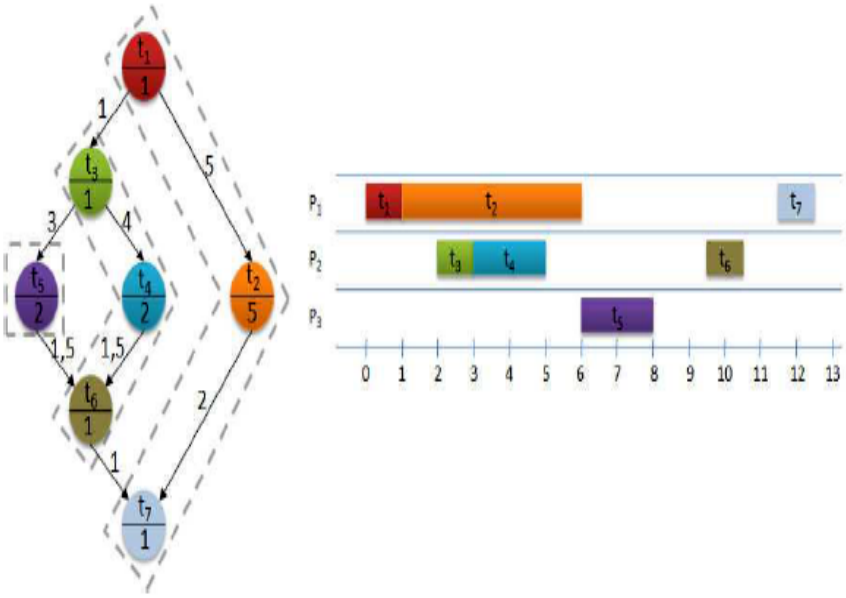

FIGURE 2.3 – Exemple1 de groupement de tâches [18]

– la deuxième phase consiste à trouver un ordonnancement et une allocation sur un processeur pour les groupes de tâches construits à l'étape précédente. Ainsi, aucune communication inter-machine ne sera faite au sein d'un groupe puisque qu'elles seront toutes exécutées par la même machine, la « Figure 2.4 » indique le groupement de tâches selon la deuxième phase.

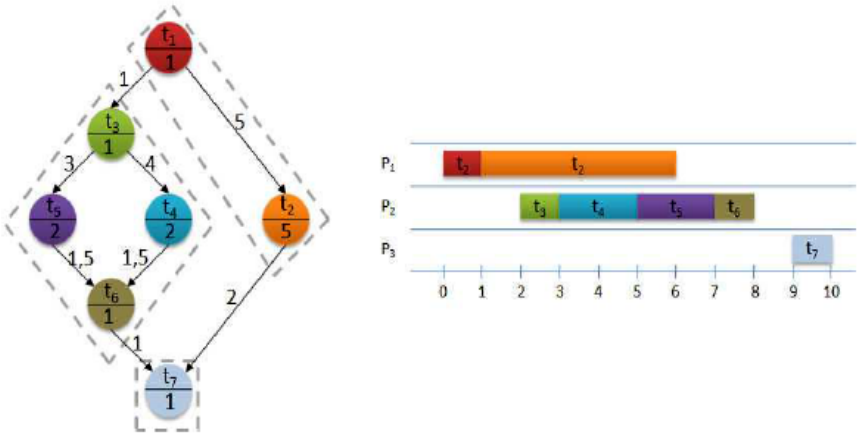

FIGURE 2.4 – Exemple2 de groupement de tâches [18]

Duplication de tâches

Les heuristiques à duplication de tâches consistent à allouer de manière redondante certaines des tâches « importantes » dont d'autres dépendent. Le but recherché est donc de réduire le temps avant que les tâches en attente ne puissent commencer, ce qui peut éventuellement améliorer le temps d'exécution global de l'application. Afin de bien comprendre l'utilité de ces heuristiques, il est présenté dans la « Figure 2.5 » un exemple d'ordonnancement d'une application sur 2 machines (par simplicité nous supposons que les machines et les liens de communication sont homogènes).

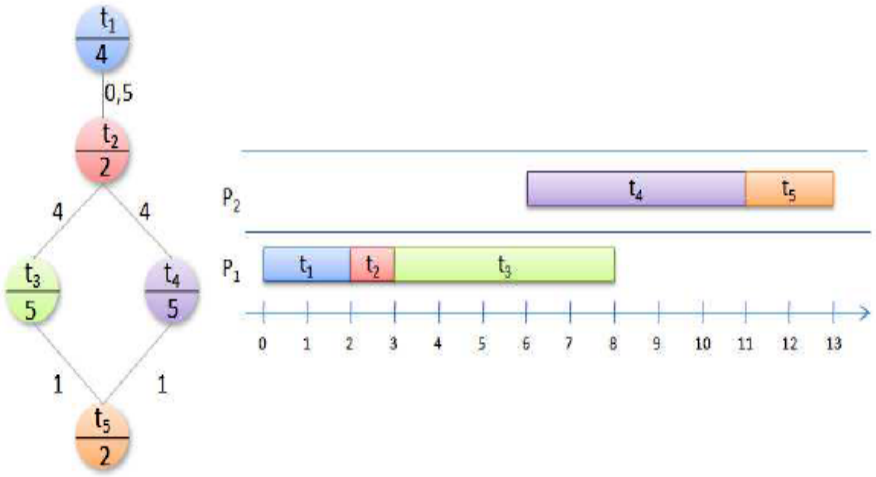

FIGURE 2.5 – Exemple d'ordonnancement d'un graphe de tâches sur deux processeurs [18]

La valuation des tâches représente leur durée d'exécution, et celle des arêtes représente la durée de transfert des données entre les deux tâches. Ici, le temps de terminaison de l'application est de 13 unités de temps. Dans l'ordonnancement précédent, beaucoup de temps est perdu à envoyer les données en sortie de la tâche (t_2) à la machine qui exécute la tâche (t_4). Si nous dupliquons la tâche (t_2) (de sorte que la machine p2 n'ait pas à attendre le transfert de la donnée issue de (t_2), puisqu'elle aura été effectuée aussi sur cette machine), tout se passe comme si nous avions désormais le graphe de tâches présenté dans la « Figure 2.5 ». La durée d'exécution de l'application sera alors réduite à 10,5 unités de temps. Dans l'absolu, si toutes les tâches hors chemin critique sont dupliquées, on obtient un ensemble de graphes de tâches de type chaîne dépendante entre elles. Ces techniques [4, 76, 123], ont été conçues pour des environnements où le nombre de ressources est important.

Méta-heuristiques

Le terme de méta-heuristiques englobe toutes les techniques qui recherchent dans l'espace des solutions possibles pour l'ordonnancement du graphe de tâches sur une plate-forme grille. Ces méthodes sont souvent utilisées face à des problèmes NP-Complet où la meilleure solution ne peut être trouvée autrement qu'en explorant toutes les solutions possibles. L'idée est donc d'explorer de manière «intelligente» l'espace des solutions afin de trouver une meilleure solution en fonction de l'objectif. La différence entre les méthodes réside dans «l'intelligence» d'exploration des solutions possibles. Bien que ce genre de techniques soit très intéressant et donnent souvent de bons résultats, il ne s'avère pas adapté adaptées à des problèmes où l'on doit trouver un ordonnancement rapidement. En effet, plus la taille du problème (nombre de tâches, nombre de dépendances, nombre de ressources) augmente, plus l'espace des solutions à explorer est grand. De plus, il faut relancer une recherche à chaque modification du problème initial.

À titre d'illustration, on citera les travaux de Mezmaz et al. [108] qui s'est servi de la grille de calcul Grid'5000 pour résoudre le problème d'ordonnancement d'atelier flow-shop Ta56 (50 travaux sur 20 machines). Il aura fallu plus de 22 ans de temps processeur cumulé pour trouver la solution exacte au problème de minimisation Ta56. Les principales méthodes utilisées dans ce domaine sont la programmation par contrainte logique, les algorithmes génétiques, la recherche tabou et le recuit simulé.

2.7.2 Ordonnancement dynamique

Les approches statiques ne sont pas toujours applicables aux grilles de calcul, à la prise en compte de reconfigurations et de pannes [93]. À l'inverse, les stratégies purement dynamiques ne demandent aucun préalable à l'exécution. Les décisions qu'elles sont amenées à prendre n'utilisent que des informations obtenues en temps réel au cours de l'exécution. Elles sont donc particulièrement adaptées aux architectures distribuées [24, 25, 29]. Elles s'utilisent avantageusement lorsque le système permet la migration de tâches, ce qui offre plus de liberté au processus de distribution de tâches et quelques références vers celles-ci. Il peut être intéressant d'utiliser un algorithme pseudo-statique pour le mappage initial des tâches [34]. parmi lesquels nous pouvons trouver :

– Dans [34], les auteurs proposent une méthode d'ordonnancement dynamique pour des applications composées de tâches avec des dépendances, ces tâches sont représentées sous forme de DAGs, tenant compte de l'état de la grille et des estimations de tous les temps d'exécution des tâches. L'objectif de l'algorithme proposé est de maximiser le nombre de travaux traités dans un intervalle de temps particulier, qui est réalisé au coût d'une exécution plus lente de différentes tâches. L'algorithme place les tâches dans l'ordre de leur arrivée dans la grille et de telle manière que les tâches parallèles finissent plus ou moins simultanément. L'idée derrière cette approche est que les tâches ayant les mêmes dépendances s'exécutent en même temps (autant que faire se peut).

– Dans [63], les auteurs présentent un modèle d'ordonnancement des tâches dans des environnements de grille de calcul. Un algorithme d'ordonnancement dynamique est proposé afin de maximiser l'utilisation des ressources et minimiser les temps de traitement des tâches. L'algorithme proposé est basé sur le groupement de tâches. Les résultats montrent que l'algorithme d'ordonnancement proposé réduit efficacement le temps de traitement des tâches.

– Les auteurs proposent dans [43] une évaluation de 12 heuristiques d'ordonnancement des tâches dépendantes dans les environnements de grille. Afin de faciliter l'évaluation des performances, un simulateur de DAG est mis en œuvre, fournissant ainsi, un ensemble d'outils pour la configuration, l'exécution et le suivi des tâches du DAG.

2.8 Gestionnaires d'exécution de workflow

Il existe beaucoup de gestionnaires d'exécution de workflows pour les grilles de calcul. À vrai dire, la plupart des grilles actuellement utilisées propose un système permettant de prendre en compte des dépendances entre des travaux à exécuter. Nous décrivons quelques uns d'entre eux :

– ***Pegasus/DAGMan*** [2] est un gestionnaire d'exécution de workflows qui s'appuie sur le module DAGman de l'intergiciel Condor-G pour l'allocation des tâches sur les ressources informatiques. Celui-ci travaille principalement sur la structure du graphe de tâches afin de le réduire pour permettre une exécution efficace. Il utilise donc d'abord un partitionnement du graphe de tâches en sous-groupes de tâches (heuristiques de clustering présentées auparavant). Les sous-groupes de tâches sont alors soumis à DAGman pour l'exécution. DAGman place les tâches dans une liste et alloue les calculs prêts aux machines disponibles. Lorsqu'une tâche se termine, ses tâches filles sont à leur tour démarrées.

– ***MOTEUR*** [61, 153] est un gestionnaire de flots de calcul. Il permet la description de flots applicatifs complexes avec un schéma de composition de données dans un formalisme très compact. La prise en compte transparente du parallélisme de services et de données est exprimée implicitement en séparant le graphe de flot et les données. MOTEUR implante des interfaces pour les services standards de type service Web ou GridRPC.

– ***GridAnt*** [6] est un système de gestion des workflows côté client. Il a été conçu en 2002 à l'Argonne National Laboratory pour les utilisateurs de grilles de calcul dans le but d'être un outil pratique pour exprimer (spécifier des pré-conditions et des tâches exécutables en parallèles) et contrôler les séquences d'exécution. GridAnt ne fournit pas de mécanisme d'allocation automatique des ressources, l'utilisateur doit lui-même spécifier les machines qui exécuteront les travaux contenus dans son workflow.

– ***Triana*** [105] est un environnement permettant la construction et l'exécution de workflows. Il est développé à l'Université de Cardiff pour le calcul distribué grâce aux services web. Il inclut le support de GridLab [140] GAT (Grid Application Toolkit) pour la soumission de jobs, la gestion des données, des fichiers et de la sécurité. Triana est surtout un outil permettant la création d'applications à partir des services disponibles sur la grille qu'il adresse. Le moteur d'exécution qu'il utilise permet d'appeler un service sur une machine spécifiée par l'utilisateur. Il déroule donc le scénario préétabli par l'utilisateur en

respectant les contraintes de dépendance. Il ne fait pas à proprement parler d'allocation ni d'ordonnancement dans le choix des travaux à effectuer.

– ***GridFlow*** [27] est un gestionnaire d'exécution de workflows développé à l'Université de Warwick, il s'appuie sur le gestionnaire de grille ARMS [26] (Agent-based Resource Management System for grid Computing) et sur TITAN, le gestionnaire d'exécution de tâches sur des ressources locales. Il intègre une interface permettant de construire dynamiquement le graphe de dépendances entre plusieurs activités. Une activité peut elle-même représenter une application avec des dépendances (sub-workflows). L'allocation des activités qui composent le workflow s'effectue en simulant l'exécution à partir des données recueillies par le moniteur de ressources (MDS). L'exécution est gérée à deux niveaux, le module ARMS s'occupe du workflow global tandis que le module TITAN gère l'exécution des activités. La technique globale d'ordonnancement et d'allocation s'apparente à une heuristique de liste. L'allocation des activités qui composent le workflow peut aussi s'effectuer sous la contrainte de l'utilisateur.

– ***Taverna*** [2] est le gestionnaire de workflows du projet myGrid. Il propose une interface graphique pour composer son application à partir de services Web. Il permet de définir un graphe d'appel à des services qu'il composera avec une description des données sur lesquelles le même graphe de services sera appelé. Il existe des opérateurs de compositions de données (all-to-all, one-to-one) ajoutant un formalisme permettant d'exprimer le parallélisme sur les données d'entrée et de sortie des services. Dans ce gestionnaire de workflows de services, il n'y a pas de stratégie d'ordonnancement et d'allocation globale du graphe de tâches sur les ressources, étant donné qu'elles sont déjà définies avant l'appel. Cependant les invocations peuvent être exécutées de manière concurrente si les services utilisés supportent des appels concurrents.

– ***ASKALON*** [81] est un projet développé par l'Université d'Innsbruck. Il permet à un développeur désirant utiliser les ressources d'une grille de composer son application à partir d'une interface graphique. ASKALON propose deux modes de représentation des workflows. Le premier, basé sur l'expression fonctionnelle avec le langage (AGWL), et l'autre, basé sur les graphes de tâches (CGWL). L'ordonnancement proposé par ce gestionnaire de workflows s'appuie sur des heuristiques de listes avec la possibilité de définir différentes politiques de sélection d'une tâche prête et d'une ressource adéquate. Ils ont aussi implanté des méta-heuristiques de type « recuit simulé ».

2.9 Ordonnancement de workflows tolérant aux fautes

Plusieurs études ont été faites sur l'ordonnancement tolérant aux fautes [79], les auteurs dans [10] introduisent un mécanisme de tolérance aux fautes appelé ordonnanceur pour l'équilibrage de charge et le masquage des défaillances. Les défaillances sont manipulées sans qu'il soit nécessaire de les détecter. En affectant une tâche à plusieurs processeurs, cela garantit également qu'une machine défaillante ou une machine lente sera remplacée automatiquement par une machine plus rapide. L'auteur dans [135] utilise un mécanisme de tolérance aux fautes basé sur la crédibilité d'ordonnancement. Ce mécanisme fournit une tolérance aux fautes de processeurs bénévoles malveillants. Javelin dans [113] utilise un ordonnancement avancé qui améliore la scalabilité. Il construit une arborescence pour suivre l'état d'un calcul. La sélection de l'hôte pour réaffecter le job utilise un algorithme basé sur une arborescence. Si un hôte tombe en panne, Javelin fournit une tolérance aux fautes à l'aide d'un schéma de réparation de l'arborescence. Les auteurs dans [111] répliquent les points de reprise pour un mécanisme de tolérance aux fautes. La réplication est basée sur un algorithme d'ordonnancement des tâches, qui calcule le minimum du temps global d'accomplissement d'un job (Minimum Total Time to Release MTTR). Le temps d'accomplissement d'un job comprend le temps de transmission des données et du job, le temps d'attente et le temps de son exécution sur une ressource. Le MTTR sélectionne la ressource en se basant, aussi, sur les exigences du job et les caractéristiques des ressources demandées.
D'autres algorithmes d'ordonnancement de workflow tolérants aux fautes ont été proposés et peuvent être classés en fonction de :

– ***Checkpointing*** : Dans les approches de point de reprise et les approches de retour en arrière, l'état de la tâche en cours d'exécution est enregistré à des points fixes pendant son exécution. Ces points sont appelés points de contrôle. L'état sauvegardé de la tâche est utilisé pour la récupération d'une ressource en cas de défaillance. Les auteurs dans [86] ont décrit un mécanisme pour l'ordonnancement de workflow tolérant aux fautes en considérant le checkpointing et la migration des tâches. Les auteurs dans [54] ont proposé une stratégie d'ordonnancement de tâches tolérante aux fautes pour les tâches indépendantes ainsi que les tâches dépendantes nommée FTTS. Cette stratégie prend en compte le taux d'échec et la capacité de calcul des ressources. Dans [172], les auteurs proposent de nouvelles

approches combinant le checkpointing avec les algorithmes d'ordonnancement de workflow existants. Le checkpointing peut améliorer la fiabilité du système, mais engage beaucoup plus de temps sur l'enregistrement des points de reprise et le recouvrement.

– ***Réplication*** : Des méthodes comme la réplication des tâches ne nécessitent aucun historique d'informations. La tâche est répliquée sur plusieurs ressources, le nombre de répliques est généré soit d'une manière statique ou dynamique en utilisant l'historique du taux de défaillance d'un nœud. Dans [3], l'auteur a proposé une stratégie d'ordonnancement de workflow basée sur la réplication. Dans [144], les auteurs avaient proposé un algorithme d'ordonnancement de workflow tolérant aux fautes nommé FTSW, qui fournit un ordonnancement tolérant aux fautes d'un job en maintenant une table de valeurs de groupe pour les travaux orphelins (les travaux non capables de contacter leurs parents en cas d'une panne du processeur) et permet aux applications de poursuivre leur exécution en cas d'échec d'une ressource. Dans cet algorithme, des copies répétées des tables sont maintenues, ce qui conduit à un calcul redondant.
Dans [14], les auteurs ont proposé FTSA (Fault Tolerant Scheduling Algorithm), une extension de l'algorithme HEFT [149]. Il attribue $(\varepsilon + 1)$ copies pour chaque tâche aux différentes ressources pour tolérer un nombre arbitraire de fautes. Ce mécanisme conduit à une grande consommation de ressources [121] ;

– ***Resoumission*** : resoumission essaie de ré-exécuter des jobs pour plaider les échecs. La resoumission ou la redondance dans le temps permet de se remettre de défauts transitoires ou d'erreurs logicielles. Dans [172], les auteurs ont proposé une nouvelle approche qui combine les techniques de tolérance aux fautes avec des algorithmes d'ordonnancement de workflow existants. Dans [118], les auteurs proposent un algorithme appelé Resubmission Impact (RI) et qui essaie d'établir une métrique décrivant l'impact de la soumission d'une tâche au temps d'exécution global d'une application de workflow.

2.10 Conclusion

Au cours de ce chapitre, nous avons présenté de manière générale le problème d'ordonnancement de graphes de tâches sur les grilles de calcul. Nous nous sommes intéressés au problème particulier de l'ordonnancement à la volée de graphes de tâches dans un environnement hétérogène. Après avoir établi un tour d'horizon sur les gestionnaires d'exécution de workflow dans les grilles de calcul, nous avons proposé des heuristiques d'ordonnancement tolérantes aux fautes. Les algorithmes d'ordonnancement de workflow qui se basent sur la réplication des tâches proposées dans ce chapitre ont comme problème la consommation de ressources. A cet effet, nous proposons dans le chapitre suivant une stratégie d'ordonnancement de workflow tolérante aux fautes avec moins de consommation de ressources, la stratégie est basée sur la réplication passive et dynamique.

Chapitre 3

Modèle et stratégie proposés

Sommaire

3.1 Introduction

Dans les deux premiers chapitres, nous avons présenté l'essentiel des notions et des techniques auxquelles nous aurons recours dans cette thèse. Dans ce chapitre, nous commencerons par une présentation du modèle hiérarchique que nous proposons pour représenter une grille de calcul. Ensuite, nous décrirons la stratégie d'ordonnancement tolérante aux fautes. Cette stratégie combine l'ordonnancement et la réplication afin de minimiser le temps d'exécution global (makespan), la fiabilité, et la consommation de ressources, tout en respectant les contraintes de précédences qui existent entre les tâches.
Le Tableau 3.1 présente les différents symboles utilisés dans ce chapitre.

TABLE 3.1 – Liste des Symboles Usuels

L_i	Niveau i
W_{ij}	Coût d'exécution de la tâche t_i assignée au VN j
σ_{c_J}	Écart-type du coût d'exécution des tâches
W_{C_J}	Coût d'exécution moyen da la tâche t_i assignée au cluster C_J : $W_{C_J} = \sum W_{ij}/nbr_{VN}$
$\overline{c}_{ij}$	Coût de communication moyen
C_J	Cluster J
$t_{i,j}$	Réplique j de la tâche i
$succ(t_i)$	Ensemble des successeurs de la tâche t_i
$pred(t_i)$	Ensemble des prédécesseurs de la tâche t_i
$avail[VN_j]$	Temps où le VN_j termine l'exécution de la tâche t_i et il est prêt à exécuter une autre tâche.
$avail[C_J]$	Temps où le cluster C_J termine l'exécution de la tâche t_i et il est prêt à exécuter une autre tâche.

3.2 Modèle de la grille

L'approche d'ordonnancement de workflow tolérante aux fautes, que nous proposons est basée sur la représentation d'une grille sous la forme d'un modèle hiérarchique.

3.2.1 Modèle de base

Le modèle de base représente une vue logique de la plus petite grille possible, à savoir un seul cluster. Dans ce modèle, un cluster représente un ensemble de ressources informatiques homogènes reliées par un réseau local et localisées géographiquement dans une même organisation (Campus universitaire, Entreprise ou Personne individuelle).

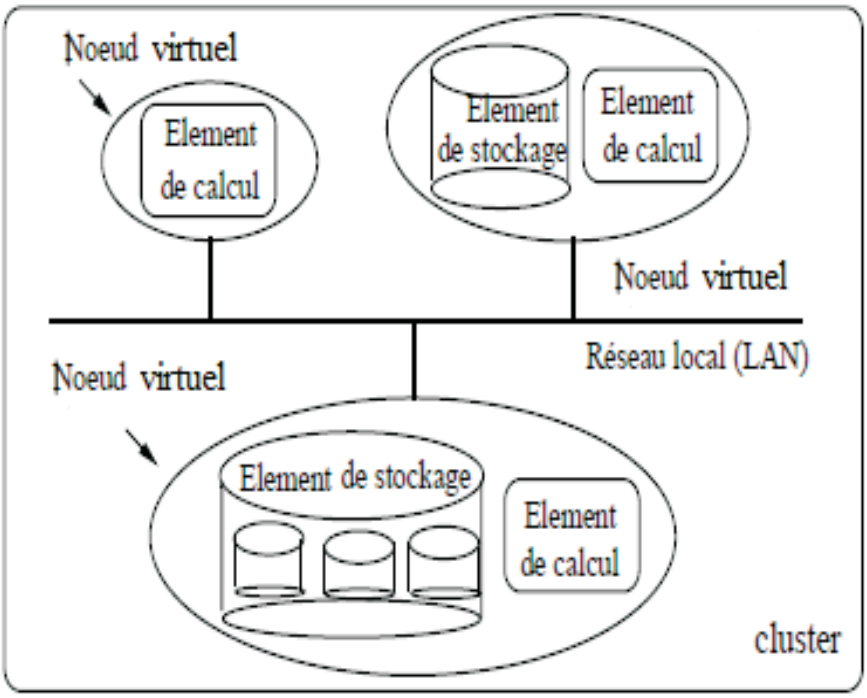

FIGURE 3.1 – Composantes d'un cluster

La « Figure 3.1 » indique les composantes d'un cluster.
Le modèle général comme indiqué dans la « Figure 3.2 », permet de représenter une grille en agrégeant G modéles de base. G est le nombre de cluster que comporte la grille. Chaque cluster a un représentant nommé FLCM [10], le FLCM :

- détecte les nœuds virtuels nécessaires ;
- assigne les tâches à ces nœuds ;
- détecte et tolère les fautes au niveau des nœuds virtuels.

La « Figure 3.3 » indique un exemple sur un modèle comportant 5 clusters. La racine de

10. Fault Local Cluster Manager

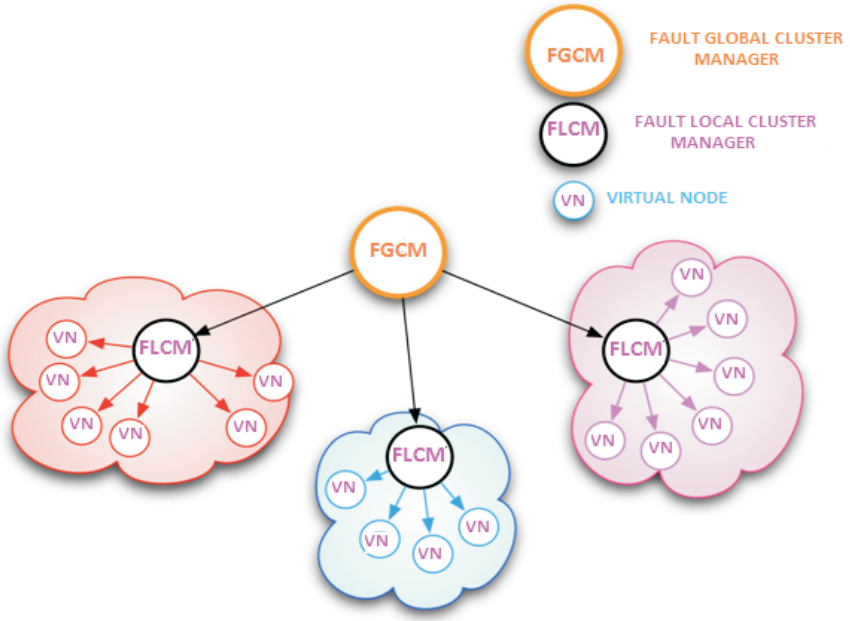

FIGURE 3.2 – Modèle proposé de la grille

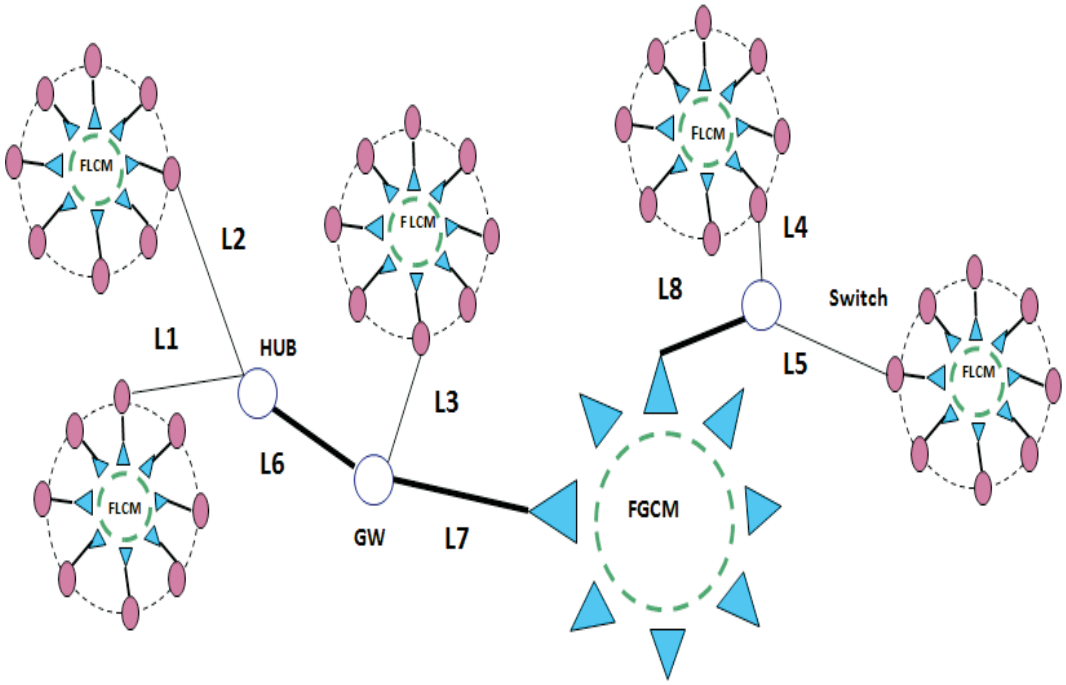

FIGURE 3.3 – Exemple d'un modèle de grille

la structure arborescente est le FGCM [11], ce dernier :

- détecte les clusters nécessaires ;
- assigne les tâches à ces clusters ;
- détecte et tolère les fautes au niveau des clusters.

Les feuilles sont les clusters, tandis que les branches de l'arbre représentent les liens virtuels. Les VLs [12] relient les feuilles et la racine. Certains (VLs) sont utilisés en commun par plusieurs clusters. Dans cette thèse, nous considérons un VN [13] comme une machine qui peut avoir un ou plusieurs PEs [14]. L'architecture du cluster peut être approchée par une structure avec une topologie en étoile.

Dans le modèle proposé nous considérons les informations suivantes :

- Pour les liens virtueles (VLs) reliant les clusters, le modèle considère la bande passante ;
- Pour les VLs reliant les VNs au FLCM, la bande passante est négligeable ;
- Le temps pour exécuter des tâches par les VNs est impliqué.

3.2.2 Caractéristiques du modèle

Le modèle d'ordonnancement tolérant aux fautes proposé dans cette thèse, est une représentation hiérarchique qui présente un aspect coopératif dans la mesure où il se base sur l'échange des informations entre entités qui présentent des intérêts communs tels que l'ordonnancement et la tolérance aux fautes globale. La coopération consiste à échanger des informations entre les gestionnaires des clusters locaux et le gestionnaire global du cluster. En outre, le modèle est extensible étant donné qu'il est capable d'intégrer de nouveaux clusters sans être obligé de reformuler ou de redimensionner ses constituants [166]. Cette facilité de gestion favorise l'utilisation de ce modèle dans les systèmes à large échelle.

- La structure du modèle facilite les flux d'informations à travers les nœuds de l'arbre.

En termes de flux d'informations, il y'a lieu d' en distinguer deux types :

1. ***Flux montant*** : ce flux concerne la circulation des informations sur les nœuds et l'état des tâches du niveau 2 (bas) vers le niveau 1(haut) (gestionnaire global de fautes des clusters). Grâce â ce flux, le gestionnaire global de fautes des clusters pourra avoir

11. Fault Global Cluster Manager
12. Virtual links
13. Virtual Node
14. Processing Elements

une vue globale sur le déroulement de la stratégie d'ordonnancement et la panne des ressources ;

2. ***Flux descendant*** : Ce flux permet de véhiculer les décisions d'ordonnancement prises par le gestionnaire global de fautes des cluster au niveaux 1 vers les gestionnaires locaux de fautes des clusters du niveau 2.

– La dynamicité puisque les connexions/déconnexion des utilisateurs sont de simples opérations d'ajout/suppression des feuilles de l'arbre [166] ;

– L'hétérogénéité car nous n'avons imposé aucune contrainte aux nœuds de l'arbre ;

– Le passage à l'échelle (ajout et retrait d'un élément de calcul, d'un cluster) [166].

3.3 Modèle de workflow

Une application workflow est représentée par un graphe acyclique orienté, $G = (V, E)$ où V est l'ensemble des tâches et E est l'ensemble des arcs entre les tâches. Chaque $arc(i, j)$ appartenant à E, représente la contrainte de précédence, telle que la tâche t_i doit terminer son exécution avant que la tâche t_j ne commence.
$data_{i,k}$ est la quantité de données requises à transmettre à partir de la tâche t_i à la tâche t_k. Dans un graphe de tâche, une tâche sans aucun parent est appelée une tâche d'entrée et une tâche sans fille est appelée une tâche de sortie, la « Figure 3.4 » présente un exemple de DAG la tâche d'entrée est la tâche numéro 1, la tâche de sortie est la tâche numéro 10.

Avant l'ordonnancement des tâches, chaque tâche a son coût d'exécution w_{ij}. Le coût de communication de l'arc (i,j) comme indiqué dans la « Figure 3.4 », est le temps de transfert des données à partir de la tâche t_i (assignée au cluster C_m) vers la tâche t_k (assignée au cluster C_n), est définie par :

$$c_{i,k} = data_{i,k}/B_{m,n} \tag{3.1}$$

$B_{m,n}$ est la bande passante entre le cluster C_m et le cluster C_n, $data_{i,k}$ étant la quantité de données requises à transmettre à partir de la tâche t_i à la tâche t_k.

Lorsque les deux tâches t_i et t_k sont programmées pour être assignées au même cluster, $c_{i,k}$ devient " nul ", parce que le coût de communication entre les nœuds est négligeable lorsqu'il est comparé au coût de communication entre les clusters.

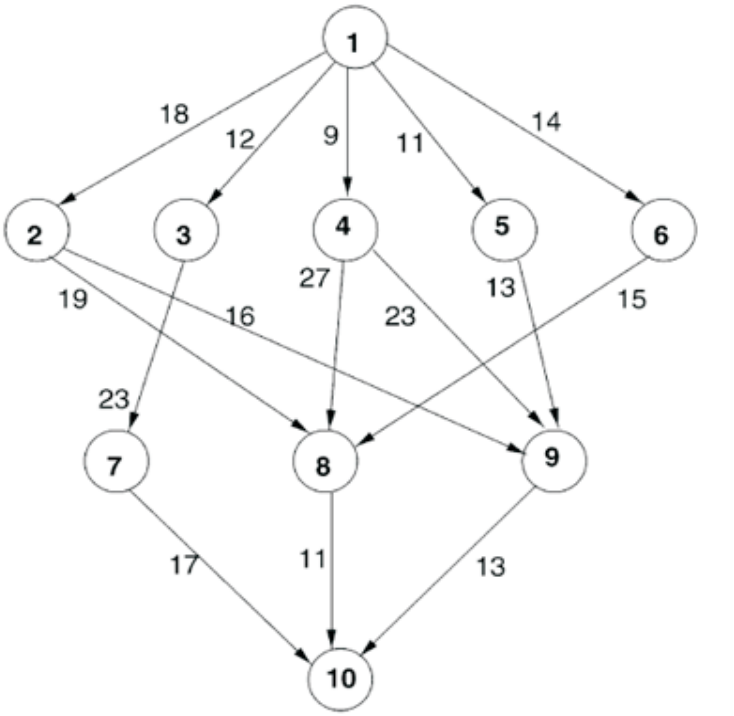

FIGURE 3.4 – Un exemple de DAG

3.4 Stratégie d'ordonnancement de tâches

On vise à réduire le temps d'exécution globale du workflow et à respecter les dépendances des tâches, la stratégie d'ordonnancement de tâches suit les phases suivantes :

3.4.1 Tri de tâches selon leur priorité

On calcule la priorité d'une tâche en parcourant le graphe vers le haut à partir de la tâche de sortie vers la tâche d'entrée, c'est la priorité des tâches. Un exemple indiquant les priorités des tâches est montré dans le Tableau 3.2.
La priorité de chaque tâche est calculée par l'équation suivante :

$$Prio(t_i) = max(\sigma_{c_J}) + max_{t_j \varepsilon succ(t_i)}(L_i * \overline{c}_{ij} + Prio(t_j)) \tag{3.2}$$

$$\sigma_c = \sqrt{\sum (W_{ij} - W_{C_J})^2 / nbr_{VN}} \tag{3.3}$$

t_i	**Prio**(t_i)
t_1	208
t_2	88
t_3	90
t_4	90
t_5	79
t_6	73
t_7	52
t_8	45
t_9	54.33
t_{10}	24

TABLE 3.2 – Exemple de priorités des tâches

3.4.2 Clustering des tâches

Les tâches du DAG sont triées par ordre décroissant de leur priorité. Avec cet ordre, elles sont divisées en différents clusters comme suit :
La première tâche est ajoutée au cluster numéro 0. Si les tâches successives en ordre décroissant de leur priorité sont dépendantes d'une tâche déjà affectée au cluster, elles sont alors sont placées dans le même cluster. Inversement, dans le cas d'indépendance, un nouveau cluster sera créé et le numéro du nouveau cluster sera le numéro du cluster actuel augmenté de un nombre. Le résultat final est un ensemble de clusters ordonnés, comme le montre la « Figure 3.5 ». Les tâches de chaque cluster sont attribuées par le FGCM au cluster qui minimise leur temps de fin d'exécution, et elles sont attribuées par le FLCM au VN appartenant à ce cluster, qui minimise leurs temps de fin d'exécution. Par exemple, si le tri donne l'ordre suivant des tâches : t_1, t_2, t_3, t_7, t_4, t_5, t_9, t_6, t_8, t_{10}, donc le clustering consiste à créer des clusters de tâches : C1 (t_1, t_2), $C2$(t_5, t_9) et C3(t_3, t_7), et assigner t_1 et t_2 au cluster pour que le coût de communication entre les deux tâches devienne nul. Cette même opération telle quelle est se répète pour les tâches t_5 et t_9, t_3 et t_7. Nous définissons le EST (Earliest Start Time) de la tâche t_i assignée au VN j :

$$EST(t_i, VN_j) = max\{avail[VN_j], max_{t_m \varepsilon pred(t_i)}(AFT_{(t_m)} + c_{im})\} \tag{3.4}$$

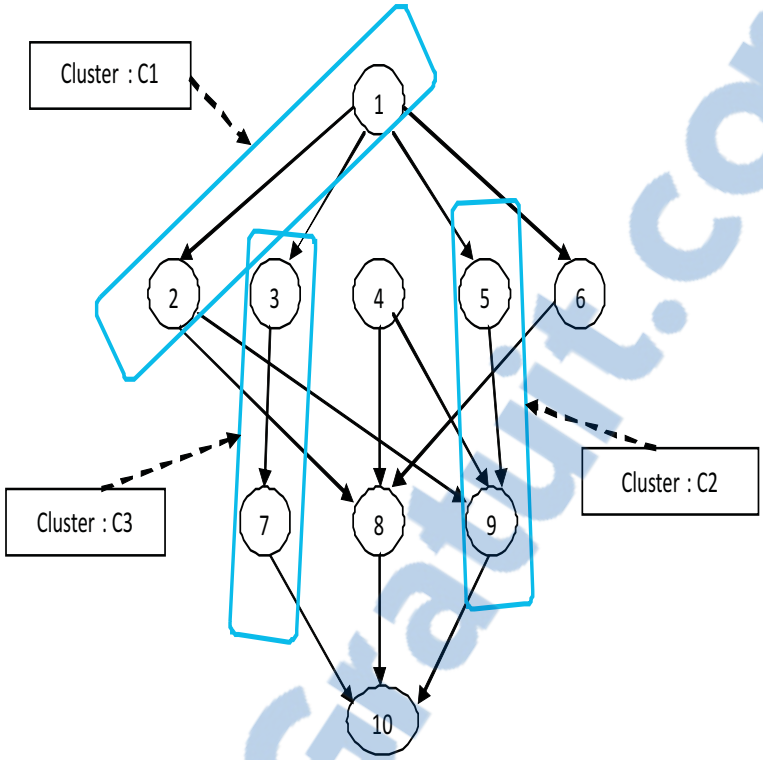

FIGURE 3.5 – Le clustering des tâches

Nous définissons le AFT(Actual Finish Time) de la tâche t_i assignée au VN j :

$$AFT(t_i, VN_j) = W_{ij} + EST(t_i, VN_j) \tag{3.5}$$

Nous définissons le EST de la tâche t_i assignée au cluster J :

$$EST(t_i, C_J) = max\{avail[C_J], max_{t_m \varepsilon pred(t_i)}(AFT_{(t_m)} + c_{im})\} \tag{3.6}$$

Nous définissons aussi le AFT de la tâche t_i assignée au cluster J :

$$AFT(t_i, C_J) = W_{C_J} + EST(t_i, C_J) \tag{3.7}$$

EST et AFT, représentent le temps estimé de début d'exécution et de fin d'exécution d'une tâche sur un clusteur.

$EST(t_i, C_j)$: est le temps estimé de début d'exécution de la tâche t_i sur le cluster C_j

$AFT(t_i, Cj)$: est le temps estimé de fin de d'exécution de la tâche t_i sur le cluster C_j.

Pour calculer le EST d'une tâche t_i, il faut que toutes les tâches qui précèdent t_i soient assignées.
D'où $succ(t_i)$ est l'ensemble des tâches successeurs de la tâche t_i, et $avail[j]$ est l'instant où le cluster C_j est disponible pour l'exécution de la tâche t_i.
Si t_k est asignée au cluster C_j, alors $avail[j]$ est le temps que le cluster C_j termine l'exécution de la tâche t_k et il est prêt à en exécuter une autre.

$$EST(n_{entry}, C_j) = 0 \tag{3.8}$$

Le temps d'arrivée d'une tâche t_m sur le cluster C_j, et le temps de fin d'exécution de la tâche t_m sur le cluster C_j, sont représentés par $EST(t_m)$ et $AFT(t_m)$ de la tâche t_m. Après que toutes les tâches dans un graphe ont été assignées, la durée prévue (c'est à dire, le temps d'exécution global de l'application workflow) sera le temps réel du début d'exécution de la première tâche t_{entry}, jusqu'à la fin d'exécution de la tâche de sortie t_{exit}. S'il y'a plusieurs tâches de sorties alors la durée totale d'exécution est calculée par :

$$makespan = maxAFT(t_{exit}) \tag{3.9}$$

Une tâche prête à être exécutée, est définie comme la tâche dont les prédécesseurs ont terminé l'exécution. Au début, la file d'attente de l'ordonnancement (SQ [15]) se compose de tâches d'entrée car elles n'ont pas de tâches parents [70]. Lorsqu'une tâche t_i est terminée, SQ est mis à jour en supprimant t_i de SQ et en insérant de nouvelles tâches.
Algorithme1 indique les instructions d'ordonnancement d'un DAG à des Clusters/VNs. d'abord, les tâches sont triées en fonction de leur priorité. Elles sont regroupées aux clusters en fonction de leurs dépendances dans la deuxième étape. Ensuite, à l'étape 3, les tâches sont assignées aux Clusters/VNs en fonction de leur EFT.

15. Schedule Queue

Algorithm 1: Algorithme d'ordonnancement

Data: DAG

Result: Makespan

```
Définir les coûts d'exécution des tâches, les coûts de communication
l'écart-type des coûts d'exécution de chaque tâche
Calculer la priorité de chaque tâche                    selon l'eq.(3.2)
Trier les tâches par ordre décroissant de leur priorité
while Il y a des tâches dans la liste d'ordonnancement do
    Ajoutez la première tâche à un cluster numéro 0
    if Les tâches successives dépendent d'une tâche du cluster i then
        Ajouter les tâches successives au cluster i
    else
        Créer un nouveau cluster, son numéro est le numéro du
        dernier cluster augmenté de un nombre
    end
end
while Il ya des tâches non ordonnancées do
    sélectionnez la première tâche d'un cluster
    assignez la tâche au cluster qui minimise son EFT        selon l'eq.(3.4)
    assignez la tâche au VN qui minimise son EFT             selon l'eq.(3.6)
    assignez la tâche successive appartenant au même groupe au même cluster
end
```

3.5 Système de tolérance aux fautes

Dans les grille de calcul le risque de défaillance est très important, car le nombre de ressources (physiques et logiques) est à la fois très important, mais également ces ressources sont dispersées sur une échelle très large et sont très diverses les unes par rapport aux autres. Différents types de fautes sont entrelacés dans l'environnement de grilles, telles que les fautes de blocage, les fautes de connection et les fautes de programme.

3.5.1 Types de fautes

Les types possibles de fautes qui peuvent se produire dans les grilles sont [123] :

– ***Fautes de blocage et fautes de temporisation*** : Généralement, la file d'attente des requêtes peut avoir une limitation sur le nombre maximal de demandes en attente dans la file d'attente. Lorsqu'une nouvelle requête arrive, et si la file d'attente est pleine, elle déclanche une défaillance de blocage. Si le temps d'attente des requêtes dans la file d'attente dépasse le temps dû, une faute de temporisation se produit ;

– ***Fautes de déconnection*** : survient lorsque les requêtes ne parviennent pas à se connecter aux ressources ;

– ***Faute du réseau*** : lorsque les travaux utilisent des ressources distantes, les canaux de communication peuvent être déconnectés, ce qui provoque une « défaillance du réseau » ;

– ***Fautes de ressources*** : la ressource partagée sur la grille peut être une ressource logicielle ou une ressource matérielle. Par conséquent, lors de l'utilisation de la ressource, son échec peut provoquer une défaillance du service.

Dans notre travail les types de fautes traitées sont :

1. **Fautes de ressources** : Le grand nombre de ressources présent dans une grille de calcul implique une éventualité non négligeable de fautes de ressources.
2. **Fautes de déconnexion** : Ces fautes correspondent à des déconnexions physiques (rupture d'un câble, équipement réseau hors service) ou logiques (panne du serveur DNS par exemple).

3.5.2 Détection de fautes

Dans cette thèse, la détection de fautes est réalisée de la manière suivante :

– ***Détection globale de fautes*** : les FLCMs envoient les résultats des tâches exécutées au FGCM, une fois qu'un VN ne parvient pas à exécuter une tâche, le FLCM envoie le numéro de la tâche défaillante au FGCM, le FGCM demande alors au FLCM contenant une copie secondaire, de l'activer pour exécution ;

– ***Détection locale de fautes*** : les nœuds virtuels envoient périodiquement des messages de vie [125] à leur FLCM, en annonçant qu'ils sont en vie. En l'absence de tels messages, les FLCMs détectent la défaillance produite au niveau de ce VN, puis ils envoient les numéros des tâches défaillantes au FGCM, ce mécanisme est illustré dans la « Figure3.6 ».

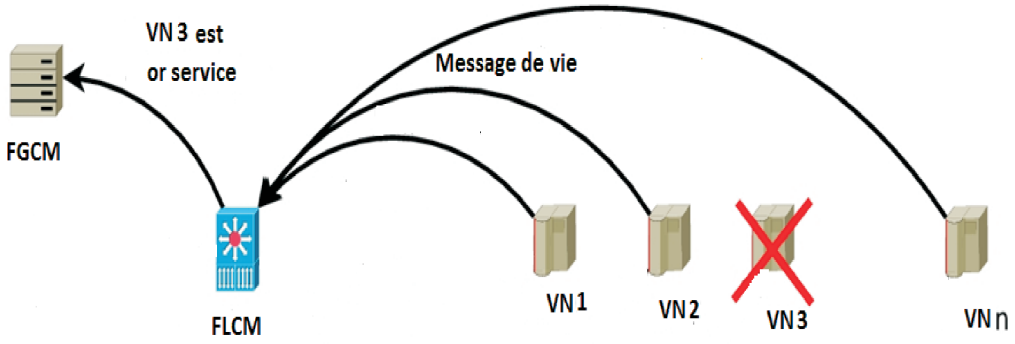

FIGURE 3.6 – Détection locale de fautes

Algorithme2 explique l'envoi périodique du message de vie en utilisant des instructions. Tout d'abord, les VNs envoient périodiquement des messages de vie à leur FLCM. Si un des VNs s'arrête d'envoyer le message, alors le FLCM détecte que ce nœud est tombé en panne. Par conséquent, il envoie les numéros des tâches défaillantes qui s'exécutent dans ce nœud au FGCM.

Algorithm 2: Algorithme de détection locale de fautes

Data: Message de vie

Result: Réponse

```
repeat
    Envoi périodique du message de vie par les VNs aux FLCMs
    if un nœud virtuel s'arrête d'envoyer le message de vie then
        le FLCM détecte que ce nœud est tombé en panne
        le FLCM envoie les numéros des tâches qui s'exécutent dans ce nœud
        au FGCM
    end
until Fin de simulation
```

Représentation structurelle de détection locale de fautes

La « Figure 3.7 » est un diagramme de séquence qui montre le scénario de détection locale de fautes. Tout d'abord, le FLCM envoie un message de vie au VNi, le VNi lui répond qu'il est en vie, la même opération se répète pour le VNj, à un moment donné le FLCM envoie un message de vie au VNj, ce dernier ne lui donne pas de réponse et le temps d'envoi du message de vie expire, à ce moment le FLCM détecte que le VNj est tombé en panne, ce processus se répète jusqu'à la fin de simulation.

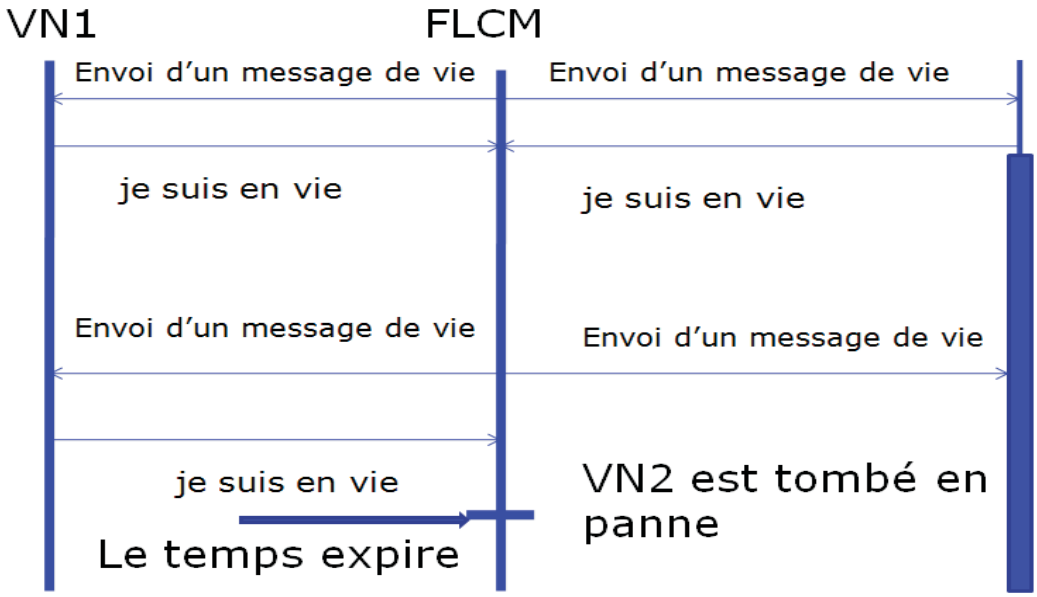

FIGURE 3.7 – Diagramme de séquence de détection locale de fautes

3.5.3 Recouvrement de fautes

Le recouvrement de fautes suit le modèle virtuel, la tolérance matérielle est obtenue en appliquant la méthode de réplication passive et dynamique des tâches, nommée PDRT [16]. La méthode couvre les phases suivantes :

Réplication spatiale d'une tâche

La réplication [124] employée a comme but la minimisation de la consommation des ressources. Les principales étapes de la réplication spatiale sont :

16. Passive and dynamic replication of tasks method

– Les tâches sont répliquées par niveau, la Figure« 3.8 » indique un DAG divisé en plusieurs niveaux ;

– Les tâches de chaque niveau sont répliquées et assignées aux VNs libres ;

– Les tâches de chaque niveau sont répliquées et assignées selon l'équation qui suit :

$$EST(t_{k+1}, j) - MAX(EFT(t_k, j) < W_{ij} \tag{3.10}$$

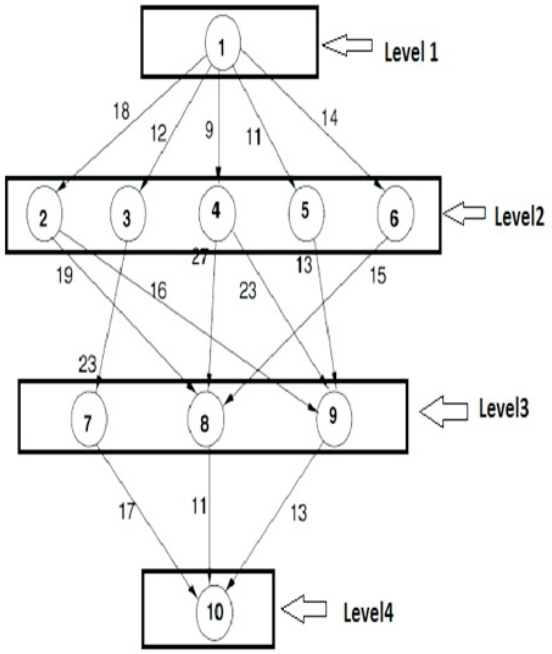

FIGURE 3.8 – DAG divisé à des niveaux

Réplication temporaire d'une tâche

Les tâches filles sont répliquées et soumises à plusieurs VNs, dès que leurs parents commencent l'exécution. Par exemple, la tâche 8 ne sera répliquée que lorsque la tâche 4 commencera son exécution, comme le montre la « Figure 3.9 ». Lorsque la tâche 4 est terminée, la file d'attente de l'ordonnancement SQ est mise à jour en supprimant les copies.

Activation temporaire d'une tâche

Une fois qu'un VN ne parvient pas à exécuter une tâche, et si une tâche t_i appartient à ce VN, sa copie secondaire est activée dans le VN libre. Ce mécanisme comme le montre la « Figure 3.9 », pourrait réduire :

– Le temps estimé de fin d'exécution des tâches ;

– Le temps d'attente et par la suite le makespan ;

– Les coûts de communication, si la copie secondaire est activée dans le cluster qui contient son parent.

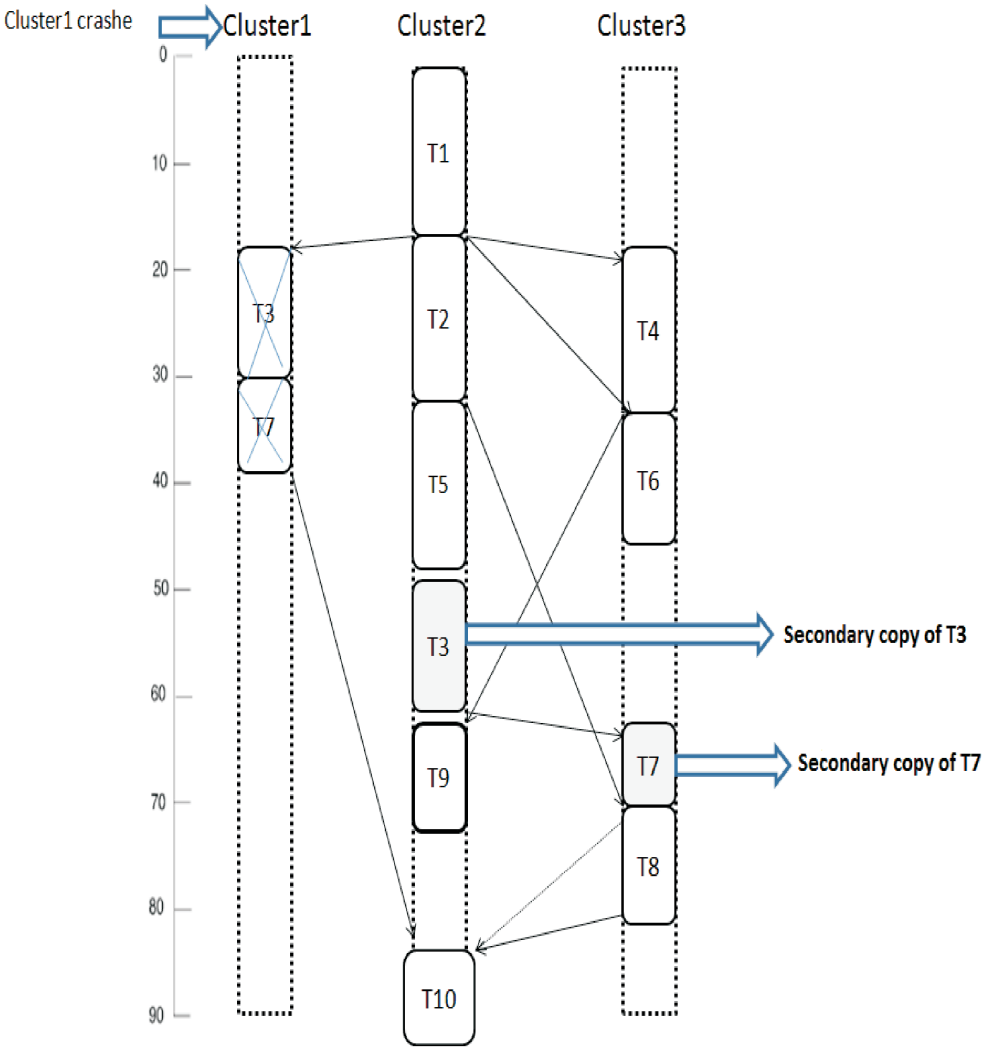

FIGURE 3.9 – Mécanisme de tolérance aux fautes

Algorithme3 explique la réplication spatiale en utilisant des instructions. Premièrement, les tâches d'entrées sont répliquées et attribuées aux VNs libres. Les tâches filles de chaque tâche mère sont répliquées dès que leurs parents commencent l'exécution dans la deuxième étape. Cette opération est répétée pour toutes les tâches du DAG.

Algorithm 3: Algorithme de réplication

Data: DAG

Result: Tâches répliquées

```
1 while il y'a des tâches non ordonnancées dans chaque cluster do
2 |   répliquer les tâches d'entrées et les assigner au VNs libres
3 |   if la tâche mère commence son exécution then
4 |   |   répliquer et assigner les tâches filles aux VNs qui sont libres
5 |   end
6 end
```

3.6 Conclusion

Dans ce chapitre, nous avons présenté un modèle pour la tolérance aux fautes. Ce modèle est totalement indépendant de toute topologie physique d'une grille de calcul. Dans le but de minimiser le temps d'exécution global des tâches soumises à la grille, tolérer les fautes au niveau de la grille et minimiser la consommation des ressources, nous avons proposé une stratégie dynamique d'ordonnancement de tâches tolérante aux fautes. La tolérance aux fautes est réalisée par la réplication passive et dynamique des tâches. Le chapitre suivant exposera les résultats expérimentaux relatifs à la stratégie proposée.

Chapitre 4

Etude Expérimentale

Sommaire

4.1 Introduction

Afin de valider et d'évaluer notre stratégie d'ordonnancement tolérante aux fautes, nous avons effectué une série d'expérimentations dont les résultats et les interprétations seront évoqués dans ce chapitre. Nous commencerons d'abord par fixer l'environnement dans lequel nous avons réalisé nos expérimentations, et définir les métriques et les notations que nous avons utilisées, ensuite nous discuterons et analyserons les résultats obtenus.

4.2 Implémentation

L'étude du comportement d'un système peut se faire selon plusieurs approches. La première est la méthode analytique qui se base sur la modélisation mathématique et l'utilisation des démonstrations pour prouver qu'une configuration est meilleure qu'une autre. Mais avec des systèmes complexes, les modélisations et les démonstrations mathématiques deviennent très difficiles voire impossibles.
Par conséquent, c'est la deuxième méthode, qui a été l'expérimentation, est utilisée. Cette approche consiste à expérimenter directement les différentes solutions afin d'en sélectinner la meilleure. L'expérimentation peut, à son tour, devenir contraignante si nous voulons expérimenter des systèmes qui nécessitent des moyens importants dont le coût peut s'avérer très lourd ou des systèmes qui feraient intervenir plusieurs paramètres qui font que les conditions expérimentales ne soient pas les mêmes. Toutefois une troisième méthode, la simulation, représenterait un bon outil. En effet, la simulation ne nécessite pas de grands moyens, et permet d'itérer autant que nécessaire les expériences tout avec que faire se peut les expériences tout avec les mêmes conditions expérimentales [12, 85]. Il existe de nombreux simulateurs de grille parmi lesquels nous pouvons citer :

– Bricks [147] (Takefusa et al., 2001), pour la simulation de systèmes de type client-serveur ;
– OptorSim [13] (Bell et al., 2003), conçu pour l'étude d'algorithmes d'ordonnancement traitant spécifiquement de la réplication ou de la migration de données,
– GridSim [22] (Buyya et al., 2002) et Simgrid [28] (Legrand et al., 2003), des outils de modélisation de ressources de calcul et réseau, et de simulation d'algorithmes d'ordonnan-

cement de grille.

Pour étudier le comportement de notre stratégie, nous l'avons intégrée dans le simulateur GridSim [22] :

Gridsim est une plate-forme de développement de simulateurs, développée à l'université de Monash en Australie depuis 2001. Gridsim est une bibliothèque de fonctions implémentée en java, l'utilisateur utilise ces fonctions pour développer son propre simulateur, la « Figure 4.1 » indique la structure du système GridSim.

Gridsim permet de définir tous les éléments d'un environnement de calcul : les clients, les éléments de calcul, les liens et les tâches. Il gère aussi les messages qui circulent dans cet environnement et il permet d'accéder aux informations essentielles de la simulation, à savoir, l'état des ressources et la durée d'exécution d'une tâche.

L'architecture de GridSim se résume à :

Modèle de simulation

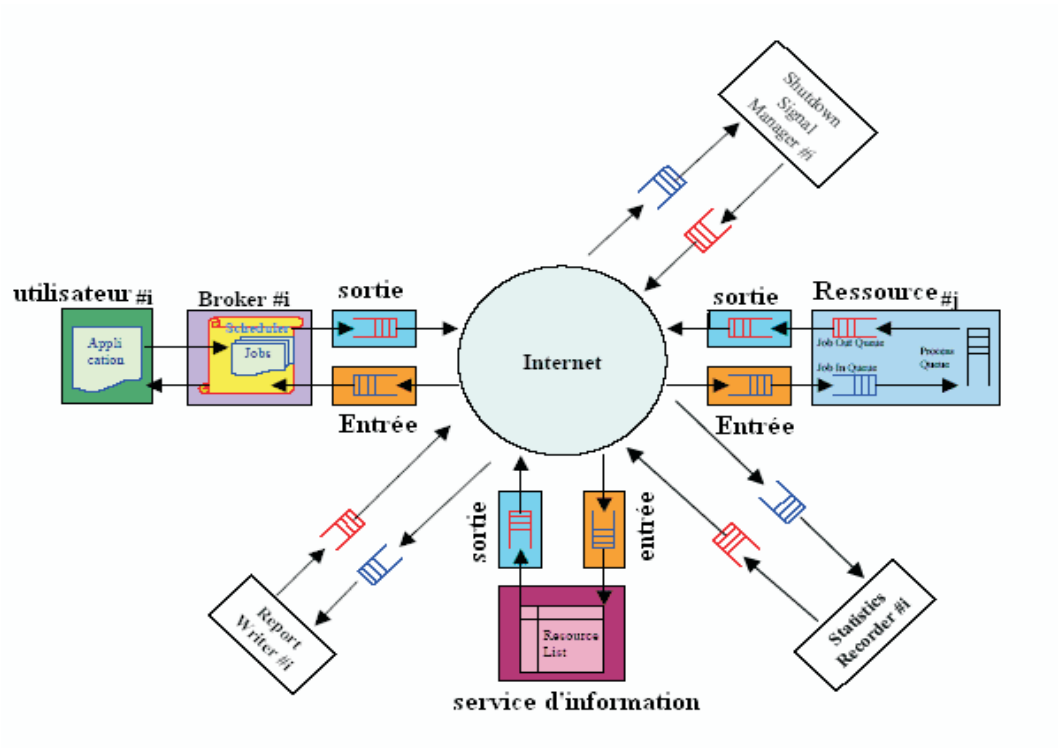

FIGURE 4.1 – Structure du système GridSim [22].

– ***Utilisateur*** : Chaque utilisateur diffère du reste des utilisateurs par :

– Types des travaux créés, par exemple le temps d'exécution du travail, nombre de réplication, etc. ;

– ***Stratégie d'optimisation*** : par exemple minimisation du coût de transfert, du temps d'exécution, ou des deux ;

– Fuseau horaire.

– ***Broker*** : Chaque utilisateur est relié à un broker. Chaque travail d'un utilisateur est d'abord soumis à ce broker. Avant d'ordonnancer les tâches, le broker obtient dynamiquement une liste des ressources disponibles, cette liste est fournie par le service d'information de la grille. Chaque broker essaye d'optimiser la politique de son utilisateur et donc, on s'attend à ce que les brokers fassent concurrence en accédant aux ressources.

– ***Resource*** : Chaque ressource peut différer du reste des ressources par :

– Coût de traitement ;

– Nombre de processeurs ;

– Vitesse de traitement ;

– Fuseau horaire.

La vitesse des ressources et le temps d'exécution du travail sont exprimés en MIPS (Million d'Instructions par Seconde).

Les données associées à chaque tâche sont :

– Taille exprimée en MI (Million d'Instructions) ;

– Temps d'exécution cumulé ;

– Temps d'attente cumulé ;

– Date d'arrivée ;

– Date de début d'exécution.

– ***Service d'information de grille*** : Fournit des services d'enregistrement des ressources et maintient une liste des ressources disponibles dans la grille. Ce service est utilisé par les brokers pour obtenir la configuration et le statut des ressources.

– ***Entrée et sortie*** : Le flux d'informations entre les entités de GridSim circule via les entités d'entrée et de sortie, comme illustré dans la « Figure4.1 ».

Modèle de réseau

Depuis la version 3.1 de GridSim le toolkit inclut des dispositifs de réseau plus sophistiqués. Entrés/Sorties sont prolongées avec de nouvelles classes réseau pour modéliser le comportement du réseau. Les nouveaux dispositifs permettent la définition des topologies de réseau

avec des routeurs, des liens, des paquets, qualité de service (QoS), générateurs du trafic.

– **Lien** : est un lien réseau avec une bande passante et une unité de transmission maximales ;

– Routeur : représente un routeur réseau avec plusieurs algorithmes de routage tel que RIP, OSPF et des tables de routage statiques ou dynamiques ;

– **Packet** : GridSim contient deux types de paquet, à savoir NetPacket et InfoPacket. NetPacket est employé pour transporter des données par le réseau, alors qu'InfoPacket est utilisé pour rassembler l'information du réseau telle que la bande passante ;

– **Générateur de trafic** : produit du trafic selon plusieurs distributions telles que Bernoulli, exponentielle et binomiale.

Comme le simulateur GridSim ne permet pas de simuler, directement, notre stratégie, nous avons procédé à des modifications pour l'adapter à nos besoins. Ces modifications ont le plus souvent porté sur l'ajout de nouvelles classes JAVA et sur la surcharge de certaines méthodes. Les classes intégrées sont montrées dans la « Figure 4.2 », il s'agit de :

– La classe FGCM : pour la représentation structurelle du FGCM, elle contient la méthode constructrice, responsable de l'assignement de tâches aux clusturs et de la détection globale de fautes ;

– La classe FLCM : pour la représentation structurelle du FLCM, elle contient la méthode constructrice, responsable de l'assignement de tâches aux VNs et de la détection locale de fautes ;

– La classe FTSA : pour la représentation structurelle de la méthode FTSA ;

– La classe HEFT : pour la représentation structurelle de la méthode HEFT ;

– La classe DFWS : pour la représentation structurelle de la méthode DFWS ;

– La classe Random : pour la représentation structurelle de la méthode aléatoire ;

– La classe RoundRobin : pour la représentation structurelle de la méthode RoundRobin ;

– La classe Workflow generator : qui permet la génération des différents workflows et l'enregistrement du workflow dans un fichier texte ;

– Workflow parcer : qui permet de lire un workflow à partir d'un fichier texte.

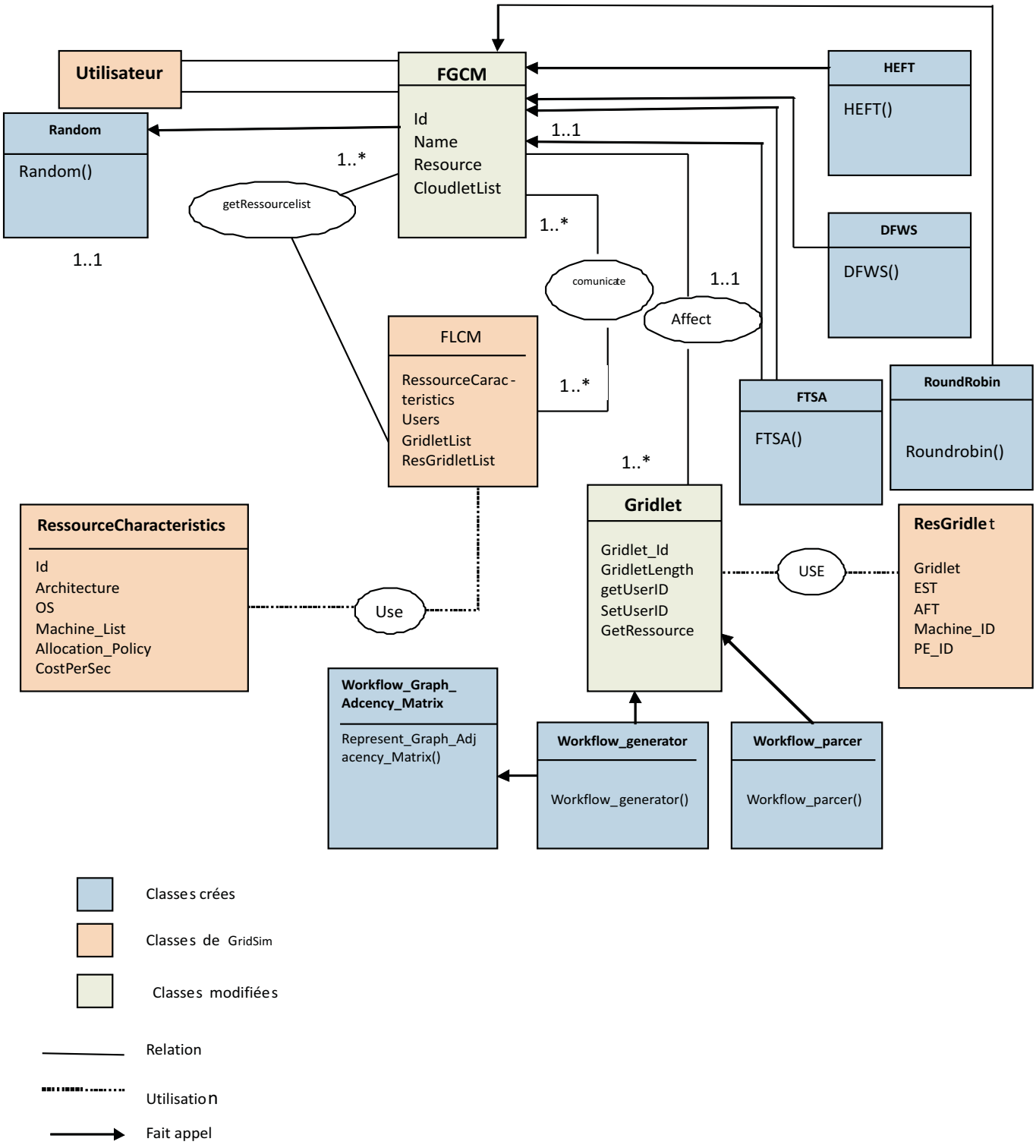

FIGURE 4.2 – Diagramme de classe des packages de GridSim

4.3 Évaluation de fiabilité

La tâche i peut être complétée avec succès par le VN_j, à condition que ce VN et le chemin de communication n'échouent pas avant la fin d'exécution de la tâche. La probabilité conditionnelle de réussite de la tâche est calculée selon la formule suivante :

$$p(t_{ij}) = e^{-((\lambda_{VN}+\mu_{VN})*((W_{ij}+c_{ij}),VN_j)))} \tag{4.1}$$

Le modèle de la grille illustré dans la « Figure 3.3 », peut être représenté par un graphe non dirigé g = (V, E) dans lequel V, | V | = n, n est l'ensemble de clusters. Le graphe non dirigé, tel qu'illustré dans la « Figure 4.3 » nécessite une technique d'évaluation de fiabilité. La technique est basée sur la théorie des graphes, et elle est décrite comme suit :

– L'ensemble des nœuds et des liens impliqués dans l'exécution du graphe donné, forme un groupe de toutes les composantes connexes de la structure virtuelle ci-dessous ;

– Chaque composante connexe représente une combinaison possible minimale d'éléments disponibles (VNs et VLs), qui garantissent le succès d'exécution du DAG ;

– L'échec de chaque composante connexe entraîne l'échec du graphe en entier ;

– Nous déterminons le coût d'exécution et le coût de communication pour tous les VNs et VLs ;

– Nous trions les composantes connexes par ordre croissant de leur makespan, et nous les divisons en groupes contenant des composantes connexes avec un makespan identique ;

– Nous supposons qu'il y a K groupes, désignés par G_1 ; G_2 ; . . . G_k, et tout groupe G_i contient des composantes connexes avec un makespan identique.

Par conséquent, la fiabilité de l'ordonnancement est la suivante :

$$SchedRelia = PR((\overline{Ecc_i}), (\overline{Ecc_{i-1}}), ..., (\overline{Ecc_3}), (Ecc_1)) \tag{4.2}$$

Avec Ecc_i est l'événement où une composante connexe du groupe G_i est disponible et $\overline{Ecc_i}$ est l'événement où une composante connexe du groupe G_i n'est pas disponible.
Supposons que les composantes connexes d'un groupe G_i sont arbitrairement triées et V_{ij} représentant un événement lorsque la jième composante connexe du groupe est disponible. L'événement Ecc_i peut être donc exprimé par :

$$Ecc_i = \cup V_{ij}$$

Et l'équation 4.2 prendrait la forme :

$$SchedRelia = PR(\cup V_{ij}, (\overline{Ecc_{i-1}}), (\overline{Ecc_3}), (\overline{Ecc_1})) \tag{4.3}$$

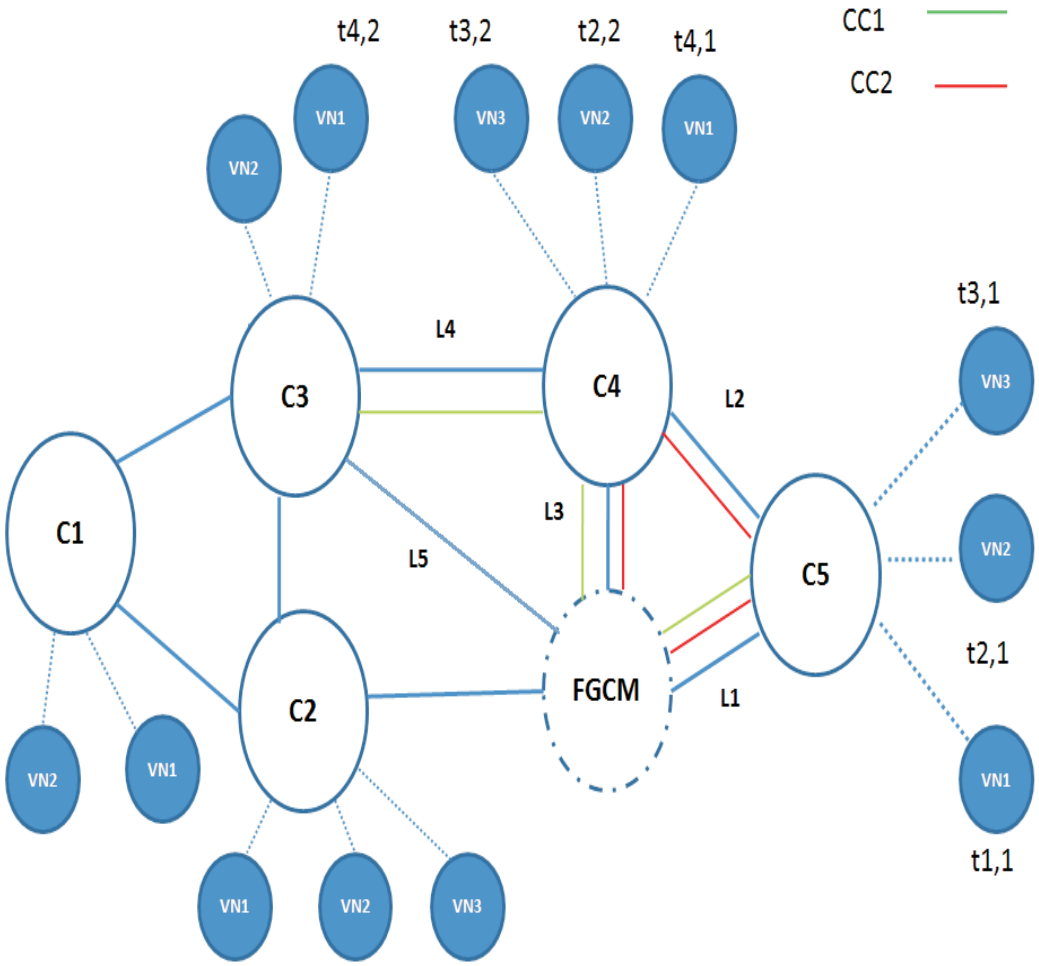

FIGURE 4.3 – Exemple de composantes connexes

D'aprés la « Figure 4.3 », nous pouvons construire les composantes connexes suivantes :

CC1 :$\{t_{11}$,t_{21},t_{31},t_{41},L1,L2,L3$\} \Longrightarrow\ CC1 : \{C4, C5, L1, L2, L3\}$

$CC2 : \{t_{11}$,t_{22},t_{31},t_{41},L1,L2,L3$\} \Longrightarrow\ CC2 : \{C4, C5, L1, L2, L3\}$

$CC3 : \{t_{11}$,t_{21},t_{32},t_{41},L1,L2,L3$\} \Longrightarrow\ CC3 : \{C4, C5, L1, L2, L3\}$

$CC4 : \{t_{11}$,t_{21},t_{31},t_{42},L1,L2,L3,L4,L5$\} \Longrightarrow\ CC4 = \{C4, C5, C3, L1, L2, L3, L4, L5\}$

$CC5 : \{t_{11}$,t_{22},t_{32},t_{41},,L3,L4,L5$\} \Longrightarrow\ CC5 : \{C4, C5, L1, L2, L3, L4, L5\}$

$CC6 : \{t_{11}, t_{22}, t_{32}, t_{42}, L1, L3, L4, L5\} \Longrightarrow CC6 : \{C4, C5, C3, L1, L3, L4, L5\}$

Supposons que les makespans des composantes connexes se présentent comme suit :

$CC1_{Makespan}$=100 ;

$CC2_{Makespan}$=110 ;

$CC3_{Makespan}$=113.56 ;

$CC4_{Makespan}$=113.56 ;

$CC5_{Makespan}$=115.36 ;

Par conséquent, les CCs sont classées en quatre groupes en fonction de leur makespan :

- G1=$\{CC1\}$ avec un makespan=100 ;
- G2=$\{CC2\}$ avec un makespan=110 ;
- G3=$\{CC3, CC4\}$ avec un makespan=113.56 ;
- G4=$\{CC5\}$ avec un makespan=115.36.

Suivant l'équation 4.2, on aura quatre groupes :

G1 :Q1=$Pr(E1)$;

$G2 : Q2$ =Pr(E2,$\overline{E1}$) $\Longrightarrow Pr(V21) * Pr(\overline{E1}/V21)$;

G3 :Q3=$Pr(E3, \overline{E1}, \overline{E2}) \Longrightarrow Pr(V31) * Pr(\overline{E1}, \overline{E2}/V31) + Pr(V32) * Pr(\overline{V31}, \overline{E1}, \overline{E2}/V32)$;

G4 :Q4=$Pr(E4, \overline{E1}, \overline{E2}, \overline{E3}) \Longrightarrow Pr(V41) * Pr(\overline{E1}, \overline{E2}, \overline{E3}/Pr(V41))$

Par conséquent, la fiabilité de l'algorithme d'ordonnancement devient :

Pr=Q1+Q2+Q3+Q4.

Algorithme1 explique le calcul de fiabilité en utilisant des instructions. Tout d'abord, nous formons les composantes connexes, puis groupons les composantes connexes à des groupes contenant un makespan identique, et enfin calculons la fiabilité de chaque groupe. La fiabilité de l'ordonnancement est la somme des fiabilités calculées des groupes.

Algorithm 4: Algorithme d'évaluation de fiabilité

Data: Graphe non dirigé

Result: Fiabilité

```
Définir les nœuds et les liens impliqués dans l'exécution du graphe donné
Déterminer le coût d'exécution et le coût de communication pour tous les
VNs et VLs
Trier les composantes connexes par ordre croissant de leur makespan
while Il y a une composante non affectée do
    Ajoutez la première composante à un groupe numéro 1
    if La composante connexe suivante a le même makespan then
        Ajouter la composante au groupe i
    else
        Créer un nouveau groupe, son numéro est le numéro du
        dernier groupe augmenté de un nombre
    end
end
while Il y'a un groupe non traité do
    Calculer la fiabilité du groupe          selon l'éq. (4.2)
    Calculer la somme des fiabilités de chaque groupe
end
```

4.4 Évaluation de performances

4.4.1 Paramètres de simulation

– *Environnement matériel et logiciel* : Toutes ces expériences ont été effectuées sur un PC core i3, machine Intel , avec 4Go de mémoire, fonctionnant sur Windows 7 ;

– Environnement de développement : Nous avons utilisé le langage Java ;

– *Nombre d'expérimentations* : Pour chaque série d'expérimentations, et afin d'obtenir des résultats viables, nous avons répété chaque expérience plusieurs fois, puis nous avons pris la moyenne des résultats obtenus ;

– Les défaillances de différents éléments (VNs et VLs) sont indépendantes ;

– Nous supposons que le FGCM n'échoue pas lors de l'exécution du service de la grille. Cette hypothèse est raisonnable car le FGCM (basé sur plusieurs serveurs) a généralement une très grande fiabilité ;

– Les échecs des VNs et VLs suivent un processus de Poisson [32] ;

– Nous avons supposé que chaque cluster contient 4 PE. Pour chaque PE, nous avons généré aléatoirement les spécificités techniques des composants (vitesse) et des tâches selon les hypothèses suivantes :

– Génération de vitesses variant entre 9 et 100 MIPS ;

– Génération de la taille des tâches de manière aléatoire entre 300 et 300000 MI (millions d'instructions) ;

– La bande passante du réseau local, reliant les PEs, est fixée à 100 MIPS ;

– Nous avons constaté que les meilleurs résultats sont obtenus pour les taux d'échecs des VNs et des VLs, $\lambda = 0.0095$, $\mu = 0.0045$;

4.4.2 Stratégies de référence

Nous avons comparé la stratégie proposée avec quatre autres stratégies comme suit :

– *HEFT* (Heterogeneous Earliest-Finish-Time) : Est un algorithme classique d'ordonnancement de liste statique. Il est largement utilisé [167] et Zhang et al. [173], ont montré qu'il fonctionne bien dans un environnement de grille multi-clusters. Par conséquent, c'est un bon moyen pour tester la performance de DFWS ;

– FTSA (Fault Tolerant Scheduling Algorithm) : Est une version étendue de l'algorithme HEFT. Il attribue ($\varepsilon + 1$) copies de chaque tâche aux différentes ressources pour tolérer un nombre arbitraire de défaillances de ressources [14] ;

– Random Strategy : Les tâches sont attribuées aléatoirement ;

– Round Robin Strategy : Les tâches sont affectées à l'aide d'une stratégie round robin.

4.4.3 Résultats de performance

Les résultats sont présentés en cinq parties. La première partie présente les résultats de makespan. La seconde donne les résultats de la consommation des ressources. La troisième partie présente les résultats de temps d'attente moyen, la quatrième partie présente les résul-

tats de speedup. Et la dernière partie présente les résultats de fiabilité. Une analyse détaillée sera donnée pour chaque ensemble d'expériences.

Les résultats de Makespan

La comparaison de la performance des quatre algorithmes se fera à travers des graphes. Dans le premier ensemble d'expériences, nous comparons la performance en termes de makespan. À partir de la « Figure 4.4 », on peut voir que l'ordre de performance des algorithmes partant du moins performant est Round Robin, Random, HEFT, FTSA, DFWS. L'algorithme Round Robin atteint la valeur du makespan la plus basse. La valeur de makepan de l'algorithme proposé DFWS est supérieur de HEFT et de FTSA. Ceci est dû à la réplication passive et dynamique et au clustering qui réduisent le coût de communication et conduisent à un makespan plus important. Pour l'algorithme FTSA, un nombre croissant de tâches entraîne une augmentation significative des répliques, ce qui conduit à un makespan plus important. Ceci est différent de notre algorithme d'ordonnancement où la réplication réduit le coût de communication si la tâche fille est répliquée au cluster qui contient la tâche mère. Nous pouvons aussi remarquer que le gain du maskpan de l'approche DFWS proposée augmente en augmentant le nombre de tâches. Ces résultats confirment que notre approche fonctionne bien dans un environnement de grille multi-clusters.

Le tableau 4.1 indique la variation du makespan en fonction du nombre de tâches.

Tâche / *Stratégie*	10	20	30	40	50	60	70	80	90	100
Random	517,04	555,48	841,64	1278,2	1487,8	1673,16	2308,4	2473,68	2443,36	3277,52
RRobin	447,04	508,16	759,8	1101,28	1217,36	1370,44	1923,68	1936,56	1900,36	2744,08
HEFT	370,04	388,00	482,00	761,00	798,00	915,20	1 285,00	1 270,00	1 327,16	2 178,32
FTSA	300,00	350,00	400,5	748,56	820,46	900,58	1202,56	1210,13	1250,17	2000,15
DFWS	280,14	310,8	345,55	720,45	790,32	800,56	900,45	1000,2	1230,33	1500,23

TABLE 4.1 – Résultats de Makespan

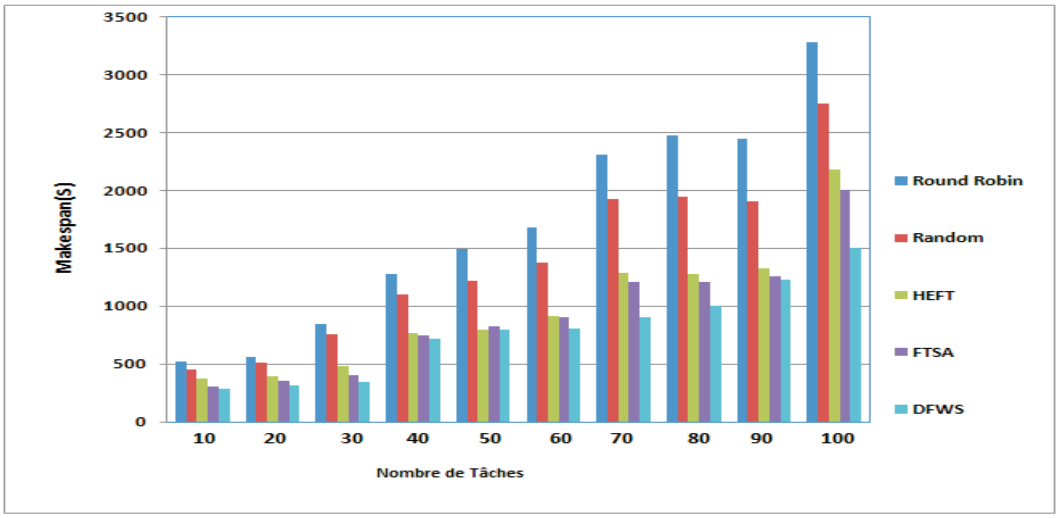

FIGURE 4.4 – Résultats de Makespan

Résultats de consommation des ressources

La « Figure 4.5 » met en exergue la comparaison entre les algorithmes (DFWS, FTSA). Leur ordre basé sur la consommation de ressources est (DFWS, FTSA).

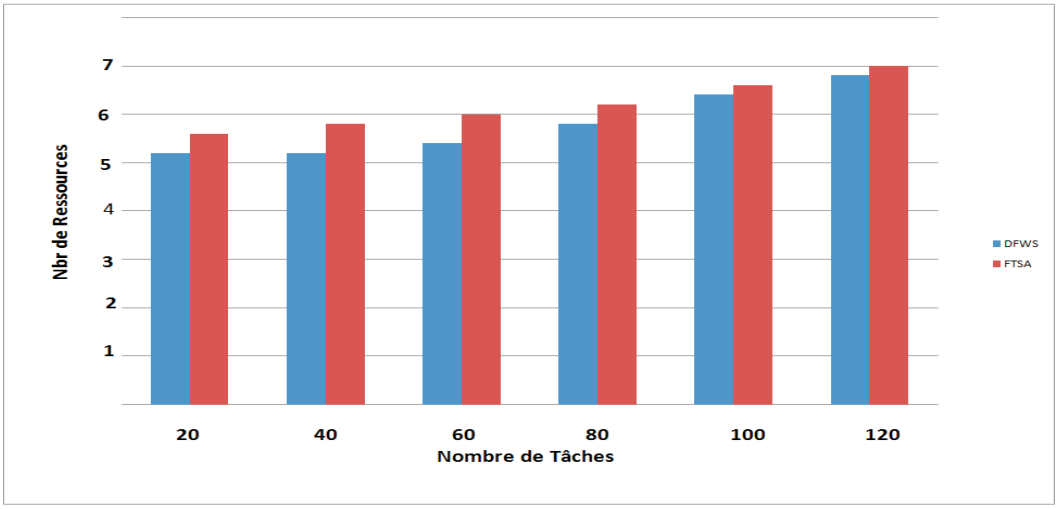

FIGURE 4.5 – Résultats de consommation de ressources

DFWS attribue plusieurs copies de chaque tâche uniquement aux NVs libres, ce mécanisme ne génère pas beaucoup de ressources supplémentaires. Contrairement à la méthode FTSA qui attribue plusieurs copies de chaque tâche à différentes ressources, ce qui conduit à une consommation énorme de ressources. En effet, accomplir plus de tâche conduit à une exécution plus lente.

Résultats de temps d'attente moyen

À partir de la « Figure 4.6 », nous pouvons voir que lorsque le nombre de tâches augmente, le temps d'attente moyen augmente. Nous remarquons également que l'algorithme DFWS nous a permis de réduire le temps d'attente moyen des tâches par rapport à l'algorithme FTSA. Ceci est dû premièrement au clustering qui réduit le temps d'attente en affectant deux tâches ayant un lient de dépendance au même cluster, en deuxième lieu à la réplication passive et dynamique, qui minimise le temps d'attente dans les files d'attente locale, de telle sorte que chaque copie secondaire est activée dans les VNs libres. Ainsi, si un VN tombe en panne, la copie secondaire activée ne doit pas attendre pour être exécutée mais s'exécute immédiatement.

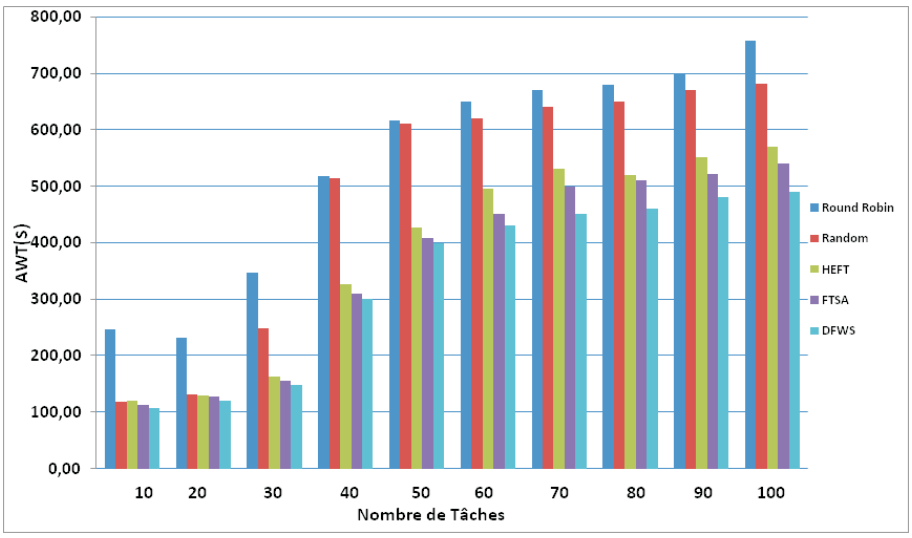

FIGURE 4.6 – Résultats de temps d'attente moyen

Le tableau 4.2 montre la variation du tempts d'attente moyen en fonction du nombre de tâches.

Tâche / *Stratégie*	10	20	30	40	50	60	70	80	90	100
Random	118,46	132,15	248,55	514,48	610,20	620,48	640,00	650,00	670,54	680,48
RRobin	231,16	246,43	347,95	517,00	617,00	650,00	670,00	680,00	700,34	758,16
HEFT	120,36	130,76	162,53	326,50	426,30	496,23	496,23	520,40	550,67	570,60
FTSA	113,26	128,95	156,12	309,34	409,24	450,33	500,00	510,56	520,67	540,89
DFWS	108,05	120,00	149,00	300,00	400,00	430,32	450,45	460,56	480,48	490,88

TABLE 4.2 – Résultats de temps d'attente moyen

Résultats de Speedup

Le speedup est calculé en divisant le temps d'exécution cumulé des tâches sur le makespan

$$Speedup = \frac{min_{Cj} \sum W_{ij}}{makespan} \tag{4.4}$$

Les résultats de speedup indiqués dans la Figure « 4.7 » montre que notre algorithme est supérieur aux autres algorithmes en terme de speedup. Ces résultats indique que l'algorithme DFWS s'exécute rapidement que les autres algorithmes. Le tableau 4.3 indique les résultats du speedup.

Résultats de fiabilité

Le graphe de comparaison entre la fiabilité avec la réplication et la fiabilité sans réplication est montré Dans la « Figure 4.8 ». Dès que les tâches sont répliquées et affectées aux différentes ressources, la fiabilité sera améliorée par rapport à l'ordonnancement sans réplication (SW), ainsi plus de répliques créera plus de composantes connexes, et comme la fiabilité de l'ordonnancement dépend du nombre des composantes connexes et que les composantes connexes augmente en augmentant le nombre de répliques et le nombre de tâches. la fiabilité augmente en augmentant le nombre de tâches.

Tâche / *Stratégie*	10	20	30	40	50	60	70	80	90	100
Random	0.3279	0.4849	0.2902	0.3218	0.34712	0.4095	0.3049	0.4338	0.2653	0.2073
RRobin	0.2680	0.3721	0.2571	0.2448	0.2984	0.3454	0.2761	0.2294	0.2247	0.1941
HEFT	0.4048	0.6509	0.4584	0.4449	0.5337	0.6311	0.4998	0.4454	0.4216	0.3731
FTSA	0.4093	0.8220	0.6337	0.4793	0.5733	0.6687	0.5129	0.4874	0.4281	0.4010
DFWS	1.2741	1.5306	1.3044	1.4521	1.8055	1.5404	1.4241	1.4755	1.4834	1.0307

TABLE 4.3 – Résultats de Speesup

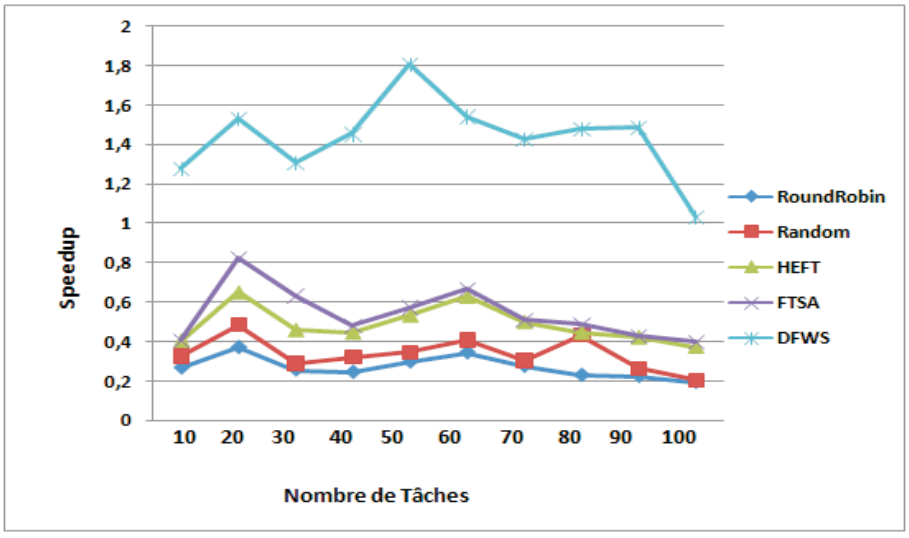

FIGURE 4.7 – Résultats de Speedup

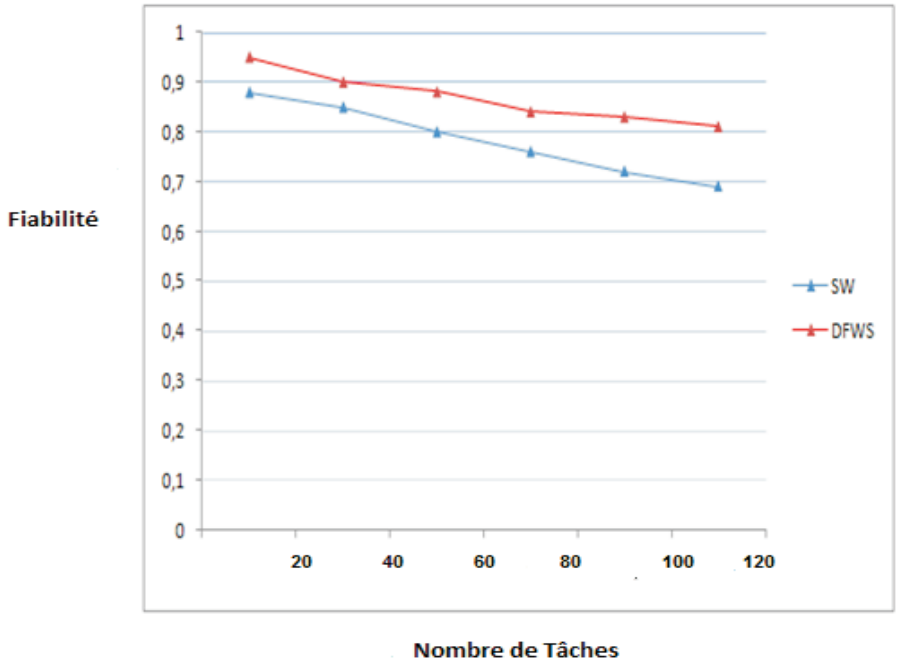

FIGURE 4.8 – Résultats de fiabilité

4.5 Conclusion

Partant d'un modèle à deux niveaux, nous avons présenté une stratégie d'ordonnancement de tâches tolérantes aux fautes. Cette stratégie est constituée de deux phases : une phase d'ordonnancement, et une phase de réplication passive. L'avantage des deux stratégies est qu'elles permettent de réduire, au maximum le makespan, et la consommation des ressources. Les expérimentations présentées dans ce chapitre nous ont permis de mettre en valeur la praticabilité de notre modèle, sur la base de ce modèle cinq stratégies d'ordonnancement ont été implémentées. La comparaison de notre approche avec les quatre autres approches (HEFT, FTSA, Random, et Round Robin), nous montre que notre approche apporte de meilleurs résultats lorsque nous prenons en compte la consommation de ressources et le makespan et le temps d'attente moyen. Nous avons démontré que l'objectif du compromis entre la performance et la fiabilité était atteint. La comparaison entre la fiabilité avant la tolérance aux fautes et après la tolérance aux fautes nous montre que notre approche apporte de meilleurs résultats. Cependant, une évaluation plus approfondie dans un environnement de grille réelle s'avère nécessaire pour aboutir à une conclusion sur l'importance des gains réalisés de par notre approche.

CONCLUSION GÉNÉRALE

1. *Synthèse*

*D*urant ces dernières années, on a remarqué une augmentation intense du volume des informations traitées, ce qui a fait naître le besoin du développement des réseaux à grande échelle et le partage géographique des informations à travers le monde entier. Ainsi les chercheurs ont réfléchi à la conception de grilles qui utilisent la puissance de calcul et l'espace de stockage de plusieurs unités de traitement interconnectées par des réseaux. Toutefois, les grilles de calcul ne sont pas encore arrivées à un niveau de maturité qui leur permettrait d'être généralisées et largement exploitées. Les principales difficultés proviennent de la nature des grilles de calcul, qui sont des architectures très hétérogènes et fortement dynamiques. De plus, leur exploitation nécessite une technicité qui n'est pas forcément à la portée de tout utilisateur. De nombreux travaux de recherche ont proposé des solutions permettant de faire face à une partie de ces difficultés alors que d'autres sont encore en cours. En ce qui nous concerne, nous nous sommes intéressés au problème d'ordonnancement de workflow tolérant aux fautes dans les grilles de calcul. Le problème d'ordonnancement de workflow dans n'importe quel système distribué est bien connu pour être NP-difficile. Il est beaucoup plus difficile lorsque l'on s'intéresse à l'ordonnancement de tâches tolérant aux fautes. Dans cette thèse, nous avons proposé une stratégie dynamique d'ordonnancement de workflow tolérante aux fautes. Partant de ce contexte, nous avons présenté un modèle pour l'ordonnancement de workflow tolérant aux fautes, adapté aux grilles de calcul et qui tient compte surtout de l'hétérogénéité des ressources et tout en étant totalement indépendant de toute architecture physique. En se basant sur ce modèle nous avons proposé une stratégie qui combine

l'ordonnancement avec la réplication des tâches, de sorte que le workflow est exécuté de manière efficace et fiable. Et ce, tout en utilisant la réplication passive et dynamique des tâches avec moins de consommation de ressources, contrairement aux stratégies de réplication citées dans le chapitre 2, et qui surmontent leurs l'inconvénients ayant trait à la consommation de ressources. Comme exemple, nous prenons, la stratégie FTSA [14] qui attribue plusieurs copies de chaque tâche aux différentes ressources. Ce mécanisme entraîne une large consommation de ressources, ainsi que la stratégie FTSW [144] qui maintient des copies répétées d'une table pour les travaux orphelins, ce qui conduit à un calcul redondant. En outre, lorsque le nombre de tâches est très élevé par rapport au nombre de ressources disponibles, la stratégie proposée ne conduit pas à la sérialisation des tâches au lieu et place d'une exécution parallèle. Ceci est dû à la de réplication passive et dynamique, qui attribue les répliques de chaque tâches aux VNs libres. Pour tester et évaluer les performances de notre stratégie, nous avons fait des simulations. Les résultats expérimentaux de fiabilité, de consommation de ressources et de makespan furent encourageants et ont montré que l'objectif du compromis entre la performance et la fiabilité était atteint. Nous avons également montré que la stratégie DFWS est meilleure que FTSA [14] en termes de consommation de ressources et de makespan.

2. Perspectives

Comme perspective :

- Nous prévoyons d'intégrer notre stratégie au middleware GLOBUS [48] ;
- Différents types de fautes sont entrelacées dans l'environnement de grilles, telles que les fautes de blocage, et les fautes du programme. Nous prévoyons également d'étudier ces types de fautes ;
- Dans ce travail, nous supposons que le FGCM ne reçoit qu'un seul workflow de l'utilisateur, nous prévoyons également d'appliquer la même stratégie avec plusieurs workflows ;
- Il serait intéressant de mettre en œuvre cette stratégie et de la tester sur une grille réelle comme Grid 5000 ou EGEE.

Bibliographie

[1] Internet. Workflow management coalition. http ://www.wfmc.org.

[2] The Condor Team. Dagman : Directed acyclic graph manager. http ://www.cs.wisc.edu/condor/dagman.

[3] J. H. Abawajy. Fault-tolerant scheduling policy for grid computing systems. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings*, pages 238–244, April 2004.

[4] I. Ahmad and Y.K.K. Yu-Kwong Kwok. A new approach to scheduling parallel programs using task duplication. In *International Conference on Parallel Processing*, volume 2, pages 47–51, Aug 1994.

[5] A. AL-Fawair. *Génération automatique de distributions/ordonnancements temps réel, fiables et tolérants aux fautes.* PhD thesis, Institut National Polytechnique de Grenoble - INPG, 2004.

[6] K. Amin, G. Laszewski, M. Hategan, N. J. Zaluzec, S. Hampton, and A. Rossi. Gridant : a client-controllable grid workflow system. In *HICSS'04 : Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSSâ04) - Track 7, page 70210.3*, Jan 2004.

[7] A. Avižienis, J.C. Laprie, and B. Randell. Dependability and its threats : A taxonomy. In *Building the Information Society : IFIP 18th World Computer Congress Topical Sessions 22-27 August 2004 Toulouse, France*, pages 91–120, 2004.

[8] E. Badidi. *Architecture et services pour la distribution de charge dans les systèmes distribués objet.* PhD thesis, Faculté des études supérieures, Université de Montréal, Mai 2000.

[9] M. Baker, R. Buyya, and D. Laforenza. Grids and grid technologies for widearea dis-

tributed computing. *Journal of Software Practice and Experience*, 32(15) :1437–1466, December 2002.

[10] A. Baratloo, M. Karaul, Z.M. Kedem, and P. Wijckoff. Charlotte : Metacomputing on the web. *Future Generation Computer Systems*, 15(5) :559 – 570, 1999.

[11] G. Belalem. Contribution et la gestion de la cohérence de répliques de fichiers dans les systèmes à large échelle. Thèse de Doctorat d'État, Faculté des sciences, Université d'Oran, Novembre 2007.

[12] G. Belalem, M. Meddeber, F. Kalfadj, A. Benddine, and S. Bouhazab. Un protocole de gestion de réplication dans un environnement de simulation optorsim. *Conférence Internationale sur l'informatique et ses applications (CIIA06), Saida, Algérie*, Mai 2006.

[13] W.H. Bell, D. G. Cameron, A. P. Millar, L. Capozza, K. Stockinger, and F. Zini. Optorsim : A grid simulator for studying dynamic data replication strategies. *The International Journal of High Performance Computing Applications*, 17(4) :403–416, 2003.

[14] A. Benoit, M. Hakem, and Y. Robert. Fault tolerant scheduling of precedence task graphs on heterogeneous platforms. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, USA*, pages 1–8, April 2008.

[15] M. Bertier. *Service de détection de défaillances hiérarchiques.* PhD thesis, Université de Paris 6, 2006.

[16] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proceedings International Conference on Dependable Systems and Networks, USA*, pages 354–363, June 2002.

[17] F. Boem, R.M.G. Ferrari, T. Parisini, and M.M. Polycarpou. A distributed fault detection methodology for a class of large-scale uncertain input-output discrete-time nonlinear systems. In *50th IEEE Conference on Decision and Control and European Control Conference*, pages 897–902, Dec 2011.

[18] R. Bolze. *Analyse et déploiement de solutions algorithmiques et logicielles pour des applications bio-informatiques à grande échelle sur la grille.* PhD thesis, Ecole normale supérieure de lyon - ENS LYON, 2008.

[19] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello. Mpich-v project : A multiprotocol automatic fault-tolerant mpi. *The International Journal of High Performance Computing Applications*, 20(3) :319–333, 2006.

[20] P. Brucker. Scheduling algorithms. In *Springer, Berlin, Heidelberg*, 2007.

[21] K. Budati, J. Sonnek, A. Chandra, and J. Weissman. Ridge : Combining reliability and performance in open grid platforms. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, HPDC '07, pages 55–64, 2007.

[22] R. Buyya and M. Murshed. Gridsim : a toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation : Practice and Experience*, 14(13-15) :1175–1220, 2002.

[23] R.D. Camargo, R. Cerqueira, and F. Kon. Strategies for storage of checkpointing data using non-dedicated repositories on grid systems. In *Proceedings of the 3rd International Workshop on Middleware for Grid Computing*, MGC'05, pages 1–6, 2005.

[24] L. C. Canon, E. Jeannotl, R. Sakellariou, and W. Zheng. Comparative evaluation of the robustness of dag scheduling heuristics. Technical report, 2008.

[25] J. Cao, A.T.S. Chan, Y. Sun, S.K. Das, and M. Guo. A taxonomy of application scheduling tools for high performance cluster computing. *Cluster Computing*, 9(3) :355–371, 2006.

[26] J. Cao, S. A. Jarvis, S. Saini, D. J. Kerbyson, and G. R. Nudd. Arms : an agent-based resource management system for grid computing. 10(2) :135–148, 2002.

[27] J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd. Gridflow : workflow management for grid computing. In *CCGrid 2003. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2003. Proceedings*, pages 198–205, May 2003.

[28] H. Casanova. Simgrid : a toolkit for the simulation of application scheduling. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 430–437, 2001.

[29] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2) :141–154, Feb 1988.

[30] A. Chakrabarti and S. Sengupta. Scalable and distributed mechanisms for integrated scheduling and replication in data grids. In *Distributed Computing and Networking : 9th International Conference, ICDCN 2008, Kolkata, India*, pages 227–238, 2008.

[31] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2) :225–267, March 1996.

[32] D. Chen, R. S. Chen, and T. H. Huang. A heuristic approach to generating file spanning trees for reliability analysis. *Computers and Mathematics with Applications*, 34(10) :115–131, 1997.

[33] M. Chen, A.X. Zheng, J. Lloyd, M.I. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *International Conference on Autonomic Computing, 2004. Proceedings*, pages 36–43, May 2004.

[34] M. Chtepen, F. Claeys, B. Dhoedt, B. Turck, P. Vanrolleghem, and P. Demeester. Scheduling of dependent grid jobs in absence of exact job length information. In *Proceedings of the 4th IEEE/IFIP International Workshop on End-to-end Virtualization and Grid Management, Samos Island, Greece*, pages 22–26, Sept 2008.

[35] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J.S. Chase. Correlating instrumentation data to system states : A building block for automated diagnosis and control. In *OSDI'04, USA*, pages 16–32, 2004.

[36] R.W. Conway, W.L. Maxwell, and L.W. Miller. *Theory of Scheduling*. Dover Books on Computer Science Series. Dover, 2003.

[37] A. Das, I. Gupta, and A. Motivala. Swim : scalable weakly-consistent infection-style process group membership protocol. In *Proceedings International Conference on Dependable Systems and Networks*, pages 303–312, 2002.

[38] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1(1) :25–39, 2003.

[39] M.E.M. Diouri, O. Gück, L. Lefevre, and F. Cappello. Energy considerations in checkpointing and fault tolerance protocols. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*, pages 1–6, June 2012.

[40] S. Djilali, T. Herault, O. Lodygensky, T. Morlier, G. Fedak, and F. Cappello. Rpc-v : Toward fault-tolerant rpc for internet connected desktop grids with volatile nodes. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC'04, 2004.

[41] F. Douglis and J. Ousterhout. Transparent process migration : Design alternatives and the sprite implementation. *Software : Practice and Experience*, 21(8) :757–785, 1991.

[42] W. Fabian and E. Rolf. Execution cost interval refinement in static software analysis. *Journal of Systems Architecture*, 47(3-4) :339 – 356, 2001.

[43] G. Falzon and M. Li. Evaluating heuristics for scheduling dependent jobs in grid computing environments. *BOOK, IGI Global*, pages 31–46, 2012.

[44] C. Ferdinand and R. Wilhelm. On predicting data cache behavior for real-time systems. *Languages, Compilers, and Tools for Embedded Systems : ACM SIGPLAN Workshop LCTES'98 Montreal, Canada, June 19–20, Proceedings*, pages 16–30, 1998.

[45] R. Fernandes, K. Pingali, and P. Stodghil. Mobile mpi programs in computational grids. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 22–31, 2006.

[46] R.M.G. Ferrari, T. Parisini, and M.M. Polycarpou. Distributed fault diagnosis of large-scale discrete-time nonlinear systems : New results on the isolation problem. In *49th IEEE Conference on Decision and Control (CDC)*, pages 1619–1626, Dec 2010.

[47] G. Florin. La tolérance aux pannes dans les systèmes répartis. Technical report, 1996. Laboratoire CEDRIC CNAM.

[48] I. Foster. Globus toolkit version 4 :software for service oriented systems. *Journal of computer Science and Technology*, 21(4) :513–520, 2006.

[49] I. Foster and C. Kesselman. *Computational grids. The Grid2 : Blueprint for a New Computing Infrastructure*. San Francisco, CA, U.S.A., 2003.

[50] W.H. Freeman and D. S. Johnson. Computers and intractability. *A Guide to the Theory of NP-Completeness*, 1990.

[51] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-g : A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3) :237–246, 2002.

[52] M. Gabel, A. Schuster, R.G. Bachrach, and N. Bjørner. Latent fault detection in large scale services. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12, June 2012.

[53] E. Gabriel, G.E. Fagg, A. Bukovsky, T. Angskun, and J.J. Dongarra. A fault tolerant communication library for grid environments. In *Proceedings of the 17th Annual ACM International Conference on Supercomputing (ICS'03)*, 2003.

[54] R. Garg and A.K . Singh. Fault tolerant task scheduling on computational grid using checkpointing under transient faults. *Arabian Journal for Science and Engineering*, 39(12) :8775–8791, 2014.

[55] S. Genaud. *Exécutions de programmes parallèles à passage de messages sur grille de calcul*. PhD thesis, Université Henri Poincaré - Nancy I, Dec 2009.

[56] S. Genaud, E. Jeannot, and C. Rattanapoka. Fault-management in p2p-mpi. *International Journal of Parallel Programming*, 37(5) :433–461, 2009.

[57] S. Genaud and C. Rattanapoka. P2p-mpi : A peer-to-peer framework for robust execution of message passing parallel programs on grids. *Journal of Grid Computing*, 5(1) :27–42, 2007.

[58] C. Germain, J. Nauroy, and K. Rafes. The Grid Observatory 3.0 - Towards reproducible research and open collaborations using semantic technologies, May 2015. EGI Community Forum 2014.

[59] J.E. Ghaffour. *Sécurité Dans Les Grilles De Calcul*. PhD thesis, Université Rennes, France, 2004.

[60] A. Girault, C. Lavarenne, M. Sighireanu, and Y. Sorel. Fault-tolerant static scheduling for real-time distributed embedded systems. In *Proceedings 21st International Conference on Distributed Computing Systems*, pages 695–698, Apr 2001.

[61] T. Glatard. *Description, deployment and optimization of medical image analysis workflows on production grids*. PhD thesis, Université de Nice Sophia Antipolis, Sophia Antipolis, 2007.

[62] C.A. Goble and D.C.D. Roure. my experiment : Social networking for workflow-using e-scientists. In *WORKS'07 : Proceedings of the 2Nd Workshop on Workflows in Support of Large-scale Science, Monterey, California, USA*, pages 1–2, 2007.

[63] S. Gomathi and D. Manimegalai. An adaptive grouping based job scheduling in grid computing. In *2011 International Conference on Signal Processing, Communication, Computing and Networking Technologies*, pages 192–196, July 2011.

[64] T. Goodale, G. Allen, G. Lanferman, J. Massó, T. Radke, E. Seidel, and J. Shalf. The cactus framework and toolkit : Design and applications. In *High Performance Computing for Computational Science — VECPAR 2002 : 5th International Conference Porto, Portugal, June 26–28, 2002 Selected Papers and Invited Talks*, pages 197–227, 2003.

[65] J.P. Goux, S. Kulkarni, J. Linderoth, and M. Yoder. An enabling framework for master-

worker applications on the computational grid. In *Proceedings of the Ninth International Symposium on High-Performance Distributed Computing*, pages 43–50, 2000.

[66] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling : a survey. In *Discrete Optimization IIProceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symposium co-sponsored by IBM Canada and SIAM Banff, Aha. and Vancouver*, volume 5, pages 287 – 326. 1979.

[67] A. Greenspan, P.J. O'Rourke, and P.R. Auld. Providing virtual machine technology as an embedded layer within a processing platform, 2008. US Patent App. 11/513,877.

[68] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *Reliable Software Technologies — Ada-Europe'96 : 1996 Ada-Europe International Conference on Reliable Software Technologies Montreux, Switzerland, June 10–14, 1996 Proceedings*, pages 38–57, 1996.

[69] F.P. Guimaraes, P. Célestin, D.M. Batista, G. N. Rodrigues, and A. C. M. A. de Melo. A framework for adaptive fault-tolerant execution of workflows in the grid : Empirical and theoretical analysis. *Journal of Grid Computing*, 12(1) :127–151, 2014.

[70] F.P. Guimaraes and A.C. Magalhaes Alves de Melo. User-defined adaptive fault-tolerant execution of workflows in the grid. *Computer and Information Technology (CIT), IEEE 11th International Conference, Cyprus*, pages 356–362, 2011.

[71] L. Guo, G. Shao, and S. Zhao. Multi-objective task assignment in cloud computing by particle swarm optimization. In *8th International Conference on Wireless Communications, Networking and Mobile Computing*, pages 1–4, Sept 2012.

[72] I. Gupta, T. D. Chandra, and G. S. Goldszmidt. On scalable and efficient distributed failure detectors. In *PODC '01*, pages 170–176, 1988.

[73] R. W. Hockney. The communication challenge for mpp : Intel paragon and meiko cs-2. *Parallel Computing*, 20(3) :389 – 398, 1994.

[74] Y. Horita, K. Taura, and T. Chikayama. A scalable and efficient self-organizing failure detector for grid applications. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID'05, pages 202–210, 2005.

[75] S. Hwang and C. Kesselman. A flexible framework for fault tolerance in the grid. *Journal of Grid Computing*, 1(3) :251–272, 2003.

[76] A. Ishfaq and K. Yu-Kwong. On exploiting task duplication in parallel program scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 9(9) :872–892, Sep 1998.

[77] M.A. Iverson, F. Güner, and G.J. Follen. Parallelizing existing applications in a distributed heterogeneous environment. In *4TH HETEROGENEOUS COMPUTING WORKSHOP (HCW'95)*, pages 93–100, 1995.

[78] S. Jafar. *Programmation des systèmes parallèles distribués : tolérance aux pannes, résilience et adaptabilité*. PhD thesis, Institut National Polytechnique de Grenoble - INPG, 2006.

[79] Y. Jiang and W. Chen. Task scheduling in grid computing environments. *Genetic and Evolutionary Computing*, 238(10) :23–32, 2014.

[80] H. Jitsumoto, T. Endo, and S. Matsuoka. Abaris : An adaptable fault detection/recovery component framework for mpis. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, March 2007.

[81] C. Junwei, S. A. Jarvis, S. Saini, and G. R. Nudd. Gridflow : workflow management for grid computing. In *CCGrid 2003. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2003. Proceedings*, pages 198–205, May 2003.

[82] A. Kalakech. *Étalonnage de la sûreté de fonctionnement des systèmes d'exploitation - Spécifications et mise en oeuvre*. PhD thesis, de Toulouse - INPT, 2005.

[83] F. Kalfadj and B. Yagoubi. Vers une heuristique d'assignement de workflows tolérante aux pannes. *Conférence Internationale sur le Traitement de l'Information Multimédia CITIM 2015, Mascara, Algérie*, 2015.

[84] F. Kalfadj and B. Yagoubi. Dynamic fault tolerant scheduling policy for workflows in grid computing. *Multiagent and Grid Systems*, 12(4) :287–302, 2016.

[85] F. Kalfadj, B. Yagoubi, and M. Meddeber. Conception d'un simulateur de grilles orienté gestion d'équilibrage. In *Conflérence Internationale sur l'Informatique et ses Applications (CIIA), Saida, Alglérie*, 2009.

[86] G. Kandaswamy, A. Mandal, and D.A. Reed. Fault tolerance and recovery of scientific

workflows on computational grids. *Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on, France*, pages 777–782, May 2008.

[87] L. Ke, J. Hai, C. Jinjun, L. Xiao, Y. Dong, and Y. Yun. A compromised-time-cost scheduling algorithm in swindew-c for instance-intensive cost-constrained workflows on a cloud computing platform. *The International Journal of High Performance Computing Applications*, 24(4) :445–456, 2010.

[88] T. Kielman, H.E. Bal, and K. Verstoep. Fast measurement of logp parameters for message passing platforms. *Parallel and Distributed Processing : 15 th IPDPS 2000 Workshops Cancun, Mexico, May 1–5, 2000 Proceedings*, pages 1176–1183, 2000.

[89] T. Kohonen. The self-organizing map. *Neurocomputing*, 21(1-3) :1 – 6, 1998.

[90] G. Kola, T. Kosar, and M. Livny. Phoenix : making data-intensive grid applications fault-tolerant. In *Fifth IEEE/ACM International Workshop on Grid Computing*, pages 251–258, Nov 2004.

[91] G. Kola, T. Kosar, and M. Livny. Faults in large distributed systems and what we can do about them. In *Euro-Par 2005 Parallel Processing : 11th International Euro-Par Conference, Lisbon, Portugal, August 30 - September, 2005. Proceedings*, pages 442–453, 2005.

[92] D. Kondo, B. Javadi, A. Iosup, and D. Epema. The failure trace archive : Enabling comparative analysis of failures in diverse distributed systems. In *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 398–407, May 2010.

[93] Y.K . Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4) :406–471, December 1999.

[94] Y.K. Kwok and I. Ahmad. Benchmarking the task graph scheduling algorithms. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 531–537, Mar 1998.

[95] S.M. Lambert. Réplication et Durabilité dans les systèmes répartis. Technical report, 2001. école Polytechnique Fédérale de Lausanne.

[96] L. Lamport. Paxos made simple, fast, and byzantine. In *OPODIS*, pages 7–9, 2006.

[97] G. Lanfermann, G. Allen, T. Radke, and E. Seidel. Nomadic migration : Fault tolerance in a disruptive grid environment. In *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium*, pages 280–280, May 2002.

[98] J.C. Laprie. Dependable computing : Concepts, challenges, directions. In *COMPSAC, Hong Gong*, 2004.

[99] J.C. Laprie. Dependable computing and fault tolerance : Concepts and terminology. In *15 th IEEE international Sympsium on Fault Tolerant Computing (FTCS), Ann, Arbor, MI, USA*, pages 2–11, juin 1985.

[100] A. Legrand, A. Su, and F. Vivien. Minimizing the stretch when scheduling flows of divisible requests. *Journal of Scheduling*, 11(5) :381–404, 2008.

[101] J. K. Lenstra and A. H. G. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1) :22–35, 1978.

[102] H. Li, D. Groep, L. Wolters, and J. Templon. Job failure analysis and its implications in a large-scale production grid. In *Second IEEE International Conference on e-Science and Grid Computing (e-Science'06)*, pages 27–27, Dec 2006.

[103] S.H. Low. A duality model of tcp and queue management algorithms. *IEEE/ACM Transactions on Networking*, 11(4) :525–536, Aug 2003.

[104] B. Ludäscher, M. Weske, T. McPhillips, and S. Bowers. Scientific workflows : Business as usual ? In *Business Process Management : 7th International Conference, BPM 2009, Ulm, Germany, September 8-10, 2009. Proceedings*, pages 31–47, 2009.

[105] S. Majithia, M. Shields, I. Taylor, and I. Wang. Triana : a graphical web service composition and execution toolkit. In *Proceedings. IEEE International Conference on Web Services, 2004*, pages 514–521, July 2004.

[106] B. Marin, C. Henri, R.Yves, V. Frédéric, and Z. Dounia. Using group replication for resilience on exascale systems. *The International Journal of High Performance Computing Applications*, 28(2) :210–224, 2014.

[107] E.N. Melnozahy, L. Alvisi, Y.M. Wang, and D.B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3) :375–408, September 2002.

[108] M. Mezmaz, N. Melab, and E.G. Talbi. A grid-enabled branch and bound algorithm

for solving challenging combinatorial optimization problems. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–9, March 2007.

[109] J. Mincer-Daszkiewicz. *A Service for Reliable Execution of Grid Applications.* PhD thesis, Warsaw University in Warsaw, Poland, 2006.

[110] S. Monnet. *Gestion des données dans les grilles de calcul : support pour la tolérance aux fautes et la cohérence des données.* PhD thesis, IRISA, Rennes, France, 2006.

[111] M. Nandagopal and V.R. Uthariaraj. Fault tolerant scheduling strategy for computational grid environment. *International Journal of Engineering Science and Technology*, 9(2) :4361–4372, 2010.

[112] A. Natrajan, M.A. Humphrey, and A.G. Grimshaw. Capacity and capability computing using legion. In *Computational Science — ICCS 2001 : International Conference San Francisco, CA, USA, May 28–30, 2001 Proceedings, Part I*, pages 273–283, 2001.

[113] M.O. Nearyl, S.P. Brydon, and P.K. Paul. Javelin++ : Scalability issues in global computing. In *Proceedings of the ACM 1999 Conference on Java Grande*, JAVA '99, pages 171–180, New York, NY, USA, 1999. ACM.

[114] A. S. Mohd Noor, M. Mat Deris, M.Y. Bin Mohd, and E. A. Sirajudin. Distributed dynamic failure detection. *JOURNAL OF SOFTWARE*, 9(5) :1342–1347, 2014.

[115] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. Pocock, A. Wipat, and P. Li. Taverna : a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17) :3045–3060, 2004.

[116] F. Paul. Fault diagnosis in dynamic systems using analytical and knowledge-based redundancy : A survey and some new results. *Automatica*, 26(3) :459 – 474, 1990.

[117] D. Pelleg, M. Ben-Yehudai, R. Harper, L. Spainhower, and T. Adeshiyan. Vigilant : Out-of-band detection of failures in virtual machines. *SIGOPS Oper. Syst. Rev.*, 42(1) :26–31, January 2008.

[118] K. Plankensteiner and R. Prodan. Meeting soft deadlines in scientific workflows using resubmission impact. *IEEE Transactions on Parallel and Distributed Systems*, 23(5) :890–901, May 2012.

[119] C. Pouzat, M. Delescluse, P. Viot, and J. Diebolt. Improved spike-sorting by modeling firing statistics and burst-dependent spike amplitude attenuation : A markov chain monte carlo approach. *Journal of Neurophysiology*, 91(6) :2910–2928, 2004.

[120] R. Presuhn. Version 2 of the protocol operations for the simple network management protocol (snmp). Technical report, USA, 2002.

[121] R. Prodan and T. Fahringer. Overhead analysis of scientific workflows in grid environments. *IEEE Transactions on Parallel and Distributed Systems*, 19(3) :378–393, March 2008.

[122] B. Ramkumar and V. Strumpen. Portable checkpointing for heterogeneous architectures. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, pages 58–67, June 1997.

[123] B. Rashmi and D.P. Agrawal. Improving scheduling of tasks in a heterogeneous environment. *IEEE Transactions on Parallel and Distributed Systems*, 15(2) :107–118, Feb 2004.

[124] B. Rashmi and D.P. Agrawal. Improving scheduling of tasks in a heterogeneous environment. *IEEE Transactions on Parallel and Distributed Systems*, 15(2) :107–118, 2004.

[125] M. Rebbah. *Tolérance aux fautes dans les grilles de calcul.* PhD thesis, Université d'Oran, 2015.

[126] X. Ren, R. Eigenmann, and S. Bagchi. Failure-aware checkpointing in fine-grained cycle sharing systems. In *Proceedings of the 16 th International Symposium on High Performance Distributed Computing*, HPDC'07, pages 33–42, 2007.

[127] H. Sabbah. *Modélisation et dimensionnement d'une plate-forme hétérogène de services.* PhD thesis, Institut National polytechnique de Grenoble, Université du Luxembourg, 2009.

[128] S. Sadi and B. Yagoubi. Communication-aware approaches for transparent checkpointing in cloud computing. *Scalable Computing : Practice and Experience*, 17(3) :251–270, 2016.

[129] S. Sadi and B. Yagoubi. Virtual machine's dependencies aware framework for fault tolerance in cloud. In *Proceedings des 3ème Journées Doctorales en Informatique à l'Université de Guelma, Guelma, Algeria*, pages 24 :1–24 :5, Déc. 2013.

[130] S. Sadi and B. Yagoubi. Improved fault tolerance through a multi-zone checkpointing approach. In *Proceedings de la 3ème édition des journées de l'étudiant à l'École na-*

tionale Supérieure d'Informatique (JEESI), Algiers, Algeria, pages 61 :1– 61 :4, Mai 2014.

[131] S. Sadi and B. Yagoubi. On the optimum checkpointing interval selection for variable size checkpoint dumps. In *Proceedings of the 5th IFIP International Conference on Computer Science and Its Applications, Saida, Algeria*, pages 599–610, Mai 2015.

[132] R.K. Sahoo, A.J. Oliner, I. Rish, M. Gupta, J.E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD'03, pages 426–435, 2003.

[133] M.A. Salehi and R. Buyya. Adapting market-oriented scheduling policies for cloud computing. *Algorithms and Architectures for Parallel Processing : 10th International Conference, ICA3PP 2010, Busan, Korea, May 21-23, 2010. Proceedings. Part I*, pages 351–362, 2010.

[134] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The lam/mpi checkpoint/restart framework : System-initiated checkpointing. *The International Journal of High Performance Computing Applications*, 19(4) :479–493, 2005.

[135] L.F.G. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. *Future Generation Computer Systems*, 18(4) :561 – 572, 2002. Best papers from Symp. on Cluster Computing and the Grid (CCGrid2001).

[136] K. Sato, A. Moody, K. Mohror, T. Gamblin, B. R. D. Supinski, N. Maruyama, and S. Matsuoka. Fmi : Fault tolerant messaging interface for fast and transparent recovery. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1225–1234, May 2014.

[137] C. Schneider, A. Barker, and S. Dobson. Autonomous fault detection in self-healing systems : Comparing hidden markov models and artificial neural networks. In *Proceedings of International Workshop on Adaptive Self-tuning Computing Systems*, ADAPT '14, 2014.

[138] C. Schneider, A. Barker, and S. Dobson. Autonomous fault detection in self-healing systems using restricted boltzmann machines. *CoRR*, abs/1501.01501, 2015.

[139] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4) :337–350, Oct 2010.

[140] E. Seidel, G. Allen, A. Merzky, and J. Nabrzyski. Gridlabâa grid application toolkit and testbed. *Future Generation Computer Systems*, 18(8) :1143 – 1153, 2002.

[141] A. B. Sharma, H. Chen, M. Ding, K. Yoshihira, and G. Jiang. Fault detection and localization in distributed systems using invariant relationships. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–8, June 2013.

[142] B. Shirazi, M. Wang, and G. Pathak. Analysis and evaluation of heuristic methods for static task scheduling. *Journal of Parallel and Distributed Computing*, 10(3) :222–232, 1990.

[143] D. Siewiorek and R. Swarz. *Reliable Computer Systems : Design and Evaluation*. Digital Press, 2014.

[144] K. Srikala and S. Ramachandram. Fault tolerant scheduling of workflows in grid computing environment (ftsw). *Communication Technologies (GCCT), 2015 Global Conference on, India*, pages 343–347, April 2015.

[145] R. Stewart, P. Trinderl, and P. Maier. *Supervised Workpools for Reliable Massively Parallel Computing*, pages 247–262. 2013.

[146] R. Strom and S. Yemin. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3) :204–226, August 1985.

[147] A. Takefusa, S. Matsuoka, H. Nakada, K. Aida, and U. Nagashima. Overview of a performance evaluation system for global computing scheduling algorithms. In *Proceedings. The Eighth International Symposium on High Performance Distributed Computing*, pages 97–104, 1999.

[148] H. Topcuoglu, S. Hariri, and W. Min-You. Task scheduling algorithms for heterogeneous processors. In *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth*, pages 3–14, 1999.

[149] H. Topcuoglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3) :260–274, Mar 2002.

[150] P. Townend, P. Groth, N. Looker, and J. Xu. Ft-grid : A fault-tolerance system for e-science. In *AHM05*, 2005.

[151] P. Townend and J. Xu. Fault tolerance within a grid environment. In *Proceedings of the UK e-Science All Hands Meeting 2003*, pages 227–275, 2003.

[152] D. B. Tracy, J. S Howard, B. Noah, L. B. Ladislau, M. Muthucumaru, I. R. Albert, P. R. James, D. Mitchell, Y. Bin, H. Debra, and F. Richard. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6) :810 – 837, 2001.

[153] G. Tristan, M. Johan, L. Diane, and P. Xavier. Flexible and efficient workflow deployment of data-intensive applications on grids with moteur. *The International Journal of High Performance Computing Applications*, 22(3) :347–360, 2008.

[154] F. Tronel. *Application des problèmes d'accord à la tolérance aux défaillances dans les systèmes distribués asynchrones*. PhD thesis, 2003. Thèse de doctorat dirigée par Raynal, Michel Informatique Rennes 1.

[155] S. Vadhiyar and J. Dongarra. Srs : A framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Processing Letters*, 13(2) :291–312, 2003.

[156] L. Valcarenghi and P. Castoldi. Qos-aware connection resilience for network-aware grid computing fault tolerance. In *Proceedings of 2005 7th International Conference Transparent Optical Networks, 2005*, volume 1, pages 417–422, July 2005.

[157] S. van der, M.M.M.P.J. Claessen, and D. Alstein. A hierarchical membership protocol for synchronous distributed systems. *Dependable Computing — EDCC-1 : First European Dependable Computing Conference Berlin, Germany, October 4–6, 1994 Proceedings*, pages 599–616, 1994.

[158] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Middleware'98 : IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, 1998.

[159] S. Varrette. *Dag Scheduling For Grid Computing Systems*. PhD thesis, University Of Udine, Italy, 2006.

[160] D. C. Verma, S. Sahu, S. Calo, A. Shaikh, I. Chang, and A. Acharya. Sriram : A scalable resilient autonomic mesh. *IBM Systems Journal*, 42(1) :19–28, 2003.

[161] A. Villemeur. *Sureté de fonctionnement des systèmes industriels : fiabilité - facteurs humains, informatisation. Eyrolles, Paris* . Dir. Et. Rech. Electr. France. Eyrolles, Paris, 1988.

[162] P. Watson. Grid computing : Making the global infrastructure a reality. Technical report, 2002.

[163] J. Weissman and B.D. Lee. The virtual service grid : an architecture for delivering high-end network services. *Concurrency and Computation : Practice and Experience*, 14(4) :287–319, 2002.

[164] N. Woo, S. Choi, and H. Jung. Providing fault tolerance on grid environments. In *Cluster Computing and the Grid (CC-Grid2003)*, 2003.

[165] N. Woo, H.Y. Yeom, and T. Park. Mpich-gf : Transparent checkpointing and rollback-recovery for grid-enabled mpi processes. In *IEICE TRANSACTIONS on Information and Systems*, volume 87, pages 1820–1828, 2003.

[166] B. Yagoubi. Equilibrage de charge dans les grilles de calcul. Thèse de Doctorat d'État, Faculté des sciences, Université d'Oran, Novembre 2007.

[167] J. Yu and R. Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3) :171–200, 2005.

[168] V. C. Zandy, B.P. Miller, and M. Livny. Process hijacking. In *Proceedings. The Eighth International Symposium on High Performance Distributed Computing (Cat. No.99TH8469)*, pages 177–184, 1999.

[169] S. Zanikolas and R. Sakellariou. A taxonomy of grid monitoring systems. *Future Generation Comp. Syst*, 21(1) :163–188, 2005.

[170] X. Zhang, F. Junqueira, K. Marzullo, R.D. Schlichting, and M. Hiltunen. Replicating nondeterministic services on grid environments. In *15th IEEE International Conference on High Performance Distributed Computing*, pages 105–116, 2006.

[171] X. Zhang, D. Zagorodnov, M. Hiltunen, K. Marzullo, and R.D. Schlichting. Fault-tolerant grid services using primary-backup : feasibility and performance. In *IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935)*, pages 105–114, 2004.

[172] Y. Zhang, C. Koelbel, and K. Kennedy. Combined fault tolerance and scheduling techniques for workflow applications on computational grids. *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, China*, pages 244–251, 2009.

[173] Y. Zhang, C. Koelbel, and K. Kennedy. Relative performance of scheduling algorithms in grid environments. *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07), Brazil*, pages 521–528, May 2007.

[174] S. Zhiao and J.D. Jack. Scheduling workflow applications on processors with different capabilities. *Future Generation Computer Systems*, 22(6) :665 – 675, 2006.

LISTE DES SYMBOLES

L_i Niveau i

W_{ij} Coût d'exécution de la tâche t_i assignée au VN j

σ_{c_J} Ècart-type du coût d'exécution des tâches

W_{C_J} Coût d'exécution moyen da la tâche t_i assignée au cluster C_J :$W_{C_J} = \sum W_{ij}/nbr_{VN}$

$\overline{c}_{ij}$ Coût de communication moyen

C_J Cluster J

$t_{i,j}$ Réplique j de la tâche i

$succ(t_i)$ Ensemble des successeurs de la tâche t_i

$pred(t_i)$ Ensemble des prédécesseurs de la tâche t_i

$avail[VN_j]$ Temps où le VN_j termine l'exécution de la tâche t_i
et il est prêt à exécuter une autre tâche.

$avail[C_J]$ Temps où le cluster C_J termine l'exécution de la tâche t_i
et il est prêt à exécuter une autre tâche.

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

Chapitre1

MTTF Minimum Time To Failure

RR Rollback-Recovery

GSI Grid Security Infrastructure

MDS Metacomputing Directory Service

GRAM Globus Resource Allocation Manager

GASS Global Access to Secondary Storage

GARA Advanced Resource Reservation and Allocation

FTS Service de Tolérance aux Fautes

VM Virtual Machine

Chapitre2

PERT Program (ou Project) Evaluation and Review Technique

UML Langage de Modélisation Unifié

OMG Object Management Group

DAG Directed Acyclic Graph

ASAP As Soon As Possible

ALAP As Late As Possible

MCT Minimum Completion Time

SRPT Shortest Remaining Processing Time

LRPT Longest Remaining Processing Time

HEFT Heterogeneous Earliest Finish Time

CPOP Critical Path on Processor

LMT Levelized Min Time

GDL Generalized Dynamic Level

GAT Grid Application Toolkit

ARMS Agent-based Resource Management System for grid Computing

FTSA Fault Tolerant Scheduling Algorithm

Chapitre3

AFT Actual Finish Time

EFT Earliest Finish Time

SQ Schedule Queue

FGCM Fault Global Cluster Manager

FLCM Fault Local Cluster Manager

VN Virtual Node

VL Virtual Link

Chapitre4

QOS Quality of Service

Multiagent and Grid Systems – An International Journal 12 (2016) 287–302
DOI 10.3233/MGS-160255
IOS Press

Dynamic fault tolerant scheduling policy for workflows in Grid computing

Kalfadj Fatima and Yagoubi Belabbas*
Department of Computer Science, University of Oran1 Ahmed Ben Bella, Oran, Algeria

Received 27 February 2016
Accepted 8 July 2016

Abstract. More and more complex scientific workflows are now executed on computational Grids. In addition to the challenges of managing and scheduling resources, reliable challenges arise because the Grid infrastructure is unreliable. Current Grid systems mainly exploit fault tolerance mechanisms to guarantee the dependable workflow execution, which wastes system resources. In this paper, we have chosen an alternative way for reliable workflow scheduling, schedule tasks replicas to resources with the objective of meeting a reliability requirement, in such a way that all tasks are replicated with less resource consumption, using Passive Dynamic Replication of Tasks method namely (PDRT).

Keywords: Grid, workflows, fault tolerance, reliability, resource consumption, replication, scheduling

1. Introduction

Grid is a collection of geographically distributed, highly heterogeneous and loosely coupled computing environments [1,2]. Recent developments in Grid infrastructure technologies allow to execute large and distributed applications on it [3,4]. Many of these applications fall in the category of workflow applications, which are often represented as DAGs (Directed Acyclic Graphs) with nodes representing computations and edges representing data movement [5,6]. Because of its decomposable nature, the workflow paradigm has emerged as the most successful paradigm for applications to be executed on Grid infrastructures [7,8]. At the same time, the recent growth in size and complexity of the Grid infrastructure causes failures appearance at all system levels, power supply, computing hardware, network, operating system, Grid middleware [9]. Therefore, fault tolerance issues must be taken into consideration [10]. The level of fault tolerance is reflected by quantifying the system dependability. Dependability means that our system can be trusted to deliver the service(s) for which it has been designed. It can be measured by reliability metric. Reliability characterizes the ability of a system to perform on demand, its service correctly. However distributed network reliability for small-scale systems has been extensively studied, these studies have raised some limitations:

(i) The models only consider the hardware failures of links and processors without taking into account the software and resource failures;

*Corresponding author: Yagoubi Belabbas, Department of Computer Science, University of Oran1 Ahmed Ben Bella, Oran, Algeria. E-mail: byagoubi@gmail.com.

ISSN 1574-1702/16/$35.00

(ii) The operational probabilities of nodes or links are constant without considering bandwidth and contention;
(iii) The network topology is made up of physical links and nodes that are static without considering dynamic changes of components and logic structures.

These assumptions need to be relaxed for Grid [11]. In a Grid network, communication between two remote sites can be logically broken even though a physical link exists between them, the authority to use some remote resources might be malformed for instance [12]. To solve such problems, we propose a new model using a virtual instead of a physical structure. This model simplifies the physical structure of a Grid computing, allows service performance (execution time, reliability) to be efficiently evaluated, and guaranties fault detection and fault tolerance.

We consider a workflow as a set of atomic tasks, interconnected in a directed acyclic graph through control flow and data flow dependencies, find a schedule for the DAG so that the system reliability can be optimized and the time constraint can be met (a fast schedule is not necessarily a reliable one). This problem can be solved by using Dynamic Fault Tolerant Scheduling strategy for Workflows namely (DFWS). This strategy couples workflow scheduling with workflow replication, in such a way that the workflow is efficiently and reliably executed using Passive Dynamic Replication of Tasks method namely (PDRT). The proposed strategy consists in: task prioritizing phase, where tasks are ordered according to their priorities, tasks clustering phase, where tasks are grouped to clusters and assigned to several resources, and replication phase, where children tasks are replicated and assigned to free resources.

The paper is organized as follows. We begin with the overview of the literature review in Section 2. Section 3 presents the system model, and some definitions according to the literature. Section 4 describes the main steps of the proposed strategy. We evaluate the performance of the proposed strategy in Section 5. Section 6 concludes the paper.

2. Literature review

In this section, we discuss some fault tolerance techniques which are used for scheduling workflows. Scheduling is a very important task which determines the performance of a whole system [13]. This task should be tolerant to the failures that occur in the computing environment [14,15]. Fault tolerant scheduling algorithms can be categorized based on:

- Checkpointing: In checkpointing and rollback approaches the state of the running job is saved at some fixed points during its execution. These points are called checkpoints. The saved state of the job is used for recovery in case of a resource failure. The authors in [16] described a mechanism for fault tolerant workflows scheduling by considering checkpointing, migration, and over-provisioning. Authors in [17] proposed Fault Tolerant Task Scheduling Strategy namely FTTS for independent as well as dependent tasks Grid applications. This strategy takes the failure rate and computing capacity of resources into consideration. In [18], authors proposed new approaches that combine checkpointing with existing workflows scheduling algorithms. This mechanism can improve system reliability but incurs much additional time on saving checkpoints and rollback recovery.
- Replication: Methods like replication of jobs do not require any history of information. The job is replicated on multiple resources, number of replicas are generated either statically or dynamically using the history of failure rate of a node. In [19], author proposed a workflows scheduling strategy based on replication.

In [20], authors proposed Fault Tolerant Scheduling of Workflows in a Grid computing environment namely FTSW, which provides fault tolerant scheduling of workflows of jobs by maintaining a group table of values for Orphan jobs (Jobs not able to contact Parent Job because of processor failure at Parent Job) and allows applications to continue their execution event if there is a resource failure. In this strategy, repeated copies of the table are maintained at nodes that execute the jobs of the workflow, which leads to redundant computation.

In [21], authors proposed FTSA (Fault-Tolerant Scheduling Algorithm), an extended version of the classic HEFT Algorithm [22]. It allocates $\varepsilon + 1$ copies for each task to different resources to tolerate an arbitrary number of resource failures. This mechanism leads to a large amount of resource overheads. However if there is enough number of resources available, then the workflow replication algorithms presents good results. Usually, the number of tasks will be very high, when compared to the number of resources available. Hence these methods of replication may lead to task serialization instead of parallel execution [23].

– Resubmission: Resubmission tries to re-execute jobs to mitigate failures. Resubmission or redundancy in time helps recover from transient faults or soft errors.

In [18], authors proposed a new approach that combines fault tolerance techniques with existing workflows scheduling algorithms. In [24], authors proposed an algorithm called Resubmission Impact (RI) that tries to establish a metric describing the impact of resubmitting a task to the overall execution time of a workflow application, and to adjust the replication size of each task accordingly. This mechanism usually has a predefined limit for the number of retries where it will attempt to resolve a failure.

To overcome the disadvantages of replication and resubmission, few techniques are discussed in [25], which finds the tradeoff between the replication and the resubmission. Tasks are replicated without considering any other parameters except the heuristic metric which may replicate all tasks even if they are not critical and may lead to resource wastage.

In this paper, we have chosen an alternative way for reliable workflow scheduling, schedule tasks replicas to resources with the objective of meeting a reliability requirement, in such a way that all tasks are replicated with less resource consumption using Passive Dynamic Replication of Tasks method namely (PDRT). The proposed strategy consists in:

(i) Tasks scheduling: Relies on task prioritizing phase that orders tasks according to their priorities, and tasks clustering phase that groups tasks to clusters and assigned them to several resources;
(ii) Tasks replication: Children tasks are replicated and assigned to free resources.

3. System model and assumptions

3.1. Workflow taxonomy

The taxonomy of workflows characterizes and classifies approaches of workflow management in the context of Grid computing [26]. As shown in Fig. 1, the taxonomy consists of sub-taxonomies based on the elements of a Grid workflow management system:

(a) Workflow design: Includes key factors involved in the workflow at build-time;
(b) Information retrieval: Retrieves information from appropriate entities;
(c) Workflow scheduling: Is a kind of global task scheduling as it focuses on mapping and managing the execution of inter-dependent tasks on shared resources that are not directly under its control;

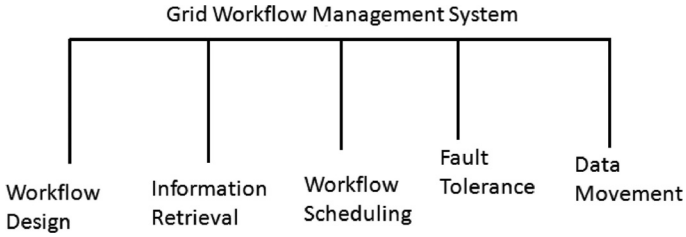

Fig. 1. Elements of a grid workflow management system.

(d) Fault tolerance: Handles failures flexibly and support reliable executions in the presence of concurrency and failures;
(e) Data movement: Manages intermediate data transfer in the workflow.

In this paper, a workflow is composed by connecting multiple scientific tasks according to their dependencies. It can be represented by a directed acyclic graph $D = (V, E)$, where V is a set of v nodes and E is a set of directed e edges [27]. A node in the DAG (as described in Table 2) represents a task which in turn is a set of instructions. The edges, denoted by $(t_i \longmapsto t_j)$, correspond to the precedence constraints among the nodes. The weight of an edge is called the communication cost of the edge and is denoted by c_{ij}, it denotes the volume of data being transmitted between $(t_i \longmapsto t_j)$. We call the source node of an edge the parent node while the sink node is called the child node. A node with no parent is called an entry node and a node with no child is called an exit node. A task is said to be ready if all its parents have finished their execution and can start its execution only after all its dependencies have been satisfied. If the two tasks t_i and t_j, are mapped to the same cluster, the communication cost between them is assumed to be zero.

3.2. Grid architecture

In this paper, we present a more reasonable virtual structure which has a tree topology.

The root of the virtual tree structure is the FGCM.[1] The leaves are the clusters, while the branches of the tree represent the Virtual Links (VLs) (as described in Table 2). The VLs link the leaves and the root.

Some channels (VLs) are used in common by multiple clusters. Each cluster in turn consists of the FLCM.[2] In this paper, we consider a VN as a site that hosts a set of nodes and each node may have a single or multiple PEs (as described in Table 2). As shown in Fig. 2, the cluster architecture can be approximated by a structure with a star topology.

In the proposed model, we will consider the following information:

(a) For VLs linking the clusters, the model considers the available bandwidth, the exchanged data from different clusters that contend for the bandwidth of the VLs, and the failure rate of the VLs;
(b) For VLs linking the VNs (as described in Table 2) to the FLCM, the model does not consider the available bandwidth;
(c) For VNs, the time for the VNs to execute tasks is involved.

[1]Fault Global Cluster Manager: Detects the available clusters, assigns tasks to those clusters for execution, detects and tolerates faults in the clusters.

[2]Fault Local Cluster Manager: Detects the available VNs, assigns tasks to those VNs for execution, detects and tolerates faults in the VNs.

Table 1
List of used symbols

L_i	Level i
W_{ij}	Computation cost of the task t_i scheduled to the VN_j
σ_{c_J}	Standard deviation of the computation costs of each task
W_{C_J}	Average computation cost of the task t_i scheduled to the cluster C_J: $W_{C_J} = \sum W_{ij}/nbr_{VN}$
$\overline{c}_{ij}$	Average communication cost
C_J	Cluster J
λ_{VN}	VN's failure rate
μ_{VN}	Communication path failure rate between the FGCM and the clusters
$t_{i,j}$	j'th replica of the task i
$succ(t_i)$	Set of immediate successor of the task t_i
$pred(t_i)$	Set of immediate predecessor of the task t_i
$avail[VN_j]$	Earliest time at which the VN_j is ready for task execution
$avail[C_J]$	Earliest time at which the cluster J is ready for task execution

Table 2
Table of abbreviations

VN	Virtual node
VL	Virtual link
DAG	Direct acyclic graph
EST	Earliest start time
EFT	Earliest finish time
AFT	Actual finish time
FTSW	Fault tolerant scheduling of workflow
FTSA	Fault tolerant scheduling algorithm
SW	Scheduling without replication
PEs	Processing elements
AWT	Average wait time
CC_i	i'th connected component
FTTS	Fault Tolerant task scheduler
nbr_{NV}	Number of VNs

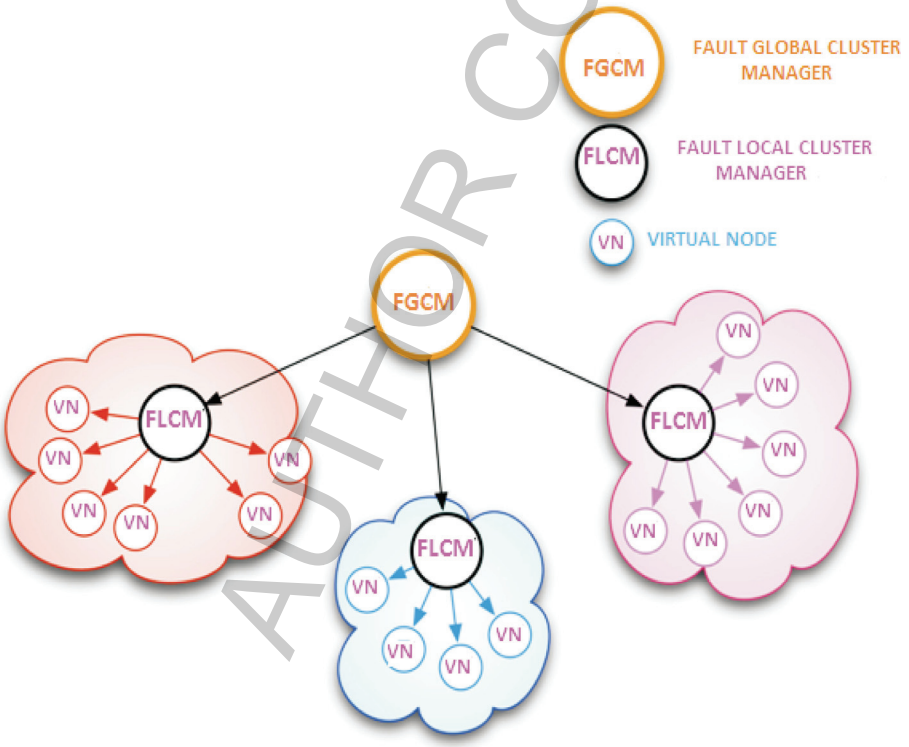

Fig. 2. Virtual structure of a grid computing.

4. Proposed method

The general workflow scheduling problem includes the problem of assigning tasks of a DAG to suitable Clusters/VNs and the problem of ordering tasks execution on each VN. It consists in the following phases:

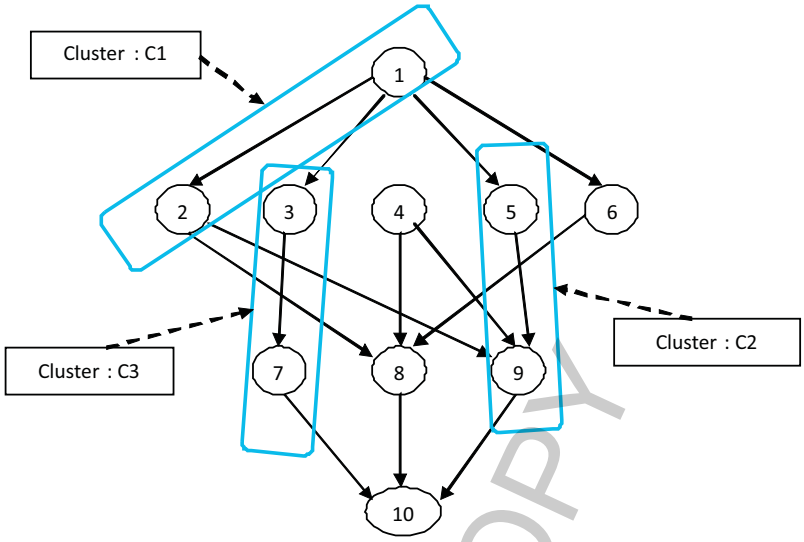

Fig. 3. Task clustering.

4.1. Task prioritizing phase

This phase requires the priority of each task to be set with a priority value, which is based on the computation and the average communication costs, the standard deviation of the computation costs of each task σ_{c_J}, and the task level (L_i) (as defined in Table 1). The task list is generated by sorting tasks by decreasing order of their priorities.

$$Prio(t_i) = \max(\sigma_{c_J}) + \max_{t_j \varepsilon succ(t_i)} (L_i * \overline{c}_{ij} + Prio(t_j)) \tag{1}$$

$$\sigma_c = \sqrt{\sum (W_{ij} - W_{C_J})^2 / nbr_{VN}} \tag{2}$$

A ready task is defined as the task whose parent tasks have completed their executions. At the beginning, the schedule queue (SQ) consists of the entry tasks because they have no parent tasks [28]. They are executed according to the generated schedule. When a task t_i is completed, SQ is updated by removing t_i from SQ and inserting new tasks.

4.2. Task clustering phase

The tasks of the DAG are sorted in descending order of their priorities. With this order, they are divided into different clusters as follows:

The first task is added to a cluster numbered 0. If the successive tasks in descending order of their priorities are dependent with one task already assigned to the cluster (namely, there is dependence between them), they are placed in the same cluster. Reversely, if there is independence, a new cluster will be created and the new cluster's number is the current cluster's number increased by one. The final outcome are a set of ordered clusters, as shown in Fig. 3. Tasks of each cluster are assigned by the FGCM to the cluster that minimizes their earliest finish times (EFTs) (as described in Table 2), and to the VN belonging this cluster that minimizes their earliest finish times (EFTs).

We define the EST (as described in Table 2) of the task t_i scheduled to the VN_j:

$$EST(t_i, VN_j) = \max\{avail[VN_j], \max_{t_m \varepsilon pred(t_i)}(AFT_{(t_m)} + c_{im})\} \quad (3)$$

We define the AFT (as described in Table 2) of the task t_i scheduled to the VN_j:

$$AFT(t_i, VN_j) = W_{ij} + EST(t_i, VN_j) \quad (4)$$

We define the EST of the task t_i scheduled to the cluster J:

$$EST(t_i, C_J) = \max\{avail[C_J], \max_{t_m \varepsilon pred(t_i)}(AFT_{(t_m)} + c_{im})\} \quad (5)$$

We also define the AFT of the task t_i scheduled to the cluster J:

$$AFT(t_i, C_J) = W_{C_J} + EST(t_i, C_J) \quad (6)$$

Algorithm 1 gives instructions of scheduling tasks of a DAG to Clusters/VNs. Firstly tasks are ordered according to their priorities. Tasks are grouped to clusters according to their dependencies in step 2. Then, in step 3 tasks are assigned to Clusters/VNs according to their EFTs.

Algorithm 1 : Scheduling Algorithm

1: Set the computation costs of the tasks, the communication costs of the edges and the standard deviation of the computation costs of each task
2: Compute the priority of each task ▷ as per Eq. (1)
3: Sort tasks by decreasing order of their priorities ▷ tasks ranking
4: **while** there are none clustered tasks in the list **do**
5: Add the first task to a cluster numbered 0 ▷ the first cluster constructed
6: **if** the successive tasks are dependent with one task of the cluster i **then**
7: Add the successive tasks to the cluster i
8: **else**
9: Create a new cluster, its number is the last cluster's number increased by one
10: Add the successive task to this cluster
11: **end if**
12: **end while**
13: **while** there are unscheduled tasks in each cluster **do**
14: Select the first task from the cluster
15: Assign the task to the cluster that minimizes its EFT ▷ as per Eq. (6)
16: Assign the task to the VN that minimizes its EFT ▷ as per Eq. (4)
17: Assign the successive task in the same group to the same cluster
18: **end while**

4.3. Fault detection

The Fault detection is crucial for providing a scalable, dependable and highly available Grid computing environment, fault detection can be divided into:

(a) Global fault detection: The FLCMs send the results of tasks executed to the FGCM, once a VN fails to run a task, the FLCM sends the number of this failed task to the FGCM, the FGCM asks the FLCM that contains its secondary copy to activate it for execution;
(b) Local fault detection: The VNs periodically send heartbeat messages [29] to their FLCMs, announcing that they are alive. In the absence of such messages, the FLCMs recognize that a failure has occurred at that VN, it then sends the number of the failed tasks to the FGCM.

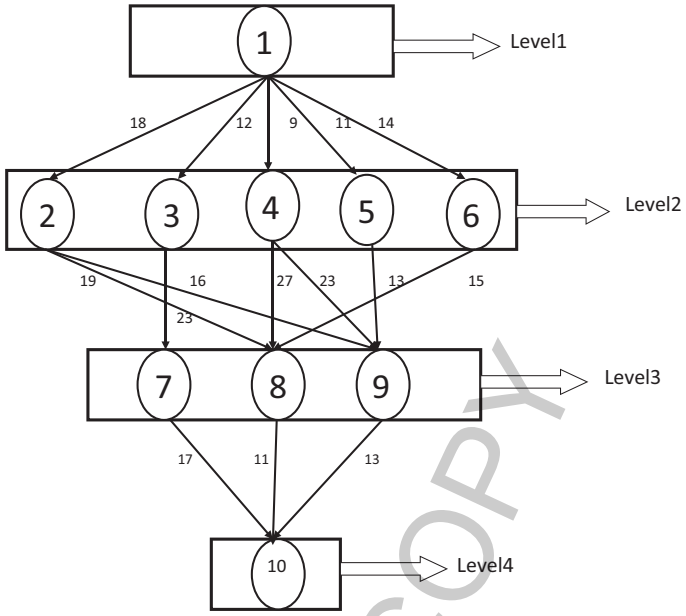

Fig. 4. DAG divided into levels.

4.4. Fault recovery

The fault recovery follows the virtual model, the hardware tolerance is achieved by applying the Passive Dynamic Replication of Tasks method namely (PDRT). It covers the following phases.

4.4.1. Spatial replication of a given task

Replication employed, should be minimized for reducing resource consumption [30]. The main steps of the spacial replication are:

(a) Tasks are replicated by levels, as shown in Fig. 4;
(b) Tasks of each level are replicated and assigned to free VNs;
(c) Tasks of each level are replicated and assigned to a VN_j, where $t_k \ldots$; t_m have been assigned according to the following formula:

$$EST(t_{k+1}, j) - MAX(EFT(t_k, j) < W_{ij} \tag{7}$$

4.4.2. Temporal replication of a given task

Children tasks are replicated and submitted to several VNs as soon as their parents start execution. For example, task 8 will be replicated only when task 4 starts its execution, as shown in Fig. 4. When the task 4 is completed, the schedule queue (SQ) is updated by removing their copies.

4.4.3. Temporal activation of the primary copy

Once a VN fails to run a task, if a task t_i belongs to this VN, its secondary copy is activated into the free VN. This mechanism as shown in Fig. 5, could reduce:

(i) The earliest finish time of tasks;

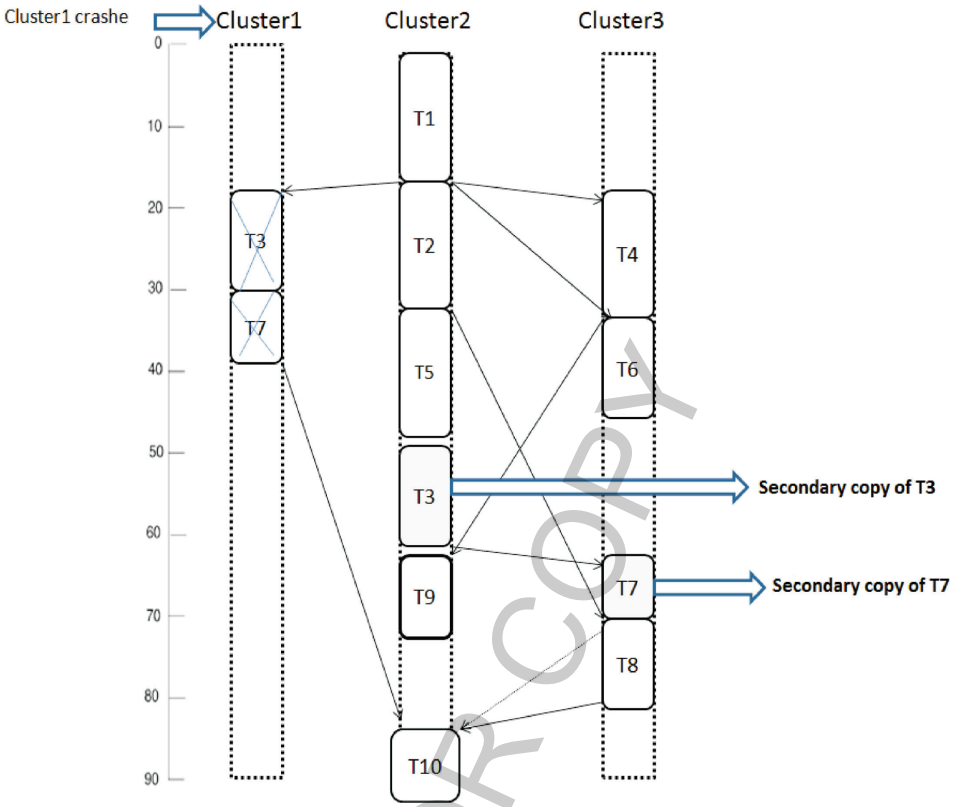

Fig. 5. Fault tolerance mechanism.

(ii) The waiting time and thereafter makespan;[3]
(iii) The communication costs, if the secondary copy is activated in the cluster which contains its parent.

Algorithm 2 gives an explanation of the spatial replication using instructions. Firstly the enter tasks are replicated and assigned to free VNs. Children tasks of each enter task are replicated as soon as their parents start execution in the second steps. This operation is repeated for all tasks of the DAG.

Algorithm 2 : Replication Algorithm

```
while there are unscheduled tasks in each cluster do
    Replicate the enter tasks and assigned them to free VNs
    if the task parent starts its execution then
        Replicate and assigned the children tasks to free VNs        ▷ as per Eq. (7)
    end if
end while
```

[3]Makespan, M, is the total time required to execute the entire workflow. The deadline D is considered as a constraint where the Makespan M should not be more than the deadline ($M \leqslant D$) [31]. The makespan of the workflow is computed according the following formula: $M = AFT - EST$

4.5. Grid reliability

4.5.1. Failure types

The possible types of failure that can occur in the Grid service are [32]:

(a) Blocking failure and time-out failure: Usually, the requests queue may have a limitation on the maximal number of requests waiting in the queue. When a new request arrives, if the queue is full, it delayed a blocking failure. If the waiting time for the request in the queue exceeds the due time, a time-out failure occurs;
(b) Matchmaking failure: Occurs when the request fails to match with resources;
(c) Network failure: When the programs are using remote resources, the communication channels may be disconnected which causes 'network failure';
(d) Resources failure: The resource shared on Grid may be software, hardware, or firmware. Hence, when using the resource, its failure can cause a service failure.

4.5.2. Reliability evaluation

Task i can be successfully completed by the VN_j, if this VN and communication path do not fail before the end of the execution. The conditional probability of the task success is calculted according to the folloing formula:

$$p(t_{ij}) = e^{((\lambda_{VN}+\mu_{VN})*(AFT(t_i,VN_j)))} \tag{8}$$

The above model of the Grid, as shown in Fig. 2 can be represented by an undirected graph $g = (V, E)$ in which V, $|V| = n$, n is the set of clusters. The undirected graph, as shown in Fig. 6 requires a technique for evaluating reliability. The technique based on graph theory, and it is described as following:

(a) The set of nodes and links involved in performing the given graph forms a group of all the connected components of the virtual structure bellow;
(b) Each connected component represents a minimal possible combination of available elements (VNs and VLs), which guarantees the success of the completion of the entire direct acyclic graph;
(c) Failure of any connected component leads to the entire graph failing;
(d) We determine the corresponding computation cost and communication cost for all VNs and VLs;
(e) We sort the connected components in an increasing order of their conditional direct acyclic graph completion time, and divide them into different groups containing connected components with identical conditional completion time;
(f) We suppose that there is K such groups, denoted by $G_1; G_2; \ldots; G_k$, and any group G_i contains connected components with identical conditional completion time.

Then the schedule reliability is:

$$SchedRelia = PR((\overline{Ecc_i}), (\overline{Ecc_{i-1}}), \ldots, (\overline{Ecc_3}), (Ecc_1)) \tag{9}$$

where Ecc_i is the event when at least one of the connected components from the group G_i is available and $\overline{Ecc_i}$ is the event when none of the connected components from the group G_i is available. Suppose the connected components in a group G_i are arbitrarily ordered, and V_{ij} represents an event when the j-th connected component in the group is available. Then, the event Ecc_i can be expressed by: $Ecc_i = \cup V_{ij}$.

And the equation above takes the form:

$$SchedRelia = PR(\overline{\cup V_{ij}}, (\overline{Ecc_{i-1}}), (\overline{Ecc_3}), (Ecc_1)) \tag{10}$$

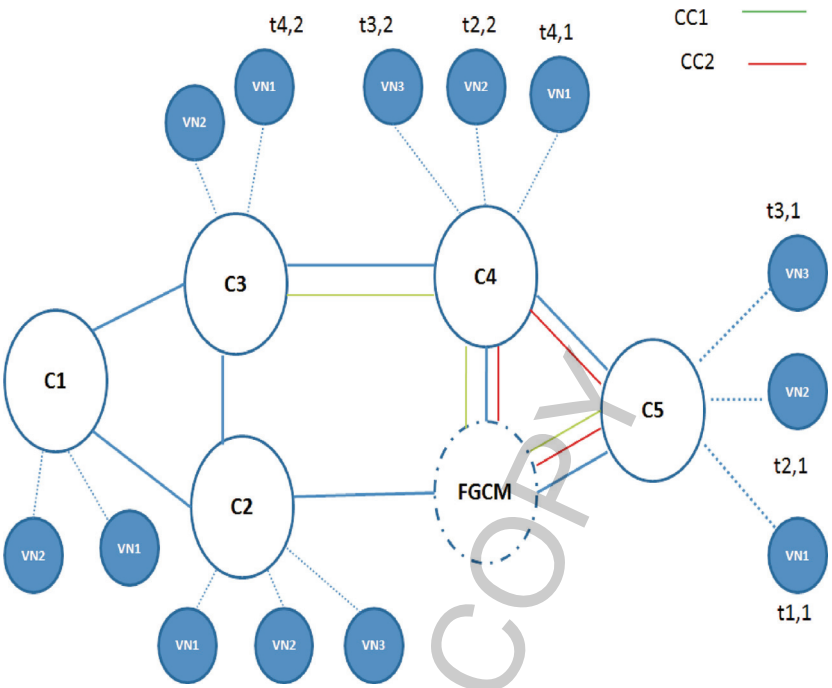

Fig. 6. Example of the connected components.

5. Performance evaluation

Our simulation infrastructure is created by using the simulator namely GridSim [5]. GridSim is a simulation toolkit that provides core functionality for different kinds of heterogeneous resources, services and application types.

5.1. Simulation parameters

All our experiments have been conducted on a core i3 Intel machine running at 2.53 GHz and equipped with 4 GB RAM. We also assume that:

(a) The structure of the Grid service can be depicted by the structure of VNs and VLs;
(b) The failures occurring in VN and VL follow Poisson's Processes [11];
(c) The failures of different elements (VNs and VLs) are independent;
(d) We assume that the FGCM and the FLCMs do not fail during the Grid service execution. This assumption is reasonable because the FGCM and FLCMs (based on multiple servers) usually have a very high reliability;
(e) We consider only network failures and resources failures.

5.2. Baseline policies

We have compared the proposed strategy (DFWS) with four other policies as follows:

(a) HEFT (Heterogeneous Earliest-Finish-Time) is a classical static list scheduling algorithm [22]. It does not take the resource failure into consideration, it is widely used [26] and Zhang et al. [33],

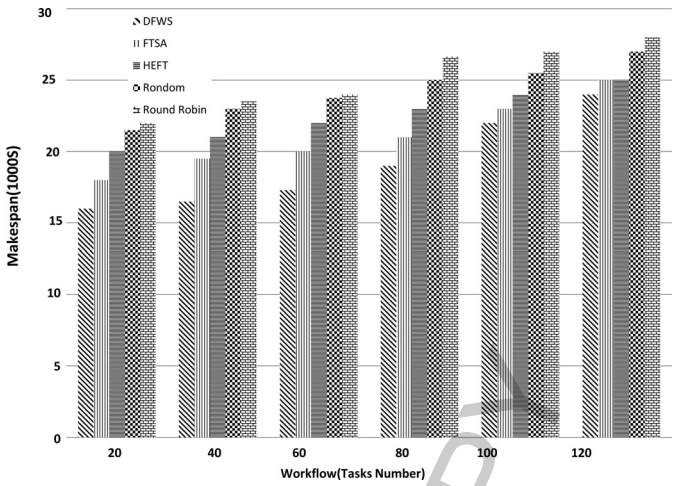

Fig. 7. Makespan results. (a) 120 is the maximum number of tasks; (b) 240 is the maximum number of tasks.

have shown that it performs well in a multi-cluster grid environment. However, it is a good reference to test the performance of the proposed strategy;

(b) FTSA (Fault-Tolerant Scheduling Algorithm) is an extended version of the classic HEFT algorithm. It allocates $\varepsilon + 1$ copies of each task to different resources to tolerate an arbitrary number of resource failures [21];
(c) Random strategy: Tasks are affected randomly to resources;
(d) Round robin strategy: Tasks are affected using a round robin strategy.

5.3. Performance results

The results are organized in four parts. The first part presents results on makespan. The second part presents results on resource consumption. The third part presents results on average wait time. The last part presents results with respect to reliability. A detailed analysis is given for each set of experiments.

5.3.1. Makespan results

The performance of the four algorithms is compared with respect to various graphs. In the first set of experiments, we compare the performance in terms of makespan. From Fig. 7, we can see that the performance order of the algorithms starting from the worst performing algorithm is Round Robin, Random, HEFT, FTSA, DFWS. The Round robin algorithm achieves the lowest makespan value. The makespan value of the proposed algorithm DFWS is very close to that of HEFT and FTSA. This is due to the Passive Dynamic Replication. For the FTSA algorithm, an increasing number of tasks leads to a significant increase of replicas. Hence leads to a more important makespan. This is different to our scheduling algorithm where replication reduces the earliest finish times of tasks, decreases the waiting time and thereafter makespan.

5.3.2. Resource consumption results

Figure 8 gives the comparison of the tow algorithms (DFWS, FTSA), and their order based on resource consumption is (DFWS, FTSA). DFWS assigns multiple copies of each task only to free resources (VNs), which does not lead to much extra resource consumption. Reversely to FTSA which assigns multiple copies of each task to different resources no matter that the failures affect its execution. This

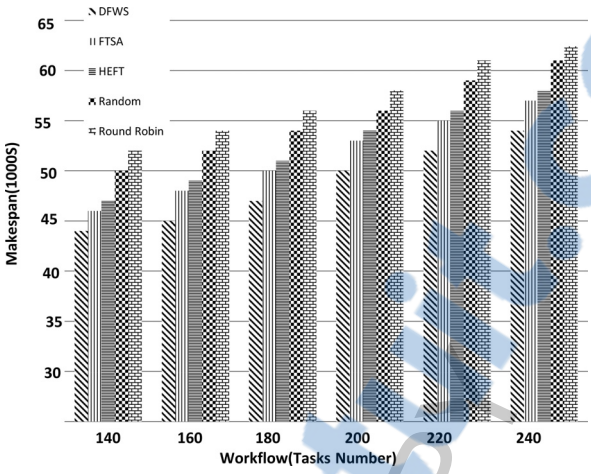

Fig. 8. Resource consumption results. (a) 120 is the maximum number of tasks; (b) 240 is the maximum number of tasks.

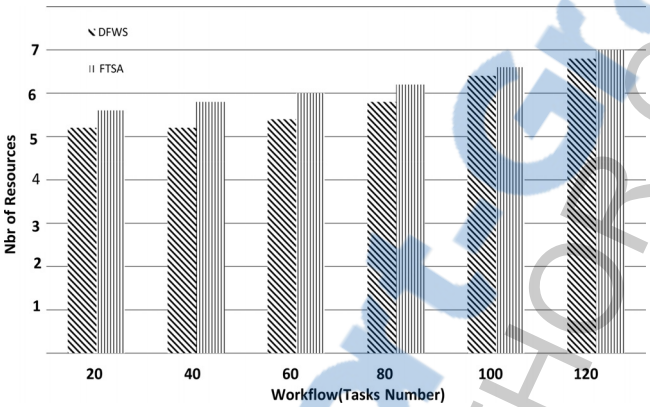

Fig. 9. Average wait time results.

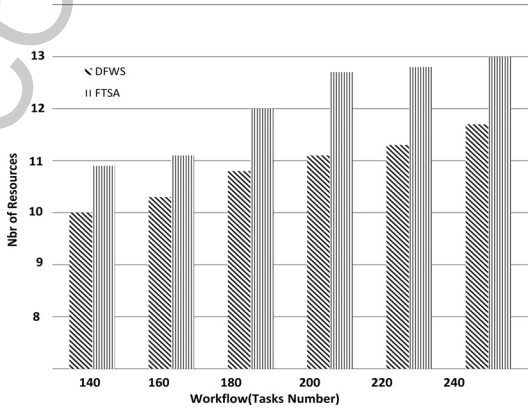

Fig. 10. Reliability results.

algorithm leads to a large amount of resource overheads. More tasks lead to longer makespan. Hence, more failures occur during the execution lead to more replicas as well as resource consumption for each task.

5.3.3. Average wait time (AWT) results

From Fig. 9, we can see that when the number of tasks increases, the Average Waiting Time increases. We also notice, that the DFWS algorithm allowed us to reduce the Average Waiting Time of tasks compared to the FTSA algorithm. This is due to the Passive Dynamic Replication method, that minimizes the waiting time in the local queues, in such a way that each secondary copy is activated in the free VN.

5.3.4. Reliability results

The graph of comparison between reliability with replication and reliability without replication is shown in Fig. 10. Since tasks are replicated and assigned to different resources, the reliability will be improved compared to scheduling without replication. Replication is one of the most important methods for improving reliability, in the proposed straetgy, all tasks are replicated and assigned to several

ressources using passive and dynamic replication. So at least one copy of each task is executed on a free VN.

6. Conclusion

Even though highly distributed environments such as Clouds and Grids are increasingly used for e-Science high performance applications, they still cannot deliver the robustness and reliability needed for widespread acceptance as ubiquitous scientific tools. In this paper, we have chosen an alternative way for reliable workflow scheduling: schedule tasks replicas to free VNs with the objective of meeting a reliability requirement. The DFWS adopts a scheduling strategy based on Passive Dynamic Replication with less resource consumption, which is different from the replication strategies cited in Section 2, and overcomes its drawbacks (resource consumption). For example FTSA strategy assigns multiple copies of each task to different resources, which leads to a large amount of resource overheads, and FTSW strategy maintains repeated copies of a table for the Orphan Jobs at each nodes that executes the jobs of the workflow, which leads to redundant computation. Furthermore, when the number of tasks is very high when compared to the number of resources available, the proposed strategy does not lead to task serialization instead of parallel execution. This is due to the Passive Dynamic Replication strategy that assigns tasks replicas to free VNs. To test and evaluate the performance of our strategy, we have made a simulation. The first experimental results of reliability, makespan and resource consumption are encouraging and showed that the objective of compromise between performance and reliability was achieved. And we have also shown that DFWS is superior to FTSA in terms of resource consumption and makespan. We also point out that the schedules generated by the DFWS strategy can tolerate faults. In the future, we plan to integrate our strategy to the middleware GLOBUS [34], various types of failures are interleaved in the Grid computing environment, such as blocking failures, time-out failures, matchmaking failures and program failures. We plan also to study these kinds of failures. In this work, we assume that the FGCM receives only one workflow from the user, we plan also to apply the strategy for multiple workflows. Finally, it would be interesting, to implement this strategy and test it on real Grid like Grid 5000 or EGEE.

References

[1] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su and D. Zagorodnov, Adaptive computing on the grid using apples, *IEEE Transactions on Parallel and Distributed Systems* **14**(4) (April 2003), 369–382.

[2] A. Kumar, An efficient supergrid protocol for high availability and load balancing, *IEEE Transactions on Computers* **49**(10) (Oct 2000), 1126–1133.

[3] G.B. Berriman, E. Deelman, J.C. Good, J.C. Jacob, D.S. Katz, C. Kesselman, A.C. Laity, T.A. Prince, G. Singh and M.-H. Su, Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand, *Proc SPIE 5493, Optimizing Scientific Return for Astronomy Through Information Technologies*, USA **5493** (2004), 221–232.

[4] G.B. Berriman, J.C. Good, A.C. Laity, A. Bergou, J. Jacob, D.S. Katz, E. Deelman, C. Kesselman, G. Singh, M.-H. Su and R. Williams, Montage: A Grid Enabled Image Mosaic Service for the National Virtual Observatory, *Astronomical Data Analysis Software and Systems (ADASS) XIII, Proceedings of the Conference Held 12–15 October, 2003 in Strasbourg*, France ASP Conference Proceedings, San Francisco: Astronomical Society of the Pacific **314** (2004), 593.

[5] R. Buyya and M. Murshed, Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing, *The Journal of Concurrency and Computation: Practice and Experience (CCPE)* **14** (2002), 1175–1220.

[6] W. Chen, R.F. da Silva, E. Deelman and T. Fahringer, Dynamic and fault-tolerant clustering for scientific workflows, *IEEE Transactions on Cloud Computing* **4**(1) (Jan 2016), 49–62.

[7] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P.J. Maechling, R. Mayani, W. Chen, R.F. da Silva and K. Wenger, Pegasus, a workflow management system for science automation, *Future Generation Computer Systems* **46** (2015), 17–35.
[8] M. Rodríguez-Pascual, A.J. Rubio-Montero, R. Mayo-García, C. Kanellopoulos, O. Prnjat, D. Darriba and D. Posada, A fault tolerant workflow for reproducible research, *Information and Computer Technology (GOCICT), 2014 Annual Global Online Conference*, Louisville, Kentucky (2014), 70–75.
[9] R. Graves and E. Deelman, Cybershake: A physics-based seismic hazard model for southern california, *Pure and Applied Geophysics* **168**(3) (2011), 367–381.
[10] Y. Taho and H. Jin, Dependable grid workflow scheduling based on resource availability, *Journal of Grid Computing* **11**(1) (2013), 47–61.
[11] D. chen, R.S. Chen and T.H. Huang, A heuristic approach to generating file spanning trees for reliability analysis, *Computers and Mathematics with Applications* **34**(10) (1997), 115–131.
[12] M.S. Lin, M.S. Chang, D.J. Chen and K.L. Ku, The distributed program reliability analysis on ring-type topologies, *Computers and Operations Research* **28**(7) (2001), 625–635.
[13] K. Nirmala and A. Tamilaras, Improving fault tolerant resource optimized aware job scheduling for grid computing, *Journal of Computer Science* **10**(5) (2014), 763–773.
[14] D. Meiländer, A. Ploss, F. Glinka and S. Gorlatch, A dynamic resource management system for real-time online applications on clouds, *Euro-Par 2011: Parallel Processing Workshops: CCPI, CGWS, HeteroPar, HiBB, HPCVirt, HPPC, HPSS, MDGS, ProPer, Resilience, UCHPC, VHPC, September 2, 2011, Revised Selected Papers, Part I, Bordeaux*, France (2012), 149–158.
[15] S. Jayadivya, J. Nirmala and M. Saira, Fault tolerant workflow scheduling based on replication and submission of tasks in cloud, *International Journal on Computer Science and Engineering (IJCSE)* **4**(6) (2012), 996–1006.
[16] G. Kandaswamy, A. Mandal and D.A. Reed, Fault tolerance and recovery of scientific workflows on computational grids, *Cluster Computing and the Grid, 2008 CCGRID '08 8th IEEE International Symposium on*, France (May 2008), 777–782.
[17] R. Garg and A.K. Singh, Fault tolerant task scheduling on computational grid using checkpointing under transient faults, *Arabian Journal for Science and Engineering* **39**(12) (2014), 8775–8791.
[18] Y. Zhang, C. Koelbel and K. Kennedy, Combined fault tolerance and scheduling techniques for workflow applications on computational grids, *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, China (2009), 244–251.
[19] J.H. Abawajy, Fault-tolerant scheduling policy for grid computing systems, *Parallel and Distributed Processing Symposium, 2004 Proceedings 18th International*, New Mexico (April 2004), 238.
[20] K. Srikala and S. Ramachandram, Fault tolerant scheduling of workflows in grid computing environment (ftsw), *Communication Technologies (GCCT), 2015 Global Conference on*, India (April 2015), 343–347.
[21] A. Benoit, M. Hakem and Y. Robert, Fault tolerant scheduling of precedence task graphs on heterogeneous platforms, *Parallel and Distributed Processing, 2008 IPDPS 2008 IEEE International Symposium on*, USA (April 2008), 1–8.
[22] H. Topcuoglu, S. Hariri and M.-Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, *IEEE Transactions on Parallel and Distributed Systems* **13**(3) (Mar 2002), 260–274.
[23] R. Prodan and T. Fahringer, Overhead analysis of scientific workflows in grid environments, *IEEE Transactions on Parallel and Distributed Systems* **19**(3) (March 2008), 378–393.
[24] K. Plankensteiner and R. Prodan, Meeting soft deadlines in scientific workflows using resubmission impact, *IEEE Transactions on Parallel and Distributed Systems* **23**(5) (May 2012), 890–901.
[25] K. Plankensteiner, R. Prodan and T. Fahringer, A new fault tolerance heuristic for scientific workflows in highly distributed environments based on resubmission impact, *e-Science, 2009 e-Science '09 Fifth IEEE International Conference on*, Oxford (Dec 2009), 313–320.
[26] J. Yu and R. Buyya, A taxonomy of workflow management systems for grid computing, *Journal of Grid Computing* **3**(3) (2005), 171–200.
[27] S. Gurmeet, S.M. Hui and V. Karan, Workflow task clustering for best effort systems with pegasus, *Proceedings of the 15th ACM Mardi Gras Conference: From Lightweight Mash-ups to Lambda Grids: Understanding the Spectrum of Distributed Computing Requirements, Applications, Tools, Infrastructures, Interoperability, and the Incremental Adoption of Key Capabilities*, USA (2008), 1–9.
[28] F.P. Guimaraes and A.C.M.A. de Melo, User-defined adaptive fault-tolerant execution of workflows in the grid, *Computer and Information Technology (CIT), 2011 IEEE 11th International Conference on*, Cyprus (2011), 356–362.
[29] M. Rebbah, Y. Slimani, A. Benyettou and L. Brounie, Dynamic hierarchical model for fault tolerance in grid computing, *World Applied Programming* **1**(5) (2011), 302–321.
[30] R. Bajaj and D.P. Agrawal, Improving scheduling of tasks in a heterogeneous environment, *IEEE Transactions on Parallel and Distributed Systems* **15**(2) (Feb 2004), 107–118.

[31] D. Poola, S.K. Garg, R. Buyya, Y. Yang and K. Ramamohanarao, Robust scheduling of scientific workflows with deadline and budget constraints in clouds, *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, Canada (May 2014), 858–865.
[32] Y.S. Dai, Y. Pan and X. Zou, A hierarchical modeling and analysis for grid service reliability, *IEEE Transactions on Computers* **56**(5) (May 2007), 681–691.
[33] Y. Zhang, C. Koelbel and K. Kennedy, Relative performance of scheduling algorithms in grid environments, *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, Brazil (May 2007), 521–528.
[34] I. Foster, Globus toolkit version 4: software for service oriented systems, *Journal of Computer Science and Technology* **21**(4) (2006), 513–520.

Authors' Bios

Fatima Kalfadj, born in 1983. Ph.D. candidate in the University of Oran1 Ahmed Ben Bella from Algeria. Her main research interests include Distributed System, Grid Computing, Replication, Fault tolerance, Load Balancing and Task Scheduling.

Belabbas Yagoubi, PhD in Computer Science is a full professor at the University of Oran1 Ahmed Ben Bella (Algeria). He is interested in research in the field of Large-Scale Distributed Systems Including Security, Fault Tolerance, Replication, Load Balancing and Task Scheduling.

Résumé

Les workflows sont généralement composés de milliers de tâches, avec des interdépendances. Exécuter un workflow sur une grille de manière efficace se confronte à un problème d'ordonnancement. Ce problème est d'autant plus difficile lorsqu'il y a plusieurs facteurs à prendre en compte par exemple la tolérance aux fautes. Cependant, le problème d'ordonnancement de workflow est vu comme un problème d'optimisation combinatoire, où il est impossible de trouver la solution globale optimale en utilisant des algorithmes ou des règles simples. Il est bien connu comme un problème NP-complet et dépend de la taille du problème à résoudre. Trouver un algorithme d'ordonnancement pour un workflow afin que la fiabilité du système est optimisée et la contrainte de temps peut être satisfaite. Ce problème peut être résolu dans cette thèse en utilisant une stratégie dynamique d'ordonnancement de workflow tolérante aux pannes nommée DFWS (Dynamic Fault tolerant Workflow Scheduling Policy). Cette stratégie combine l'ordonnancement du workflow avec la réplication des tâches, de sorte que le workflow est exécuté de manière efficace et fiable en utilisant la réplication passive et dynamique des tâches avec moins de consommation de ressources.

Mots clés :

Grille de calcul; Tolérance aux fautes; Consommation de ressources; Réplication; Ordonnancement; Système fiable; Cluster; NP-complet; DFWS; Workflow.