الجمهورية الجزائرية الديمقراطية الشعبية
République Algérienne Démocratique et Populaire
وزارة التعليم العالي والبحث العلمي
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
جامعة مصطفى اسطمبولي معسكر
Université Mustapha Stambouli de Mascara

كلية العلوم الدقيقة
Faculté des Sciences Exactes
Département Informatique

Thèse

Présentée par :

Miloud KHALDI

En vue de l'obtention du diplôme de **Doctorat 3ème Cycle LMD**

Tolérance aux fautes à base de clusterisation dans un environnement de système distribué à large échelle

---

Spécialité : Technologie de l'Information et de la Communication (TIC)

Soutenue le :

Devant le jury composé de :

Président	Mohammed DEBAKLA	MCA	Université de Mascara
Directeur de thèse	Boudjelal MEFTAH	MCA	Université de Mascara
Co-Directeur de thèse	Mohammed REBBAH	MCA	Université de Mascara
Examinateur	Ghalem BELALEM	Professeur	Université d'Oran 1
Examinatrice	Meriem MEDDEBER	MCA	Université de Mascara

Année universitaire 2019-2020

Dédicaces

Je dédie ce modeste travail à :

L'affable, honorable, aimable mère : Tu représentes pour moi le symbole de la bonté par excellence, la source de tendresse et l'exemple du dévouement qui n'a pas cessé de m'encourager de prier pour moi. Ta prière et ta bénédiction m'ont été d'un grand secours pour mener à bien mes études. Aucune dédicace ne serait être assez éloquente pour exprimes ce que tu mérites pour tous les sacrifices que tu n'as cessés de consentir depuis ma naissance, durant mon enfance et même à l'âge adulte. Je te dédie ce travail en témoignage de mon profond amour. Puisse Dieu, le tout puissant, te préserver et t'accorder santé, longue vie et bonheur.

A la mémoire de mon père que je n'oublierai jamais et qui aurait été comblé de bonheur, s'il était encore de ce monde. Que Dieu ait son âme et l'abrite dans son immense paradis.

Mon cher frère, mes chères sœurs, mes neveux et mes nièces : En témoignage de l'attachement, de l'amour et de l'affection que je porte pour vous. Je vous remercie pour votre hospitalité sans égal et votre affection si sincère. Je vous dédie ce travail avec tous mes vœux de bonheur, de santé et de réussite.

A toute ma famille : Je tiens à exprimer ma profonde gratitude pour votre soutien indéfectible.

KHALDI Miloud

Remerciements

Un grand merci à mes directeurs de thèse, le Docteur Boudjelal MEFTAH, merci d'avoir fait confiance en moi et merci de votre disponibilité et de vos encouragements, et le Docteur Mohammed REBBAH, merci d'avoir accepté de m'encadrer pour la troisième fois et d'avoir pris le temps de m'expliquer, de m'apprendre c'est quoi la recherche... J'ai beaucoup appris à tes côtés. Merci de m'avoir fait découvrir les plaisirs du travail bien fait. Je tiens également à te remercier pour ton écoute et pour tous les conseils que tu m'as donnés sur bien des sujets. Tes qualités humaines m'ont profondément touché, Merci pour tout ...

J'exprime également ma profonde gratitude au Docteur Mohammed DEBAKLA, pour l'honneur qu'il m'a fait en présidant mon jury de thèse. Je remercie tout particulièrement le Professeur Ghalem BELALEM et le Docteur Meriem MEDDEBER pour l'honneur qu'ils me font en participant à mon jury de thèse.

Mes vifs remerciements vont aussi aux Professeurs Ahmed YAHIAOUI, Khaled BENMERIEM et Fatima DEBBAT.

Je remercie également mes amis de l'Université de Mascara, Samir SETAOUTI, Rachid KHALLADI, Djamel HAMDADOU, Youcef FEKIR, Omar SMAIL, Chakir MOKHTARI et Brahim KHALDI, ainsi que Leila BIDI. Avec eux j'ai passé de très bons moments pendant le parcours de thèse.

Je tiens aussi à remercier les enseignants qui m'ont aidé et soutenu durant toutes mes études.

Mes derniers remerciements vont à ma famille, et à mes amis et collègues, car sans leurs constants soutiens et leurs encouragements, je n'aurai pas pu mener à bien ce travail.

KHALDI Miloud

ملخص

توفر بيئات الأنظمة الموزعة على نطاق واسع للعملاء قاعدة موارد غير محدودة لحساب البيانات وتخزينها. يمكن تمييز هذه البيئات في شكل شبكة حاسوبية أو الحوسبة السحابية أو أنظمة نظير إلى نظير. موارد هذه الأنظمة عرضة للفشل بسبب الأعطال من جميع الأنواع (الانهيار، الانفصال، البيزنطية الخ). الهدف الرئيسي من هذه الأطروحة هو (أ) اقتراح تقنيات للتسامح مع الخطأ على أساس تجميع الموارد في بيئات الشبكة الحاسوبية و (ب) تجميع مهام نظام سير العمل العلمي في بيئات الحوسبة السحابية. تسمح هذه التقنيات بتعميم الإشكاليات المثارة عن طريق تطوير إثنين من المناهج التجريبية المتسامحة مع الأخطاء وكذا تنفيذها وتقييمها عن طريق المحاكاة. كإسهام أول لهذه الأطروحة، نقترح نموذج للتسامح مع الخطأ FT-GRC الذي يسعى إلى البحث عن البديل الأنسب للعقدة الفاشلة بواسطة تجميع موارد الحوسبة دون أي نسخ متماثل. يستند هذا النموذج على الرسوم البيانية الملونة الديناميكية التي يمكن أن تأخذ في الاعتبار الخصائص الأساسية الثلاث للحوسبة الشبكية، وهي عدم التجانس والديناميكية والقابلية للتطور. تستخدم آلية التسامح مع الخطأ المقترحة خوارزمية تجميع من نوع 1-قفزة تسمى HCC لتحويل الشبكة إلى مجموعة من الكتل المترابطة ومن خلال إدخال دالة تقييم تتيح حساب مستوى أداء كل عقدة في الشبكة وفقًا لخصائصها المادية والمنطقية. وقد أتاح لنا الجمع بين هذه التقنيات تحديد اختيار البدائل على النحو الأمثل من خلال البحث عن بدائل في نفس المجموعة (التسامح مع الخطأ داخل الكتلة) ثم من خلال البحث عن أقرب البدائل (التسامح مع أخطاء بين الكتل). وبالتالي، استخدمنا الترحيل الدوري للمهام من العقدة الفاشلة إلى البدائل المناسبة. في الإسهام الثاني، نقترح نموذجًا جديدًا للتسامح مع الخطأ يسمى FT-HCC لنظام سير العمل العلمي الذي يتكون من مجموعة من المهام المجمعة والتي يتم تنفيذها في بيئة الحوسبة السحابية. قمنا بتطبيق هذا النهج على خمسة مسارات سير عمل علمي واسعة الاستخدام مع نموذج سير عمل مختلف وخصائص حسابية مختلفة. تقليل وقت التنفيذ وكذا تكلفة التنفيذ هو الهدف الرئيسي لنموذج التسامح مع الخطأ المقترح من أجل زيادة أداء سير العمل. لقد أخذنا في الاعتبار الأخطاء الداخلية مثل فشل الحاسب المضيف، بما في ذلك مهام سير العمل والأجهزة الافتراضية (VM). من وجهة نظر التجميع، اقترحنا استراتيجيتين لتجميع المهام قابلتين للتسامح مع الخطأ لتحسين أداء تنفيذ سير العمل: مجموعة حاسمة (CC) تعتمد على المسار الحاسم ومجموعة أفقية (HC) تستند على المستويات الأفقية لسير العمل. تقنية التسامح مع الخطأ المقترحة تعمل على تطبيق النسخ المتماثل النشط لمهام المسار الحاسم، والنسخ المتماثل السلبي للمهام غير الحاسمة. كذلك، يتم استغلال وقت الخمول للأجهزة الافتراضية في النسخ المتماثل السلبي.

الكلمات المفتاحية:

الشبكات الحاسوبية، الحوسبة السحابية، التسامح مع الخطأ، التجميع، مسارات سير العمل العلمي، مستوى الأداء.

Résumé

Les environnements de systèmes distribués à large échelle offrent aux clients un parc de ressources illimitées pour le calcul et le stockage des données. Ces environnements peuvent être structurés sous forme de grille de calcul, de Cloud Computing ou de systèmes pair à pair. Les ressources de tels systèmes sont sujettes à des pannes de toutes nature (crash, déconnexion, byzantine etc.). L'objectif principal de cette thèse est (i) la proposition des techniques de tolérance aux fautes basé sur la clusterisation des ressources dans des environnements de grille de calcul et (ii) une clusterisation des tâches d'un système de workflow scientifique dans des environnements de Cloud Computing. Elles généralisent la problématique soulevée par le développement de deux heuristiques de clustering tolérant aux fautes, leurs implémentations et leurs évaluations par des simulations. Comme première contribution, nous proposons FT-GRC un modèle de tolérance aux fautes qui cherche à trouver le substitut le plus adéquat au nœud défaillant par la clusterisation des ressources de calcul sans aucune réplication. Ce modèle est basé sur des graphes colorés dynamiques qui puissent tenir compte des trois caractéristiques fondamentales des grilles, à savoir l'hétérogénéité, la dynamicité et le passage à l'échelle. Le mécanisme de tolérance aux fautes proposé utilise un algorithme de clustering à 1 saut (*1*-hop) appelé HCC pour transformer la grille en un ensemble de clusters interconnectés et par l'introduction d'une fonction de *scoring* qui permettes de calculer le niveau de performance de chaque nœud de la grille en fonction de leurs caractéristiques physiques et logiques. La combinaison de ces techniques, nous a permis de déterminer le choix des substituts de manière optimale en cherchant les substituts dans le même cluster (tolérance aux fautes local intra-cluster) puis par la recherche des plus proches substituts (tolérance aux fautes inter-cluster). Ainsi, nous avons utilisé une migration périodique des jobs du nœud défaillant vers les substituts appropriés. Dans la deuxième contribution, nous proposons un nouveau modèle de tolérance aux fautes appelée FT-HCC pour un système de workflow scientifique composé d'un ensemble de tâches clusterisées soumises dans un environnement de Cloud Computing. Nous avons appliqué cette approche à cinq workflows scientifiques en temps réel avec un modèle de workflow et des caractéristiques de calcul différents. Réduire le temps d'exécution (*makespan*) et le coût d'exécution est l'objectif principal de notre modèle de tolérance aux fautes afin d'augmenter les performances du workflow. Nous avons considéré les fautes internes tels que les défaillances de l'hôte, y compris les tâches de workflow et les instances VM. Du point de vue clustering, nous avons proposé deux stratégies de clustering de tâches tolérantes aux fautes pour améliorer les performances d'exécution de workflow : un Clustering Critique (CC) basé sur le chemin critique et un Clustering Horizontal (HC) basé sur les niveaux du workflow. La technique de tolérance aux fautes proposée applique une réplication active pour les tâches du chemin critique, et une réplication passive pour les tâches non critiques. Ainsi, nous exploitons le temps d'inactivité des VMs pour la réplication passive.

Mots clés : Grille de calcul, Cloud Computing, Tolérance aux fautes, Clustering, Workflow scientifique, Niveau de performance.

Abstract

Large-scale distributed systems environments provide customers with an unlimited pool of resources for calculation and data storage. These environments can be characterized in the form of Grid Computing, Cloud Computing or peer-to-peer (P2P) systems. The resources of such systems are subject to failures of all kinds (crash, disconnection, byzantine etc.). The main objective of this thesis is (i) the proposal of fault tolerance techniques based on the resource clustering in grid computing environments and (ii) a task clustering of a scientific workflow system in Cloud Computing environments. They generalize the problem raised by the development of two fault-tolerant clustering heuristics, their implementations and their evaluations by simulations. As a first contribution, we propose FT-GRC a fault tolerance model that seeks to find the most suitable substitute for the failed node by the clustering of the grid resources. This model is based on dynamic colored graphs which can take into account the three basic characteristics of grid computing, such as dynamicity, heterogeneity and scalability. The proposed fault tolerance mechanism uses a *1*-hop clustering algorithm called HCC to transform the grid into a set of interconnected clusters and by the introduction of a scoring function that calculates the level of performance of each node of the grid according to their physical and logical characteristics. The combination of these techniques allowed us to determine the choice of substitutes in an optimal way by looking for substitutes in the same cluster (intra-cluster fault tolerance) then by the search for the closest substitutes (inter-cluster fault tolerance). Thus, we used a periodic migration of jobs from the failed node to the appropriate substitutes. In the second contribution, we propose a new fault tolerance model called FT-HCC for a scientific workflow system composed of a set of clustered tasks submitted in a Cloud Computing environment. We applied this approach to five real-time scientific workflows with different workflow models and computational characteristics. Reducing execution time (makespan) and execution cost is the main objective of our fault-tolerance model in order to increase workflow performance. We considered internal faults such as host failures, including workflow tasks and VM instances. From a clustering point of view, we proposed two fault tolerant task clustering strategies to improve workflow execution performance: a Critical Clustering (CC) based on the critical path and a Horizontal Clustering (HC) based on the workflow levels. The proposed fault tolerance technique applies active replication for critical path tasks, and passive replication for non-critical tasks. Thus, we exploit the idle time of VMs for passive replication.

Keywords: Grid Computing, Cloud Computing, Fault tolerance, Clustering, Scientific workflow, Performance level.

Table des matières

Table des figures

Liste des tableaux

Introduction générale

Problématique

L'augmentation constante des besoins en terme de puissance de calcul informatique a toujours été un défi auquel la communauté informatique a été (et est toujours) confrontée. Même si les évolutions technologiques de ces dernières années ont permis de mettre en place des machines de plus en plus puissantes, ces dernières restent très insuffisantes par rapport aux besoins en terme de calcul. Partant de cette constatation, les solutions se sont orientées vers une approche qui consiste à regrouper le plus grand nombre de machines pour fournir une puissance de calcul très élevée. Le calcul distribué, ou système distribué de manière général pourrait résoudre des problèmes scientifiques à grande échelle par l'interconnexion des ressources hétérogènes distribuées géographiquement telles que les dispositifs de stockage, les sources de données et les ressources de calcul utilisées par les utilisateurs du monde entier comme une seule ressource fusionnée. Nous pouvons le définir ainsi comme une collection de sites autonomes connectés à l'aide d'un réseau de communication. Cette définition implique une propriété importante des systèmes distribués, à savoir une distribution transparente pour l'utilisateur et pour les développeurs d'applications. Aujourd'hui, un certain nombre de nouveaux concepts et termes liés au calcul distribué ont fait surface et promettent de fournir des services de calcul évoluant du pair-to-peer (P2P) au grille de calcul et récemment au Cloud Computing.

Les systèmes distribués sont souvent constitués de matériel hétérogène où tout composant peut tomber en panne à tout moment. La défaillance des systèmes distribués à large échelle est et sera une réalité. Les hôtes, les réseaux, les disques et les applications tombent souvent en panne, redémarrent, disparaissent et se comportent de manière inattendue. Par conséquent, comme le travail de calcul est réparti sur plusieurs ressources, la fiabilité globale de l'application diminue. Pour faire face à ce problème, des techniques de tolérance aux fautes ou des mécanismes de gestion des erreurs doivent être mis en place. Assurer la sûreté de fonctionnement permet aux utilisateurs d'avoir un certain niveau de confiance et d'augmenter la complexité des décisions d'allocation des ressources dans ces systèmes distribués à large échelle, hétérogènes et hautement dynamiques. Comme il est très difficile (voir impossible) de concevoir un système qui puisse fonctionner sans présence de fautes matérielles et logicielles, il est donc nécessaire de mettre en place un système de tolérance aux fautes qui puisse offrir un certain nombre de propriétés liées à la sûreté de fonctionnement de ces systèmes. C'est dans ce cadre général que se situe cette thèse par la proposition d'un modèle de tolérance aux fautes basé sur le clustering

des tâches dans des environnements de Cloud Computing et un clustering des ressources dans des environnements de grille de calcul.

Objectifs de la thèse

Nos travaux de recherche portent sur la proposition et la mise en œuvre des techniques de tolérance aux fautes pour les systèmes de calcul distribué à large échelle, qui tiennent compte des caractéristiques intrinsèques des grilles de calcul et de Cloud Computing. A travers ces travaux de recherche, nous essayons d'atteindre les objectifs suivants :

- Notre premier objectif concerne la mise en place d'un mécanisme de tolérance aux fautes basé sur la clusterisation des ressources dans des environnements de grille de calcul, qui puisse tenir compte des trois propriétés de base des grilles, à savoir l'hétérogénéité, la dynamicité et le passage à l'échelle. C'est ainsi que nous avons modélisé toute grille par un Graphe Coloré Dynamique, où les nœuds (ressources) sont représentés par des sommets et les arcs représentent les connexions entre ces nœuds. Grâce à un jeu de couleurs, nous arrivons à modéliser l'hétérogénéité des ressources (couleurs déférentes) selon le niveau de performance de chaque nœud, ce qui permettra par la suite de choisir les substituts les plus appropriées en cas de défaillance en fonction de leurs couleurs. La structure de graphe, nous a permis également de prendre en charge la dynamicité d'une grille, et ce par l'ajout ou la suppression de sommets et d'arcs. Le modèle proposé utilise une technique de clustering de ressources tolérante aux fautes pour déterminer de manière optimale le choix des substituts les plus proches en se basant sur le niveau de performance (identique ou plus fiable) du nœud défaillant par une tolérance aux fautes intra-cluster. Si le résultat n'est pas suffisant, nous complétons l'ensemble des substituts par une tolérance aux fautes inter-cluster. Nous avons utilisé ainsi une migration périodique des jobs du nœud défaillant vers les substituts adéquats.
- Le deuxième objectif de cette thèse, est d'intégrer des techniques de tolérance aux fautes basées sur la cluterisation d'un système de workflows scientifiques composé d'un ensemble de tâches dépendantes soumises (ordonnancées) dans un environnement de Cloud Computing pour assurer la continuité de service et offrir une meilleure qualité de service. Pour cela, nous avons proposé un nouveau modèle de clustering de tâches tolérant aux fautes, appelé FT-HCC. Le temps d'exécution (*makespan*) et le coût d'exécution du workflow sous contrainte de deadline sont considérés dans cette approche afin d'augmenter les performances.

Contributions

Les principales contributions de cette thèse, peuvent se résumer comme suit à travers deux axes :

1. **Modèle de tolérance aux fautes dans les grilles de calcul par la clusterisation des ressources :** A travers une heuristique de tolérance aux fautes basée sur la clusterisation dédié aux grilles de calcul, que nous avons appelée FT-GRC. Cette approche cherche à trouver le substitut le plus adéquat au nœud défaillant par une technique de clusterisation

des ressources de la grille tolérante aux fautes sans réplication des ressources de calcul. Cette technique est basée sur des graphes colorés dynamiques, où l'ensemble des sommets représente les nœuds de la grille et chaque arc représente les communications entre les entités associées aux nœuds. Ainsi, nous transformons toute grille de calcul en un ensemble de clusters interconnectés en utilisant un algorithme de clustering à 1 saut (*1*-hop) appelé HCC (*High Connectivity Clustering*) dans lesquels chaque cluster est géré par un nœud chef (*cluster-head*). En outre, les nœuds membres (*members*) sont responsables de l'exécution des jobs et les nœuds passerelles (*gateways*) plus de leurs rôles des *membres* se servent de relais entre les clusters et comme nœuds clés dans le plus court chemin trouvé entre les clusters. Le mécanisme de tolérance aux fautes proposé utilise une fonction de *scoring* pour déterminer le substitut approprié (identique ou plus fiable) pour chaque nœud défaillant en calculant le niveau de performance de chaque nœud de la grille suivant les valeurs des attributs de leur vecteur d'état (caractéristiques physiques et logiques) ; chaque nœud est classifié ensuite en fonction de leur niveau de performance en trois classes (performant, moins performant et plus performant). Nous colorons les sommets du graphe par trois couleurs de base (Vert, Rouge, Bleu) selon le niveau de performance du nœud correspondant (Vert pour performant, Rouge pour moins performant et Bleu pour plus performant). Ensuite, le clustering est exploité pour déterminer le choix des substituts de manière optimale en cherchant les substituts dans le même cluster (tolérance aux fautes local intra-cluster) puis par la recherche des plus proches substituts (tolérance aux fautes inter-cluster). Nous avons utilisé ainsi une migration périodique des jobs du nœud défaillant vers les substituts appropriés.

2. **Modèle de tolérance aux fautes dans le Cloud Computing par la clusterisation des tâches d'un système de workflows scientifiques :** Dans cet axe, une nouvelle heuristique appelée algorithme FT-HCC s'est proposée pour améliorer le mécanisme d'ordonnancement et de tolérance aux fautes des tâches de workflow clusterisées, et nous les avons appliquées à cinq workflows scientifiques en temps réel (LIGO, Montage, Epigenome, SIPHT et Cybershake) soumises dans un environnement Cloud hautement distribué. Ces applications de workflow ont un modèle de workflow et des caractéristiques de calcul différents. L'objectif principal de notre modèle de tolérance aux fautes est de réduire le makespan total et le coût d'exécution, qui sont utilisées pour augmenter les performances du workflow. Nous avons considéré les fautes internes tels que les défaillances de l'hôte, y compris les tâches de workflow et les instances VM. Pour améliorer l'approche de tolérance aux fautes du point de vue clustering, nous avons proposé deux stratégies de clustering de tâches tolérantes aux fautes pour améliorer les performances d'exécution de workflow dans un environnement défectueux : un Clustering Critique (CC) basé sur le chemin critique et un Clustering Horizontal (HC) basé sur les niveaux du workflow. La technique de tolérance aux fautes proposée applique une réplication active pour les tâches du chemin critique afin de minimiser le makespan, et une réplication passive pour les tâches non critiques. Ainsi, nous exploitons le temps d'inactivité des VMs pour la réplication passive ce qui permet une réduction considérable du coût d'exécution total.

Structure du manuscrit de thèse

L'ensemble des travaux de cette thèse sont synthétisés dans ce manuscrit composé de quatre chapitres, outre une introduction et une conclusion générale avec des perspectives.

Chapitre 1 : Tolérance aux fautes dans les systèmes distribués à large échelle Ce chapitre présente dans une première partie un état de l'art sur les systèmes distribués à large échelle en particulier les systèmes pair-à-pair (P2P), les grilles de calcul et le Cloud Computing, ainsi que leurs concepts de base, leurs objectifs, leurs architectures, leurs caractéristiques, suivis des défis rencontrés. La deuxième partie de ce chapitre est consacrée à la tolérance aux fautes qui couvre la base théorique et les aspects liés à la sûreté de fonctionnement. Nous mettons en évidence les différentes fautes possibles dans les systèmes distribués à large échelle, les moyens pour les détecter. Finalement, nous détaillons les différentes techniques et mécanisme de tolérance aux fautes utilisés dans ces systèmes.

Chapitre 2 : Clustering dans les systèmes distribués à large échelle Dans ce chapitre, nous dressons un état de l'art qui permet de passer en revue les différentes représentations graphiques et modèles de systèmes distribués à large échelle. Nous mettons, en particulier, l'accent sur les notions de base de clustering, les domaines d'application, ainsi que les techniques et les algorithmes proposées dans la littérature pour organiser les nœuds (tâches ou ressources) en clusters dans les systèmes distribués à large échelle.

Chapitre 3 : Tolérance aux fautes dans les grilles de calcul par la clusterisation des ressources Nous avons consacré ce chapitre pour présenter notre mécanisme de tolérance aux fautes basé sur la clusterisation des ressources dans des environnements de grille de calcul. Ce modèle est basé sur des graphes colorés dynamiques capables d'abstraire toutes les caractéristiques fondamentales des grilles de calcul. Deux techniques de tolérance aux fautes sont proposés sur la base de ces graphes. La première technique applique un algorithme de clustering à *1*-saut appelé HCC pour clusteriser les ressorces de la grille de calcul. La deuxième technique utilise une fonction de *scoring* pour calculer le niveau de performance de chaque nœud dans la grille en se basant sur leurs attributs physiques et logiques. La combinaison de ces deux techniques, nous a permis de déterminer le choix des substituts appropriés (identiques ou plus fiables) de manière optimale dans le même cluster et/ou dans les clusters les plus proches. Ce chapitre donne les résultats de base des expérimentations et l'analyse des performances en utilisant un simulateur développé en Java avec la bibliothèque GraphStream, dédiée à l'exploitation de graphes colorés dynamiques.

Chapitre 4 : Tolérance aux fautes pour un système de workflow scientifique dans un environnement de Cloud Computing Ce chapitre décrit en détail notre modèle de tolérance aux fautes proposé basé sur la clusterisation des tâches d'un système de workflow scientifique exécuté dans un environnement de Cloud Computing. Dans ce modèle, nous avons combiné entre deux techniques de clustering et de réplication des tâches pour réduire le makespan total et le coût d'exécution total et améliorer les performances du workflow en présence de fautes. En outre, il présente sa preuve d'exactitude et la comparaison des performances réalisée via l'étude de simulation par l'outil WorkflowSim.

Le manuscrit de thèse se termine par une conclusion générale, qui rappelle à la fois la problématique abordée dans cette thèse, ainsi que nos principales contributions tant sur le plan théorique que sur le plan pratique. Cette présentation est suivie par une liste de quelques perspectives de recherche.

Chapitre 1

Tolérance aux fautes dans les systèmes distribués à large échelle

Sommaire

1.1 Introduction

Le calcul distribué interconnecte des ressources géographiquement distribuées telles que les dispositifs de stockage, les sources de données et tout type de ressources utilisées pour le calcul. Aujourd'hui, un certain nombre de nouveaux concepts et termes liés au calcul distribué ont fait surface et promettent de fournir des services de calcul évoluant du P2P au grille de calcul et maintenant au Cloud Computing.

La défaillance des systèmes distribués à grande échelle restera pour toujours est une réalité. Les hôtes, les réseaux, les disques et les applications tombent souvent en panne, redémarrent, disparaissent et se comportent de manière inattendue. Le soutien au développement d'applications tolérantes aux fautes a été identifié comme l'un des principaux défis techniques à relever pour le déploiement réussi de ces systèmes.

L'objectif principal de ce chapitre est de présenter les principaux concepts liés aux systèmes distribués à large échelle. Nous présentons, dans ce chapitre, les définitions de base des systèmes distribués, nous donnons des exemples de systèmes distribués et leurs caractéristiques, suivis des défis rencontrés.

La deuxième partie de ce chapitre a été consacrée à la tolérance aux fautes qui couvre la base théorique, les aspects et les différents techniques et mécanismes utilisés.

1.2 Systèmes distribués à large échelle

Un système distribué est constitué d'un ensemble de sites reliés entre eux par un réseau de communication. L'histoire des systèmes distribués est étroitement liée à l'évolution des réseaux de communication. Ainsi, l'apparition vers le milieu des années 70 du réseau Ethernet, réseau local à haut débit utilisant un réseau à diffusion, ont marqué une étape importante dans l'émergence des systèmes distribués. Les premiers systèmes distribués, utilisant des réseaux locaux, ont été réalisés en interconnectant plusieurs systèmes homogènes (notamment des systèmes Unix). La plupart de ces systèmes étendent simplement le système de fichiers pour offrir un accès transparent aux fichiers locaux ou distants ; d'autres permettent la création et l'exécution de processus à distance (Morin, 1998). Tanenbaum définit un système distribué comme une "collection d'ordinateurs indépendants qui paraissent aux utilisateurs du système comme un système unique et cohérent" (Tanenbaum, 1999). On trouve deux points essentiels dans cette définition. Le premier est l'usage du mot indépendant ; cela veut dire que, du point de vue architecture, les machines sont capables d'opérer indépendamment. Le deuxième point est que cette séparation est cachée à l'utilisateur laissant apparaître une seule machine "virtuelle". Plusieurs facteurs ont contribué au développement des systèmes distribués : (i) les ordinateurs sont devenus plus petits et meilleur marché ; (ii) les technologies de communication ont progressé au point où il est très facile et peu coûteux de connecter un ensemble d'ordinateurs ; (iii) les débits et la sécurité dans les réseaux ont connu des progrès notoires et (iv) la croissance explosive de l'Internet et du World Wide Web dans le milieu des années 90 a permis d'avoir des systèmes distribués au-delà de leurs domaines d'application traditionnels, tels que l'automatisation industrielle, la défense et les télécommunications, et presque dans tous les

domaines, y compris le commerce électronique, les services financiers, la santé, la gouvernance et le divertissement (Rebbah M. , 2015).

1.2.1 Avantages de la distribution

Les propriétés suivantes rendent les systèmes distribués de plus en plus présents dans l'informatique actuelle et du future (Tanenbaum, 1999):

- **Collaboration et connectivité :** Une motivation importante pour les systèmes distribués est leur capacité à rassembler de larges quantités d'information et des services distribués, tels que les sites de commerce électronique, les encyclopédies. La popularité de la messagerie instantanée et les forums de discussion sur Internet met en évidence une autre motivation pour les systèmes distribués : garder le contact entre personnes, collaborer, etc.
- **Economie :** Les réseaux informatiques qui intègrent les PDA, les ordinateurs portables, les PC et les serveurs offrent souvent un meilleur rapport performance/prix par rapport aux gros ordinateurs centralisés. Par exemple, ils supportent la décentralisation et peuvent partager des périphériques coûteux, comme les serveurs de fichiers de grande capacité et des imprimantes à haute résolution. De même, les composants logiciels et les services peuvent être exécutés sur des sites avec des attributs de qualités de performance réservés jusque-là à certaines catégories d'ordinateurs ou applications.
- **Performance et scalabilité :** Les services applicatifs suscitent, avec le temps, plus d'utilisateurs d'où la nécessité d'augmenter les performances de systèmes distribués pour supporter cette charge de travail. L'augmentation significative des performances peut être obtenue en utilisant la puissance de calcul combinée de nœuds (ou sites) de calcul en réseau.
- **Tolérance aux fautes :** Une des conséquences immédiates de l'informatique distribuée est de tolérer les défaillances d'un système. Les éléments d'un système distribué (nœuds, réseaux, services, etc.) sont susceptibles de tomber en panne suite à des défaillances. Ces défaillances doivent être gérées avec transparence et sans affecter le fonctionnement global du système distribué. D'une manière générale, la mise en œuvre de la tolérance aux fautes nécessite l'utilisation de la réplication à travers les nœuds et les réseaux du système. Elle permet de minimiser les points de défaillance unique, ce qui peut améliorer la fiabilité du système face à des défaillances partielles.

1.2.2 Défis de la distribution

En dépit de l'omniprésence croissante et l'importance des systèmes distribués, les développeurs de logiciels pour les systèmes distribués font face à plusieurs défis (Schmidt, Stal, Rohnert, & Buschmann, 2000):

- **Complexités inhérentes** qui surviennent de défis inhérents au domaine : Par exemple, les composants d'un système distribué résident souvent dans des espaces d'adressage séparés et donc la communication intra-nœuds a besoin de nouveaux mécanismes, politiques et protocoles. De plus, la synchronisation et la coordination est plus compliquée dans un système distribué puisque les composants peuvent être exécutés en parallèle et la communication peut être asynchrone et non-déterministe. Les réseaux qui connectent des composants dans les systèmes distribués introduisent des effets supplémentaires, tel que la latence, l'instabilité, les fautes transitoires, la surcharge, etc.

(Voelter, Kircher, & Uwe, 2004).

- **Complexités accidentelles** qui surviennent de limitations des outils logiciels et des techniques de développement telles que les API's non-portables et les débogueurs distribués non compatibles.
- **Méthodes et techniques inadéquates**, les méthodes d'analyse des logiciels et les techniques de conception populaires sont conçues pour un seul processus. Le développement de systèmes distribués de qualité avec le respect des exigences de performance, tel que vidéo-conférence ou le contrôle du trafic aérien a été laissé aux experts qualités d'architectures des logiciels.

1.3 Modèle client/serveur

Le modèle client/serveur désigne un mode de comportement et de communication entre plusieurs applications qui s'exécutent sur des ordinateurs connectés en réseau. Ce modèle définit un comportement synchrone entre un client et un serveur, dans la mesure où une fois que le client a soumis une requête au serveur, il doit attendre une réponse de ce dernier pour reprendre son exécution (Oluwatosin, 2014).

1.4 Système Pair-à-pair

Le premier système Pair-à-Pair ou Peer-to-Peer (P2P) est apparu en 1999 (Vu, Lupu, & Ooi., 2010). Il s'agissait du logiciel Napster (Saroiu, Gummadi, & Gribble, 2003), logiciel permettant la diffusion de fichiers musicaux (principalement au format mp3). Ce système possédait une structure centralisée, à savoir que les pairs (ou nœuds) du réseau envoient à un serveur central la liste des fichiers qu'ils mettant à disposition des utilisateurs. Ce système est apparu aux yeux du grand public comme une véritable révolution, puisqu'il s'agissait du premier logiciel massivement utilisé et qui substituait le modèle client/serveur. L'apparition de Napster a donc modifié la hiérarchie existante entre les différents ordinateurs connectés, en permettant à chacun de devenir un serveur (Schollmeier, 2001).

1.4.1 Caractéristiques

Les différentes caractéristiques intrinsèques au modèle P2P garantissent aux systèmes un fonctionnement à large échelle (Mario, 2003). Un très grand nombre de pairs peuvent interagir dans le réseau, de manière à permettre le partage d'une grande quantité de ressources. Ainsi, le comportement global du réseau résulte uniquement des interactions locales entre les pairs. Les pairs sont autonomes et volatiles, ce qui nécessite de gérer leurs connections et déconnexions. La caractéristique principale de ce type de systèmes (Mario, 2003) est l'absence d'une infrastructure fixe, ce qui les rend difficile à contrôler, à évaluer dans leur ensemble (Antoniu, Hatcher, Jan, & Noblet, 2005), mais facilite leur passage à l'échelle. La dynamicité des utilisateurs, des nœuds, des routes et des flux échangés (Rowstron & Druschel, 2001), ainsi que l'auto-organisation d'un système sont les mots clés utilisés principalement pour le partage de ressources (Liben-Nowell, Balakrishnan, & Karger, 2002).

1.4.2 Avantages et inconvénients du modèle P2P

Les avantages des systèmes P2P sont nombreux (Shen, Yu, Buford, & Akon, 2009). Ils donnent accès à un grand nombre de ressources et disposent d'une administration transparente. Un système P2P évolue sans machine(s) dédiée(s) pour son administration. Ces systèmes sont conçus de telle sorte qu'aucun des pairs ne soit indispensable au fonctionnement général : le système n'est pas paralysé par la défaillance d'un ou plusieurs pairs (Olivier, 2005). Les qualités de ce système (robustesse, disponibilité, performances, etc.) augmentent avec le nombre d'utilisateurs, et présentent de nombreux avantages (décentralisation, pas de coûts d'infrastructure, etc.). En contrepartie, de par la volatilité des pairs, le système est non fiable. De plus, des problèmes de sécurité peuvent survenir à cause de certains pairs malveillants. Ce système ne peut pas assurer la confidentialité des données échangées et les vitesses de transfert sont aléatoires.

1.5 Grille de calcul

L'augmentation constante des besoins en terme de puissance de calcul informatique a toujours été un défi auquel la communauté informatique est confrontée (Foster I. , Kesselman, Nick, & Tuecke, 2002). Même si les évolutions technologiques de ces dernières années ont permis d'aboutir à la création de machines de plus en plus puissantes, celles-ci ne fournissent pas suffisamment de puissance pour effectuer des calculs d'une complexité très élevée, ou traitant un grand volume de données. Le calcul parallèle ou distribué fournit une solution à ce problème à condition de disposer d'infrastructures matérielles appropriées. Une des solutions consisterait à répartir un calcul complexe sur un ensemble de machines, reliées entre elles par des réseaux rapides. Parmi ces infrastructures, nous pouvons citer les grilles de calcul (Berman, Fox, & Hey, 2003).

1.5.1 Définition

Le concept de grille de calcul étant encore relativement récent, nous pouvons en trouver plusieurs définitions de ce concept que ce soit dans la littérature scientifique ou sur Internet. Nous reprendrons, dans ce chapitre, les deux définitions données par I. Foster et C. Kesselman (Foster & Kesselman, The Grid 2 : Blueprint for a New Computing Infrastructure, 2003), et discuterons brièvement les différentes notions abordées dans celles-ci.

La première définition donnée par I. Foster et C. Kesselman date de 1998, dans le livre *" The Grid : Blueprint for a New Computing Infrastructure "* (Foster & Kesselman, The Grid 2 : Blueprint for a New Computing Infrastructure, 2003) : *" Une grille de calcul est une* ***infrastructure*** *matérielle et logicielle fournissant un accès* ***fiable*** *(dependable),* ***cohérent*** *(consistent),* ***à taux de pénétration élevé*** *(pervasive) et* ***bon marché*** *(inexpensive) à des capacités de traitement et de calcul ".*

C'est une **infrastructure** car une grille devra fournir des ressources (calcul, stockage, etc.) à grande échelle. Cela nécessite une quantité significative de matériels qui constituera les

ressources de la grille et une quantité importante de logiciels pour bien utiliser, contrôler et superviser cet ensemble de ressources.

La nécessité d'un service **fiable** est fondamentale. Les utilisateurs d'une telle infrastructure s'attendent à recevoir un service prédictible, continu et performant.

La cohérence suggère, tout comme dans une grille d'électricité, la présence de services standards, accessibles via des interfaces standards et opérantes selon des paramètres standards.

Un **taux de pénétration élevé** permet de garantir que les services seront facilement accessibles par une large population.

Finalement, l'aspect **bon marché** est très important d'un point de vue viabilité économique.

En 2000, dans l'article *"The Anatomy of the Grid"* (Foster, Kesselman, & Tuecke, 2001), cette définition sera modifiée en y ajoutant des éléments sociaux et des politiques d'accès. Ainsi, les grilles de calcul sont concernées par *"le partage de ressources et la résolution coordonnée de problèmes dans des **organisations virtuelles** dynamiques et multi-institutionnelles"*.

La notion d'Organisation Virtuelle (OV) (*Virtual Organization*) est définie comme un ensemble d'individus et/ou d'institutions qui partagent des ressources et des services et qui sont soumis à des politiques de sécurité spécifiant les autorisations de ce partage. Ces organisations virtuelles peuvent varier fortement en taille, en structure, en but ou en durée. Elles nécessitent des mécanismes de partage très flexibles permettant de gérer notamment l'accès aux ressources, qui doit pouvoir être reconfiguré avec précision. Une ressource appartenant à une certaine OV est disponible pour certaines personnes, sous certaines conditions et à certains instants.

1.5.2 Caractéristiques d'une grille

Une grille de calcul consiste à exploiter pleinement les ressources de l'intégralité d'un parc informatique (serveurs et PC). C'est une forme d'informatique distribuée basée sur le partage dynamique des ressources entre des participants, des organisations et des entreprises dans le but de pouvoir les mutualiser, et faire ainsi exécuter des applications de calcul intensif ou des traitements utilisant de très gros volumes de données. Les grilles de calcul possèdent quatre principales caractéristiques (Baker, Buyya, & Laforenza, 2002):

1. **Existence de plusieurs domaines administratifs :** les ressources sont géographiquement distribuées et appartiennent à différentes organisations, chacune ayant ses propres politiques de gestion et de sécurité. Ainsi, il est indispensable de respecter les politiques de chacune de ces organisations.
2. **Hétérogénéité des ressources :** les ressources dans une grille sont de nature hétérogène en terme de matériels, de logiciels, d'accès, etc.
3. **Passage à l'échelle :** une grille pourra être constituée de quelques dizaines de ressources à des millions voire des dizaines de millions de ressources. Cela pose de nouvelles contraintes sur les applications et les algorithmes de gestion des ressources.

4. **Dynamicité des ressources :** les grilles sont caractérisées par leur aspect dynamique (arrivée de nouveaux membres, départ des membres existants, etc.). Cela pose des contraintes sur les applications telles que l'adaptation au changement dynamique du nombre de ressources, la tolérance aux fautes et aux délais d'allocation, etc.

1.5.3 Architecture d'une grille de calcul

L'architecture d'une grille de calcul est organisée en couches (voir Figure 1.1) (Foster, Kesselman, & Tuecke, 2001). Une couche est une abstraction représentant un ensemble de fonctions de la grille. Chaque couche fait appel aux services de toutes les couches inférieures. La couche Fabrique ou infrastructure matérielle fournit les ressources. Ce sont, d'un point de vue physique, des ressources telles que des processeurs pour le calcul, des unités de stockage ou des ressources réseau. La couche Fabrique est en relation directe avec le matériel pour mettre à la disposition des utilisateurs les ressources partagées. Lorsqu'une demande d'accès à une ressource est formulée, par le biais d'une opération de partage d'un niveau supérieur, des composants logiciels du niveau Fabrique sont invoqués. Le rôle de ces composants est d'agir directement sur les ressources de la grille.

La couche Connectivité implémente les principaux protocoles de communication et d'authentifications nécessaires aux transactions sur un réseau de type grille. Les protocoles de communication permettent l'échange des données à travers les ressources du niveau Fabrique. Ces protocoles d'authentification s'appuient sur les services de communication pour fournir des mécanismes sécurisés de vérification de l'identité des utilisateurs et des ressources.

La couche Ressource ou intergiciel utilise les services des couches Connectivité et Fabrique pour collecter des informations sur les caractéristiques des ressources, les surveiller et les gérer. Elle s'occupe également de l'aspect facturation et fournit les intergiciels nécessaires à la gestion des ressources, la coordination de l'accès, l'ordonnancement des tâches, etc.

La couche Collective ou environnement et outils de programmation regroupe tous les outils et les paradigmes pouvant aider les développeurs à concevoir et à développer des applications pouvant tourner sur une grille. On y trouve plus particulièrement des compilateurs, des librairies, des outils de développement d'applications parallèles ainsi que des interfaces de programmation ou API (découverte et réservation des ressources, des mécanismes de sécurité, stockage, etc.) que les développeurs d'applications pourront utiliser.

Enfin, la couche la plus haute du modèle est la couche Application qui correspond aux applications qui sont de nature variée : projets scientifiques, médicaux, financiers, ingénierie, etc.

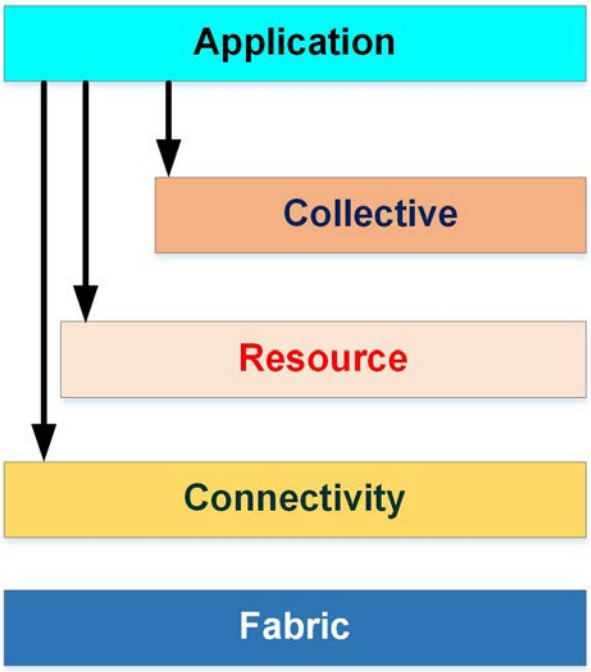


FIGURE 1.1 – Architecture d'une grille de calcul (Foster, Kesselman, & Tuecke, 2001)

1.5.4 Topologies d'une grille de calcul

Il existe trois types de topologies de grille (Ferreira, Berstis, Armstrong, Kendzierski, & al, 2003): Intragrilles (*Intragrids*), Extragrilles (*Extragrids*) et Intergrilles (*Intergrids*) :

- **Intragrille :** C'est la grille la plus simple, car elle est composée d'un ensemble relativement limité de ressources et de services et appartenant à une organisation unique. Les principales caractéristiques d'une telle grille sont l'interconnexion à travers un réseau performant et haut débit, un domaine de sécurité unique et maîtrisé par les administrateurs de l'organisation et un ensemble relativement statique et homogène de ressources.
- **Extragrille :** C'est une agrégation de plusieurs intragrilles. 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.
- **Intergrille :** Elle consiste à agréger les grilles de multiples 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 d'un ensemble très dynamique de ressources.

1.5.5 Cadres d'utilisation de grille

Une grille de calcul offre un éventail de possibilités pour tous les domaines pouvant bénéficier de ses capacités de traitement et de stockage. Nous présenterons, brièvement, dans

ce qui suit cinq grandes classes d'applications pour lesquelles une grille pourrait apporter une nouvelle vision de traitement de ses problèmes (Berman, Fox, & Hey, 2003).

1. **Calcul distribué :** Les applications de calcul distribué sont évidemment d'excellentes candidates pour être utilisées 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. Parmi les principaux défis que doit relever l'architecture d'une grille pour de telles applications :
   - L'ordonnancement à grande échelle des processus.
   - La souplesse des algorithmes et des protocoles qui doivent être capables de gérer un nombre de nœuds pouvant aller de la dizaine à des centaines voire des milliers de machines.
   - La tolérance des algorithmes aux temps de latence inhérents à la taille de la grille.
   - La possibilité d'atteindre et de maintenir un haut niveau de performances dans un système très hétérogène.
2. **Calcul haut débit :** Une grille de calcul sera utilisée pour ordonnancer en parallèle un nombre important de tâches. Comme domaines d'application, nous pouvons citer la recherche de clés cryptographiques, les simulations de molécules, l'analyse du génome, etc.
3. **Calcul à la demande :** Ce type d'applications utilise une grille afin de satisfaire des besoins à court terme en ressources, tels qu'ils ne peuvent être satisfaits en local pour des raisons pratiques ou de rentabilité. Ces ressources peuvent être du temps de calcul, des logiciels, des données ou des capteurs spécialisés.
4. **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. Généralement, ces types de traitement sont également de grands consommateurs de puissance de calcul et de bande passante. Les systèmes de prévisions météorologiques modernes utilisent énormément de données récoltées aux quatre coins du globe (comme des observations satellites par exemple). Le processus complet implique des transferts et des traitements de plusieurs dizaines de Téraoctets de données.
5. **Informatique collaborative :** Le but des applications collaboratives est de permettre les interactions entre les personnes. Elles sont souvent structurées sous forme d'espaces virtuels partagés entre les utilisateurs. La plupart de ces applications permettent de partager des ressources comme des données ou des résultats de simulations.

1.5.6 Outils d'expérimentations des grilles

Les expérimentations sur les grilles sont difficiles à manipuler, car cela nécessite des outils efficaces pour le contrôle et l'observation des paramètres d'une grille (Almond & Snelling, 1999). Il existe deux méthodes pour travailler sur les grilles : soit l'utilisation des outils de simulation, ou des expérimentations sur des grilles physiques. Les simulateurs, bien qu'ils soient intéressants, ne permettent pas de capturer des conditions réelles d'utilisation d'une grille (codes exécutables, etc.) et de reproduire les conditions expérimentales. A l'inverse, les expérimentations réelles ne permettent pas le contrôle des paramètres, les observations fines et

la reproductibilité. La Figure 1.2 montre quelques outils d'expérimentations sur les grilles (Cappello, Primet, Richard, Cérin, & Sens, 2005).

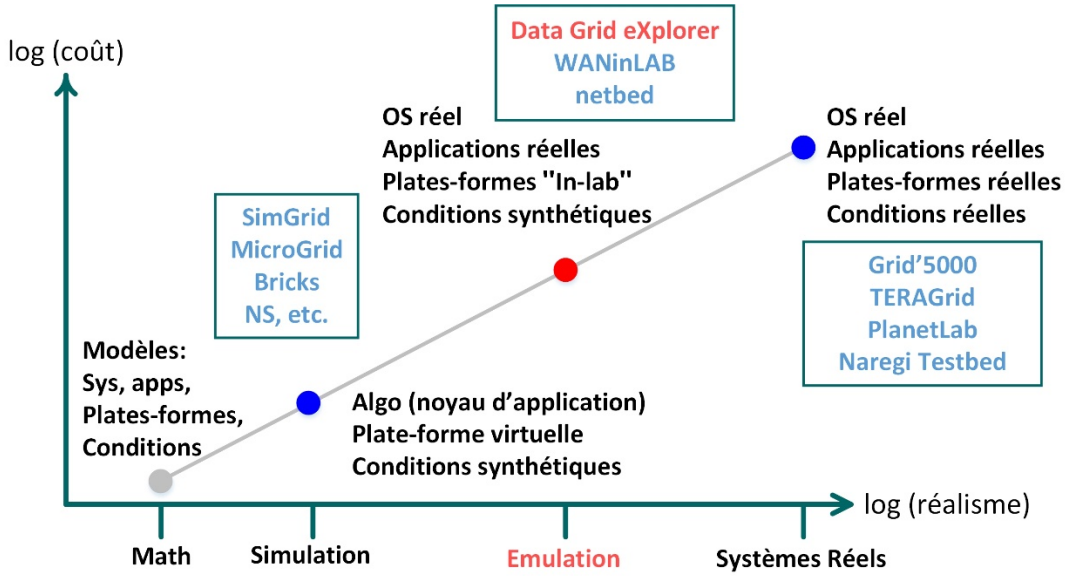


FIGURE 1.2 – Outils d'expérimentations sur les grilles (Cappello, Primet, Richard, Cérin, & Sens, 2005)

1.5.6.1 Middlewares

Les grilles de calcul nécessitent une couche logicielle responsable de la gestion, de la coordination et de l'accès aux différentes ressources (Foster, Kesselman, & Tuecke, 2001). Elles utilisent le concept de middleware ou d'intergiciel pour désigner cette couche. Un middleware est la brique de base regroupant l'ensemble des éléments logiciels pour la mise en œuvre d'une grille. Il comprend notamment les fonctions suivantes (Joshy & Fellenstein, 2003) :

- Le partage et l'allocation des différentes ressources de la grille suivant des critères techniques de performance, mais également des critères économiques et d'éventuelles contraintes utilisateurs.
- L'exécution, l'ordonnancement et l'administration de la grille, intégrant toutes les fonctions de monitoring et de gestion.
- L'ensemble des procédures de sécurisation de la grille, notamment les outils d'authentification des utilisateurs, la gestion des restrictions d'accès, la confidentialité des données et des résultats.
- Les outils collaboratifs permettant aux divers acteurs de travailler ensemble et d'échanger des documents, des données, des logiciels, des résultats, etc., en garantissant leur cohérence au cours de l'ensemble des manipulations.
- Les outils d'évaluation des performances et de mesure de la qualité de services.
- Les outils de développement et les interfaces utilisateurs pour le déploiement des applications.

Ces middlewares s'appuient sur des protocoles standards de l'Internet tels que FTP (*File Transfer Protocol*), LDAP (*Light Directory Access Protocol*), http (*HyperText Transfert*

Protocol). Parmi les middlewares les plus utilisés actuellement, il faut citer LEGION (Grimshaw, Wulf, & The Legion Team, 1997), UNICORE (Almond & Snelling, 1999) et GLOBUS (Foster & Kesselman, Globus: a Metacomputing Infrastructure Toolkit , 1997).

1.5.6.2 Simulateurs de grille

A défaut de disposer de grilles réelles, il existe différents outils de simulation des grilles dont les plus importants sont :

SimGrid : SimGrid (Casanova, 2001) est un logiciel développé par Henri Casanova et le groupe de recherche AppLe de l'université de Californie à San Diego. C'est un simulateur modulaire écrit en C. Il a été spécifiquement conçu pour permettre l'étude du comportement d'applications distribuées sur des plates-formes réalistes (de la grille au réseau de stations de travail).

GridSim : GridSim (Buyya & Murshed, 2002) est un simulateur de grille, développé par Rajkumar Buyya et Manzur Murshed. Il a été utilisé pour simuler les algorithmes d'ordonnancement simples ou multiples dans les systèmes distribués tels que des clusters de grilles. Il fournit un service complet pour la simulation de différentes classes de ressources hétérogènes, utilisateurs, applications et ordonnanceurs.

EDGSim : EDGSim (Crosby, 2003) est une simulation du flux de jobs, de données physiques et de l'information autour d'une grille informatique.

OptorSim : OptorSim (Bell, et al., 2003) est un simulateur de grille de données (*Data Grid*) développé en Java. Il a été conçu pour étudier l'efficacité des algorithmes d'optimisation de répliques dans une grille de données.

GangSim : GangSim (Dumitrescu & Foster, 2005) est une amélioration de l'outil du Ganglia Monitoring Toolkit (Massie, Chun, & Culler, 2004) pour les organisations virtuelles. Il simule un environnement qui comprend effectivement un très grand nombre de ressources, où des centaines d'établissements et des milliers d'individus contrôlent collectivement des dizaines ou des centaines de milliers d'ordinateurs et systèmes associés de stockage.

1.6 Cloud Computing

En 1960, John McCarthy envisageait (voir Figure 1.3) que le calcul serait un jour fourni en tant qu'utilité (Rimal, Choi, & Lumb, 2009). Le Cloud Computing est la réalisation de cette vision. Avec des technologies telles que les services web, l'architecture orientée services, le web 2.0 et la virtualisation matérielle devenant populaires et largement acceptées, ils ont ouvert la voie aux environnements de Cloud Computing.

Le Cloud Computing (informatique dans le nuage) a été reconnu comme un terme générique pour décrire une catégorie de services informatiques à la demande initialement offerts par des fournisseurs commerciaux, tels qu'Amazon, Google et Microsoft. Il désigne un modèle sur lequel une infrastructure informatique est considérée comme un "Cloud", à partir duquel les

entreprises et les particuliers accèdent à des applications à la demande de n'importe où dans le monde (Buyya, Yeo, Venugopal, Broberg, & Brandic, 2009).

Le principe de base de ce modèle est d'offrir le calcul, le stockage et les logiciels " en tant que service ".

1.6.1 Définition

De nombreux praticiens dans les sphères commerciales et académiques ont tenté de définir exactement ce qu'est le "Cloud Computing" et quelles sont les caractéristiques uniques qu'il présente. Buyya et al (Buyya, Yeo, Venugopal, Broberg, & Brandic, 2009) l'ont défini comme suit : Le "Cloud" est un système de calcul parallèle et distribué constitué d'un ensemble de calculateurs interconnectés et virtualisés qui sont dynamiquement approvisionnés et présentés comme une ou plusieurs ressources de calcul unifiées sur la base d'accords de niveau de service (*Service Level Agreement*) établis par négociation entre le fournisseur de services et les consommateurs".

Vaquero et al. (Vaquero, Rodero-Merino, Caceres, & Lindner, 2009) ont déclaré : " Les Clouds constituent un grand pool de ressources virtualisées facilement utilisables et accessibles (tel que le matériel, les plates-formes de développement et/ou les services). Ces ressources peuvent être reconfigurées dynamiquement pour s'adapter à une charge variable (passage à l'échelle), ce qui permet également une utilisation optimale des ressources. Ce pool de ressources est généralement exploité par un modèle de paiement à l'utilisation (*pay-per-use*) dans lequel les garanties sont offertes par le fournisseur d'infrastructures au moyen d'accords de niveau de service personnalisés (*Service Level Agreements*) ".

Un rapport de l'Université de Californie à Berkeley (Armbrust, Fox, Griffith, Joseph, & Katz, 2009) a résumé les principales caractéristiques du Cloud Computing comme suit " (1) l'illusion de ressources informatiques infinies ; (2) l'élimination d'un engagement préalable des utilisateurs du Cloud ; et (3) la possibilité de payer pour l'utilisation ... au besoin ... ".

L'Institut National des Normes et de la Technologie (National Institute of Standards and Technology NIST) (Mell & Grance, 2011) caractérise le Cloud Computing comme "... un modèle de paiement à l'utilisation (*pay-per-use*) pour permettre un accès pratique et à la demande à un pool partagé de ressources informatiques configurables (par exemple, réseaux, serveurs, stockage, applications, services) qui peuvent être rapidement approvisionnées et libérées avec un minimum d'effort de gestion ou d'interaction avec le fournisseur de services".

Dans une définition plus générique, Armbrust et al. (Armbrust, Fox, Griffith, Joseph, & Katz, 2009) définissent le Cloud comme "le matériel et le logiciel d'un centre de données qui fournissent des services". De même, Sotomayor et al. (Sotomayor, Montero, Llorente, & Foster, 2009) soulignent que le terme "Cloud" est plus souvent utilisé pour désigner l'infrastructure informatique déployée sur un centre de données comme fournisseur d'infrastructure en tant que service.

Bien qu'il existe plusieurs autres définitions, il semble y avoir des caractéristiques communes entre les plus notables énumérées ci-dessus, qu'un Cloud devrait avoir : (i) paiement à l'utilisation (pas d'engagement permanent, prix d'utilité) ; (ii) capacité élastique et illusion de ressources infinies ; (iii) interface libre-service ; et (iv) ressources abstraites ou virtualisées.

En plus de calcul brute et du stockage, les fournisseurs de Cloud Computing offrent généralement une large gamme de services logiciels. Ils comprennent également des API et des outils de développement qui permettent aux développeurs de créer des applications évolutives en toute transparence sur leurs services. L'objectif ultime est de permettre aux clients d'exécuter leur infrastructure informatique quotidienne "dans le Cloud".

Un grand battage médiatique a entouré le domaine du Cloud Computing à ses débuts, souvent considéré comme le changement le plus important dans le monde des technologies de l'information depuis l'avènement d'Internet (Carr, 2009). Au milieu d'un tel battage, une grande confusion règne lorsqu'on essaie de définir ce qu'est le Cloud Computing et quelles infrastructures informatiques peuvent être qualifiées de "Clouds".

En effet, le rêve, de longue date, de fournir l'informatique en tant que service public a été réalisé avec l'avènement du Cloud Computing (Armbrust, Fox, Griffith, Joseph, & Katz, 2009). Cependant, au fil des années, plusieurs technologies ont mûri et ont contribué de manière significative à rendre le Cloud Computing viable. Dans cette section, nous retraçons les racines du Cloud Computing en examinant les principaux progrès technologiques qui ont contribué de manière significative à l'avènement de ce domaine émergent. Nous expliquons également les concepts et les développements en catégorisant et en comparant les efforts de recherche et de développement les plus pertinents dans le Cloud Computing, en particulier les Clouds publics, les outils de gestion et les frameworks de développement. Les réalisations pratiques les plus importantes du Cloud Computing sont listées, en mettant l'accent sur les aspects architecturaux et les caractéristiques techniques innovantes.

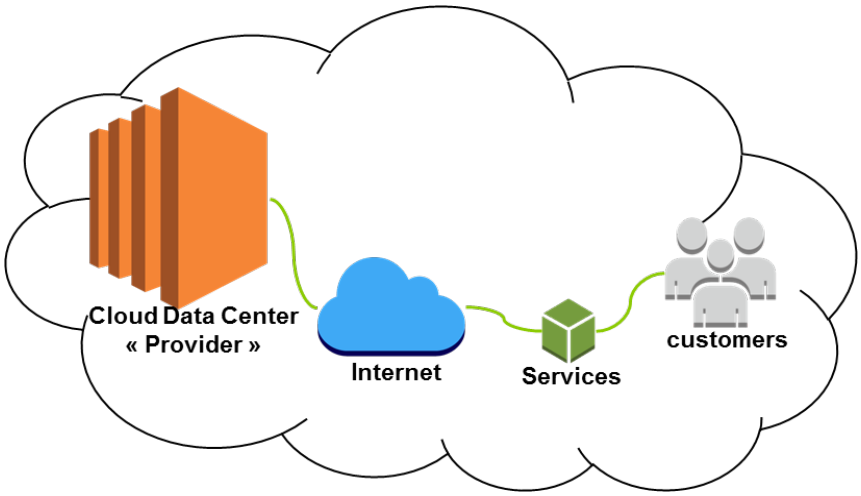


FIGURE 1.3 – Vision du Cloud Computing par John McCarthy (Rimal, Choi, & Lumb, 2009)

1.6.2 Les couches et les types de Clouds

Les services de Cloud Computing sont divisés en trois classes, selon le niveau d'abstraction de la capacité fournie et le modèle de service des fournisseurs, à savoir (1) Infrastructure en tant que service, (2) Plate-forme en tant que service, et (3) Logiciel en tant que service (Mell & Grance, 2011). La Figure 1.4 illustre l'organisation en couches de la pile cloud, de l'infrastructure physique aux applications.

Ces niveaux d'abstraction peuvent également être considérés comme une architecture en couches où les services d'une couche supérieure peuvent être composés à partir des services de la couche sous-jacente (Youseff, Butrico, & Da Silva, 2008). Le modèle de référence de Buyya et al. (Buyya, Pandey, & Vecchiola, 2009) explique le rôle de chaque couche dans une architecture intégrée. Un middleware de noyau gère les ressources physiques et les machines virtuelles (VMs) déployées au-dessus ; en outre, il fournit les fonctionnalités requises (par exemple, la comptabilité et la facturation) pour offrir des services de paiement à l'achat de locataires multiples (multi-tenant *pay-as-you-go*). Les environnements de développement Cloud sont construits sur des services d'infrastructure pour offrir des capacités de développement et de déploiement d'applications. A ce niveau, divers modèles de programmation, bibliothèques, API et éditeurs permettent la création d'une gamme d'applications commerciales, Web et scientifiques. Une fois déployées dans le Cloud, ces applications peuvent être consommées par les utilisateurs finaux.

1.6.2.1 Infrastructure en tant que service (Infrastructure as a Service IaaS)

L'offre à la demande de ressources virtualisées (calcul, stockage et communication) est connue sous le nom de "*Infrastructure as a Service*" (IaaS) (Sotomayor, Montero, Llorente, & Foster, 2009). *Une infrastructure Cloud* permet le provisionnement à la demande de serveurs exécutant plusieurs choix de systèmes d'exploitation et une pile logicielle personnalisée. Les services d'infrastructure sont considérés comme la couche inférieure des systèmes de Cloud Computing (Nurmi, et al., 2009).

Amazon Web Services (AWS) offre principalement IaaS, qui dans le cas de son service EC2 (*Elastic Compute Cloud*), propose des machines virtuelles (VMs) avec une pile logicielle qui peut être personnalisée de la même manière qu'un serveur physique ordinaire. Les utilisateurs ont le privilège d'effectuer de nombreuses activités sur le serveur, telles que : le démarrer et l'arrêter, le personnaliser en installant des packages logiciels, y attacher des disques virtuels et configurer des autorisations d'accès et des règles de pare-feu.

1.6.2.2 Plateforme en tant que service (Platform as a Service PaaS)

En plus des Clouds orientés infrastructure qui fournissent des services informatiques et de stockage bruts, une autre approche consiste à offrir un niveau d'abstraction plus élevé pour rendre un Cloud facilement programmable, connu sous le nom de "*Platform as a Service*"

(PaaS). *Une plateforme Cloud* offre un environnement dans lequel les développeurs créent et déploient des applications et n'ont pas nécessairement besoin de savoir combien de processeurs ou combien de mémoire ces applications utiliseront. En outre, de multiples modèles de programmation et services spécialisés (par exemple, l'accès aux données, l'authentification et les paiements) sont offerts comme éléments constitutifs de nouvelles applications.

Google AppEngine, un exemple de plateforme en tant que service, offre un environnement évolutif pour le développement et l'hébergement d'applications Web, qui devraient être écrites dans des langages de programmation spécifiques tels que Python ou Java. Les blocs de construction comprennent un cache d'objets en mémoire, un service de messagerie, un service de messagerie instantanée, un service de manipulation d'images et l'intégration avec le service d'authentification des comptes Google.

1.6.2.3 Logiciel en tant que service (Software as a Service SaaS)

Les applications résident au sommet de la pile Cloud. Les services fournis par cette couche sont accessibles aux utilisateurs finaux via des portails Web. Par conséquent, les consommateurs passent de plus en plus de programmes informatiques installés localement à des services logiciels en ligne qui offrent les mêmes fonctionnalités. Les applications bureautiques traditionnelles telles que le traitement de texte et les feuilles de calcul sont désormais accessibles en tant que service sur le Web. Ce modèle de fourniture d'applications, connu sous le nom de "*Software as a Service*" (SaaS), allège le fardeau de la maintenance logicielle pour les clients et simplifie le développement et les tests pour les fournisseurs (voir Figure 1.4) (Youseff, Butrico, & Da Silva, 2008; Hayes, 2008).

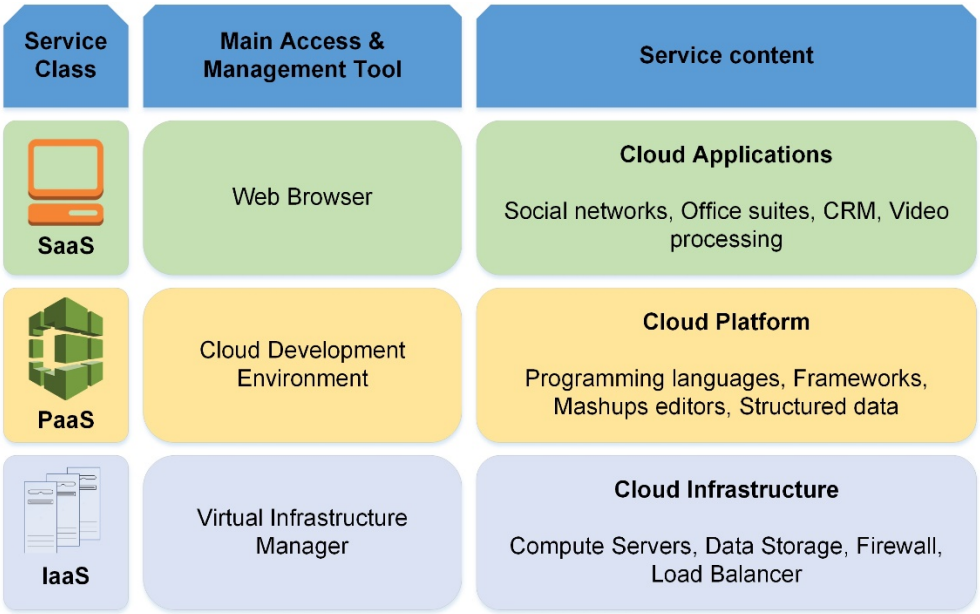


FIGURE 1.4 – Architecture en couches du Cloud

1.6.3 Modèles de déploiement

Bien que le Cloud Computing ait émergé principalement de l'apparition des utilités informatiques publics, d'autres modèles de déploiement, avec des variations dans la localisation physique et la distribution, ont été adoptés. En ce sens, quelle que soit sa classe de service, un Cloud peut être classé comme public, privé, communautaire ou hybride (Mell & Grance, 2011) en fonction du modèle de déploiement, comme le montre la Figure 1.5.

Armbrust et al. (Armbrust, Fox, Griffith, Joseph, & Katz, 2009) proposent des définitions pour le Cloud public comme "Cloud mis à la disposition du grand public selon un système de paiement à l'utilisation (*pay-as-you-go*)" et le Cloud privé en tant que "centre de données interne d'une entreprise ou d'une autre organisation, non mis à la disposition du grand public".

Dans la plupart des cas, l'établissement d'un Cloud privé signifie la restructuration d'une infrastructure existante par l'ajout de la virtualisation et d'interfaces de type Cloud. Cela permet aux utilisateurs d'interagir avec le centre de données local tout en bénéficiant des mêmes avantages des Clouds publics, notamment l'interface en libre-service, l'accès privilégié aux serveurs virtuels et la facturation à l'utilisation.

Un *Cloud communautaire* est "partagé par plusieurs organisations et soutient une communauté spécifique qui a des préoccupations communes (par exemple, la mission, les exigences de sécurité, la politique et les considérations de conformité) (Mell & Grance, 2011)".

Un *Cloud hybride* prend forme lorsqu'un Cloud privé est complété par la capacité de calcul des Clouds publics (Sotomayor, Montero, Llorente, & Foster, 2009). L'approche consistant à louer temporairement de la capacité pour faire face à des pics de charge est connue sous le nom "Cloud-bursting" (explosion des Clouds) (Jaeger, Lin, Grimes, & Simmons, 2009).

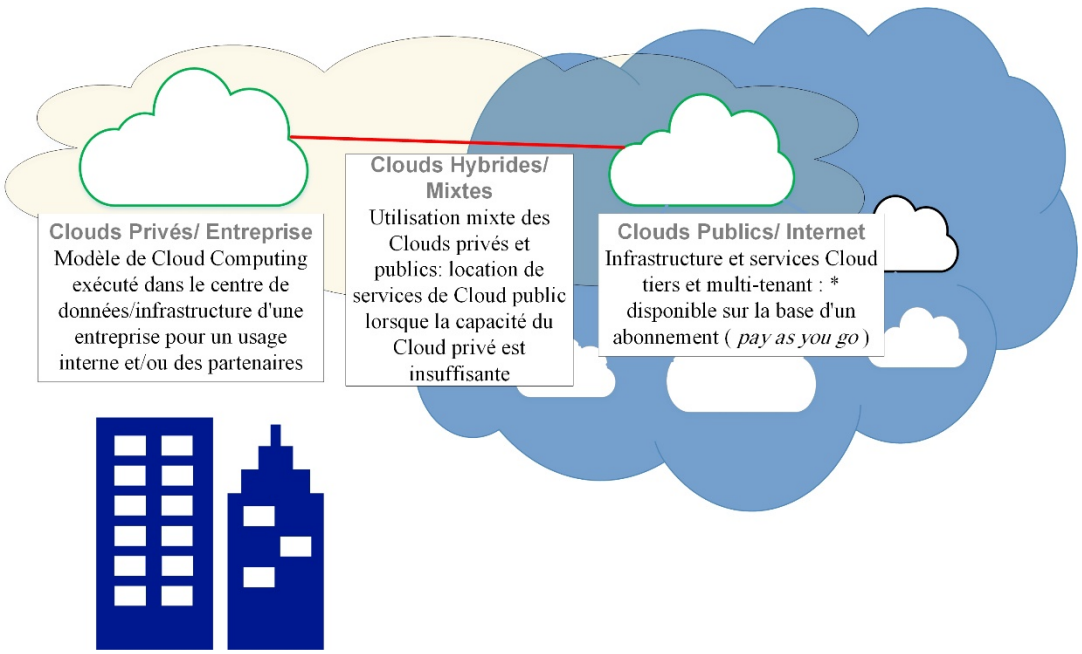


FIGURE 1.5 – Modèles de déploiement du Cloud

1.6.4 Outils de développement Cloud

Les ressources d'infrastructure sont offertes par des Clouds publics et privés. Les Clouds publics sont offerts par de nombreux fournisseurs comme Amazon AWS, Google Compute Engine, Microsoft Azure, IBM Cloud et bien d'autres. Des Clouds privés pourraient être construits à l'aide d'Openstack[1], eucalyptus[2] et VMware[3] pour n'en nommer que quelques-uns. Les fournisseurs de Cloud offrent de nombreuses solutions de stockage qui peuvent être utilisées par les systèmes de gestion de workflow (*Workflow Management Systems*). Certaines solutions de stockage proposées sont par Amazon S3, BigTable de Google et Microsoft Azure Storage. Oracle offre également une base de données basée sur le Cloud comme un service pour les entreprises.

1.6.5 Fonctionnalités souhaitées d'un Cloud

Certaines caractéristiques du Cloud sont essentielles pour permettre des services qui représentent vraiment le modèle de Cloud Computing et répondent aux attentes des consommateurs, dont les offres Cloud doivent être (i) en libre-service, (ii) mesurées et facturées par utilisation, (iii) élastiques, et (iv) personnalisable.

1.6.5.1 Libre-service (*Self-Service*) et à la demande (*On demande*)

Les consommateurs de services de Cloud Computing s'attendent à un accès aux ressources à la demande et quasi instantané. Pour répondre à cette attente, les Clouds doivent permettre un accès en libre-service (self-service) afin que les clients puissent demander, personnaliser, payer et utiliser les services sans l'intervention d'opérateurs humains (Mell & Grance, 2011).

1.6.5.2 Mesure et facturation à l'utilisation (*Pay-as-you-go*)

Le Cloud Computing élimine l'engagement initial des utilisateurs, leur permettant de demander et d'utiliser uniquement la quantité nécessaire. Les services doivent être tarifés sur une base à court terme (par exemple, à l'heure), ce qui permet aux utilisateurs de libérer (et de ne pas payer) les ressources dès qu'elles ne sont pas nécessaires (Armbrust, Fox, Griffith, Joseph, & Katz, 2009). Pour ces raisons, les Clouds doivent implémenter des fonctionnalités permettant de négocier efficacement des services tels que la tarification, la comptabilité et la facturation (Buyya, Yeo, Venugopal, Broberg, & Brandic, 2009). Des mesures doivent être effectuées en conséquence pour différents types de services (par exemple, stockage, traitement et bande passante) et l'utilisation doit être rapidement signalée, offrant ainsi une plus grande transparence (Mell & Grance, 2011).

[1] https://www.openstack.org/

[2] https://www.eucalyptus.cloud/

[3] https://www.vmware.com/

1.6.5.3 Modèles de tarification

Les fournisseurs de Cloud ont différents modèles de tarification. Comme indiqué précédemment, différents fournisseurs de Cloud facturent différemment et un seul fournisseur de Cloud peut provisionner la même ressource via plusieurs modèles de tarification.

Par exemple, les instances Amazon EC2 sont provisionnées de trois manières principales : 1) instance à la demande (*on demande*), où l'utilisateur paie par heure. 2) instances ponctuelles (*spot*) : lorsqu'un utilisateur soumissionne pour l'instance et si le prix d'enchère (appel d'offre) est supérieur au prix spot, les instances sont louées à l'utilisateur. Et lorsque le prix d'enchère (appel d'offre) tombe en dessous du prix spot, l'instance est terminée. 3) instance réservée, ici, l'utilisateur paie un prix initial et réserve l'instance pour une période de temps. En outre, lorsqu'ils utilisent effectivement l'instance, ils paient un prix nominal supplémentaire pour celle-ci.

1.6.5.4 Élasticité

Le Cloud Computing donne l'illusion de ressources de calcul infinies disponibles à la demande (Armbrust, Fox, Griffith, Joseph, & Katz, 2009). Par conséquent, les utilisateurs s'attendent à ce que les Clouds fournissent rapidement toutes les ressources demandées et à tout moment. En particulier, il est prévu que les ressources supplémentaires puissent être (a) fournies, éventuellement automatiquement, lorsque la charge d'une application augmente et (b) libérées lorsque la charge diminue (augmentation et diminution) (Mell & Grance, 2011).

1.6.5.5 Configuration dynamique

Les ressources Cloud sont fournies via le Web par des interfaces graphiques et des APIs faciles à gérer. Elles permettent aux fournisseurs de Cloud de fournir des services que les utilisateurs peuvent configurer dynamiquement et gérer de manière transparente.

1.6.5.6 Personnalisation

Dans un Cloud à locataires multiples, il existe souvent une grande disparité entre les besoins des utilisateurs. Ainsi, les ressources louées à partir du Cloud doivent être hautement personnalisables. Dans le cas des services d'infrastructure, la personnalisation signifie permettre aux utilisateurs de déployer des appliances virtuelles spécialisées et de bénéficier d'un accès privilégié (root) aux serveurs virtuels. D'autres classes de services (PaaS et SaaS) offrent moins de flexibilité et ne conviennent pas à l'informatique à usage général (Armbrust, Fox, Griffith, Joseph, & Katz, 2009), mais devraient néanmoins fournir un certain niveau de personnalisation.

1.6.5.7 Service Level Agreement (SLA)

Le Cloud constitue un point d'accès unique pour tous les services disponibles partout dans le monde, sur la base de contrats commerciaux qui garantissent la satisfaction des exigences de QoS des clients selon des contrats de niveau de service (SLA) spécifiques. Le SLA est un contrat négocié et convenu entre un client et un fournisseur de services. En d'autres termes, le fournisseur de services est tenu d'exécuter les demandes de service d'un client dans le cadre des

exigences de qualité de service (QoS) négociées pour un prix donné. Le but de l'utilisation des SLA est de définir une base formelle de performances et de disponibilité que le fournisseur garantit de livrer. Les contrats SLA enregistrent le niveau de service, spécifié par plusieurs attributs ; tels que la disponibilité, la facilité d'entretien, la performance, le fonctionnement, la facturation ou même des pénalités en cas de violation du SLA (Ardagna, Trubianb, & Zhangc, 2007).

1.7 Tolérance aux fautes dans les systèmes distribués à large échelle

La tolérance aux fautes dans les systèmes distribués est un domaine de recherche qui a été et qui reste très largement étudié (Pankaj, 1994). 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.

1.7.1 Sûreté de fonctionnement

La sûreté de fonctionnement (*dependability*) des systèmes informatiques est définie 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. Cette sûreté est basée sur les notions suivantes (Cohen, Goldszmidt, Kelly, Symons, & Chase, 2004) :

Faute (Fault) : c'est toute cause (événement, action, circonstance) pouvant provoquer une erreur (Sonna Momo, 2001). 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 peut être créée de manière intentionnelle ou accidentelle, à cause des phénomènes physiques ou à cause des imperfections humaines. Durant l'exécution du système, la faute reste dormante jusqu'à ce qu'un événement intentionnel ou accidentel provoque son activation (Kalla, 2004).

Erreur (Error) : 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 va donner un résultat incorrect ou imprécis par rapport à celui attendu (Kalla, 2004). Cet état peut rester non détecté longtemps (latence de la faute) mais peut conduire à court ou à long terme à une défaillance (Florin, 1996).

Défaillance (Failure) : elle survient lorsque le service délivré par le système ne correspond plus à sa spécification (Sonna Momo, 2001).

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. Parmi ces propriétés, nous trouvons :

Disponibilité (*Availability*) : Probabilité pour qu'un système soit disponible à un instant *t* (Sonna Momo, 2001).

Fiabilité (*Reliability*) : Probabilité pour qu'un système soit continûment en fonctionnement sur une période donnée (entre 0 et *t*) (Jafar, 2006).

Sûreté (*Safety*) : C'est une propriété qui respecte la non occurrence de défaillance catastrophique (Sonna Momo, 2001), similaire à la fiabilité mais par rapport aux conséquences catastrophiques causées par les fautes (Jafar, 2006).

Sécurité-confidentialité (*Security*) : Cette propriété concerne l'occurrence des accès non autorisés ou l'acquisition non autorisée d'informations (Sonna Momo, 2001). Cette propriété évalue la capacité du système à fonctionner en dépit de fautes intentionnelles et d'intrusions illégales (Jafar, 2006).

Intégrité (*Integrity*) : L'intégrité d'un système définit son aptitude à assurer des altérations approuvées des données (Jafar, 2006).

Maintenabilité (*Maintenability*) : Définit l'aptitude aux réparations et aux évolutions (Sonna Momo, 2001). C'est la probabilité pour qu'un système en panne à l'instant 0 soit réparé à l'instant *t* (Florin, 1996).

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.

Quatre classes de méthodes de traitement de fautes peuvent être distinguées (Cohen, Goldszmidt, Kelly, Symons, & Chase, 2004) :

Prévention des fautes (*Fault prevention*) : cette méthode vise à empêcher l'occurrence ou l'apparition de fautes par le développement des systèmes informatiques de manière à éviter l'introduction de fautes de conception ou de fabrication et à empêcher que des fautes ne surviennent en phase opérationnelle (Kalakech, 2005).

Tolérance aux fautes (*Fault tolerance*) : elle consiste à délivrer un service correct en dépit de l'occurrence de fautes (Sonna Momo, 2001). Le degré de tolérance aux fautes se mesure par la capacité du système à délivrer son service en présence de fautes (Jafar, 2006).

Elimination des fautes (*Fault removal*) : cette méthode consiste à réduire le nombre et la sévérité des fautes dans le but de les éliminer du système (Sonna Momo, 2001).

Prévision des fautes (*Fault forecasting*) : elle consiste à estimer le nombre de fautes (physiques, de conception ou malveillantes) courantes et futures ainsi que leurs conséquences (Sonna Momo, 2001). La sûreté de fonctionnement peut être illustrée par le schéma de la Figure 1.6.

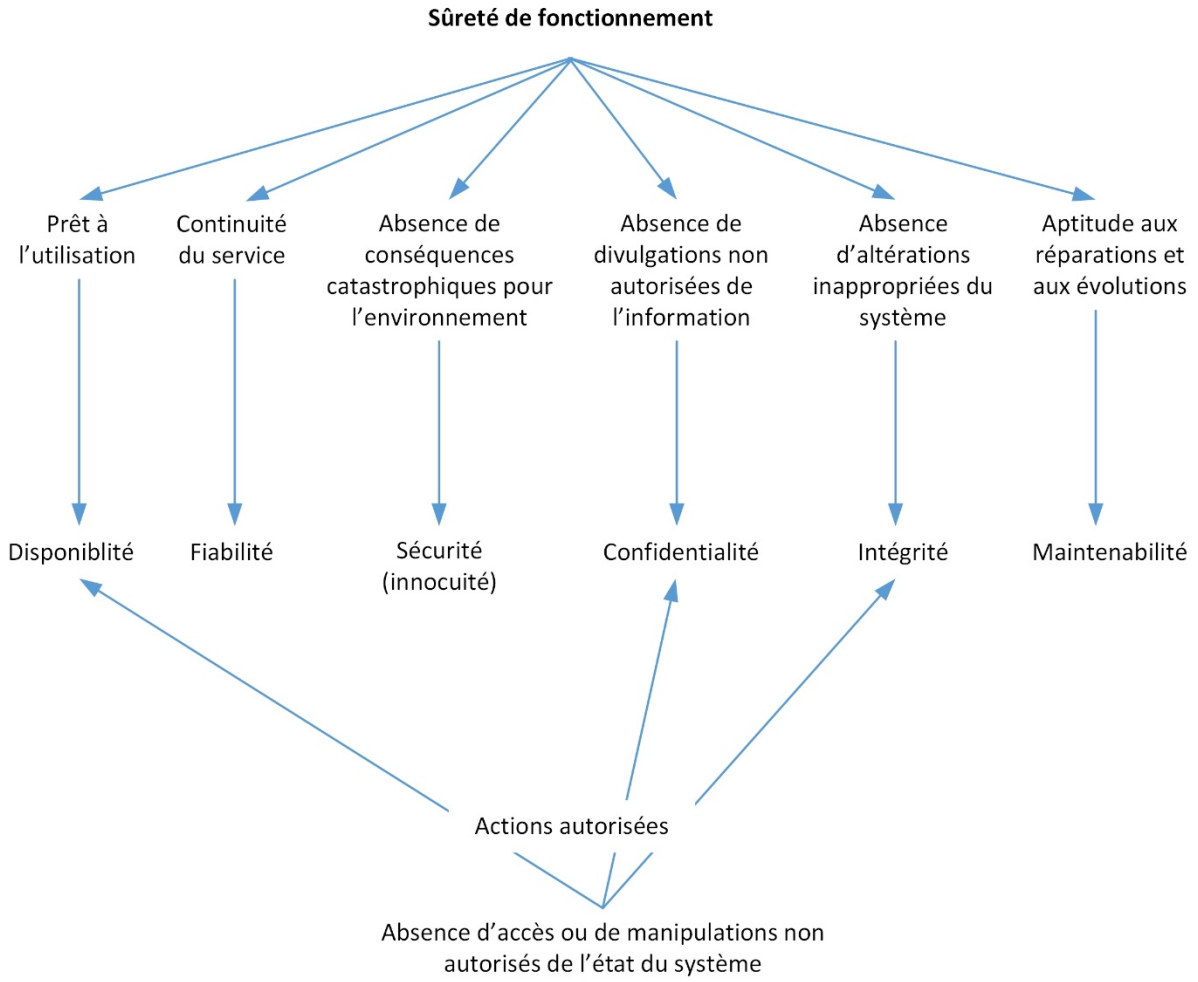


FIGURE 1.6 – Arbre de la sûreté de fonctionnement (Laprie, 2004)

1.7.2 Taxonomie des fautes

Une faute est généralement utilisée pour décrire un défaut au niveau d'abstraction le plus bas (Gärtner, 1999). Une faute peut provoquer une erreur qui, à son tour, peut conduire à une défaillance, c'est-à-dire lorsqu'un système ne s'est pas comporté conformément à sa spécification.

Dans les systèmes distribués, en particulier à cause de ses composants hétérogènes, plusieurs types de défaillances peuvent se produire, ce qui affecte les applications en cours d'exécution. Ces défaillances incluent, sans s'y limiter, les débordements (*overflow*), les dépassements de délai (*timeout*), les ressources manquantes, les défaillances du réseau, du matériel, des logiciels et des bases de données (Dai, Yang, Dongarra, & Zhang, 2009). Les défaillances sont généralement considérés comme étant (Huedo, Montero, & Llorente, 2006):

- Lié au matériel
- Lié au job
- Lié au système
- Lié au réseau.

En (Schroeder & Gibson, 2010), près de dix ans de données réelles sur les défaillances de 22 systèmes de calcul de haute performance (HPC) ont été étudiées et ont conclu que les défaillances matérielles étaient le type de défaillance le plus courant, allant de 30 % à plus de 70 % selon le type de matériel, tandis que de 10 % à 20 % des défaillances étaient des défaillances logicielles.

1.7.3 Types de fautes

Un critère pour classifier les fautes est la nature des fautes. En utilisant ce critère, nous distinguons généralement cinq types de fautes possibles (Avizienis, Laprie, & Randell, 2004). Ainsi, nous pouvons distinguer ces fautes suivant qu'elles surviennent sur l'état ou sur le code d'un élément :

1. **Fautes d'état** : le changement des variables d'un élément peut être la conséquence de perturbations dues à l'environnement (par exemple des ondes électromagnétiques), 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 sensées prendre lors d'une exécution normale du système.
2. **Fautes de code** : le changement arbitraire du code d'un élément résulte la plupart du temps d'une attaque (par exemple le remplacement d'un élément par un adversaire malicieux), mais certains types moins graves peuvent correspondre à des bogues où à une difficulté à supporter la charge d'un élément du système.
3. **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.
4. **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 reprend son activité normale.
5. **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.

Ces différents types de défaillance sont imbriqués les uns dans les autres. En effet, si un système peut tolérer que des composants agissent de manière totalement imprévisible (défaillance byzantine), il peut alors également tolérer un composant qui agit selon sa spécification mais qui omet des parties de son activité (défaillance par omission). De même, la défaillance franche est un cas particulier de défaillance par omission : l'omission concerne alors tout ce qui se passe après la défaillance.

1.7.4 Etapes de la tolérance aux fautes

Plusieurs phases successives, non obligatoirement toutes présentes, font partie d'un processus de tolérance aux fautes (Girault, Lavarenne, Sighireanu, & Sorel, 2001) :

- **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 pas désormais une défaillance.

1.7.5 Techniques de détection des fautes

Les détecteurs de fautes sont un élément central dans les systèmes distribués tolérants aux fautes. La capacité d'un détecteur de fautes pour fonctionner de manière complète et efficace, en présence d'une messagerie non fiable ainsi que des composants sujetés 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 était par Chandra et Toueg (Chandra & Toueg, 1996). Les auteurs ont montré 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 de déterminer si un processus à distance s'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. Ce résultat a lancé une vague de recherches théoriques pour la classification des détecteurs de fautes.

1.7.5.1 Cas des systèmes synchrones/asynchrones

Dans un système synchrone, détecter une défaillance est une issue triviale. Puisque les délais sont liés et connus, les défaillances sont détectées à l'aide d'un délai de garde.

Un système asynchrone est un système pour lequel il n'y a aucune hypothèse temporelle sur les temps de transmission des messages ou sur les temps de calcul des processeurs. Comme dans le cas d'absence de communication en provenance d'un processus pendant une durée *t*, celui-ci est considéré potentiellement défaillant (suspect). Un temporisateur (délai de garde) est amorcé et un message spécial est envoyé à ce processus. En l'absence de réponse avant la fin du temporisateur, le processus est jugé défaillant et la tolérance à cette faute peut commencer. Cette technique simple n'est pas optimale, puisqu'elle a pour inconvénient de distinguer difficilement un processus lent d'un processus mort.

1.7.5.2 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 peut permettre 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 sont nécessaires (chaque message de vie '*Pong*' étant réclamé explicitement par un message '*Ping*') (Monnet, 2006).

1.7.5.3 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 (Monnet, 2006).

1.7.6 Techniques de tolérance aux fautes

La majorité des techniques de tolérance aux fautes sont basées 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 revenir en cas d'une défaillance ou bien une duplication informationnelle (redondance de données, codes, signatures) (Jafar, 2006).

1.7.6.1 Tolérance aux fautes par duplication

Trois approches fondamentales du problème de la tolérance aux fautes, basées sur la duplication (Guerraoui & Schiper, 1996), sont proposées dans la littérature : l'approche masquante à base de la redondance active, l'approche recouvrante à base de la duplication passive et l'approche hybride à base de la duplication passive et active.

Duplication active : La duplication active peut être utilisée de manière efficace pour masquer la faute de plusieurs composants matériels (capteurs, processeurs, média de communication et actionneurs) dans un système distribué. Elle est bien adaptée à toutes les hypothèses de défaillance : silence (latence) sur défaillance, fautes transitoires, fautes temporelles et fautes byzantines (ou quelconques). Par exemple, lorsqu'une défaillance d'un processeur se produit, toutes les tâches exécutées sur ce processeur deviennent inactives, ce qui conduit à une défaillance du système. Afin de tolérer ces fautes, la duplication active permet de masquer k fautes de processeurs en répliquant activement chaque tâche t sur k+1 processeurs distincts. Pour cela, chaque réplique t_i de t doit recevoir ses données d'entrées en k+1 exemplaires. Par exemple, dans la Figure 1.7 b, la tâche *A* (resp. *B*) est répliquée en deux exemplaires, qui sont allouées à deux processeurs distincts *P1* et *P2* (resp. *P2* et *P4*) afin de masquer une faute de processeur. Dans cet exemple, la réplique *B2* de *B* reçoit ses données d'entrée en deux exemplaires (message *m*). De même, la perte d'un message dans un réseau de communication due aux fautes de déconnexion, peut être masquée par la transmission de ce message via

plusieurs routes disjointes. Par exemple, dans la Figure 1.7 c, la communication *A.B* (message *m*) est réalisée sur deux routes disjointes reliant le processeur *P1* à *P4*. Précisément, le processeur *P1*, exécutant l'opération *A*, envoie au processeur *P4*, exécutant l'opération *B*, la communication *A.B* via les deux routes disjointes *P1,P2,P4* et *P1,P3,P4*.

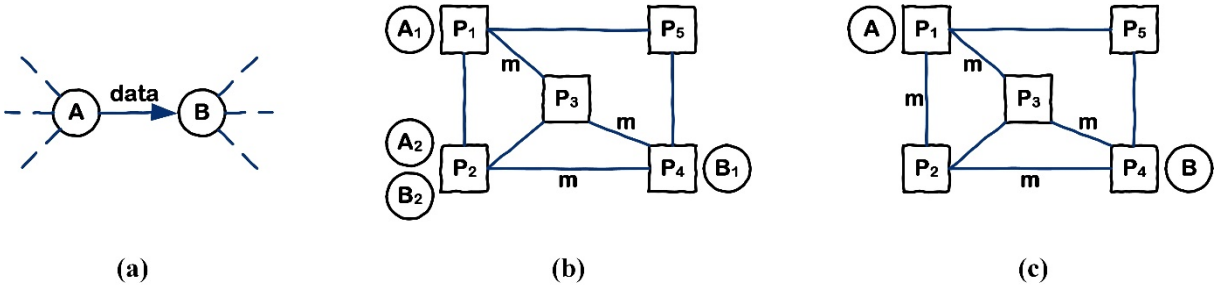


FIGURE 1.7 – Duplication active (Tronel, 2003)

Duplication passive : Comme dans la duplication active, afin de tolérer *k* fautes de processeurs ou de déconnexions, chaque tâche t est répliquée sur *k*+1 processeurs distincts. Cependant, une seule copie t_1, appelée primaire, est exécutée tandis que les autres copies t_i ($i \neq 1$), appelées sauvegardes ou secondaires, surveillent la copie primaire. Si le processeur exécutant la copie primaire devient défaillant, une copie de sauvegarde sera sélectionnée pour remplacer la copie primaire. Par exemple, dans la Figure 1.8 a, la tâche *A* (resp. *B*) est répliquée en deux exemplaires, qui sont alloués à deux processeurs distincts *P1* et *P2* (resp. *P2* et *P4*) afin de tolérer une faute de processeur ou de déconnexion. Dans cet exemple, afin de tolérer une faute de déconnexion, le processeur *P1*, exécutant la copie primaire *A1*, envoie au processeur *P4*, exécutant la copie primaire *B1*, la communication *A.B* via une seule route. Si le processeur *P1* est défaillant, le processeur *P2* détecte la faute, exécute la copie de sauvegarde *A2* et envoie le message *m* à *P4* via une nouvelle route, comme cela est montré dans la Figure 1.8 b. La perte d'un message, dans le cas de la duplication passive des communications, due aux fautes de déconnexion, peut être tolérée par la retransmission de ce message via des routes disjointes dans un réseau. Cette technique de duplication nécessite des mécanismes particuliers de détection de fautes, et le choix du primaire ainsi que son intégration après la réparation.

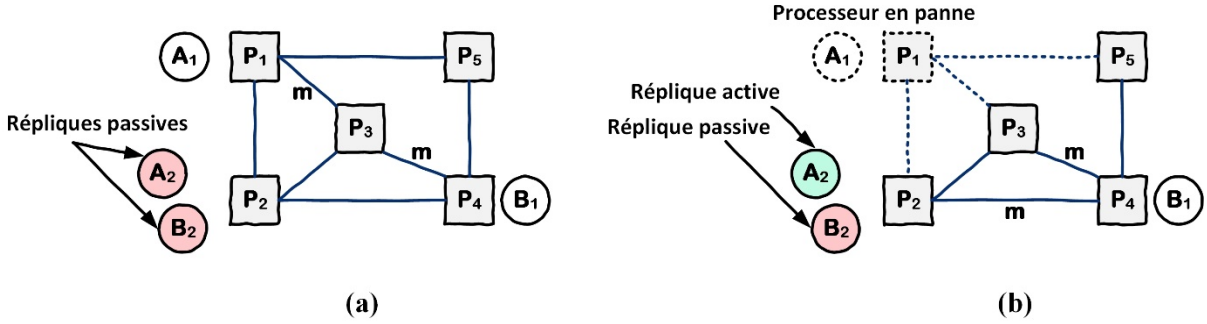


FIGURE 1.8 – Duplication passive

Duplication hybride : La duplication hybride (Felber, Défago, Eugster, & Schiper, 1999) est une combinaison de la duplication active et passive, dans laquelle par exemple, on utilise la duplication active pour les tâches et la duplication passive pour les communications, comme

cela est montré dans la Figure 1.9. Dans cette Figure, les deux tâches *A* et *B* sont répliquées activement en deux exemplaires, tandis que la communication *A.B* ne sera envoyée que par la première réplique *A1* de *A* (message *m*). Une propriété intéressante de la duplication active se situe dans le fait qu'une faute n'augmente pas la latence d'un système temps-réel, ce qui n'est pas le cas dans la duplication passive, où la faute de la réplique primaire peut de manière significative augmenter la latence du système. Cependant, la duplication passive présente l'avantage de réduire la surcharge sur les processeurs et les médias de communication, ce qui permet une meilleure exploitation des ressources matérielles offertes par l'architecture. Les duplications active et passive sont deux techniques complémentaires (Felber, Défago, Eugster, & Schiper, 1999). Ainsi, combiner efficacement ces deux techniques, pour tolérer les fautes des processeurs et/ou les fautes de déconnexion, est intéressant.

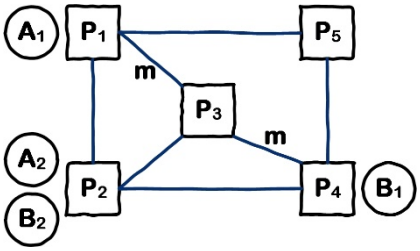


FIGURE 1.9 – Duplication hybride

1.7.6.2 Technique 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* (Elnozahy, Alvisi, Wang, & Johnson, 2002). Ces deux techniques sont basées sur les points de reprise (*checkpoint*) ou sur les fichiers logs. A la suite d'une faute, le rollback-recovery restaure l'état du système associé au plus récent ensemble consistant de points de reprise, qui est appelé ligne de recouvrement (*Recovery Line*) (Randell, 1975). Le roolback basé sur les fichiers logs (ou fichiers de trace) utilise le fait qu'une 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 (Strom & Yemini, 1985). Un tel événement peut être la réception d'un message d'un autre processus ou un événement interne au processus lui-même, comme par exemple les événements probabilistes (Mohsin, 2011). L'envoi d'un message, par contre, n'est pas un événement indéterministe (Chandy & Lamport, 1985). Le *RR* basé sur l'enregistrement suppose que tous les événements indéterministes peuvent être identifiés et que les déterminants correspondants peuvent être enregistrés sur un support fiable de stockage. Durant l'exécution, chaque processus enregistre les déterminants de tous les événements indéterministes qu'il observe sur un support fiable de stockage. Un déterminant est un ensemble d'informations qui permet de ré-exécuter les programmes de telle sorte que les évènements indéterministes soient exécutés à chaque fois de la même façon (leurs effets sur le déroulement du programme est le même). De plus, chaque processus effectue des points de reprise pour réduire le rollback à la suite d'une faute et avant la réexécution. A la suite d'une faute, le processus défaillant est ré-exécuté en utilisant les points de reprise et les déterminants

déjà enregistrés pour se comporter de la même façon que l'exécution défaillante (notamment l'ordre de réception des messages).

1.7.7 Fiabilité

La fiabilité dans le contexte des applications logicielles peut avoir plusieurs significations, en particulier pour les applications fonctionnant dans des systèmes distribués. Souvent, la fiabilité est définie comme la probabilité que le système puisse exécuter une tâche entière avec succès (Wan, Huang, Yang, & Chen, 2011; Pezoa & Hayat, 2012). Une définition similaire, pour les applications s'exécutant dans des environnements distribués, est que la fiabilité est la probabilité qu'une application logicielle remplisse ses fonctions prévues pendant une période de temps spécifiée, et est couramment utilisée pour les applications avec des contraintes de temps. Enfin, la fiabilité peut également être définie comme la probabilité qu'une tâche produise le résultat correct (Brun, Bang, Edwards, & Medvidovic, 2015). Cette dernière est généralement utilisée avec le modèle de faute byzantine.

La fiabilité d'un système dépend fortement de la façon dont le système est utilisé (Immonen & Niemelä, 2008). Pour déterminer la fiabilité d'un système, il faut prendre en compte tous les facteurs qui influent sur la fiabilité (Ahmed & Wu, 2013). Cependant, il n'est pas possible d'inclure tous les facteurs. Dans (Zhang & Pham, 2000), trente-deux (32) facteurs affectant la fiabilité du logiciel sont énumérés, à l'exclusion des facteurs environnementaux tels que les défaillances du matériel et des liaisons. D'autres conditions environnementales affectant la fiabilité incluent la quantité de données transmises, la bande passante disponible et le temps de fonctionnement (Dai, Yang, Dongarra, & Zhang, 2009; Jaeger, Lin, Grimes, & Simmons, 2009).

Pour les applications distribuées, la probabilité de défaillance augmente car elle dépend du nombre de ressources utilisées (Wan, Huang, Yang, & Chen, 2011). La plupart des modèles de fiabilité sont basés sur le temps moyen de défaillance (*Mean-Time-To-Failure* MTTF) ou le temps moyen entre défaillances (*Mean-Time-Between-Failures* MTBF) des ressources (Raghavendra & Makam, 1986). Conventionnellement, le MTTF fait référence aux ressources non réparables, tandis que MTBF fait référence aux objets réparables (Litke, Skoutas, Tserpes, & Varvarigou, 2007).

Définition 7.1. *Le MTTF (Mean-Time-To-Failure) d'un composant est le temps moyen nécessaire à un composant pour tomber en panne, étant donné qu'il était opérationnel au temps zéro*.

Définition 7.2. *Le MTBF (Mean-Time-Between-Failure) pour un composant est le temps moyen entre les défaillances successives pour ce composant*.

Le MTBF peut être calculé comme suit

$$MTBF = \frac{temps\ total}{nombre\ de\ défaillances} \tag{1.1}$$

À partir de *t* et du *MTBF*, le taux de défaillance λ utilisé dans l'équation (1.1) peut être calculé comme suit : $\lambda = t / MTBF$. L'équation (1.1) peut donc être réécrite comme suit

$$P\ (k\ défaillances\ pendant\ le\ temps\ t) = \frac{\left(\frac{t}{MTBF}\right)^{k} . e^{-\left(\frac{t}{MTBF}\right)}}{k!} \qquad (1.2)$$

La probabilité de survie correspond à zéro défaillances, et peut être exprimée comme suit

$$P\ (0\ défaillances\ pendant\ le\ temps\ t) = \frac{\left(\frac{t}{MTBF}\right)^{0} . e^{-\left(\frac{t}{MTBF}\right)}}{k!} = e^{-\left(\frac{t}{MTBF}\right)} \qquad (1.3)$$

1.8 Conclusion

Des technologies telles que P2P, les grilles de calcul et maintenant le Cloud Computing, ont toutes pour but de permettre l'accès à de grandes quantités de puissance de calcul en agrégeant les ressources et en offrant une vue unique du système.

Ces systèmes distribués sont souvent constitués de matériel hétérogène et l'un des composants, souvent très nombreux, peut tomber en panne à tout moment. Par conséquent, comme le travail de calcul est réparti sur plusieurs ressources, la fiabilité globale de l'application diminue. Pour faire face à ce problème, une conception tolérante aux faute ou un mécanisme de gestion des erreurs doit être mis en place. Assurer la fiabilité augmente la complexité des décisions d'allocation des ressources et des mécanismes de tolérance aux fautes dans les systèmes distribués hautement dynamiques.

À travers ce chapitre, nous avons donné les définitions de base des systèmes distribués et des défis tels que l'hétérogénéité, la dynamicité, la scalabilité, la gestion des fautes, etc. Nous avons également montré l'importance de la tolérance aux fautes pour garantir la fiabilité des systèmes et les techniques associées.

Chapitre 2

Clustering dans les systèmes distribués à large échelle

Sommaire

2.1 Introduction

L'utilisation de systèmes distribués à large échelle pour les calculs scientifiques, le partage et la diffusion des données augmente rapidement. Chaque nœud disponible dans ce système de calcul distribué diffère considérablement en termes de capacités de ressources telles que les performances du processeur, la bande passante, la disponibilité de la mémoire, etc. Ainsi, l'identification de la ressource est nécessaire pour satisfaire aux exigences de l'application (Kee, Logothetis, Huang, Casanova, & Chien, 2005).

Le clustering a été traditionnellement utilisé dans de nombreuses applications exécutées sur des systèmes distribués pour regrouper les tâches dans différents niveaux ou pour partitionner une topologie de réseau en groupes de nœuds ou de clusters interconnectés (Sood, Kour, & Kumar, 2016).

Un cluster est défini comme un ensemble d'objets qui ont un degré plus élevé de similarité les uns avec les autres par rapport aux objets qui ne sont pas dans le même cluster. Il a été appliqué à diverses applications dans de nombreux domaines tels que le marketing, la biologie, la reconnaissance de formes, le Web mining, et l'analyse des réseaux sociaux. De multiples algorithmes existent pour organiser les nœuds (tâches ou ressources) en clusters. Cependant, il n'existe pas de solution universelle à tous les problèmes (Nerurkar, Shirke, Chandane, & Bhirud, 2018; Mirarchi, et al., 2017).

Dans ce chapitre, nous dressons un état de l'art qui passe en revue les notions de base de clustering dans les systèmes distribués à large échelle, les domaines d'application, ainsi que les solutions proposées dans la littérature pour organiser les nœuds (tâches ou ressources) en clusters.

2.2 Représentations graphiques

Les graphes sont déployés dans de nombreux domaines du calcul parallèle et distribué, où les graphes sont utilisés pour la représentation des réseaux de communication dans des systèmes distribués. Les graphes sont également utilisés pour la représentation de la tâche, de la ressource, de la communication et de la structure de dépendance des programmes (Berge, 1976).

2.2.1 Concepts de base des graphes

Afin de discuter les modèles de graphes, il est nécessaire de définir les graphes et d'établir une terminologie. Les trois définitions initiales des graphes, des chemins et des cycles sont basées sur les notations données par Cormen et al. (Cormen, Leiserson, Rivest, & Stein, 2001).

Définition 2.1 (Graphe) Un graphe G est une paire ($\mathbf{V}$, $\mathbf{E}$), où $\mathbf{V}$ et $\mathbf{E}$ sont des ensembles finis. Un élément v de $\mathbf{V}$ est appelé sommet et un élément e de $\mathbf{E}$ est appelé arête. Une arête est une paire de sommets (u, v) où $u, v \in \mathbf{V}$, et par convention la notation e_{uv} est utilisée pour une arête entre les sommets u et v.

Dans un graphe dirigé, une arête e_{uv} a une direction distincte, du sommet u au sommet v, d'où $e_{uv} \neq e_{vu}$, et une telle arête sera appelée **arête ou arc dirigée**. Des auto-boucles (*self-loops*) d'un sommet à lui-même (c'est-à-dire e_{uv} avec $u = v$) sont possibles. Dans un graphe non orienté, e_{uv} et e_{vu} sont considérés comme la même arête non dirigée, donc $e_{uv} = e_{vu}$, et puisque les auto-boucles sont interdites $u \neq v$ pour tout arête e_{uv}.

On dit que l'arête e_{uv} est incidente sur les sommets u et v, et si e_{uv} est une arête dirigée, on dit que e_{uv} quitte le sommet u et entre dans le sommet v. De même, si $e_{uv} \in \mathbf{E}$, le sommet v est adjacent au sommet u et dans un graphe non dirigé, mais pas dans un graphe dirigé, le sommet u est adjacent au sommet v (c'est-à-dire que dans un graphe non dirigé, la relation d'adjacence est symétrique). L'ensemble $\{v \in \mathbf{V} : e_{uv} \in \mathbf{E}\}$ de tous les sommets v adjacents à u est dénoté **adj**(u). Pour l'arête dirigée e_{uv}, u est son sommet source et v son sommet de destination.

Le degré d'un sommet est le nombre d'arêtes qui lui sont incidentes. Dans un graphe dirigé, il est possible de distinguer entre le degré *out* (c'est-à-dire le nombre d'arêtes quittant le sommet) et le degré *in* (c'est-à-dire le nombre d'arêtes entrant dans le sommet).

La Figure 2.1 montre la représentation picturale de deux graphes : La Figure 2.1(a) un graphe non dirigé et la Figure 2.1(b) un graphe dirigé. Les sommets sont représentés par des cercles, les arêtes non dirigées par des lignes et les arêtes dirigées par des flèches. Les deux graphes sont composés des quatre sommets u, v, w, x. Dans le graphe non dirigé, le sommet v a un degré de 2, puisque les arêtes e_{vx} et e_{vw} sont incidentes sur lui. Ainsi, les sommets x et w sont adjacents à v, et, comme le graphe est non dirigé, v est également adjacent à eux.

Dans le graphe dirigé, v a un degré de 3 causé par les deux arêtes entrantes e_{uv} et e_{xv} (*in-degree* = 2) et l'arête sortante e_{vw} (*out-degree* = 1). Le sommet x est adjacent au sommet w, causé par l'arête e_{wx}, dont la source est w et la destination est x. Les arêtes e_{xw} et e_{wx} sont identiques dans le graphe non dirigé mais sont distinctes dans le graphe dirigé. L'une des arêtes quittant le sommet u y entre également et construit ainsi l'auto-boucle e_{uu}.

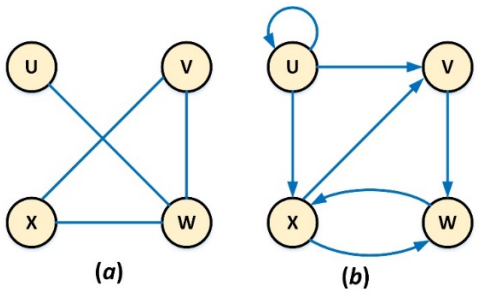


FIGURE 2.1 – Représentation picturale de deux exemples de graphes : (a) graphe non dirigé et (b) graphe dirigé. Les deux graphes sont constitués des sommets u, v, w, x et de diverses arêtes ; le sommet u a une auto-boucle (b)

Définition 2.2 (Chemin) Un chemin (*path*) p dans un graphe $G = (\mathbf{V}, \mathbf{E})$ d'un sommet v_0 à un sommet v_k est une séquence $<v_0, v_1, v_2, \ldots, v_k>$ de sommets tels qu'ils sont connectés par les arêtes $e_{v_{i-1}v_i} \in \mathbf{E}$, pour $i = 1, 2, \ldots, k$. Un chemin p contient les sommets $v_0, v_1, v_2, \ldots, v_k$ et les arêtes $e_{01}, e_{12}, e_{23}, \ldots, e_{(k-1)k}$ (e_{ij} est l'abréviation de $e_{v_iv_j}$) et le fait qu'un sommet v_i ou une

arête e_{ij} est membre du chemin p est dénoté par $v_i \in p$ et $e_{ij} \in p$, respectivement. Par conséquent, le chemin p d'un sommet v_0 à un sommet v_k est également défini par la séquence $<e_{01}, e_{12}, e_{23}, \ldots, e_{(k-1)k}>$ d'arêtes. Parfois $p(v_0 \rightarrow v_k)$ est écrit pour indiquer que le chemin va du sommet v_0 à v_k. Un chemin est simple si tous les sommets sont distincts. La longueur d'un chemin correspond au nombre d'arêtes du chemin. Un sous-chemin (*subpath*) d'un chemin $p = <v_0, v_1, v_2, \ldots, v_k>$ est une sous-séquence contiguë des sommets $<v_i, v_{i+1}, \ldots, v_j>$ avec $0 \leq i \leq j \leq k$. Deux chemins $p_1 = <v_0, v_1, \ldots, v_i>$ et $p_2 = <u_0, u_1, \ldots, u_j>$ peuvent être concaténés pour construire un nouveau chemin $p = <v_0, v_1, \ldots, vi, u_1, \ldots, u_j>$, si $v_i = u_0$.

Dans l'exemple de graphe non dirigé de la Figure 2.1, la séquence $<u, w, x>$ des sommets forme un chemin simple comme le fait $<u, x, v, w>$ dans le graphe dirigé, mais la séquence $<w, x, v, w, u>$ est un chemin dans le graphe non dirigé, ce qui n'est pas simple. Le chemin $<u, x, v>$ de longueur 2 dans le graphe dirigé est un sous-chemin du chemin $<u, x, v, w>$ de longueur 3.

Définition 2.3 (Cycle) Un chemin $p = <v_0, v_1, v_2, \ldots, v_k>$ forme un cycle si $v_0 = v_k$ et que le chemin contient au moins une arête. Le cycle est simple si, en plus, les sommets $v_1, v_2, \ldots, v_k$ sont distincts.

Le même cycle est formé de deux chemins $p = <v_0, v_1, \ldots, v_{k-1}, v_0>$ et $p' = < v'_0, v'_1, \ldots, v'_{k-1}, v'_0 >$, s'il existe un entier r avec $1 \leq r \leq k$-1 de sorte que $v'_i = v_{(i+r) mod\ k}$ pour i = 0, 1,. . . , k-1. Un sous-chemin d'un cycle p est une sous-séquence contigüe de sommets de n'importe lequel des chemins, y compris p, formant le même cycle que p. Un graphe sans cycle est *acyclique*.

Un exemple de cycle simple dans la Figure 2.1 est le chemin $<v, w, x, v>$ dans les deux échantillons de graphe, tandis que $<v, w, x, w, x, v>$ établit un cycle dans le graphe dirigé, ce qui n'est pas simple. Les deux chemins $<v, w, x, v>$ et $<w, x, v, w>$ décrivent le même cycle (r = 2) et $<v, w>$ est un sous-chemin de ce cycle.

Définition 2.4 (Relations de Sommet) Dans un graphe dirigé $G = (\mathbf{V}, \mathbf{E})$, les relations suivantes sont définies. Un sommet u est le prédécesseur du sommet v et par conséquent v est le successeur de u, si et seulement si l'arête $e_{uv} \in \mathbf{E}$, $u, v \in \mathbf{V}$. Le sommet v est le successeur du sommet u s'il est adjacent au sommet u. L'ensemble $\{x \in \mathbf{V} : e_{xv} \in \mathbf{E}\}$ de tous les prédécesseurs de v est dénoté par **pred**(v) et l'ensemble $\{x \in \mathbf{V} : e_{vx} \in \mathbf{E}\}$ de tous les successeurs de v, est dénoté par **succ**(v). Un sommet w est appelé *ancêtre* du sommet v s'il existe un chemin $p(w \rightarrow v)$ de w à v, et l'ensemble de tous les ancêtres de v est dénoté par **ance**(v) = $\{x \in \mathbf{V} : \exists p\ (x \rightarrow v) \in G\}$. Logiquement, un sommet w est appelé descendant du sommet v s'il existe un chemin $p(v \rightarrow w)$, et l'ensemble de tous les descendants de v est dénoté par **desc**(v) = $\{x \in \mathbf{V}: \exists p\ (v \rightarrow x) \in G\}$. Dans ce dernier cas, on dit parfois que le sommet w est accessible à partir de v.

Il est évident que tous les prédécesseurs sont également des ancêtres et tous les successeurs sont également des descendants. Des notations alternatives sont parfois utilisées le cas échéant, par exemple, enfant ou parent.

Voici quelques exemples du graphe dirigé de la Figure 2.1 (*b*) : le sommet x est un prédécesseur de v, qui est un successeur de u ; par conséquent, l'ensemble des prédécesseurs de v est **pred**(v)

= {*u*, *x*}. L'ensemble des successeurs de *w* ne comprend que le seul sommet *x*, **succ**(*w*) = {*x*}. Enfin, le sommet *u* est un ancêtre de *w* et *v* est un descendant de *w*, mais aussi son ancêtre.

Définition 2.5 (Sommet Source et Sommet Puits) Dans un graphe dirigé *G* = (**V**, **E**), un sommet $v \in \mathbf{V}$ n'ayant pas de prédécesseurs, **pred**(*v*) = ∅, est appelé sommet *source* et un sommet $u \in \mathbf{V}$ n'ayant pas de successeurs, **succ**(*u*) = ∅, est appelé sommet *puits*.

Les notations alternatives pour le sommet source et le sommet puits sont respectivement le sommet d'entrée et le sommet de sortie. L'ensemble des sommets source dans un graphe dirigé *G* est noté source(*G*) = $\{v \in \mathbf{V} : \mathbf{pred}(v) = \emptyset\}$ et l'ensemble des sommets puits par **sink**(*G*) = $\{v \in \mathbf{V} : \mathbf{succ}(v) = \emptyset\}$.

2.2.2 Représentation informatique et complexité des graphes

Pour l'analyse de la complexité des algorithmes basés sur les graphes, dans le temps et l'espace, il est essentiel de considérer la représentation informatique des graphes. Il existe deux façons standard de représenter un graphe *G* = (**V**, **E**) : comme une collection de listes d'adjacence ou comme une matrice d'adjacence (Cormen, Leiserson, Rivest, & Stein, 2001).

2.2.2.1 Représentation de la liste d'adjacence

Un graphe peut être représenté par un tableau de |**V**| listes d'adjacence, une pour chaque sommet dans **V**. La liste d'adjacence appartenant au sommet $u \in \mathbf{V}$ contient des pointeurs vers tous les sommets *v* qui sont adjacents à *u* ; par conséquent, il existe une arête $e_{uv} \in \mathbf{E}$. En d'autres termes, dans la liste d'adjacence du sommet *u*, les éléments de **adj**(*u*) sont stockés dans un ordre arbitraire. La Figure 2.2 montre les représentations de la liste d'adjacence des deux exemples de graphe de la Figure 2.1; dans la Figure 2.2 (*a*) celle du graphe non dirigé et dans la Figure 2.2 (*b*) celle du graphe dirigé.

Pour le graphe dirigé, la somme des longueurs des listes d'adjacence est |**E**|, car pour chaque arête e_{uv} le sommet de destination *v* apparaît une fois dans la liste du sommet *u*. Pour un graphe non dirigé, chaque arête e_{uv} apparaît deux fois, une fois dans la liste de *u* et une fois dans la liste de *v*, en raison de la symétrie de l'arête non dirigé ; ainsi, la somme des longueurs de la liste d'adjacence est 2|**E**|. Il est clair que cette forme de représentation décrit complètement un graphe *G*, car il y a une liste pour chaque sommet et au moins une entrée pour chaque arête. La quantité de mémoire requise pour un graphe, dirigé ou non, est par conséquent $\boldsymbol{O}(\mathbf{V} + \mathbf{E})$.

Dans la notation asymptotique précédente de la complexité, une convention de notation commune a été adoptée. Le signe | | pour la cardinalité (ou taille) des ensembles a été omise et elle a été écrite $\boldsymbol{O}(\mathbf{V} + \mathbf{E})$ au lieu de $\boldsymbol{O}(|\mathbf{V}| + |\mathbf{E}|)$. Cela doit être utilisé dans toutes les notations asymptotiques, mais seulement là, car il les rend plus lisibles et non ambigu.

La représentation de la liste d'adjacence a l'inconvénient qu'il n'existe pas de moyen plus rapide de déterminer si une arête e_{uv} fait partie d'un graphe *G* que de rechercher dans la liste d'adjacence de *u*.

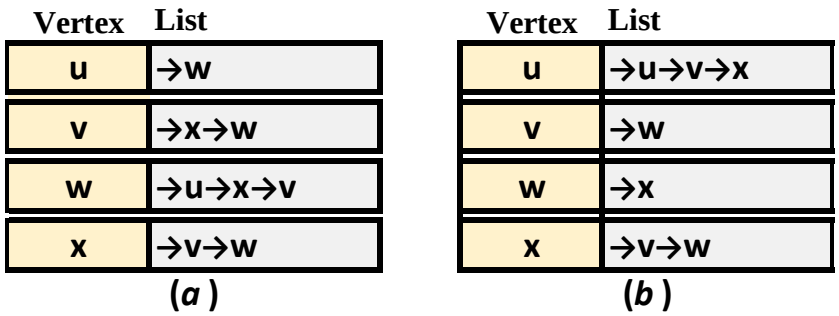


FIGURE 2.2 – Représentations de la liste d'adjacence des deux graphes de la Figure 2.1: (*a*) pour le graphe non dirigé et (*b*) pour le graphe dirigé

2.2.2.2 Représentation de la matrice d'adjacence

La représentation alternative d'un graphe en tant que matrice d'adjacence surmonte cette lacune. Un graphe est représenté par une matrice A $|\mathbf{V}| \times |\mathbf{V}|$ et on suppose que les sommets sont indexés 1, 2, . . ., $|\mathbf{V}|$ d'une manière arbitraire. Chaque élément a_{ij} de la matrice A a l'une des deux valeurs possibles : 1 si l'arête $e_{ij} \in \mathbf{E}$ et 0 autrement. La Figure 2.3 illustre les deux matrices d'adjacence pour les graphes de la Figure 2.1 avec les sommets *u*, *v*, *w*, *x* numérotés 1, 2, 3, 4, respectivement.

En raison de la symétrie des arêtes non dirigés, la matrice d'un graphe non dirigé est symétrique le long de sa diagonale principale, ce qui peut être observé dans la matrice de la Figure 2.3 (*a*). Comme la matrice est de taille $|\mathbf{V}| \times |\mathbf{V}|$, la mémoire requise pour la représentation de matrice d'adjacence est $\boldsymbol{O}(\mathbf{V}^2)$.

Pour de nombreux algorithmes, la liste d'adjacence est la forme de représentation préférée, car elle fournit un moyen compact de représenter des graphes clairsemés (peu denses) - ceux pour lesquels $|\mathbf{E}|$ est bien inférieur à $|\mathbf{V}|^2$. Pour les graphes denses, ou lorsque la détermination rapide de l'existence d'une arête est cruciale, la matrice d'adjacence est préférée.

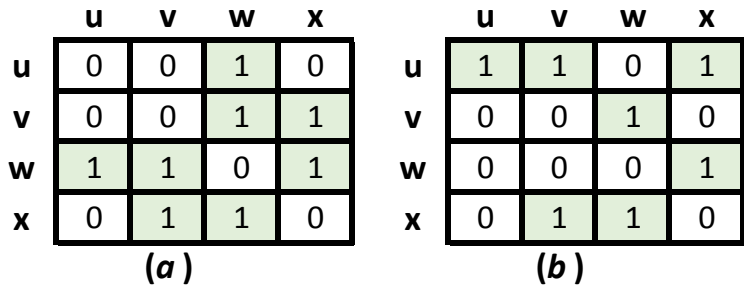


FIGURE 2.3 – Représentations de la matrice d'adjacence des deux graphes de la Figure 2.1: (*a*) pour le graphe non dirigé et (*b*) pour le graphe dirigé

2.2.3 Graphe en tant que modèle de programme

Sinnen et Sousa (Sinnen & Sousa, 1999) classifient certains modèles théoriques des graphes bien connus pour la représentation des calculs parallèles et distribués. Ils ont extrait plusieurs caractéristiques qui sont communes à la plupart des modèles de graphes. Sur la base de leurs constatations, un modèle de graphe générale est défini et ses propriétés sont analysées dans cette section.

Définition 2.6 (Modèle de Graphe) Dans une abstraction de la théorie des graphes, un programme se compose de deux types d'activités : le calcul et la communication. Le calcul est associé aux sommets d'un graphe et la communication à ses arêtes. Un sommet est appelé *nœud* et le calcul qui lui est associé *tâche*. Une tâche peut aller d'une instruction/opération atomique (c'est-à-dire une instruction qui ne peut pas être divisée en instructions plus petites) à des threads ou des déclarations composées telles que des boucles, des blocs de base et des séquences de ceux-ci. Toutes les instructions ou opérations d'une tâche sont exécutées dans un ordre séquentiel ; il n'y a pas de parallélisme au sein d'une tâche. Un nœud est à tout moment impliqué dans une seule activité : le calcul ou la communication.

Définition 2.7 (Nœuds Stricts) Les nœuds sont stricts en ce qui concerne à la fois leurs entrées et leurs sorties : c'est-à-dire qu'un nœud ne peut pas commencer l'exécution tant que toutes ses entrées ne sont pas arrivées, et qu'aucune sortie n'est disponible tant que le calcul n'est pas terminé et qu'à ce moment-là toutes les sorties sont disponibles pour la communication simultanément.

Un graphe adhérant aux définitions ci-dessus reflète avec sa structure de nœuds et d'arêtes la décomposition d'un programme en tâches et leur structure de communication. Comme mentionné précédemment, un tel graphe n'est pas capable de représenter la dépendance de contrôle, mais uniquement la structure de dépendance des données. En revanche, la dépendance aux données est très bien décrite par la structure d'un tel graphe. L'un des modèles de graphe les plus importants est le graphe de dépendance qui expose tous les types de dépendance aux données.

2.2.3.1 Coûts de calcul et de communication

De nombreux domaines d'utilisation des modèles de graphes de programmes nécessitent des connaissances sur les coûts de calcul et de communication des nœuds et des arêtes, respectivement, sauf si on suppose qu'ils sont uniformes. Généralement, ces coûts sont mesurés dans le temps - le temps que prend un calcul ou une communication sur le système cible respectif - et sont incorporés dans le modèle de graphe en attribuant des poids aux éléments du graphe. En termes de modèle de graphe général de la définition 2.6, cela peut être défini comme suit.

Définition 2.8 (Coûts de Calcul et de Communication) Soit $G = (\mathbf{V}, \mathbf{E}, w, c)$ un graphe représentant un programme $\mathcal{P}$ selon la définition 2.6. Le poids non négatif $w(n)$ associé au nœud $n \in \mathbf{V}$ représente son coût de calcul et le poids non négatif $c(e)$ associé au arête $e \in \mathbf{E}$ représente son coût de communication.

2.2.3.2 Critères de comparaison

Avoir défini un principe commun pour les modèles de graphes implique qu'il existe des caractéristiques qui permettent de les différencier. Sinnen et Sousa (Sinnen & Sousa, 1999) présentent plusieurs critères pour analyser et distinguer les différents modèles. Cela n'est pas surprenant étant donné que la création d'un graphe implique la décomposition du programme en tâches et la connaissance de leurs relations de communication. Les trois principaux critères sont :

- *Type de calcul*. Quel type de calcul peut être représenté efficacement avec le modèle de graphe ? Les modèles de graphe peuvent être distingués sur la base de granularité et la capacité à refléter les calculs itératifs et la régularité.
- *Architectures distribuées*. Pour quelle architecture de système distribué le modèle de graphe est-il généralement utilisé ? Les systèmes distribués génériques peuvent être classés en fonction de leur type d'instruction et de flux de données, de leur architecture de mémoire et de leur modèle de programmation.
- *Algorithmes et techniques*. Quels sont les algorithmes et les techniques qui peuvent être appliqués au modèle de graphe ? En général, les graphes sont utilisés pour l'analyse des dépendances, les transformations de programmes, le mappage et l'ordonnancement.

Outre ces sujets principaux, des questions pratiques telles que la représentation de calculateur et la taille des graphes sont également discutées par Sinnen et Sousa (Sinnen & Sousa, 1999).

Au cours de revu suivant des modèles de graphe individuels, ces critères seront utiles pour expliquer la motivation du modèle de graphe respectif et discuter de ses caractéristiques distinctives.

2.2.4 Graphe de tâches

Dans la littérature, le graphe de tâches est souvent appelé DAG (*Directed Acyclic Graph*), qui décrit purement les propriétés théoriques du modèle de graphe, à savoir qu'il s'agit d'un graphe acyclique dirigé. Afin d'éviter toute ambiguïté avec d'autres graphe acycliques dirigés, le nom graphe de tâche est utilisé dans ce qui suit.

Définition 2.9 (Graphe de Tâches (DAG)) Un graphe de tâches est un graphe acyclique dirigé $G = (\mathbf{V}, \mathbf{E}, w, c)$ représentant un programme $\mathcal{P}$ selon le modèle de graphe de la définition 2.6. Les nœuds dans $\mathbf{V}$ représentent les tâches de $\mathcal{P}$ et les arêtes dans $\mathbf{E}$ les communications entre les tâches. Une arête $e_{ij} \in \mathbf{E}$ du nœud n_i au n_j, $n_i, n_j \in \mathbf{V}$, représente la communication du nœud n_i au nœud n_j. Le poids positif $w(n)$ associé au nœud $n \in \mathbf{V}$ représente son coût de calcul et le poids non négatif $c(e_{ij})$ associé au arête $e_{ij} \in \mathbf{E}$ représente son coût de communication.

Pendant l'exécution du programme $\mathcal{P}$, chaque tâche représentée (c'est-à-dire le nœud) est exécutée exactement une fois. Il est présupposé que le programme $\mathcal{P}$ est un programme réalisable, d'où la propriété acyclique du graphe. Aucune communication ou dépendance n'existe dans $\mathcal{P}$ autre que celles reflétées par les arêtes de G. Le graphe des tâches caractérise

les poids des nœuds et des arêtes représentant les coûts de calcul et de communication, respectivement, avec la différence notable que le poids du nœud est ici défini comme positif comme dans la définition 2.8. Cette petite différence est importante pour certaines propriétés du graphe de tâches.

La Figure 2.4 illustre un exemple de graphe de tâches pour un petit programme fictif ; les nœuds sont nommés par les lettres de *a* à *k*, tandis que les poids des nœuds et des arêtes sont notés à côté des éléments du graphe respectifs.

Certains des premiers algorithmes d'ordonnancement et ceux utilisés dans des conditions restreintes utilisent des graphes de tâches simplifiés, dans la mesure où les coûts de communication et parfois de calcul sont négligés. En ce sens, le graphe de tâches défini ici est un modèle général, qui peut être utilisé dans ces algorithmes en ignorant les coûts de calcul et/ou de communication ou en les mettant à zéro.

Les arêtes du graphe de tâches reflètent uniquement la dépendance des données de flux. Si d'autres relations de dépendance existaient dans une version préliminaire du programme $\mathcal{P}$, elles ont été éliminées avant la construction du graphe de tâches *G*.

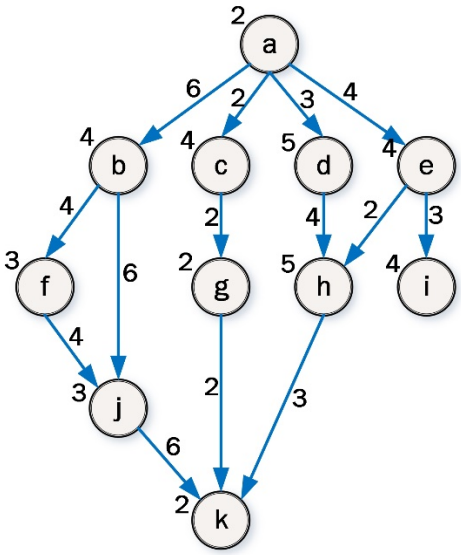


FIGURE 2.4 – Graphe de tâches (DAG) pour un programme fictif. Les nœuds sont nommés par les lettres *a* – *k* ; les poids des nœuds et des arêtes sont notés à côté d'eux.

2.3 Ordonnancement des tâches

L'ordonnancement statique fait généralement référence à l'ordonnancement au moment de la compilation, par opposition à l'ordonnancement dynamique, où les tâches sont prévues pendant l'exécution du programme, donc au moment de l'exécution. Il s'ensuit que la structure de calcul et de communication du programme et du système distribué cible doit être entièrement connue au moment de la compilation.

La discussion dans ce chapitre est orientée vers ce qui peut être considéré comme le problème général d'ordonnancement : aucune restriction n'est imposée sur le graphe de tâches d'entrée - il peut avoir des coûts de calcul et de communication arbitraires ainsi qu'une structure arbitraire - et le nombre de processeurs est limité.

Plusieurs autres problèmes d'ordonnancement surviennent en restreignant le graphe de tâches, par exemple, en ayant des coûts de calcul ou de communication unitaires, ou en employant un nombre illimité de processeurs.

2.3.1 Notions de base

Le problème d'ordonnancement a été introduit comme l'affectation spatiale et temporelle des tâches aux processeurs. L'affectation spatiale, ou mappage, est l'allocation de tâches aux processeurs.

Définition 2.10 (Allocation de Processeur) Une allocation de processeur *A* du graphe de tâches *G* = (**V**, **E**, *w*, *c*) (Définition 2.9) sur un ensemble fini **P** de processeurs est la fonction d'allocation de processeur *proc* : **V** → **P** des nœuds de *G* aux processeurs de **P**.

L'affectation temporelle est l'attribution d'une heure de début (*start time*) à chaque tâche. Même si l'ordonnancement ne fait référence qu'à l'attribution des heures de début, elle suppose l'allocation des tâches aux processeurs et, par conséquent, les deux sont généralement définies par un calendrier (*schedule*).

Définition 2.11 (Ordonnancement) Un Ordonnancement (*schedule*) *S* du graphe de tâches *G* = (**V**, **E**, *w*, *c*) sur un ensemble fini **P** de processeurs est la paire de fonctions (t_s, *proc*), où

- t_s: **V** → Q_0^+ est la fonction d'heure de début des nœuds de *G*.
- *proc*: **V** → **P** est la fonction d'allocation des processeurs des nœuds de *G* aux processeurs de **P**.

Ainsi, les deux fonctions t_s et *proc* décrivent l'affectation spatiale et temporelle des tâches, représentées par les nœuds du graphe de tâches, aux processeurs d'un système distribué cible, représentés par l'ensemble **P**. Le nœud $n \in \mathbf{V}$ est prévu pour démarrer l'exécution à $t_s(n)$ sur le processeur $proc(n) = P$, $P \in \mathbf{P}$, qui est noté $t_s(n, P)$; par conséquent,

$$t_s\,(n, P) \Leftrightarrow\ t_s(n), proc\,(n) = P, P \in \mathbf{P} \qquad (2.1)$$

La définition 2.11 décrit un ordonnancement, mais elle ne garantit pas la conformité avec les contraintes de priorité du graphe de tâches. Les conditions à cet effet seront établies, mais le modèle du système distribué cible doit d'abord être défini.

Définition 2.12 (Système Distribué Cible - Modèle Classique) Un système distribué cible est constitué d'un ensemble de processeurs **P** identiques connectés par un réseau de communication. Ce système a les propriétés suivantes :

1. *Système dédié*. Le système distribué est dédié à l'exécution du graphe de tâches ordonnancées. Aucun autre programme ou tâche n'est exécuté sur le système pendant l'exécution du graphe des tâches ordonnancées.
2. *Processeur dédié*. Un processeur $P \in \mathbf{P}$ ne peut exécuter qu'une seule tâche à la fois et l'exécution n'est pas préemptive.
3. *Communication locale négligeable*. Le coût de communication entre les tâches exécutées sur le même processeur, la communication locale, est négligeable et donc considéré comme nul. Cette hypothèse est basée sur l'observation que pour de nombreux systèmes distribués, la communication à distance (c'est-à-dire la communication interprocesseur) a un ou plusieurs ordres de grandeur plus coûteux que la communication locale (c'est-à-dire la communication intraprocesseur).
4. *Sous-système de communication*. La communication interprocesseur est effectuée par un sous-système de communication dédié. Les processeurs ne sont pas impliqués dans la communication.
5. *Communication simultanée*. La communication interprocesseur dans le système est effectuée simultanément ; il n'y a pas de conflit pour les ressources de communication.
6. *Entièrement connecté*. Le réseau de communication est entièrement connecté. Chaque processeur peut communiquer directement avec tout autre processeur via une liaison de communication identique dédiée.

Sur la base de ce modèle, la signification des coûts de calcul et de communication des nœuds et des arêtes dans un graphe de tâches, respectivement, peut être définie. Les poids des éléments du graphe de tâches ont été introduits en tant que coûts abstraits dans la définition 2.9.

Définition 2.13 (Coûts de Calcul et de Communication) Soit un système distribué constitué d'un ensemble de processeurs **P**. Les coûts de calcul et de communication d'un graphe de tâches $G = (\mathbf{V}, \mathbf{E}, w, c)$ exprimés respectivement en poids des nœuds et des arêtes sont définis comme suit :

- w: $\mathbf{V} \rightarrow Q^+$ est la fonction de coût de calcul des nœuds $n \in \mathbf{V}$. Le coût de calcul $w(n)$ du nœud n est le temps pendant lequel la tâche représentée par n occupe un processeur de **P** pour son exécution.
- c: $\mathbf{E} \rightarrow Q_0^+$ est la fonction de coût de communication de l'arête $e \in \mathbf{E}$. Le coût de communication $c(e)$ de l'arête e est le temps que la communication représentée par e prend à partir d'un processeur d'origine dans $P_i \in \mathbf{P}$ jusqu'à ce qu'elle arrive complètement à une destination différente processeur en $P_j \in \mathbf{P}$. En d'autres termes, $c(e)$ est le délai de communication entre l'envoi du premier élément de données jusqu'à la réception du dernier.

L'heure de fin d'un nœud est donc l'heure de début du nœud plus son temps d'exécution (c'est-à-dire son coût).

Définition 2.14 (Heure de Fin du Nœud) Soit S un ordonnancement pour le graphe de tâches $G = (\mathbf{V}, \mathbf{E}, w, c)$ sur l'ensemble **P**. L'heure de fin du nœud n est

$$t_f(n) = t_s(n) + w(n) \tag{2.2}$$

Encore une fois, $t_f(n, P)$ est écrit pour dénoter $t_f(n)$ et $proc(n) = P$.
En raison de la propriété 2 du modèle de système, il faut s'assurer que deux nœuds n'occupent pas un processeur en même temps.

Condition 2.1 (Allocation Exclusive de Processeur) Soit *S* un ordonnancement pour le graphe de tâches, $G = (\mathbf{V}, \mathbf{E}, w, c)$ sur l'ensemble **P**. Pour deux nœuds quelconques $n_i, n_j \in \mathbf{V}$:

$$proc(n_i) = proc(n_j) \Rightarrow \begin{cases} t_s(n_i) < t_f(n_i) \leq t_s(n_j) < t_f(n_j) \\ or \quad t_s(n_j) < t_f(n_j) \leq t_s(n_i) < t_f(n_i) \end{cases} \tag{2.3}$$

Contrairement au temps d'exécution d'une tâche dans le système cible, qui est simplement le coût du nœud respectif, le temps de communication d'une arête e_{ij} dépend des processeurs du nœud d'origine et de destination. Comme l'indique la propriété 3, la communication locale, c'est-à-dire la communication entre les tâches sur le même processeur, est négligée ; par conséquent, cela ne prend aucun temps - il n'y a pas de retard. Le temps de communication à distance, c'est-à-dire entre les tâches sur différents processeurs, est donné par le poids de l'arête représentant. L'heure à laquelle une communication arrive au processeur de destination est définie comme l'heure de fin de l'arête.

Définition 2.15 (Heure de Fin de l'Arête) Soit $G = (\mathbf{V}, \mathbf{E}, w, c)$ un graphe de tâches et **P** un système distribué. L'heure de fin de $e_{ij} \in \mathbf{E}$, $n_i, n_j \in \mathbf{V}$, communiqué par le processeur P_{src} à $P_{dst}, P_{src}, P_{dst} \in \mathbf{P}$, est

$$t_f(e_{ij}, P_{src}, P_{dst}) = t_f(n_i, P_{src}) + \begin{cases} 0 & si\ P_{src} = P_{dst} \\ c(e_{ij}) & sinon \end{cases} \tag{2.4}$$

Ainsi, l'heure d'arrivée de e_{ij} au processeur P_{dst} est donnée par l'heure de fin du nœud n_i, c'est-à-dire l'heure à laquelle les données pour e_{ij} sont rendues disponibles par n_i plus l'heure de communication de e_{ij}, soit local (0) ou distant ($c(e_{ij})$).

Suivant les propriétés 4 à 6 du modèle de système, une communication e_{ij} est toujours lancée dès que son nœud d'origine n_i se termine : il n'y a aucun retard dû à une forme quelconque de contention.

Enfin, une condition peut être formulée qui garantit la conformité de l'ordonnancement *S* avec les contraintes de précédence du graphe de tâches *G*.

Condition 2.2 (Contrainte de Précédence) Soit *S* un ordonnancement pour le graphe de tâches $G = (\mathbf{V}, \mathbf{E}, w, c)$ sur le système **P**. Pour $n_i, n_j \in \mathbf{V}$, $e_{ij} \in \mathbf{E}$, $P \in \mathbf{P}$,

$$t_s(n_j, P) \geq t_f(e_{ij}, proc(n_i), P) \tag{2.5}$$

La condition 2.2 formule les conséquences d'une contrainte de précédence imposée par une arête $e_{ij} \in \mathbf{E}$ sur l'heure de début du nœud de destination $n_j \in \mathbf{V}$ dans l'ordonnancement *S*. La stricness du nœud oblige n_j à retarder son démarrage jusqu'à ce que la communication de e_{ij} soit complètement arrivée au processeur *P*.

La condition 2.2 garantit l'exécution des nœuds dans l'ordre de précédence.

Les deux conditions 2.1 et 2.2 sont les seules contraintes mises sur un ordonnancement. Une conséquence très notable est le chevauchement possible du calcul et de la communication. Alors que les communications sortantes d'un nœud sont envoyées sur le réseau vers leurs destinations, le processeur du nœud peut procéder au calcul.

Un ordonnancement, dont les nœuds respectent ces deux conditions, respecte les contraintes de précédence du graphe de tâches et les propriétés du modèle de système et un tel ordonnancement est appelé de *faisable*.

Définition 2.16 (Ordonnancement Faisable) Soit *S* un ordonnancement pour le graphe de tâches *G* = (**V**, **E**, *w*, *c*) sur le système **P**. *S* est faisable si et seulement si tous les nœuds $n \in \mathbf{V}$ et les arêtes $e \in \mathbf{E}$ sont conformes aux conditions 2.1 et 2.2.

Désormais, tous les ordonnancements envisagés sont faisables, sauf indication contraire, et l'attribut "faisable" est donc omis.

La condition 2.2 oblige le nœud n_j à attendre que toutes les communications entrantes $e_{ij} \in \mathbf{E}$, $n_i \in \mathbf{pred}(n_j)$ soient arrivées à son processeur d'exécution. Le premier temps à laquelle le nœud n_j peut démarrer l'exécution est appelée *temps de disponibilité des données (Data Ready Time (DRT)).*

Définition 2.17 (Data Ready Time (DRT)) Soit *S* un ordonnancement pour le graphe de tâches *G* = (**V**, **E**, *w*, *c*) sur le système **P**. Le temps de disponibilité des données (DRT) d'un nœud $n_j \in \mathbf{V}$ sur le processeur $P \in \mathbf{P}$ est

$$t_{\mathrm{dr}}\left(n_j, P\right) = \max_{n_i \in \mathbf{pred}(n_j)} \left\{ t_f\left(e_{ij}, proc(n_i), P\right) \right\} \tag{2.6}$$

Si $\mathbf{pred}(n_j) = \emptyset$, c'est-à-dire que n_j est un nœud source, $t_{\mathrm{dr}}(n_j) = t_{\mathrm{dr}}(n_j, P) = 0$, pour tout $P \in \mathbf{P}$.

Notez que le DRT peut être déterminé pour tout processeur *P* de **P**, indépendamment du processeur auquel n_j est alloué. Les contraintes sur l'heure de début $t_s(n)$ d'un nœud *n* peuvent désormais être formulées à l'aide du DRT.

Condition 2.3 (Contrainte DRT) Soit *S* un ordonnancement pour le graphe de tâches *G* = (**V**, **E**, *w*, *c*) sur le système **P**. Pour $n \in \mathbf{V}$, $P \in \mathbf{P}$,

$$t_s\left(n, P\right) \geq t_{\mathrm{dr}}(n, P) \tag{2.7}$$

Ainsi, la condition 2.3 fusionne les contraintes imposées sur l'heure de début d'un nœud par toutes les arêtes entrantes selon l'équation. (2.5).

L'heure de fin d'un processeur *P* est l'heure à laquelle le dernier nœud ordonnancé sur *P* se termine.

Définition 2.18 (Heure de Fin du Processeur) Soit *S* un ordonnancement pour le graphe de tâches *G* = (**V**, **E**, *w*, *c*) sur le système **P**. L'heure de fin du processeur $P \in \mathbf{P}$ est

$$t_f(P) = \max_{n \in \mathbf{V}: proc(n)=P} \{t_f(n)\} \quad (2.8)$$

Un ordonnancement se termine lorsque le dernier des nœuds du graphe de tâches *G* se termine.

Définition 2.19 (Durée d'Ordonnancement (Makespan)) Soit *S* un ordonnancement pour le graphe de tâches *G* = (**V**, **E**, *w*, *c*) sur le système **P**. La durée d'ordonnancement (schedule length (sl)) de *S* est

$$sl(S) = \max_{n \in \mathbf{V}} \{t_f(n)\} - \min_{n \in \mathbf{V}} \{t_s(n)\} \quad (2.9)$$

Si $\min_{n \in \mathbf{V}} \{t_s(n)\} = 0$, cette expression se réduit à

$$sl(S) = \max_{n \in \mathbf{V}} \{t_f(n)\} \quad (2.10)$$

Tous les ordonnancements considérés dans ce texte commencent à l'unité de temps 0 ; ainsi, l'expression (2.10) suffit comme définition de la durée de l'ordonnancement. Une autre désignation de la durée de l'ordonnancement, qui est assez courante dans la littérature, est *makespan*.

L'ensemble des processeurs utilisés dans un ordonnancement *S* donné est défini comme l'ensemble de tous les processeurs sur lesquels au moins un nœud est exécuté.

Définition 2.20 (Processeurs Utilisés) Soit *S* un ordonnancement pour le graphe de tâches *G* = (**V**, **E**, *w*, *c*) sur le système **P**. L'ensemble des processeurs utilisés est

$$\mathbf{Q} = \bigcup_{n \in \mathbf{V}} proc\,(n) \quad (2.11)$$

Evidemment, pour tout ordonnancement *S*

$$|\mathbf{Q}| \leq |\mathbf{P}| \quad (2.12)$$

Pour conclure les définitions de base, le temps séquentiel d'un graphe de tâches est défini.

Définition 2.21 (Temps Séquentiel) Soit *G* = (**V**, **E**, *w*, *c*) un graphe de tâches. Le temps séquentiel de *G* est

$$seq(G) = \sum_{n \in \mathbf{V}} w(n) \quad (2.13)$$

ce qui correspond au temps d'exécution de *G* sur un seul processeur (rappelez-vous que la communication locale est gratuite).

2.3.2 Exemple d'ordonnancement

Les définitions et conditions ci-dessus sont illustrées en examinant un exemple d'ordonnancement. La Figure 2.5 (*b*) représente un diagramme de Gantt d'un ordonnancement pour l'exemple de graphe de tâches de la Figure 2.4 sur trois processeurs. Un diagramme de Gantt est une représentation graphique intuitive et courante d'un ordonnancement, dans laquelle chaque objet ordonnancé (c'est-à-dire, un nœud) est dessiné sous forme de rectangle. La position du nœud (c'est-à-dire la position de son rectangle) dans le système de coordonnées couvert par l'axe du temps et de l'espace (c'est-à-dire l'axe du processeur) est déterminée par le processeur alloué au nœud et son heure de début, tandis que la taille du rectangle reflète le temps d'exécution du nœud.

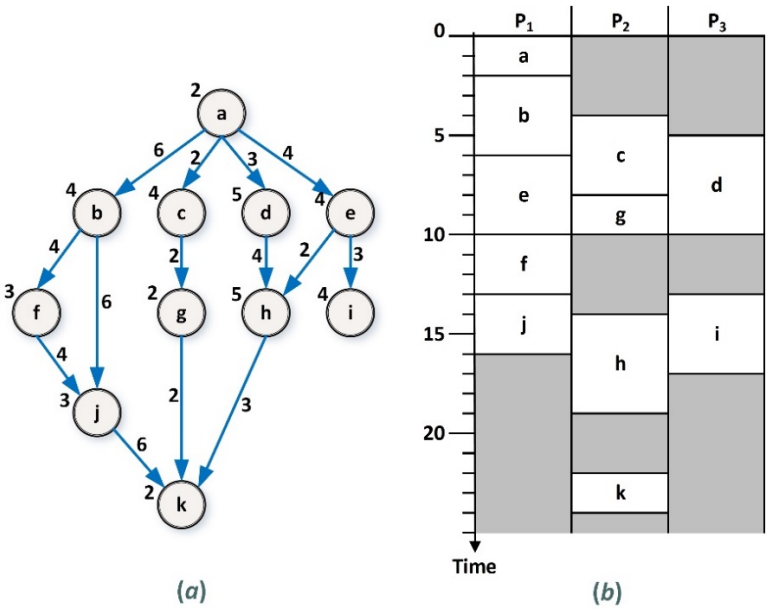


FIGURE 2.5 – Diagramme de Gantt (*b*) d'un ordonnancement pour l'exemple de graphe de tâches (*a*) de la Figure 2.4 sur trois processeurs

Quelques exemples de cette Figure illustrent les définitions et conditions précédentes. L'heure de début du nœud *a* exécuté sur le processeur P_1 est $t_s(a) = 0$ (c'est-à-dire $t_s(a, P_1) = 0$) et, avec un coût de calcul de $w(a) = 2$, son heure de fin est $t_f(a) = t_f(a, P_1) = 2$. Le nœud *d* commence à $t_s(d) = 5$ et se termine à $t_f(d) = t_s(d) + w(d) = 5 + 5 = 10$. Son heure de début $t_s(d) = 5$ est la plus proche possible, car elle doit attendre les données du nœud *a*, $t_f(e_{ad}, P_1, P_3) = t_f(a) + c(e_{ad}) = 2 + 3 = 5$. Alors dans ce cas, la communication est distante et provoque donc un retard de 3 unités de temps correspondant au poids de l'arête e_{ad}. Le nœud *b*, en revanche, reçoit également des données du nœud *a*, mais comme la communication est locale - *a* et *b* sont sur le même processeur - cela ne prend pas de temps : $t_s(b) = t_f(e_{ab}, P_1, P_1) = t_f(a) = 2$. Un bon exemple pour l'influence de diverses contraintes de précédence sur l'heure de début d'un nœud est le nœud *h*. Les deux communications entrantes e_{dh} et e_{eh} donnent un temps de disponibilité des données (DRT) de $t_{dr}(h, P_2) = 14$. Le responsable est la communication du nœud *d*, puisque *d* se termine à $t_f(d) = 10$ et la communication de e_{dh} dure 4 unités de temps, tandis que e_{eh} arrive à P_2 à $t_f(e_{eh}, P_1, P_2) = 10 + 2 = 12$. Donc le nœud *h* commence à son DRT. En revanche, le nœud *e* commence

à $t_s(e) = 6$, qui est plus tard que son DRT ($t_{dr}(e, P_1) = t_f(e_{ae}, P_1, P_1) = t_f(a) = 2$), car le processeur P_1 est occupé avec le nœud b jusqu'à ce temps. La durée de l'ordonnancement $sl = t_f(k) = 24$, c'est-à-dire l'heure de fin du dernier nœud *k*, car le premier nœud *a* commence à l'unité de temps 0.

2.3.3 Propriétés du graphe de tâches

Les concepts présentés ci-après sont utilisés dans les techniques d'ordonnancement du texte restant, en particulier pour l'attribution de priorités aux nœuds. Dans de nombreux algorithmes d'ordonnancement, l'ordre dans lequel les nœuds sont considérés a une influence significative sur l'ordonnancement résultant et sa longueur. L'évaluation de l'importance des nœuds avec un schéma de priorité est donc un élément fondamental de l'ordonnancement.

La discussion commence par la définition de la longueur d'un chemin dans le graphique des tâches, sur la base des coûts de calcul et de communication définis précédemment.

Définition 2.22 (Longueur du Chemin (Path Length)) Soit $G = (\mathbf{V}, \mathbf{E}, w, c)$ un graphe de tâches. La longueur d'un chemin *p* dans *G* est la somme des poids de ses nœuds et arêtes :

$$len(p) = \sum_{n \in p, \mathbf{V}} w(n) + \sum_{e \in p, \mathbf{E}} c(e) \tag{2.14}$$

La longueur de calcul d'un chemin *p* dans *G* est la somme des poids de ses nœuds :

$$len_w(p) = \sum_{n \in p, \mathbf{V}} w(n) \tag{2.15}$$

Notez que la définition de la longueur du chemin dans un graphe de tâches diffère de la définition générale 2.2, où la longueur d'un chemin est égale au nombre d'arêtes. La longueur de chemin $len(p)$ peut être interprétée comme le temps que prend le chemin *p* pour son exécution si toutes les communications entre ses nœuds sont des communications interprocesseurs, ce qui se produit, par exemple, lorsque chaque nœud de *p* est alloué à un processeur différent. En raison de l'ordre séquentiel inhérent au chemin, aucun des nœuds n'est exécuté simultanément avec un autre nœud. La longueur du chemin de calcul $len_w(p)$ peut être interprétée comme le temps d'exécution du chemin *p* lorsque toutes les communications entre ses nœuds sont locales, c'est-à-dire qu'elles ont un coût nul. Par conséquent, tous les nœuds de *p* sont exécutés sur le même processeur. Dans un graphe de tâches $G = (\mathbf{V}, \mathbf{E}, w)$ sans coûts de communication, la longueur du chemin est nécessairement définie comme elle l'est dans Eq. (2.15).

Lorsque les allocations de processeur des nœuds de p sont connues, la longueur du chemin peut être déterminée en tenant compte du fait que les communications locales sont gratuites. Dans un algorithme d'ordonnancement, il peut être souhaitable de calculer une longueur de chemin sur la base d'un ordonnancement partiel : c'est-à-dire que certaines allocations de processeur sont données et d'autres ne le sont pas. La longueur du chemin déterminée pour une allocation

de processeur *A* (partielle) donnée (définition 2.2) est indiquée par *len*(*p*, *A*), la longueur du chemin allouée. Les communications entre les nœuds dont les allocations de processeur sont inconnues sont supposées être distantes. La longueur du chemin allouée *len*(*p*, *A*) est ainsi comprise entre la longueur du chemin *len*(*p*) et la longueur du chemin de calcul $len_w(p)$:

$$len(p) \geq len(p, A) \geq len_w(p) \tag{2.16}$$

Le schéma ci-dessus pour la distinction entre les différentes longueurs de chemin est utilisé tout au long de ce texte. Toutes les définitions basées sur les longueurs de chemin ne seront formulées que pour *len*(*p*) mais sont implicitement valables pour *len*(*p*, *A*) et $len_w(p)$ également. Les définitions correspondantes utilisent le même schéma de distinction : autrement dit, la longueur du chemin allouée est paramétrée avec *A* et l'indice *w* est utilisé pour la longueur de calcul.

2.3.3.1 Chemin Critique

Un concept important pour l'ordonnancement est le chemin critique - le plus long chemin dans le graphe de tâches.

Définition 2.23 (Chemin Critique (Critical Path)) Un chemin critique *cp* d'un graphe de tâches *G* = (**V**, **E**, *w*, *c*) est le plus long chemin de *G*

$$len(cp) = \max_{p \in G}\{len(p)\} \tag{2.17}$$

Le chemin critique de calcul cp_w et le chemin critique alloué *cp*(*A*) pour une allocation de processeur *A* sont définis en conséquence.

Les nœuds d'un chemin critique *cp*, composé de *l* nœuds, sont dénotés par $n_{cp,1}$, $n_{cp,2}$, ..., $n_{cp,l}$. Il est évident qu'il peut y avoir plus d'un chemin critique, car plusieurs chemins peuvent avoir la même longueur maximale.

Notez qu'en général

$$cp \neq cp_w \neq cp(A) \tag{2.18}$$

d'où suit pour leurs longueurs de chemin

$$len(cp_w) \leq len(cp) \quad et \quad len\big(cp(A)\big) \leq len(cp) \tag{2.19}$$

Des inégalités équivalentes sont valables pour la longueur du chemin de calcul et la longueur du chemin alloué et les chemins critiques correspondants.

2.3.3.2 Granularité

Pour un graphe de tâches, la notion de granularité décrit généralement la relation entre le calcul et les coûts de communication. Souvent, la granularité d'un graphe de tâche est définie comme la relation entre le poids minimal du nœud et le poids maximal de l'arête.

Définition 2.24 (Granularité du Graphe de Tâches) Soit $G = (\mathbf{V}, \mathbf{E}, w, c)$ un graphe de tâches. La granularité de G est

$$g(G) = \frac{\min_{n \in \mathbf{V}} w(n)}{\max_{e \in \mathbf{E}} c(e)} \tag{2.20}$$

Selon cette définition, un graphe de tâche est dit à gros grain si $g(G) \geq 1$. La granularité grossière est une propriété souhaitable d'un graphe de tâche. L'un des objectifs de l'ordonnancement des tâches est toujours de minimiser le coût de la communication. Ceci est réalisé en ayant autant de communication locale (c'est-à-dire, communication entre les tâches sur le même processeur) que possible.

Malheureusement, cet objectif est en conflit avec l'autre objectif d'ordonnancement, à savoir la répartition des tâches entre les processeurs. Les graphes à granularité grossière peuvent être parallélisés (c'est-à-dire ordonnancés) plus efficacement, puisque le coût de la communication par parallélisation infligé est raisonnable à faible par rapport au coût de calcul.

Il n'est donc pas surprenant que certains résultats théoriques de l'ordonnancement des tâches puissent être établis pour les graphes de tâches à gros grain. Par exemple, des limites sur la longueur de l'ordonnancement par rapport à la longueur optimale de l'ordonnancement peuvent être établies pour les graphes à gros grain (Darte, Robert, & Vivien, 2000; Gerasoulis & Yang, 1992).

Certains des algorithmes de clustering qui seront présentés après dans ce texte tentent essentiellement d'améliorer (c'est-à-dire d'augmenter) la granularité d'un graphe de tâches. Les coûts de communication sont éliminés en regroupant les nœuds et en rendant ainsi la communication locale, donc gratuite.

Définition 2.25 (Grain) Soit $G = (\mathbf{V}, \mathbf{E}, w, c)$ un graphe de tâches. Le grain du nœud $n_i \in \mathbf{V}$ est

$$grain(n_i) = \min\left\{\frac{\min_{n_j \in \mathbf{pred}(n_i)} w(n_j)}{\max_{e_{ji} \in \mathbf{E}, n_j \in \mathbf{pred}(n_i)} c(e_{ji})}, \frac{\min_{n_j \in \mathbf{succ}(n_i)} w(n_j)}{\max_{e_{ij} \in \mathbf{E}, n_j \in \mathbf{succ}(n_i)} c(e_{ij})}\right\} \tag{2.21}$$

Si $\mathbf{pred}(n_i) \cup \mathbf{succ}(n_i) = \emptyset$, c'est-à-dire que le nœud est indépendant, le grain n'est pas défini.

À titre d'exemple pour cette définition, considérons la Figure 2.6, où le nœud *n* a

$$grain(n) = \min\left\{\frac{\min\{2,3,4\}}{\max\{1,1,1\}}, \frac{\min\{5,6\}}{\max\{2,2\}}\right\} = \min\left\{\frac{2}{1}, \frac{5}{2}\right\} = 2$$

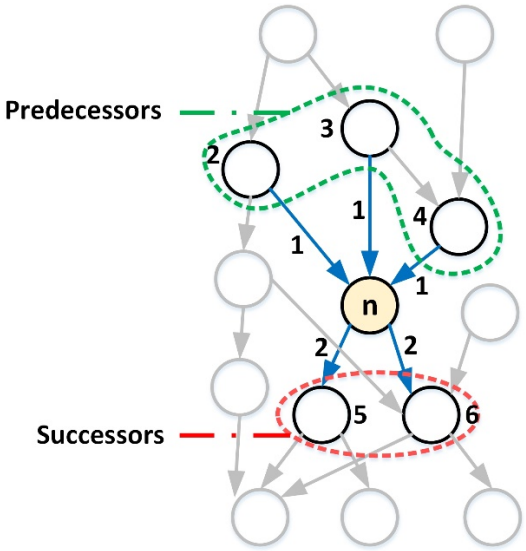


FIGURE 2.6 – Un graphe de tâches, où le nœud *n* est présenté comme exemple pour la définition 2.25 du grain ; le nœud *n* a *grain*(*n*) = min {min{2, 3, 4}/max{1, 1, 1}, min{5, 6}/max{2, 2}} = 2

2.4 Clustering des tâches

L'ordonnancement des tâches dans le modèle classique est un compromis entre la minimisation des coûts de communication interprocesseur et la maximisation de la simultanéité (concurrence) de l'exécution des tâches (Sinnen O. , 2007). Une idée naturelle est donc de déterminer d'abord - avant l'ordonnancement proprement dit - quels nœuds doivent toujours être exécutés sur le même processeur. Les candidats évidents pour le regroupement sont les nœuds qui dépendent les uns des autres, en particulier les nœuds du chemin critique.

2.4.1 Notion de clustering

Le clustering est une technique qui suit cette idée. Elle ne convient donc qu'à l'ordonnancement avec des coûts de communication. Il s'agit essentiellement d'une technique d'ordonnancement pour un nombre illimité de processeurs. Néanmoins, elle est souvent proposée comme étape initiale dans l'ordonnancement d'un nombre limité de processeurs. Pour faire la distinction entre le nombre limité de processeurs (physiques) et le nombre illimité de processeurs (virtuels) supposés lors de l'étape de clustering. Ces derniers sont appelés "clusters", d'où le terme "clustering". Dans le clustering, les nœuds correspondants aux tâches du graphe sont mappés et ordonnancés dans ces clusters.

Définition 2.26 (Clustering) Soit $G = (\mathbf{V}, \mathbf{E}, w, c)$ un graphe de tâches. Un clustering C est un ordonnancement de G sur un système distribué implicite $\mathbf{C}$ avec un nombre "illimité" de processeurs ; c'est-à-dire, $|\mathbf{C}| = |\mathbf{V}|$. Les processeurs $C \in \mathbf{C}$ sont appelés clusters (Sinnen O. , 2007).

Les algorithmes d'ordonnancement basés sur le clustering pour un nombre limité de processeurs se composent de plusieurs étapes. Dans les algorithmes basés sur le clustering, trois étapes sont

nécessaires : (1) le clustering, (2) le mappage des clusters aux processeurs (physiques) et (3) l'ordonnancement des nœuds. L'algorithme 2.1 décrit une heuristique générique d'ordonnancement basée sur le clustering en trois étapes pour un nombre limité de processeurs.

Algorithme 2.1 Algorithme générique d'ordonnancement basée sur le clustering en trois étapes (G = (**V**, **E**, w, c), **P**)

▷ *Clustering des nœuds*

(1) Trouver un clustering C de G

▷ *Mappage des clusters aux processeurs*

(2) Assigner des clusters de **C** aux processeurs (physiques) de **P**

▷ *Ordonnancement des nœuds*

(3) Attribuer l'heure de début $t_s(n)$ à chaque nœud $n \in \mathbf{V}$

La motivation souvent citée pour le clustering a été donnée par Sarkar (Sarkar, 1989): si les tâches sont mieux exécutées dans le même processeur (cluster) d'un système idéal, c'est-à-dire un système qui possède plus de processeurs (clusters) que de tâches, elles devraient également être exécutées sur le même processeur dans n'importe quel système réel.

2.4.2 Algorithmes de clustering

Strictement parlant, la première étape de l'algorithme 2.1 peut être effectuée par n'importe quel algorithme d'ordonnancement adapté à un nombre illimité de processeurs. Cependant, dans la littérature, le terme clustering désigne un certain type d'algorithmes, avec plusieurs aspects caractéristiques. La discussion suivante est basée sur (Darte, Robert, & Vivien, 2000; El-Rewini, Lewis, & Ali, 1994; Gerasoulis & Yang, 1992).

Les algorithmes de clustering commencent par un *clustering initial* C_0 du graphe de tâches G. Habituellement, chaque nœud $n \in \mathbf{V}$ est alloué à un cluster distinct $C \in \mathbf{C}$.

L'algorithme de clustering effectue ensuite *des étapes de raffinement incrémentielles* allant du clustering C_{i-1} au clustering C_i, dans lequel les clusters sont *fusionnés*. Cela signifie que les nœuds de ces clusters sont fusionnés en un seul cluster. Si les nœuds communicants sont exécutés dans le même cluster, leur communication devient locale et donc son coût est nul selon le modèle de système cible (définition 2.12). Cela peut être bénéfique pour le temps d'exécution total du graphe, car il élimine les coûts de communication. Normalement, une fusion de clusters n'est effectuée que si la longueur de l'ordonnancement du nouveau clustering C_i diminue ou au moins reste la même par rapport à la longueur de l'ordonnancement du clustering actuel C_{i-1}. Les étapes de raffinement sont effectuées jusqu'à ce que tous les candidats à la fusion aient été considérés. L'algorithme 2.2 résume ce principe d'algorithmes de clustering.

Algorithme 2.2 Principe des algorithmes de clustering ($G = (\mathbf{V}, \mathbf{E}, w, c)$)

Créer un clustering initial C_0: allouer chaque nœud $n \in \mathbf{V}$ à un cluster distinct $C \in \mathbf{C}$, $|\mathbf{C}| = |\mathbf{V}|$

$i \leftarrow 0$

Répéter

$i \leftarrow i + 1$

Sélectionner les clusters candidats à fusionner

Créer un nouveau clustering C_i : fusionner les clusters candidats en un seul cluster

si $sl(C_i) > sl(C_{i-1})$ **alors** ▷ *la fusion augmente la longueur l'ordonnancement actuel*

$C_i \leftarrow C_{i-1}$ ▷ *rejeter le nouveau clustering* C_i

fin Si

Jusqu'à tous les candidats à la fusion ont été considérés

La Figure 2.7 montre quelques exemples de clusterings d'un simple graphe de tâches (Figure 2.7 (*a*)), qui a souvent été utilisé pour illustrer le clustering (Gerasoulis & Yang, 1992). Un clustering initial est illustré dans la Figure 2.7 (*b*). Chaque nœud est alloué à un cluster distinct, symbolisé par la ligne pointillée grise entourant chaque nœud. La Figure 2.7 (*c*) illustre le clustering, où les clusters de nœuds *a* et *b* du clustering initial ont été fusionnés en un cluster. Il peut s'agir du clustering produit par un algorithme après la première étape de raffinement. Enfin, la Figure 2.7 (*d*) illustre un clustering avec seulement deux clusters.

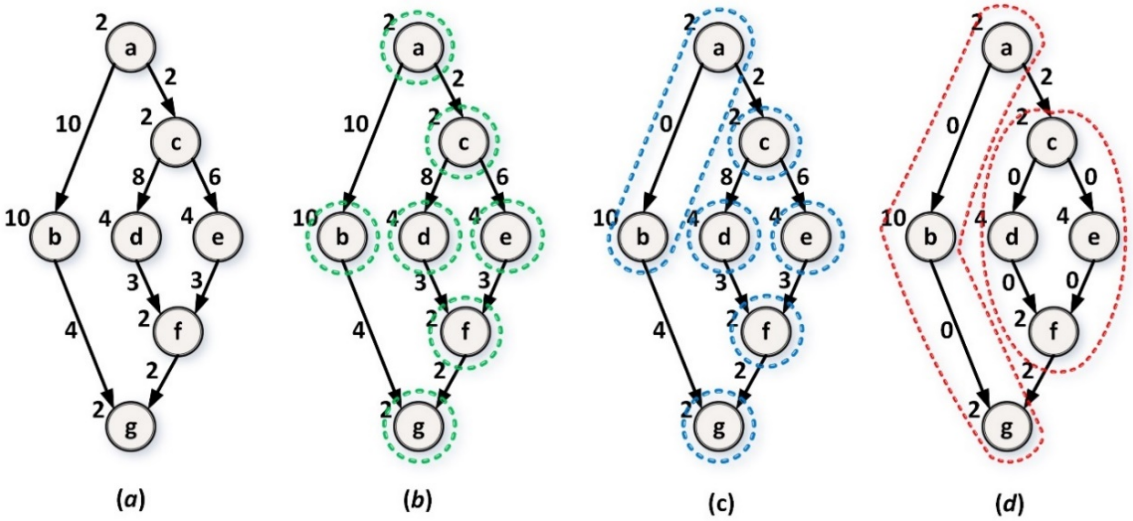


FIGURE 2.7 – Clusterings d'un simple graphe de tâche: (*a*) graphe de tâche simple pour les algorithmes de clutering (Gerasoulis & Yang, 1992); (*b*) clustering initial; (*c*) clustering après fusion des clusters de nœuds *a* et *b*; (*d*) clustering avec seulement deux cluster

2.5 Clustering de ressources

Le clustering de ressources a de nombreuses applications telles que les systèmes P2P, les grilles de calcul, le Cloud Computing, les réseaux mobiles ad-hoc, les réseaux de capteurs, etc. Dans un schéma de clustering, les nœuds sont divisés en différents groupes virtuels. Dans une structure de cluster, les nœuds peuvent se voir attribuer un statut ou une fonction différente, comme chef de cluster (*cluster-head*) ou membre de cluster (*cluster-member*). Un *cluster-head* serve normalement comme coordinateur local pour son cluster, en assurant la transmission intra-cluster, le transfert de données, etc. Un *cluster-member* est généralement appelé un nœud ordinaire, qui est un nœud non-*cluster-head* sans aucun lien inter-cluster. Dans le clustering à *1* saut (*1*-hop), la distance entre un *cluster-head* et l'un de ses membres est de *1* saut et le clustering à *k* sauts (*k*-hop) représente les clusters où le nombre maximum de sauts entre un nœud d'un cluster et son *cluster-head* est *k* sauts. Les *cluster-heads* sont directement connectées les unes aux autres. De plus, les nœuds de ces systèmes ne peuvent communiquer qu'avec leurs voisins (Sirohi & Yadav, 2016; Tsai, Kang, Hu, & Chiang, 2016; Yu & Chong, 2005).

Une revue de la littérature pertinente révèle certains problèmes de clustering des ressources dans les plates-formes distribuées à large échelle, telles que BOINC (Anderson D. P., 2004), WCG ou SETI@Home (Anderson, Cobb, Korpela, Lebofsky, & Werthimer, 2002). Ces plates-formes hautement distribuées se caractérisent par leur puissance de calcul globale élevée, leur hétérogénéité en termes de performances des ressources et par le dynamisme de leur topologie, dû aux arrivées et aux départs de nœuds.

En effet, pour les problèmes impliquant de grands ensembles de données, très peu de ressources peuvent être capables de stocker les données associées à une tâche, et donc peuvent être en mesure de participer aux calculs. Dans ce contexte, plusieurs algorithmes distribués sont proposés pour un large ensemble de ressources permettant de construire des clusters, où chaque cluster sera responsable du traitement des tâches et du stockage des données associées (Gil, Kim, & Lee, 2014; Beaumont, Bonichon, Duchon, Eyraud-Dubois, & Larchevêque, 2008).

2.5.1 Définition

Le processus de clustering consiste à un découpage virtuel de l'ensemble des nœuds constituant le cluster en un ensemble de groupes proches géographiquement.

Le réseau d'interconnexion des différents nœuds des clusters est modélisé par un graphe connexe non orienté $G = (\mathbf{V}, \mathbf{E})$ où :

- L'ensemble des sommets **V** représentent les nœuds (ressources).
- L'ensemble des arrêtes **E** représentent les liens de communication.

La Figure 2.8 donne une illustration d'un modèle de graphe.

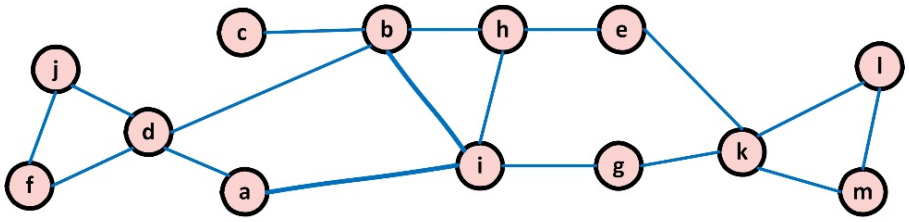


FIGURE 2.8 – Exemple d'un graphe

2.5.2 Techniques de clustering

Le clustering consiste à découper le réseau en groupes d'entités appelés clusters en donnant au réseau une structure hiérarchique. Chaque cluster est représenté par un nœud particulier appelé *cluster-head* (voir Figure 2.9). Ce nœud est élu comme *cluster-head* selon une métrique spécifique ou une combinaison de métriques telles que l'identifiant, le degré, le poids, la densité, etc.

Le *cluster-head* agit comme un coordinateur local dans son cluster. Un cluster est donc composé d'un *cluster-head* et de nœuds ordinaires appelés *cluster-membres*.

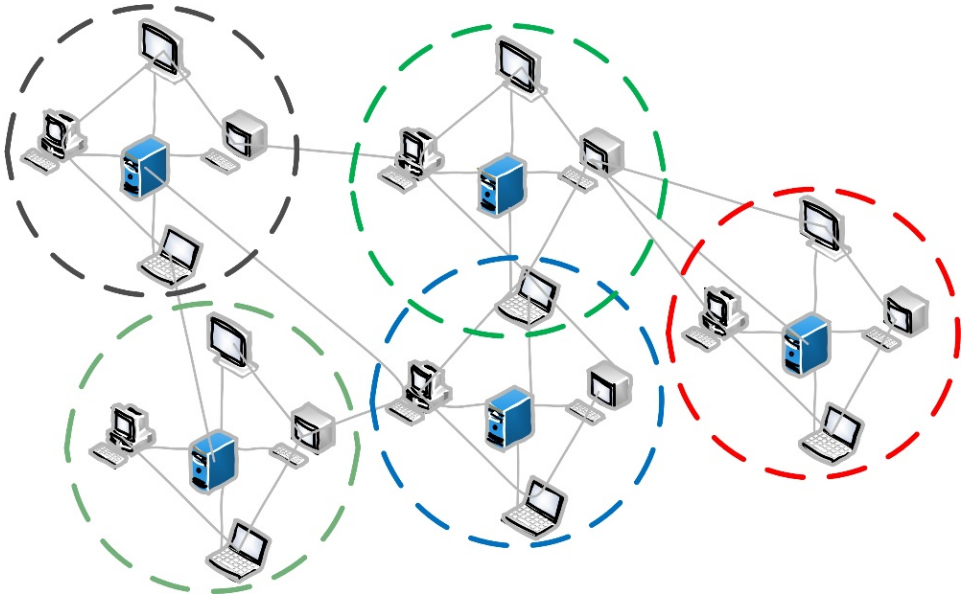


FIGURE 2.9 – Structure en clusters

2.5.2.1 Formation de clusters

Il existe plusieurs méthodes de formation de clusters. La plus répandue s'exécute comme suit :

1. Chaque nœud devra connaître son voisinage par le biais des messages "*Hello*",
2. Chaque nœud prend la décision selon sa connaissance locale de la topologie pour être *cluster-head* ou non,
3. Le nœud choisi comme *cluster-head* diffuse son statut dans son voisinage et invite ses voisins qui ne sont pas encore affiliés à d'autres clusters de le rejoindre.

2.5.2.2 Election de *cluster-head*

La phase d'élection de *cluster-head* appelée aussi la phase *Set-up* utilise une métrique spécifique ou une combinaison de métriques pour chaque nœud telle que le plus grand/petit ID dans son voisinage, le degré de connectivité (le nombre de voisinage à 1 saut), la puissance de transmission, l'énergie restante, ou bien un poids qui représente une combinaison de quelques métriques.

Nœuds	***a***	***b***	***c***	***d***	***e***	***f***	***g***	***h***	***I***	***g***	***k***	***L***	***m***
Degré	2	4	1	4	2	2	2	3	4	2	4	2	2

TABLEAU 2.1 – Degré des nœuds du graphe de la Figure 2.8

2.5.2.3 Communication intra-cluster et inter-cluster

Chaque *cluster-head* se charge des communications à l'intérieur de son cluster et maintient les informations de routage lui permettant de joindre les autres *cluster-heads*. De plus, comme les *cluster-heads* ne sont pas directement reliés, des nœuds passerelles (*gateways*) sont aussi élus et utilisés pour les communications entre *cluster-heads*.

2.5.2.4 Maintenance des clusters

Dans le but de s'adapter aux changements de la topologie du réseau, une mise à jour des clusters est dynamiquement réalisée dans le cas où un *cluster-head* ou un *membre* migre d'un cluster C_i à un autre C_j. D'autre part, si le *cluster-head* garde son statut le plus longtemps possible, même s'il ne possède pas le poids maximum dans son propre cluster alors il perdra son rôle une fois qu'il se déconnecte.

2.5.3 Avantages de clustering

Le principe du clustering consiste à organiser le réseau en une structure hiérarchique. Cette structure hiérarchique permet de :

- Optimiser la bande passante en minimisant la quantité d'information échangée afin de maintenir les tables de routage,
- Rendre la communication hiérarchique. L'idée de communication hiérarchique est d'établir des schémas de communication différents à l'intérieur des clusters (intra-clusters) et entre les clusters (inter-clusters).
- Redistribuer les ressources à travers le réseau.
- Optimiser la diffusion d'information afin de ne pas dégrader les performances du réseau. L'idée est de permettre à certains nœuds de relayer l'information afin qu'elle soit diffusée dans tout le réseau,
- Faciliter la réutilisation des ressources, cela permet d'améliorer la capacité du système.

2.5.4 Inconvénients de clustering

- Les clusters ne garantissent pas une protection contre les défaillances du stockage partagé ; les défaillances des services réseau ; les erreurs de fonctionnement ; les incidents de site (sauf si une solution de cluster dispersé sur le plan géographique a été mise en œuvre).
- Coût de maintenance de la structure élevé.
- Goulot d'étranglement au niveau des *cluster-heads*

2.5.5 Les algorithmes de clustering

Dans la littérature, il existe plusieurs solutions pour organiser un réseau en clusters. Toutes ces solutions sont destinées à identifier un sous ensemble de nœuds du réseau (appelés clusters). Les clusters sont identifiés par leur *cluster-head*. Les différents algorithmes se distinguent sur le critère de sélection des *cluster-heads*, c'est à dire la métrique. Cette métrique peut être une métrique spécifique comme l'identifiant, le degré, grand/petit ID, l'énergie, ... ou une combinaison de métriques. Il existe deux grandes catégories d'algorithmes de clustering :

2.5.5.1 Clusters à *1* saut

Les algorithmes de clustering à *1* saut (*1*-hop) construisent des clusters où chaque nœud du réseau est à distance *1* de son *cluster-head*, il existe de nombreux algorithmes de clustering à *1* saut (*1*-clusters) :

Ephremides, Weiselthier et Baker ont proposé dans (Ephremides, Wieselthier, & Baker, 1987) l'un des premiers algorithmes de clustering, il s'agit de l'algorithme de plus petit ID appelé aussi Linked Cluster Architecture (LCA).

Gerla et Tsai ont proposé dans (Gerla & Tsai, 1995) un algorithme de clustering appelé High Connectivity Clustering (HCC), cet algorithme est basé sur le degré de connectivité (nombre de voisins du nœud) pour construire des clusters au lieu des identités des nœuds.

2.5.5.2 Clusters à *k* sauts

Les algorithmes de clustering à *k* sauts (*k*-clusters) permettent de construire des clusters où chaque nœud est au maximum à *k* sauts (*k*-hop) de son *cluster-head*. Ces algorithmes sont souvent des extensions des algorithmes de clustering à *1* saut.

Dans (Ramalingam, Subramani, & Perumalsamy, 2002), les auteurs ont proposé d'utiliser une métrique particulière nommée "associativité". Le but est de privilégier les nœuds stables dans le choix du *cluster-head*.

Dans (Amis A. D., Prakash, Vuong, & Huynh, 2000), les auteurs ont proposé une heuristique appelé Max-Min *d*-cluster afin de construire des clusters à *d* sauts non-recouvrants, où *d* est le paramètre de l'heuristique. Le nombre de clusters construits est fonction du paramètre *d*, c'est à dire plus la valeur du paramètre *d* est importante, moins on aura de clusters. L'algorithme utilise l'identifiant du nœud pour le choix du *cluster-head*.

Dans (Amis & Prakash, 2000), les auteurs ont proposé une version améliorée de l'algorithme Max-Min *d*-cluster. Le but est d'apporter une équité entre les nœuds et donner ainsi à chaque nœud l'opportunité de servir comme *cluster-head*. Par exemple quand un nœud reste longtemps comme *cluster-head*, il épuise rapidement ses ressources (batterie par exemple). L'algorithme utilise l'identifiant virtuel (VID) comme métrique dans le choix du *cluster-head*.

Dans (Krishna, Vaidya, Chatterjee, & Pradhan, 1997), les auteurs ont proposé un algorithme générant des clusters à *k* sauts. Ils utilisent la distance entre les nœuds comme critère dans le choix du *cluster-head*.

Les auteurs de DDR (Nikaein, Labiod, & Bonnet, 2000) proposent également une structure sans *cluster-head*. Contrairement à la plupart des algorithmes de formation de *k*-clusters, les nœuds ne nécessitent que de la connaissance de leur *1*-voisinage.

Dans (Yu & Chong, 2003), les auteurs ont proposé un algorithme de clustering appelé 3-hop Between Adjacent Clusterheads (3hBAC), 3hBAC construit des clusters disjoints et impose 3 sauts entre deux *cluster-heads* de clusters adjacents, le critère du choix de *cluster-head* est le degré du nœud (nombre de voisins du nœud).

2.5.5.3 Clusters hiérarchiques

Il existe également des propositions de structures hiérarchiques à plusieurs niveaux, où les clusters sont ensuite regroupés avec d'autres clusters de niveaux supérieurs. Bien que la majorité des algorithmes peuvent être appliqués récursivement sur des clusters pour former des clusters de niveau supérieur.

Dans (Banerjee & Khuller, 2001), Banerjee et Khuller se basent sur un arbre couvrant, construit grâce à un parcours en largeur, pour la construction de *k*-clusters. Les clusters sont formés par branche, en fusionnant récursivement deux sous arbres de l'arbre couvrant jusqu'à obtenir une taille correcte. Le processus est alors réitéré jusqu'à obtenir le nombre de niveaux souhaité.

La structure de cellules hiérarchiques de SAFARI (Du, et al., 2008) est basée sur une auto-sélection des nœuds en tant que drums (*cluster-heads*). Le nombre de niveaux hiérarchiques s'établit automatiquement en fonction de la topologie sous-jacente des nœuds. Les clusters de niveau i sont groupés en clusters de niveau $i + 1$ et ainsi de suite, les simples nœuds étant considérés comme des cellules de niveau 0. Chaque *cluster-head* de niveau i se choisit un *cluster-head* de niveau $i + 1$. Tous les *cluster-heads* de niveau i ayant choisi le même *cluster-head* de niveau $i + 1$ appartiennent au même cluster de niveau $i + 1$. Un *cluster-head* u de niveau i décide de monter ou descendre son niveau en fonction du nombre de *cluster-heads* de niveau $i + 1$ et $i - 1$ qui existent à une certaine distance. S'il n'existe aucun *cluster-head* de niveau supérieur à une distance plus petite que D_i (D_i constante dépendant du niveau i du *cluster-head*) de u, u décide d'augmenter son niveau. Si deux *cluster-heads* de même niveau sont à moins de $h \times D_i$ ($0 < h < 1$, facteur d'hystérésis) sauts, le *cluster-head* de plus grand identifiant descend son niveau. Un *cluster-head* de niveau i est également *cluster-head* de tout niveau j tel que $0 < j < i$. Cet algorithme construit des *k*-clusters hiérarchiques, où k dépend du niveau du nœud i :

$k = D_i$. D_1 doit être fixé. À partir de là, D_i dépendant de D_{i-1}, le rayon des clusters de chaque niveau est fixé.

2.5.6 Applications

De nombreux travaux ont été réalisés autour de clustering de ressources dans la littérature. Selon les domaines d'application des clusters, on peut distinguer plusieurs types ou classes de clusters dont les principaux :

- La haute disponibilité (High Availability – HA) des clusters : améliorer la disponibilité du cluster.
- L'équilibrage de charge (Load Balancing – LB) des clusters : augmenter les performances des clusters.
- Le calcul haute performance (High Performance Computing – HPC) : augmentation de la performance par le fractionnement des tâches entre les différents nœuds.
- Le calcul à haut débit (High Throughput Computing – HTC) des clusters : maximiser le nombre de jobs exécutés.
- Tolérance aux fautes (Fault Tolerance – FT) : dans ce projet, on s'intéresse particulièrement à ce type de clusters.
- Accélérer le calcul d'un job distribué (Speed up the computation of one distributed job – FLOPS).

2.5.7 Clustering des ressources dans les systèmes distribués à lagre échelle

2.5.7.1 Clustering des ressources peer-to-peer

La mise en réseau peer-to-peer (P2P) s'intéresse principalement sur les problèmes d'évolutivité (scalabilité) inhérents à la distribution des ressources sur un grand nombre de processus en réseau (Sood, Kour, & Kumar, 2016). En raison de la large échelle et du manque de connaissances sur la structure complète du réseau au niveau de chaque pair (peer), un défi majeur dans la conception de système pair-à-pair (peer-to-peer) est d'effectuer efficacement la découverte de données et la recherche par les peers. La technique de clustering de réseaux peut faciliter considérablement ces opérations (Garcés-Erice, Biersack, Felber, Ross, & Urvoy-Keller, 2003; Kwon & Ryu, 2003). Le clustering P2P dispose de nombreuses applications telles que les systèmes de partage de fichiers P2P, les réseaux mobiles ad hoc, les réseaux de capteurs P2P, etc. (Li, Lao, & Cui, 2011). Cependant, il est très difficile de concevoir un protocole de clustering pour de tels réseaux car, par conception, il n'y a pas de point central d'administration ni même d'entité ayant une connaissance complète du réseau. De plus, les nœuds de ces systèmes ne peuvent communiquer qu'avec leurs voisins. Li et al. (Li, Lao, & Cui, 2006), ont proposé un protocole de clustering de réseau distribué appelé SCM-based Distributed Clustering (SDC), pour les réseaux peer-to-peer. Dans ce protocole, le clustering est ajusté dynamiquement sur la base de Scaled Coverage Measure (SCM) qui est une mesure de précision

de clustering pratique. En échangeant des messages avec les voisins, les pairs peuvent rejoindre ou quitter un cluster de manière dynamique afin d'améliorer la précision de clustering de l'ensemble du réseau. SDC est un protocole entièrement distribué qui ne nécessite que des informations sur les voisins. Il peut également gérer la dynamicité des nœuds localement avec une très faible surcharge (overhead) de messages tout en conservant une bonne qualité de clustering. Ramaswamy et al. (Ramaswamy, Gedik, & Liu, 2005), ont décrit un schéma de clusterisation de nœuds distribués basé sur la connectivité (CDC). Ce schéma présente une solution évolutive (scalable) et efficace pour découvrir les clusters basés sur la connectivité dans les réseaux P2P. Contrairement aux algorithmes de clustering de réseaux (graphes) centralisés, le schéma CDC est complètement décentralisé et il suppose uniquement la connaissance des nœuds voisins au lieu d'exiger une connaissance globale du réseau (graphe). Une caractéristique importante du schéma CDC est sa capacité à clusteriser automatiquement l'ensemble du réseau ou à découvrir des clusters autour d'un ensemble donné de nœuds. Pour faire face à la dynamicité typique des réseaux P2P, ils ont fourni des mécanismes permettant d'incorporer de nouveaux nœuds dans les clusters existants appropriés et de gérer gracieusement le départ des nœuds dans les clusters. Smail et al. (Smail, Cousin, & Snoussaoui, 2017), ont présenté une revue des protocoles de routage à chemins multiples (multipathes) et des protocoles de routage basés sur des techniques de clustering dans les réseaux ad hoc sans fil (protocole ES-CMR) afin d'utiliser efficacement l'énergie des nœuds et de minimiser le retard de bout en bout (end-to-end delay). Les auteurs utilisent une structure de clustering pour réduire la surcharge (overhead) de contrôle de routage.

2.5.7.2 Clustering des ressources utility computing

Utility computing est considérée comme la prochaine génération d'évolution des technologies de l'information qui décrit comment les besoins de calcul des utilisateurs peuvent être satisfaits dans la future industrie IT. Son analogie est dérivée du monde réel où les fournisseurs de services maintiennent et fournissent aux consommateurs des services publics (utility) tels que l'électricité, le gaz et l'eau (Mittal, Kesswani, & Goswami, 2013; Kaur & Rai, 2014; Kaur E. R., 2015).

2.5.7.3 Clustering des ressources de la grille de calcul

L'objectif du Grid computing est de permettre un partage coordonné des ressources et la résolution de problèmes dans des organisations virtuelles multi-institutionnelles et dynamiques (Mittal, Kesswani, & Goswami, 2013; Kaur & Rai, 2014). En règle générale, les techniques de clustering sont utilisées pour classer les ressources en fonction de leurs propriétés dynamiques dans les grilles. Gil et al. (Gil, Cho, & Hong, 2015), ont proposé un schéma de vérification des résultats avec un clustering des ressources qui peut tolérer les erreurs des résultats des tâches. Le schéma détermine également le nombre de réplications par tâche nécessaires pour la vérification des résultats sur la base des ressources classées par degré de confiance et par probabilité de renvoi des résultats. Les résultats de la simulation montrent que leur schéma de vérification des résultats peut réduire le temps d'exécution de tâches entières même si peu de ressources sont utilisées.

2.5.7.4 Clustering des ressources Cloud

Le Cloud Computing est un paradigme de calcul dans lequel les différentes tâches sont affectées à une combinaison de connexions, de logiciels et de services accessibles sur le réseau (Kaur & Rai, 2014; Cook, Robinson, Ferrag, Maglaras, & Helge Janicke, 2018). Les ressources et services de calcul peuvent être fournis et utilisés efficacement, rendant la vision de l'utilité de calcul réalisable. Malathy et al. (Malathy, Somasundaram, & Duraiswamy, 2013), ont proposé une approche de clustering des ressources qui utilise deux techniques complémentaires : les histogrammes d'utilisation des ressources pour fournir une assurance statistique des capacités des ressources et l'agrégation des ressources basées sur le clustering pour atteindre l'évolutivité (scalability). L'analyse a été effectuée à l'aide du simulateur CloudSim, dans lequel le nombre maximal de cloudlets utilisés dans le système Cloud est de 100 et le nombre de clusters de ressources est limité à 5. Ainsi, le système Cloud peut être utilisé efficacement et des améliorations supplémentaires peuvent être apportées pour la tolérance aux fautes dans le système.

2.6 Conclusion

Depuis quelques années, l'informatique distribuée est dans une évolution technologique et économique. Ces technologies sont de plus en plus matures et sophistiquées. La même observation peut être appliquée aux technologies réseaux qui tendent à devenir illimitée d'un point de vue applicatif. La réponse à ces changements est de passer à un modèle informatique distribuée permettant d'exploiter pleinement les ressources et de fournir de grandes capacités de calcul, nécessaires aux applications de calcul scientifique. Le clustering est imposé comme une solution particulièrement attractive en termes de rapport performance/coût et ont nettement modifié le panorama de calcul distribué. Dans ce chapitre, nous avons fait un survol sur les différentes techniques de clustering de tâches et de ressources existantes, leurs évolutions, puis, nous avons présenté les différents travaux de clustering depuis l'émergence des systèmes distribués. Ceci a conduit les chercheurs à faire des choix stratégiques, entre la mise à jour des anciennes techniques de clustering tolérantes aux fautes et à les adapter aux nouveaux systèmes ou le développement de nouvelles techniques répondant convenablement aux nouvelles contraintes.

Chapitre 3

Tolérance aux fautes dans les grilles de calcul par la clusterisation des ressources

Sommaire

3.1 Introduction

L'informatique décrit la façon dont les ordinateurs et les systèmes de calcul fonctionnent et comment ils sont construits et programmés. Ses principaux aspects de la théorie, des systèmes et des applications proviennent des disciplines de la technologie, de l'ingénierie, de la conception, des mathématiques, des sciences sociales et des sciences physiques. La popularité croissante d'Internet, la disponibilité de calculateurs puissants, les réseaux à haut débit et les composants de base à faible coût changent notre façon de traiter l'informatique. Au cours des années 1990 jusqu'en 2012, Internet a radicalement changé le monde de l'informatique. Tout a commencé par le calcul parallèle puis avancée vers le calcul distribué. Au fil des décennies, le calcul distribué a été une composante essentielle du calcul scientifique. Il comprend un ensemble de processus qui coopèrent pour atteindre un objectif spécifique commun (Sood, Kour, & Kumar, 2016). Cependant, la tendance dans ces systèmes massivement scalable est vers l'utilisation de peer-to-peer et d'utility computing. Ce dernier est essentiellement le Cloud et le grid computing (Kahanwal & Singh, 2012; Cook, Robinson, Ferrag, Maglaras, & Helge Janicke, 2018). Le grid computing (Baker, Buyya, & Laforenza, 2002) est une architecture distribuée à large échelle qui se compose de fédérations de clusters géographiquement distribuées rassemblant des milliers de nœuds sur Internet (Kulatunga, Argent-Katwala, & Knottenbelt, 2007). À cette échelle, les défaillances de nœuds et de réseaux ne sont plus des exceptions mais appartiennent au comportement normal du système. Ainsi, les applications de grille doivent tolérer les fautes et leurs évaluations doivent tenir compte de ces défaillances (Díaz, Pardo, Martín, & González, 2008; Zaharia, et al., 2012).

Dans la littérature, il existe différents types de fautes qui rendent un service de grille peu fiable (Yao, Li, & Lei, 2017). Ces fautes peuvent être divisées en trois catégories : fautes crash, fautes de déconnexion et les fautes byzantines. La tolérance aux fautes est une issue importante et complexe dans les systèmes de grille de calcul. Diverses techniques ont été étudiées pour détecter et tolérer les fautes des systèmes de calcul distribué (Jin, Zou, Chen, Sun, & Wu, 2003; Rajkumar & Swaminathan, 2016). Nous abordons le problème de la tolérance aux fautes dans les grille de calcul en développant un modèle basé sur les graphes colorés dynamiques pour la recherche de substituts du nœud défaillant. Notre contribution est double, la première est la proposition pour chaque nœud défaillant dans la grille deux ensembles de substituts (identiques et plus performants), suivant les valeurs des attributs du vecteur d'état de chaque nœud et la deuxième est que nous tolérons les fautes sans aucune réplication et sans aucune restructuration des ressources de la grille. Nous résumons les contributions dans les points suivants:

(i) Nous modélisons une grille de calcul sous forme d'un graphe coloré dynamique dans lequel chaque nœud de la grille est décrit par deux classes de propriétés (attributs) : capacités de nœud (mémoire, espace disque disponible, vitesse CPU, charge CPU, type de processeur) et propriétés de QoS (nombre de fautes, nombre de déconnexions, MTBF, MTTR). Ses valeurs peuvent être statiques ou dynamiques, ce qui est plus réaliste (Rebbah M., Slimani, Benyettou, & Brunie, 2016). Dans notre technique de coloration, nous colorons chaque attribut avec trois couleurs (vert, bleu et rouge) en

fonction de son niveau de performance (performant, plus performant et moins performant) en utilisant une fonction de *scoring* proposée.

(ii) Notre technique de tolérance aux fautes détermine pour chaque nœud défaillant trois ensembles de substituts (identiques, plus efficaces et moins efficaces) en fonction de leurs performances en utilisant la fonction de *scoring*.

(iii) Notre modèle utilise une technique de clustering des ressources pour déterminer de manière optimale le choix des substituts les plus proches du nœud défaillant par une tolérance aux fautes intra-cluster. Si le résultat n'est pas suffisant, nous complétons l'ensemble des substituts par une tolérance aux fautes inter-cluster.

(iv) Notre modèle ne nécessite pas de technique de réplication des ressources de calcul.

Le reste du chapitre est organisé comme suit : La section 2 présente quelques travaux connexes dans les domaines de la tolérance aux fautes dans les grilles de calcul. Le modèle de grille est introduit dans la section 3. La section 4 décrit la technique de tolérance aux fautes proposée. Dans la section 5, les résultats expérimentaux de notre modèle proposé sont présentés et discutés. Enfin, la section 6 conclut ce chapitre et donne quelques perspectives de recherche.

3.2 Travaux connexes

La prise en charge du développement d'applications tolérantes aux fautes a été identifiée comme l'un des principaux défis techniques à relever pour réussir le déploiement de grilles de calcul (Bertier, Marin, & Sens, 2003). Les calculateurs d'une grille ne doivent pas nécessairement avoir le même système d'exploitation ou les mêmes caractéristiques matérielles. Les grilles sont généralement peu connectées, souvent dans un réseau décentralisé, plutôt que d'être localisées dans un seul emplacement, comme le sont souvent des nœuds d'un cluster. L'organisation décentralisée surmonte les limites des organisations centralisées ou hiérarchiques en ce qui concerne la tolérance aux fautes, la scalabilité et l'autonomie (Rebbah, Slimani, & Benyettou, 2014). Un certain nombre de chercheurs ont étudié comment les services de gestion de réplication de données décentralisés peuvent être utilisés pour améliorer la tolérance aux fautes dans les grilles de données. Chervenak et al. (Chervenak, et al., 2009), ont décrit un service de localisation de réplique décentralisé pour Globus toolkit en distribuant des index redondants. Cela a été fait pour maintenir les informations sur les répliques de données de manière cohérente. Zhang et al. (Zhang, et al., 2006), ont proposé un algorithme pour localiser dynamiquement les serveurs de répliques de données dans une grille afin d'optimiser les performances et d'améliorer la tolérance aux fautes, bien que les problèmes de scalabilité n'aient pas été examinés. Abbes et al. (Abbes & Crin, 2010), ont proposé d'éliminer la nécessité d'un serveur centralisé, afin de supprimer ainsi le point de défaillance unique et le goulot d'étranglement des Desktop Grids existantes. Plutôt, chaque calculateur peut jouer alternativement le rôle de client ou de serveur. Ils ont conçu le protocole PastryGrid pour la Desktop Grid afin de prendre en charge une classe plus large d'applications, en particulier pour les applications distribuées avec priorité entre les tâches. Aliaa et al. (Aliaa, Atef, & Mohammed, 2010), décrivent un système de gestion de ressources basé sur les agents, ARMS, qui comprend une stratégie de réclamation de service décentralisée adaptable pour réduire le

coût du processus de réclamation au sein du système ARMS. Une composante de contrôle de performance est mise en place pour comparer la stratégie développée avec celles définies dans la littérature. Cette stratégie est plus adaptable aux changements continus de la charge de travail (workload) et de la structure.

La sélection des nœuds pour une application tolérante aux fautes est importante et défiante. Afin de trouver le substitut approprié pour un nœud défaillant, (Rebbah M., Slimani, Benyettou, & Brunie, 2016), ont proposé un modèle tolérant aux fautes pour les systèmes de grille de calcul, nommé DCFT. Ce modèle est basé sur un graphe coloré dynamique pour montrer à la fois la structure et la dynamicité d'une grille sans technique de réplication ni restructuration des composants d'une grille. Le modèle de tolérance aux fautes proposé comprend deux étapes. Dans la première étape, chaque nœud est décrit par un vecteur d'état. En outre, chaque attribut du vecteur d'état est coloré en trois couleurs (vert, bleu et rouge) en fonction de son niveau de performance. Dans la deuxième étape, les nœuds d'une grille sont classés en trois catégories : les ressources de calcul identiques en termes de performances, les plus performantes et les moins performantes. Les couleurs des nœuds sont utilisées pour développer une nouvelle stratégie de tolérance aux fautes basée sur le niveau de performance.

Dans (Gil, Cho, & Hong, 2015), les auteurs ont présenté un schéma de vérification des résultats basé sur le clustering des ressources afin de prendre en charge efficacement la vérification des résultats, qui est essentielle pour garantir l'exactitude des résultats des tâches des Desktop Grids. Les erreurs survenues dans les Desktop Grids peuvent être regroupées en deux causes : internes et externes. Les causes internes incluent les erreurs d'E/S survenant sur un disque local, comme le stockage des résultats des tâches, les problèmes de précision dus aux ressources hétérogènes de différents types de CPU et de système d'exploitation, et les erreurs de calcul du CPU dues à l'overclocking. Une cause externe typique est une altération de résultat créée intentionnellement par des ressources malveillantes. Le schéma proposé peut tolérer les erreurs de résultats de tâche et peut également déterminer le nombre de réplications par tâche nécessaires pour la vérification des résultats. Dans cette partie, nous développons une approche de tolérance aux fautes pour les nœuds défaillants dans un environnement de grille de calcul par clustering de ressources. Ce modèle tente de traiter les fautes crash et les fautes de déconnexion des nœuds sans réplication des ressources de calcul en se basant sur l'algorithme de clustering à *1*-saut HCC (*High Connectivity Clustering*) proposé par Gerla et Tsai (Gerla & Tsai, 1995).

3.3 Modèle de grille

Le modèle proposé tente de tolérer les fautes dans une grille de calcul, par un choix pertinent du nœud substitut, en cas de défaillance d'un nœud de la grille. Ce choix est basé sur le calcul de niveau de performance de chaque nœud dans la grille. En cas de défaillance d'un nœud dans la grille, on trouve le substitut le plus adéquat à le remplacer et continuer à délivrer le service (Rebbah M., Slimani, Benyettou, & Brunie, 2016).

3.3.1 Modèle de graph

Nous modélisons toute grille de calcul par un graphe non dirigé *G(X,U)*, où les sommets $X=\{1,2,\ldots,|X|=n\}$ représentent l'ensemble des ressources (nœuds) de la grille ; et les arêtes $U=\{u_1, u_2, \ldots, u_m\}$ représente les *m* connexions (liens) entre les ressources (nœuds) de la grille ; chaque arête $u_i \in U$ est attribué d'un poids non-négatif. Dans notre modèle, ces poids représentent la bande passante. La distance entre deux nœuds *x* et *y*, *d(x, y)*, est la somme des poids des arêtes le long d'un plus court chemin entre *x* et *y*. Pour représenter cette dynamicité, on note par Pour prendre en compte la dynamicité des noeuds de la grille, le graphe *G* sera dynamique, c.à.d. à tout instant *t*, le graphe peut subir des adjonctions et/ou des suppressions des sommets ou des arêtes. Pour représenter cette dynamicité, on note par $G_t(X_t, U_t)$, l'ensemble des sommets et arêtes présents à l'instant *t.* La fonction $Voisin_t(x_i)$ renvoie tous les sommets adjacents (voisins) du sommet x_i à l'instant *t*. Pour considérer l'hétérogénéité des nœuds, on associe à chaque sommet du graphe une couleur indiquant le type du nœud. Nous proposons deux définitions de graphe coloré de grille qui ne tiennent pas en compte la dynamicité (Définition 1) et une définition globale de graphe coloré de grille dynamique qui résume toutes les caractéristiques d'une grille (Définition 2) (Rebbah M., Slimani, Benyettou, Meftah, & Brunie, 2015).

Définition 1 : Graphe coloré
Un graphe coloré *G(X, U, C, E)* est un quadruplet défini comme suit :
X : ensemble fini non vide qui représente les sommets.
$U : X \rightarrow X$ *:* ensemble des arêtes.
C : ensemble non vide des couleurs.
$E : X \rightarrow C$ *:* ensemble des couleurs attribuées à chaque sommet.

Définition 2 : Graphe dynamique coloré
Un graphe dynamique coloré ξest défini par un triplet (G_0, T, P) comme suit :
$G_0(X, U, C, E)$ *:* un graphe coloré statique initial (au temps 0).
T : est une base temporelle continue ou discrète.
P : est un processus d'évolution.

3.3.2 Paramètres des noueds

Chaque nœud de la grille possède deux classes de propriétés (paramètres) : Les capacités de nœud et les propriétés non fonctionnelles liées à la qualité de service. Les valeurs des attributs des propriétés statiques sont fixées, telles que la configuration matérielle prédéfinie par le fabricant, par rapport aux valeurs des attributs des propriétés dynamiques qui varient dans le temps et en fonction de l'état de la ressource.

Nous sommes intéressés par certains paramètres fonctionnels (statiques : mémoire, vitesse CPU, type de processeur ; dynamiques : espace disque disponible, charge CPU) et certains paramètres de qualité de service (statiques : MTBF, MTTR ; dynamiques : nombre de fautes, nombre de déconnexions), pour chaque attribut, nous définissons un intervalle de valeur moyenne (voir Tableau 3.1).

Attribut	Type	Description	Valeur moyenne
C1 : Capacités du nœud			
Mémoire	S		2 GB-4 GB
Espace disque disponible	D		100 GB-300 GB
Vitesse CPU	S		2 Ghz-3 Ghz
Charge CPU	D		10-20
Type de processeur	S	Nombre de cœurs CPU	2-5
C2 : Propriétés non fonctionnelles (QoS)			
Nombre de fautes	D	0 très instable; 1 instable; 2 stable; 3 très stable	
Nombre de déconnexions	D	0 très instable; 1 instable; 2 stable; 3 très stable	
MTBF : Mean Time Between Failure	S		10-20
MTTR : Mean Time To Repair	S		10-20

TABLEAU 3.1 – Propriétés de la grille (S: statique, D: dynamique)

3.4 Tolérance aux fautes

La tolérance aux fautes dans les grilles de calcul est basée généralement sur la recherche d'un ou plusieurs substituts capables de remplacer le nœud défaillant, l'efficacité du mécanisme de tolérance aux fautes dépend généralement, de la qualité du choix des répliques plus qu'elle dépend de la technique de tolérance aux fautes mises en œuvre (Jin, Zou, Chen, Sun, & Wu, 2003; Chandrasekaran & Vaideeswaran, 2014). Dans ce papier, nous proposons un mécanisme de tolérance aux fautes basé sur des critères de choix des répliques qu'on appelle des substituts. Notre approche transforme une grille en un ensemble de clusters interconnectés et utilise plus tard une fonction de *scoring* pour déterminer le substitut approprié pour chaque nœud défaillant en calculant le niveau de performance de chaque nœud (performant, moins performant, plus performant). Le substitut doit être identique ou plus fiable que le nœud défaillant. Si le substitut appartient au même cluster du nœud défaillant, on parle ici d'une tolérance aux fautes Intra Cluster. Sinon le substitut est un membre d'un autre cluster voisin, ce type est appelé tolérance aux pannes inter-cluster.

3.4.1 Clusterisation

Le clustering consiste en une division virtuelle (découpage) du réseau en groupes de nœuds géographiquement proches les uns des autres. Ces groupes sont des clusters. Ils sont généralement identifiés par un chef particulier, un chef de groupe également appelé "*cluster-head*". Dans la plupart des algorithmes de clustering, les clusters sont construits à partir d'une métrique particulière qui attribue un chef à chaque nœud. Le cluster est alors composé du nœud chef et de tous les nœuds qui lui sont attachés (Wang & Bao, 2007).

De nombreuses solutions de clustering ont été proposées. La majorité d'entre eux propose l'utilisation d'une métrique qui permet aux nœuds de choisir un chef (coordinateur). Cette métrique peut être, par exemple, l'identifiant ou le degré des nœuds, une valeur de mobilité des nœuds, ou une somme pondérée de tous ces éléments. D'autres solutions cherchent d'abord à déterminer un ensemble dominant connecté sur lequel les clusters sont construits. Une grande partie des solutions de clustering crée des clusters à *1*-saut (appelés *1*-clusters), c'est-à-dire des clusters où chaque nœud se trouve à un saut de son chef de cluster. Les protocoles donnant naissance aux *k*-clusters (clusters où chaque nœud est au maximum à *k*-sauts de son *cluster-head*) sont plus récents et plus rares (Chatterjee, Das, & Turgut, 2002).

Le modèle proposé lance une phase de construction des clusters basé sur l'algorithme de clustering à *1*-saut HCC (*High Connectivity Clustering*) proposé par Gerla et Tsai (Gerla & Tsai, 1995). Cet algorithme est basé sur le degré de connectivité (nombre de voisins du nœud) pour construire des clusters. Un nœud est élu *cluster-head*, s'il a la plus haute connectivité parmi tous ses voisins à *1*-saut. Si deux nœuds ont le même degré de connectivité, alors le nœud ayant le plus petit identifiant deviendra *cluster-head*. L'algorithme HCC génère un nombre réduit de clusters puisqu'il favorise les nœuds ayant le plus fort degré d'être *cluster-heads*, c'est à dire les nœuds qui couvrent plus de nœuds voisins. Mais cet algorithme souffre des changements fréquents de *cluster-heads*, à cause du critère de sélection du *cluster-head*. Ainsi dans cet algorithme, les *cluster-heads* ne sont pas susceptibles de garder leur statut très longtemps puisque leurs degrés changent fréquemment. Cet algorithme construit à la fois des clusters recouvrants (c'est à dire qu'il existe des nœuds du réseau qui appartiennent à 2 clusters où plus simultanément appelées nœud de passage, passerelles où *gateway*) et non-recouvrants. Dans l'algorithme HCC, l'opération de maintenance des clusters est très coûteuse parce que la défaillance d'un nœud peut détruire la structure et nécessite une reconstruction entière de la structure (voir Figure 3.1).

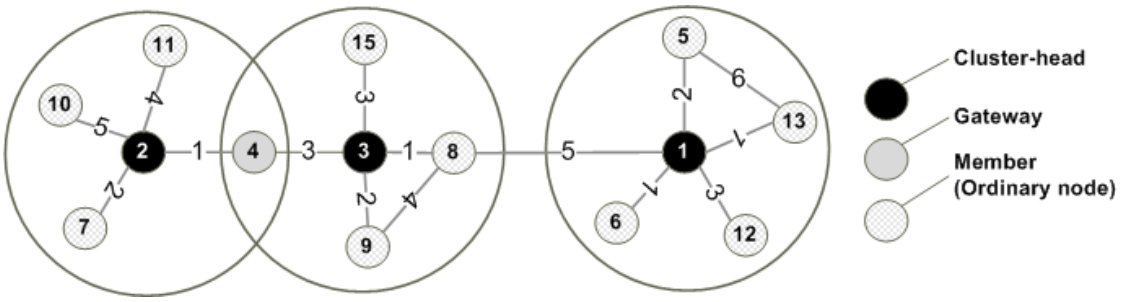


FIGURE 3.1 – Construction de clusters avec HCC

3.4.2 Scoring (Règle de coloration des somets)

Notre objectif est de trouver pour chaque nœud de la grille qui tombe en panne des substituts capables de le remplacer, la détermination des substituts dépend d'un ensemble d'attributs (propriétés). On observe pour chaque ressource de la grille un ensemble de *k* attributs $A=\{a_1, a_2, \ldots, a_k\}$ (voir Tableau 3.1) ; les valeurs de ces attributs peuvent être statiques ou changent de valeur suivant l'état de la ressource (dynamique), chaque ressource possède un vecteur d'état $V_t(x)= [va_1, va_2, \ldots, va_k]$ représentant les valeurs des attributs *A* à l'instant *t*. Un intervalle de

valeurs possibles $I_i = [v_{i\,min} .. v_{i\,max}]$ est défini au préalable pour chaque attribut a_i, les valeurs $v_{i\,min}$,$v_{i\,max}$ sont considérées à partir de la configuration existante des ressources de la grille. Ensuite, on va calculer le *score* de performance (*S*%) pour chaque ressource selon la valeur de chaque attribut et l'intervalle de valeurs possibles. Ce dernier (*score*) est utilisé pour décider (juger) la performance de la ressource en définissant un intervalle de performance $P=[s_{min}\%, s_{max}\%]$, les valeurs s_{min}, s_{max} sont estimées en fonction des ressources disponibles de la grille afin d'avoir un gain maximum en termes de ressources.

$$S_i(Node_x) = \frac{[v_i(Node_x) - v_{i\,min}]}{v_{i\,max} - v_{i\,min}} \times 100\% \qquad (3.1)$$

$S_i(Node_x)$: Le score de l'attribut *i* du nœud *x*,

$I_i=[v_{i\,min}, v_{i\,max}]$: L'intervalle de valeurs possibles pour l'attribut *i*,

$v_i(Node_x)$: La valeur de l'attribut *i* du nœud *x*.

$$S(Node_x) = \frac{\sum_1^k S_i(Node_x) \times w_i}{\sum_1^k w_i} \qquad (3.2)$$

$S(Node_x)$: Le score du nœud *x*,

k: Le nombre d'attribut pour chaque nœud,

w_i: La pondération (weighting) de l'attribut *i*.

Si $S(Node_x) \in P=[s_{min}\%,s_{max}\%]$, le nœud est performant *(performing)* on le colore par la couleur *verte*,

Si $S(Node_x) < s_{min}$, le nœud sera considéré comme moins performant *(less performing)* et on le colore en *rouge*,

Si $S(Node_x) > s_{max}$, le nœud sera considéré comme plus performant *(more performing)* et on le colore en *bleu*.

Exemple :

$Node_x$: $node_0$:

A = {mémoire, espace disque disponible, vitesse CPU, charge CPU, …, …},

$I_{mémoire}=[2GB, 4GB]$,

$I_{disque}=[100GB, 300GB]$,

……

$V(node_0)= [3GB, 250GB, 2.4 Ghz,…]$

Pour une pondération $W_i=1$ de chaque attribut

$S_{mémoire}(node_0)= [(3-2) / (4-2)]*100\% = 50\%$

……

$S(node_0)=56\%$

Pour $P=[40\%,60\%]$, $S(node_0) \in P$ => $node_0$ est performant (couleur vert)

Nous modélisons la grille par un graphe dynamique coloré où chaque sommet *x* possède un vecteur d'état $V_t(x)$ composé de *k* attributs coloré par trois couleurs de base (*rouge*, *vert*, *bleu*) suivant le *score* par rapport à l'intervalle de performance prédéfini *P* (voir Figure 3.2).

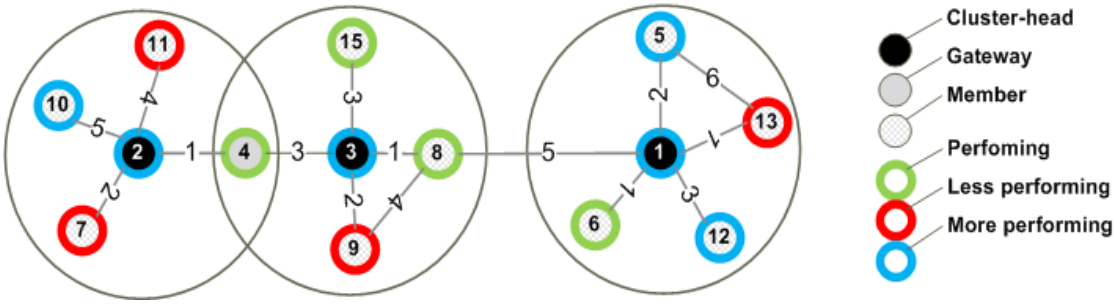


FIGURE 3.2 – Graphe dynamique coloré clusterisé

3.4.3 Classes de substituts

La tolérance aux fautes est basée sur la recherche des substituts capables de remplacer le nœud défaillant. Les substituts sélectionnés sont offerts en trois critères :

- ***Rapidité*** : Les substituts offerts tiennent en compte d'offrir les premiers substituts libres trouvés.
- ***Performance*** : On cherche les substituts ayant des caractéristiques plus proches à celui du nœud défaillant.
- ***Qualité de Service*** : On ajoute aux substituts sélectionnés, ceux ayant des bonnes QoS, comme fréquence de défaillance, fréquence de déconnexion, MTTR, MTBF, …

Le module de tolérance aux fautes assure la continuité de service en sélectionnant les substituts en trois niveaux par le calcul du niveau de performance de chaque nœud en utilisant la fonction de *scoring* :

- ***Substituts identiques*** : Représentent les substituts ayant des performances identiques au nœud défaillant (*performant*/*performant*, *moins performant*/*moins performant*, *plus performant*/*plus performant*), ces derniers sont les plus appropriés à tolérer la faute,
- ***Substituts plus performants*** : Sont en général de qualité supérieure (plus fiable) au nœud défaillant (*moins performant*/*performant*, *moins performant*/*plus performant*, *performant*/*plus performant*).
- ***Substituts moins performants*** : Sont en général de qualité inférieure au nœud défaillant, on sélectionne le minimum des substituts pouvant tolérer la faute en cas d'inexistence des substituts identiques ou plus performants.

3.4.4 Sélection des substituts

On suppose qu'il y a un détecteur de fautes fiable pour chaque cluster au niveau de *cluster-head* (Xia, Jiang, Sun, & Yang, 2011; Ahn, 2007). Il fait deux fonctions complémentaires : la première est de s'assurer que le nœud est toujours opérationnel ; la deuxième est dans le cas où

le nœud tombe en panne, le détecteur de fautes déclenche la fonction de tolérance aux fautes (les nœuds envoient au *cluster-head* la liste des jobs à tolérer).

3.4.4.1 Tolérance aux fautes Intra-Cluster

La technique de tolérance aux fautes proposée sera appliquée au niveau des clusters. Si un nœud tombe en panne, la faute sera détectée par son *cluster-head.* Ce dernier va chercher un ou plusieurs substituts localement dans son cluster, capables de le remplacer où ils acceptent de ré-exécuter ses jobs (c.à.d. un substitut ayant un niveau de performance identique au nœud défaillant sinon plus performant). Une fois les substituts trouvés, le *cluster-head* migre les jobs du nœud défaillant aux substituts.

3.4.4.2 Tolérance aux fautes Inter-Cluster

Dans le cas où il est impossible de trouver un substitut dans le même cluster, nous cherchons un complément des substituts dans les clusters voisins les plus proches (en se basant sur le plus court chemin entre les *cluster-heads*), nous favorisons alors les substitues ayant un niveau de performance identique sinon plus élevé.

Si le nœud défaillant est de type *cluster-head,* alors une phase de maintenance (mise à jour) sera déclenchée. Nous obtiendrons une nouvelle structure avec différents clusters (élection de nouveaux *cluster-heads*), les nœuds de la grille peuvent garder l'état (*cluster-head*, *gateway*, *member*) ou changent l'état (*member* devient un *gateway*, *gateway* devient un *member*, …).

3.5 Expérimentations

Notre infrastructure de simulation est créée par le développement de notre simulateur de tolérance fautes en Java, que nous avons nommé FT-GRC. Ce dernier intègre la bibliothèque *GraphStream* (GraphStream A Dynamic Graph Library, 2020), dédiée à la création et la manipulation des graphes colorés dynamiques utilisés dans les simulations. Nous évaluons donc les performances de notre modèle à travers une simulation axée sur des événements discrets. Ensuite, nous mesurons son comportement en régime transitoire en tenant en compte la structure des clusters, les classes des substituts, le nombre des jobs défaillants et le niveau de tolérance. L'influence du degré moyen des nœuds, de l'intervalle de performance, du taux d'injection de fautes et du nombre maximum de jobs par nœud sur les résultats obtenus est analysée et discutée.

3.5.1 Résultats graphiques

Chaque série de simulation est effectuée sur un graphe coloré dynamique clusterisé qui modélise la grille de calcul, généré de manière aléatoire comme suit : étant donné un ensemble de nœuds (sommets) et un nombre de liens (arêtes) ajoutées entre les paires de nœuds selon un degré moyen des nœuds, et les poids (pondérations) des arêtes sont sélectionnés au hasard suivant un intervalle de valeurs prédéfini (voir Figure 3.3).

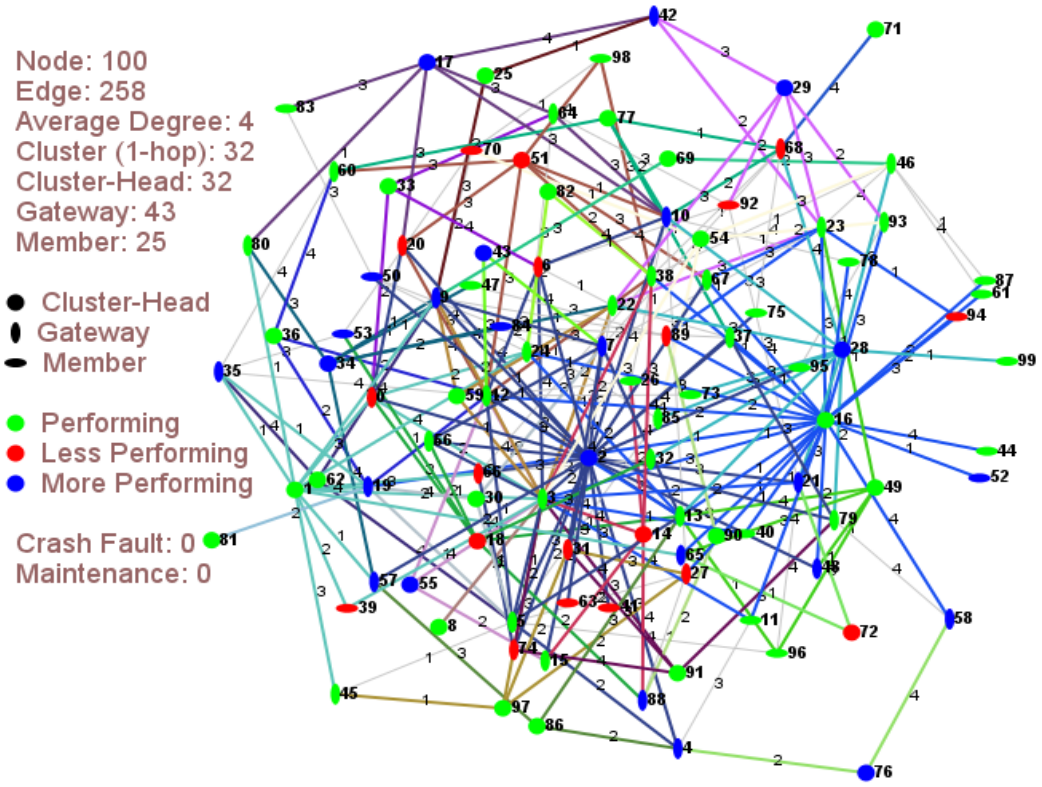


FIGURE 3.3 – Génération de graphe

3.5.1.1 Configuration expérimentale

Nous utilisons dans notre simulation une configuration de grille de calcul composée de 180 nœuds. Pour chaque nœud de la grille, nous considérons deux classes de propriétés (voir Tableau 3.1) : les capacités des nœuds et les propriétés de *QoS*. Cependant, ces caractéristiques individuelles et la topologie d'interconnexion varient d'un nœud à l'autre. Nous supposons que tous les systèmes d'un même site partagent un réseau local et que chacun de ces nœuds est connecté à tous les autres réseaux locaux à l'aide d'une connexion point à point.

Nous générons un ensemble de jobs avec les paramètres : le nombre maximum de jobs par nœud (*Max_Job_by_Node*), l'heure d'arrivée et le temps d'exécution du job. Afin de générer différentes charges de travail (*workloads*), nous varions ces paramètres pour des intervalles de temps différents. Le temps d'exécution de chaque job est sélectionné uniformément au hasard avec 12 s en moyenne et le nombre maximum de jobs par nœud (*Max_Job_by_Node*) est sélectionné de manière aléatoire avec 20 en moyenne. Ainsi, le nombre de jobs moyen augmente à 3600.

Pour évaluer l'efficacité de notre technique de tolérance aux fautes, nous définissons un paramètre de taux d'injection de fautes (*Fault_Injection_Rate*). Pour chaque série de simulation, le *Fault_Injection_Rate* est sélectionné ente 10% et 40%. Ainsi, le nombre de jobs défaillants passe de 360 (c'est-à-dire *Fault_Injection_Rate* = 10%) à 1440 (c'est-à-dire *Fault_Injection_Rate* = 40%).

3.5.1.2 Clustering

Nous faisons varier le nombre de nœuds du graphe de 20 jusqu'à 180 avec un degré moyen *Average_Degree* = 5 et nous calculons pour chaque cas le nombre moyen de nœuds en tant que *cluster-heads*, *gateways* et *members*. Nous avons constaté que le nombre de nœuds *gateways* est toujours supérieur aux nœuds *cluster-heads* et aux nœuds *members*, tandis que les nœuds *members* sont toujours inférieurs aux autres classes. Le nombre de *gateways* et de *members* est suffisant pour exécuter les jobs soumis (voir Figure 3.4).

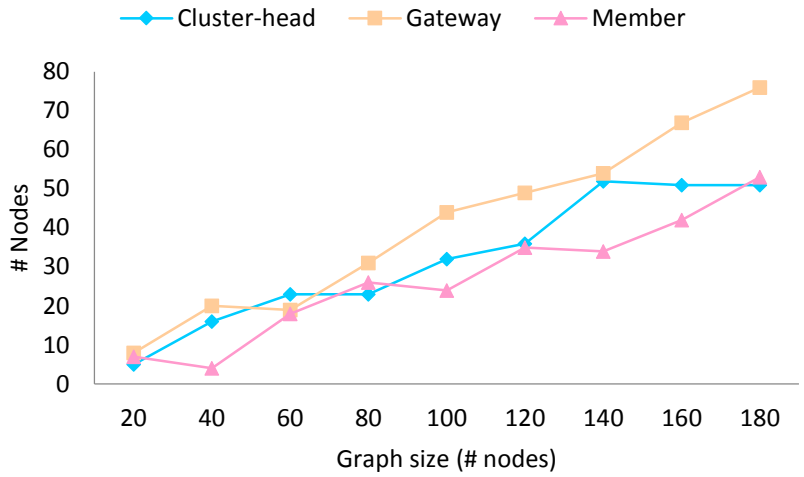


FIGURE 3.4 – Clusterisation des nœuds

3.5.1.3 Classification (*scoring*)

Dans cette série de tests, nous avons augmenté le nombre de nœuds de 20 jusqu'à 180 avec un degré moyen *Average_Degree* = 5 et un intervalle de performances *P* = [40%, 60%]. Sur la base des propriétés fonctionnelles et non fonctionnelles, nous distinguons pour chaque nœud trois classes : performing (*performing*), moins performant (*less performing*) ou plus performant (*more performing*). Les résultats obtenus montrent que les nœuds les plus performants et les moins performants restent toujours proches les uns des autres, tandis que les nœuds performants sont toujours plus supérieurs. Ce surplus de nœuds performants sera très utile pour tolérer les fautes des nœuds moins performants et plus performants (voir Figure 3.5).

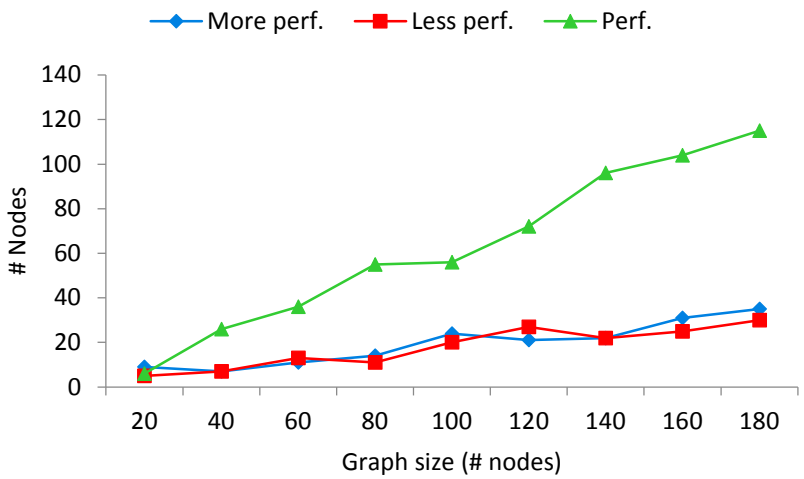


FIGURE 3.5 – Classification des nœuds

3.5.1.4 Tolérance aux fautes par rapport au clustering

Dans cette série d'expérimentations, nous faisons varier le nombre de nœuds dans le graphe avec une capacité *Max_Job_by_Node* = 20 et un taux d'injection de fautes *Fault_Injection_Rate* = 10%. Nous calculons le nombre de jobs défaillants pour être tolérer plus tard. Après la tolérance aux fautes des jobs, nous calculons la tolérance aux fautes intra-cluster et inter-cluster (voir Figure 3.6). Dans les mêmes conditions, nous faisons varier le nombre de nœuds dans le graphe avec une capacité *Max_Job_by_Node* = 20 et un *Fault_Injection_Rate* = 40% (voir Figure 3.7). Nous avons comparé le type de tolérance aux fautes obtenu par rapport au taux d'injection de fautes. Nous avons constaté que le taux d'injection de fautes augmente la tolérance aux fautes inter-cluster, ce qui est généralement coûteux.

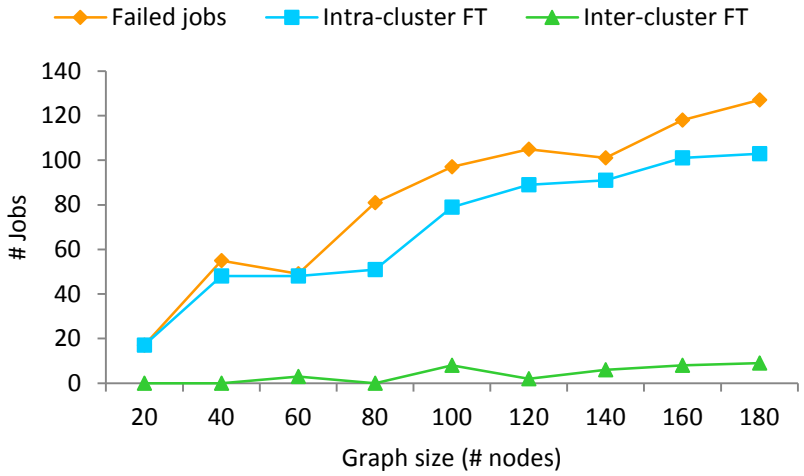


FIGURE 3.6 – Tolérance aux fautes intra-cluster et inter-cluster en fonction du nombre de nœuds (*Fault_Injection_Rate* = 10%)

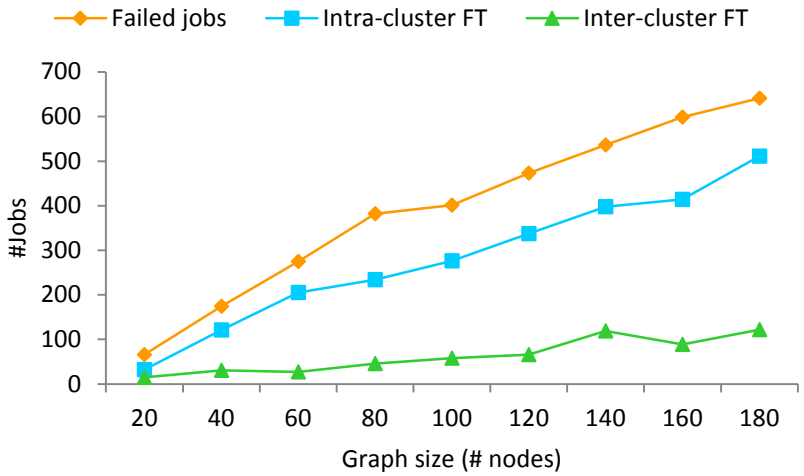


FIGURE 3.7 – Tolérance aux fautes intra-cluster et inter-cluster en fonction du nombre de nœuds (*Fault_Injection_Rate* = 40%)

3.5.1.5 Tolérance aux fautes par rapport à la classification (scoring)

Nous faisons varier le nombre de nœuds dans le graphe avec un degré moyen *Average_Degree* = 5, un intervalle de performances *P* = [40%, 60%], une capacité *Max_Job_by_Node* = 20 et un taux d'injection de fautes *Fault_Injection_Rate* = 10%. Lorsqu'un nœud tombe en panne, il peut être toléré par un nœud qui appartient à la même classe (identique) ou à une classe plus performante ou moins performante au pire des cas. Dans cette expérimentation, nous nous intéressons à ces classes de tolérance aux fautes. La majorité des tolérances aux fautes sont dans le même niveau de performance (identique), ce qui permet un gain des qualités de ressources de la grille et permet de sélectionner le nœud approprié (voir Figure 8 et Figure 9).

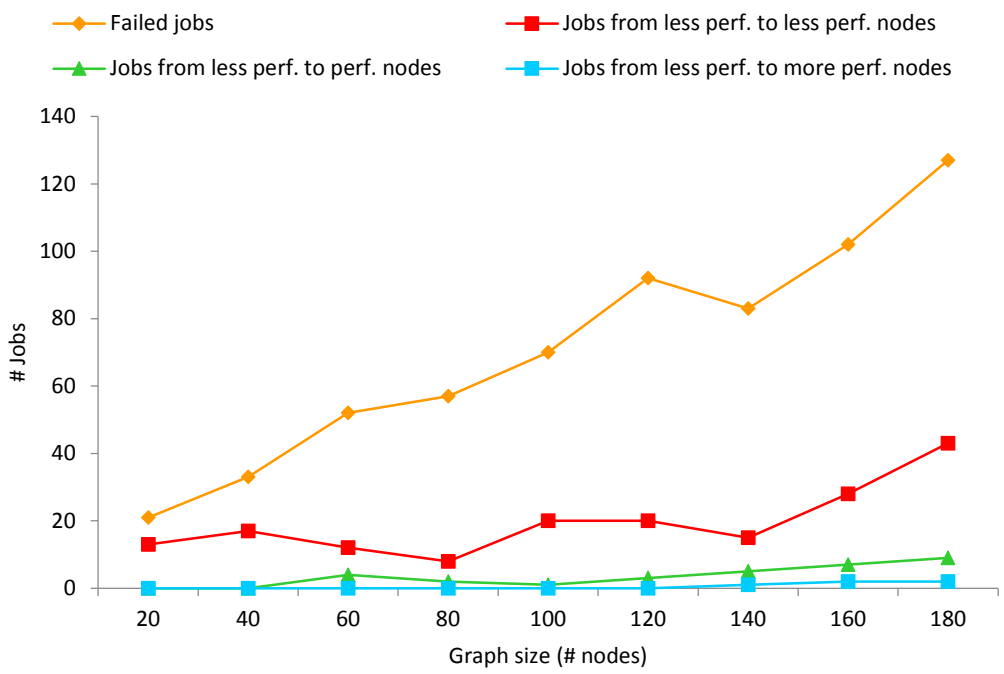


FIGURE 3.8 – Nombre de jobs défaillants migrés à partir de nœuds moins performants

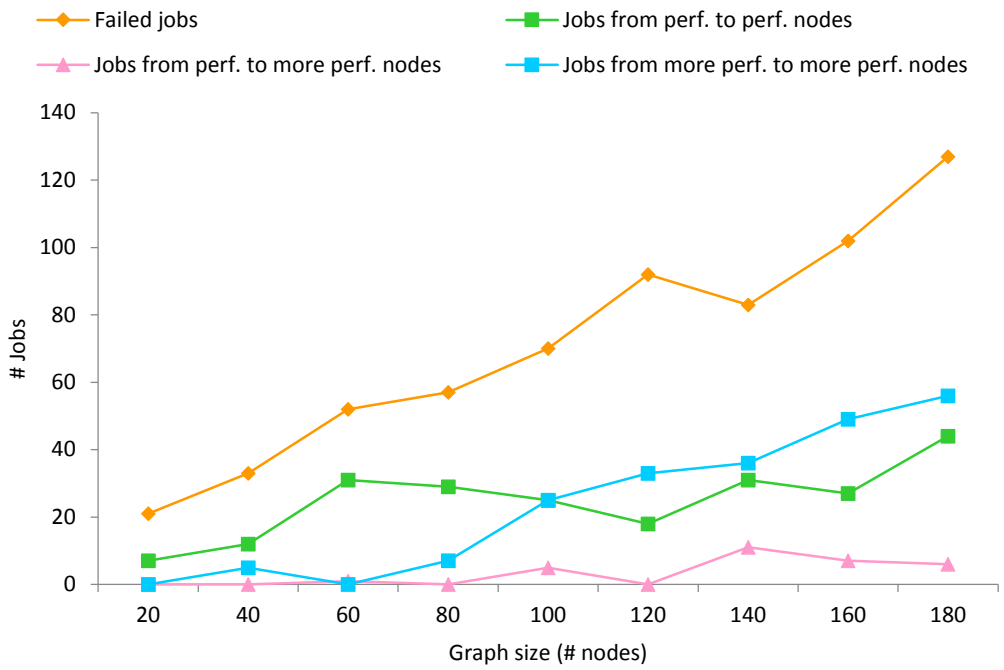


FIGURE 3.9 – Nombre de jobs défaillants migrés à partir de nœuds performants et plus performants

3.5.2 Résultats de la grille

3.5.2.1 Stratégies de base

Nous évaluons le modèle tolérant aux fautes proposé par rapport à deux autres méthodes (stratégies) comme suit :

- Méthode décentralisée (Decent_M) : Cette méthode gère les défaillances des nœuds de manière entièrement décentralisée, la sélection des nœuds substituts dépend du critère de voisinage. Ainsi, le mécanisme de tolérance aux fautes est activé lors de la détection d'une défaillance d'un nœud de la grille. Lorsqu'un nœud de la grille découvre que son voisin ne répond pas par un message de vie (*heartbeat*) par exemple, il déclenche un processus de réaction pour faire face à la défaillance. Tout d'abord, il rejette tous les jobs assignés par le nœud voisin défaillant. Ensuite, il réordonnance ses jobs, qui ont été affectés aux nœud de la grille défaillant un par un (Chauhan & Nitin, 2012).
- Méthode centralisée (Cent_M) : Dans cette stratégie, les substituts du nœud défaillant sont sélectionnés dans une organisation hiérarchique ou arborescente. La défaillance d'un nœud enfant est détectée par son parent. Ce dernier s'occupe de trouver des nœuds locaux qui peuvent tolérer les jobs du nœud défaillant, sinon la faute sera tolérée à un niveau supérieur. Une fois les substituts trouvés, chaque job est affecté au nœud le plus proche (Rebbah, Mokhtari, Khaldi, Bourasi, & Smail, 2010).

3.5.2.2 Évaluation des performances

Nous avons injecté un taux de fautes (*Fault_Injection_Rate*) allant de 10 à 50% pour un nombre de nœuds égal à 180 nœuds avec un degré moyen *Average_Degree* = 5, une capacité *Max_Job_by_Node* = 20 et une bande passante réseau moyenne = 100 Mbps. En outre, nous avons calculé le taux de jobs tolérés, le niveau de voisinage moyen et le surcoût d'exécution des travaux *JEOH* (*Job Execution Overhead*) pour les méthodes FT-GRC, Decent_M et Cent_M.

La Figure 3.10 montre le taux de jobs tolérés. Initialement (avec *Fault_Injection_Rate* = 10%), FT-GRC n'est pas meilleur que les méthodes Decent_M et Cent_M, car il existe un nombre suffisant de substituts pouvant recevoir les jobs défaillants. Cependant, à mesure que le *Fault_Injection_Rate* augmente (*Fault_Injection_Rate* ≥ 10%), le taux de jobs tolérés par FT-GRC est supérieur à celui tolérés par les méthodes Decent_M et Cent_M. Le taux de jobs tolérés par TF-GRC est augmenté de 12% par rapport à la méthode Decent_M et de 23% par rapport à la méthode Cent_M, lorsque la simulation se termine (à *Fault_Injection_Rate* = 50%). Ainsi, l'impact de notre modèle est significatif en utilisant le niveau de performance pour déterminer les substituts appropriés pour chaque nœud défaillant.

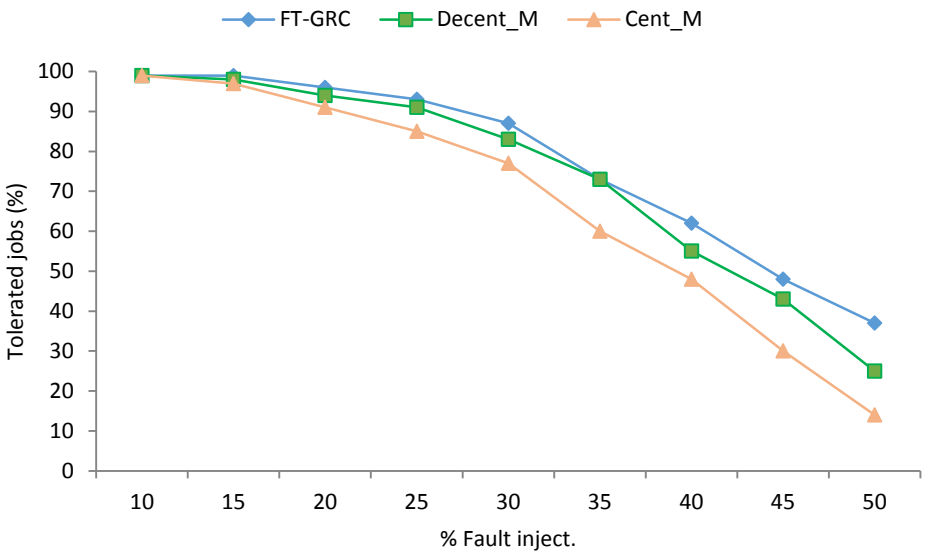


FIGURE 3.10 – Taux de jobs tolérés par rapport au taux d'injection de fautes

La Figure 3.11 montre le niveau de voisinage moyen des substituts. Dans l'approche FT-GRC, le niveau de voisinage moyen des substituts augmente de 1 à 4 sauts et de 1 à 12 sauts pour l'approche Decent_M. Le niveau de voisinage moyen le plus élevé est d'environ 21 sauts pour l'approche Cent_M (à *Fault_Injection_Rate* = 50%). La différence de niveau de voisinage moyen entre FT-GRC et l'approche Cent_M est d'environ 8 sauts (deux fois le niveau de voisinage moyen le plus élevé pour FT-GRC). Le niveau de voisinage moyen pour Decent_M et Cent_M augmente considérablement avec l'augmentation du *Fault_Injection_Rate*. Leur critère de sélection des substituts est basé sur la propagation dans le voisinage. FT-GRC utilise le chemin le plus court entre les clusters pour sélectionner le substitut qui réduit considérablement le nombre de sauts.

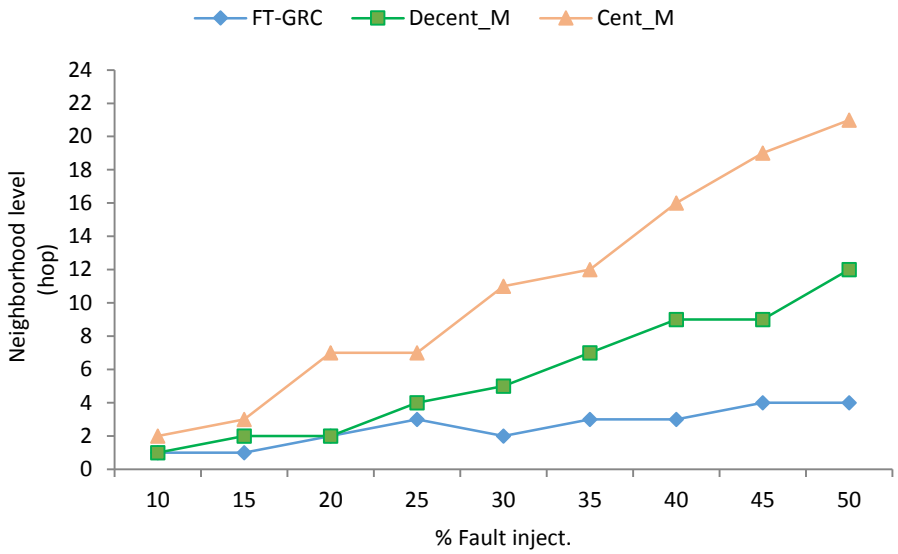


FIGURE 3.11 – Niveau de voisinage moyen par rapport au taux d'injection de fautes

Une mesure de surcoût *JEOH* (*Job Execution Overhead*) est défini comme la fraction du temps de réponse du job :

$$JEOH\ (\%) = \frac{Total\ Job\ Execution\ Time}{\sum_j (ET_j - QT_j)} \tag{3.3}$$

Où QT_j et ET_j sont les temps où le job j est mis en file d'attente sur la grille et quand il se termine.

La Figure 3.12 montre la surcharge d'exécution des jobs (*JEOH*). On observe que la *JEOH* de FT-GRC est très faible, et inférieur à 3%. Dans l'état où la demande de ressources informatiques est importante, la charge CPU et la mémoire disponible deviennent très importantes. Le Cent_M est le plus élevé jusqu'à un *JEOH* de 8,27% et s'approche du point de saturation de façon exponentielle. Decent_M donne de meilleurs résultats que Cent_M en sélectionnant les substituts les plus proches. FT-GRC considère les paramètres de bande passante pour réduire la latence et le temps de réponse du job en utilisant le chemin le plus court pour sélectionner le substitut (intra-cluster puis inter-cluster).

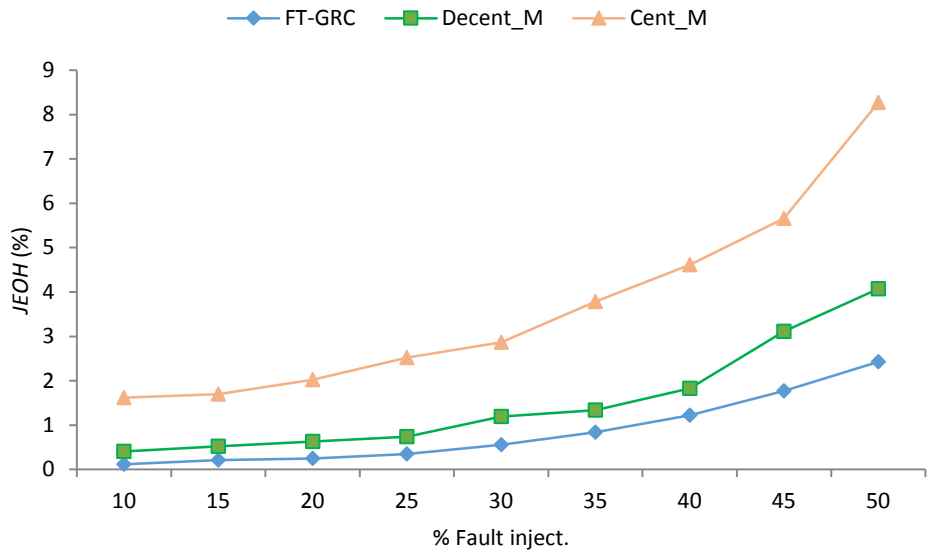


FIGURE 3.12 – Surcharge d'exécution des jobs *JEOH* (bande passante réseau moyenne = 100 Mbps)

3.6 Conclusion

Dans ce chapitre, nous avons proposé une technique de tolérance aux fautes basée sur une modélisation des grilles de calcul par un graphe dynamique coloré. Après une phase de clusterisation des nœuds de la grille, nous transformons la grille à un ensemble de cluster interconnectés entre eux. Chaque cluster est géré par un *cluster-head*, les nœuds *members* sont chargés d'exécuter les jobs et les nœuds *gateways* plus de leurs rôles des membres se servent de relais entre les clusters et nœuds clés dans les plus courts chemins trouvés entre les clusters.

En fonction des valeurs des attributs fondamentaux dans l'exécution des jobs, nous appliquons une fonction *scoring*, pour détecter les niveaux de performance de chaque nœud. Nous classifions les nœuds de chaque cluster en fonction de leurs niveaux de performance en trois catégories (moins performants, performants et plus performants). Le mécanisme de tolérance aux fautes proposé exploite la fonction *scoring* pour déterminer les substituts adéquats pour chaque nœud défaillant ; et plus tard, il exploite la clusterisation pour déterminer de manière optimal le choix des substituts, en entamant la recherche localement dans le même cluster. Si nous ne pouvons pas les trouver dans le cluster, on cherche les substituts les plus proches, en appliquant l'algorithme de DJIKSTRA, pour trouver le chemin minimal.

Les résultats des expérimentations montrent l'efficacité de la méthode de *scoring* et le gain obtenu en cherchant les substituts dans le cluster puis par la recherche des plus proches substituts.

Nous prévoyons d'étendre cette approche à un autre modèle de grille, autres algorithmes de clustering (*k*-saut), autre méthode de classification (K-Means) et d'autres mécanismes de tolérance au fautes (points de reprises).

Chapitre 4

Tolérance aux fautes pour un système de workflow scientifique dans un environnement de Cloud Computing

Sommaire

4.1 Introduction

Les workflows scientifiques se composent de milliers de tâches de calcul de granularité fine qui dépendent des contrôles et des données (Taylor, Deelman, Gannon, & Shields, 2014). Ces workflows scientifiques sont utilisés pour simuler des applications scientifiques dans différents domaines de l'astronomie (Katz, et al., 2006), de la sismologie (Maechling, et al., 2007), de la biologie (Lathers, et al., 2006), etc. Comme les workflows peuvent être composés d'un grand nombre de tâches, une puissance de calcul est nécessaire pour les faire exécuter (Juve, et al., 2013). Le problème d'ordonnancement consiste à faire mapper des tâches de ces workflows sur des systèmes distribués (Singh, et al., 2008; Chen & Deelman, 2012; da Silva, Glatard, & Desprez, 2013) qui sont souvent répartis sur des zones étendues (Zhang, Squillante, Sivasubramaniam, & Sahoo, 2004; Schroeder & Gibson, 2006) tels que les Grilles (Deelman, et al., 2004; Duan, Prodan, & Fahringer, 2006; da Silva & Glatard, 2013) et les Cloud Computing (Bresnahan, Freeman, La Bissoniere, & Keahey, 2011; Deelman, Singh, Livny, Berriman, & Good, 2008; Berriman, Juve, Deelman, Regelson, & Plavchan, 2010). L'ordonnancement est particulièrement difficile pour des workflows scientifiques à large échelle qui comportent de nombreuses tâches de calcul. Les workflows scientifiques migrent progressivement vers une plate-forme Cloud qui fournit des ressources de calcul, de stockage et de communication à faible coût (Vinay, Kumar, Raghavendra, & Venugopal, 2018). Les méthodes de clustering des tâches (Maheshwari, et al., 2012; da Silva, Glatard, & Desprez, 2014; Chen & Deelman, 2012) combinent plusieurs tâches en un seul job afin de minimiser le temps d'exécution et diminuer sa charge. Le clustering de tâches est la technique la plus utilisée pour gérer les surcharges du système et augmenter la granularité des tâches exécutées sur des plates-formes distribuées (Chen, da Silva, Deelman, & Fahringer, 2016). L'ordonnancement d'un workflow scientifique sur les ressources du Cloud réduit considérablement le makespan et les coûts d'exécution, mais il est également sujet à des défaillances de ressources. Les défaillances sont généralement dues à des fautes matérielles, logicielles et de virtualisation, etc. Les défaillances au point d'un workflow scientifique sont classées en fautes de tâche et fautes de machine virtuelle (VM) (Zhang, Squillante, Sivasubramaniam, & Sahoo, 2004). Cependant, les défaillances sont incertaines lors de l'exécution d'un workflow scientifique composé de plusieurs milliers des tâches (Vinay, Kumar, Raghavendra, & Venugopal, 2018). La défaillance d'une tâche est considérée comme un grand problème dans l'exécution du workflow. Si une tâche au sein d'un job clusterisé est interrompue par des événements inattendus lors de son exécution, le job est marqué comme ayant échoué, même si les autres tâches du même job ont terminé leur calcul avec succès (Chen, da Silva, Deelman, & Fahringer, 2016).

Diverses techniques ont été introduites pour faire face à l'impact négatif des fautes de tâche sur l'exécution des workflows scientifiques. La technique la plus utilisée consiste à resoumettre la tâche ayant échoué (Zhang, Mandal, Koelbel, & Cooper, 2009; Montagnat, et al., 2010; Kandaswamy, Mandal, & Reed, 2008). Cependant, une nouvelle tentative de tâches en cluster peut être coûteuse car les tâches réussies dans le même job doivent être recalculées, ce qui entraîne un gaspillage des cycles de ressources. De plus, rien ne garantit que les tâches soumises à nouveau se termineront avec succès. Comme alternative, les tâches peuvent être répliquées pour éviter les défaillances spécifiques à une unité de traitement (Plankensteiner, Prodan, &

Fahringer, 2009). Cependant, la réplication des tâches peut également consommer plus de ressources, en particulier lors de l'exécution de jobs pendant une longue durée. Les calculs de tâches peuvent être périodiquement sauvegardés (*checkpointed*) afin de limiter le nombre total des jobs recalculés et réduire la consommation des ressources. Cependant, la surcharge du système pour effectuer des points de reprise (*checkpoints*) peut réduire ses avantages (Zhang, Squillante, Sivasubramaniam, & Sahoo, 2004).

Les travaux existants (Chen, da Silva, Deelman, & Fahringer, 2016) dans le clustering de tâches tolérant aux fautes pour les workflows scientifiques sont basés sur les informations statiques des sites d'exécution. Aujourd'hui, avec l'aide de l'approvisionnement des ressources (Juve, Deelman, Vahi, & Mehta, 2010), nous pouvons affecter dynamiquement des ressources à plusieurs sites d'exécution (Chen & Deelman, 2012), puis exécuter les jobs clusterisés sur ces sites. Ce travail se concentre sur l'approvisionnement des jobs clusterisés, qui permettrait de lancer plusieurs machines virtuelles (VMs) en même temps et de construire un cluster virtuel. Une telle nouvelle approche, nous oblige à considérer le problème d'approvisionnement des ressources et le problème de clustering de tâches tolérant aux fautes afin de minimiser à la fois le makespan et le coût d'exécution du workflow.

Dans ce chapitre, nous proposons une nouvelle méthode d'ordonnancement tolérante aux fautes des workflows scientifiques dans un environnement de Cloud Computing (Khaldi, Mohammed, Meftah, & Smail, 2019). Nous améliorons les techniques de clustering existantes (avec réplication de tâches) dans un environnement défaillant si les défaillances transitoires satisfont le modèle de défaillance proposé. De plus, la technique de clustering de tâches tolérante aux fautes proposée prend en compte à la fois les paramètres de makespan et de coût d'exécution du workflow, permettant ainsi aux utilisateurs de spécifier leur restriction de délai (*deadline*) lors de l'ordonnancement. L'objectif de nos recherches est de réduire le makespan et le coût d'exécution du workflow. Cependant, la méthode de définir le nombre de ressources (instances VM) en fonction de l'échelle du workflow reste un défi. Nos contributions, dans ce travail, incluent un ensemble de techniques et d'heuristiques pour résoudre le problème de l'intégration du clustering de tâches tolérant aux fautes et de l'approvisionnement des ressources.

Notre méthode est implémentée et évaluée dans une approche basée sur la simulation sur différentes configurations de workflows scientifiques (cinq applications scientifiques largement utilisées). Nous étendons notre travail par; 1) évaluer l'impact de la variance de la distribution du makespan, du coût d'exécution et du deadline du workflow, 2) caractériser l'impact de l'exécution du workflow lorsque des défaillances transitoires se produisent aux tâches d'un workflow (taux d'échec des tâches), et 3) étudier le gain de performances de l'utilisation de notre approche de clustering de tâches tolérante aux fautes par rapport aux autres méthodes de clustering existantes appliquées sur un ensemble plus large de workflows.

Le reste de ce chapitre est structuré comme suit. La section 2 donne un aperçu des travaux connexes. La section 3 décrit le modèle de workflow, de Cloud et de faute utilisé. La section 4 présente notre méthode de clustering tolérante aux fautes. La section 5 analyse les résultats expérimentaux et cette partie se termine par une discussion et une conclusion.

4.2 Travaux connexes

Des recherches ont été menées sur la prédiction et le recouvrement des fautes, mais seuls quelques travaux sont réalisés, en particulier pour les fautes de tâches dans l'exécution du workflow sur l'environnement Cloud. Amoon et al. (Amoon, El-Bahnasawy, Sadi, & Wagdi, 2019) ont avoué que la défaillance est une partie difficile à laquelle le fournisseur de Cloud doit faire face pour garantir le SLA. Ils ont proposé une méthode réactive de tolérance aux fautes basée sur des points de reprise (checkpointing) à des intervalles flexibles. Ying et al. (Ying, Yu, & He, 2018) ont proposé un framework de calcul en mémoire, qui réduit le recalcul de la tâche avec un algorithme de recouvrement utilisé pour recouvrer efficacement la défaillance. Les travaux réalisés par Zhao et al. (Zhao, Melliar-Smith, & Moser, 2010) et Jhawar et al. (Jhawar, Piuri, & Santambrogio, 2012) utilise le protocole de messages de vie (heartbeat). Il détecte les fautes de type crash et les fautes de répliques, mais la principale difficulté est qu'il n'y a pas de mécanisme pour identifier la faute de la tâche dans les workflows scientifiques. Poola et al. (Poola, Ramamohanarao, & Buyya, 2014) ont proposé une méthode d'ordonnancement de workflow tolérante aux fautes, utilisant des instances ponctuelles (spot). Padmakumari et al. (Padmakumari & Umamakeswari, 2019) ont proposé un modèle cognitif de tolérance aux fautes avec trois phases pour tolérer les tâches et les défaillances d'instances VM de manière proactive.

Les problèmes liés à la tolérance aux fautes ont déjà été traités dans des systèmes de gestion des workflows scientifiques (WFMS), tels que Pegasus WFMS (Deelman, et al., 2015). Il a incorporé un système qui surveille au niveau du job et relance les tâches échouées. Les données de provenance sont également suivies et analysées pour déterminer la cause des défaillances (Samak, et al., 2011). Une synthèse des techniques de prévention, de détection et de recouvrement des fautes dans la Grille WFMS actuelle est disponible dans (Plankensteiner K. , Prodan, Fahringer, Kertész, & Kacsuk, 2008). Elle fournit un ensemble de techniques de recouvrement telles que les points de reprise des tâches, la réplication, la resoumission et la migration. Dans ce travail, nous combinons certaines techniques de réplication de tâches (réplication active et passive) avec notre méthode de clustering pour améliorer ses performances et sa fiabilité.

Le contrôle de la granularité des tâches couplées a été abordé par les algorithmes de clustering qui prennent en compte le temps d'exécution des tâches, la taille du fichier de tâches, la charge CPU et les contraintes de ressources (Muthuvelu, et al., 2005; Muthuvelu, Chai, & Eswaran, 2008). Muthuvelu et al. (Muthuvelu N. , Chai, Chikkannan, & Buyya, 2010) ont développé un algorithme d'ordonnancement en ligne qui regroupe un ensemble de tâches en fonction de l'utilisation du réseau de ressources, du deadline d'application et du budget de l'utilisateur. De plus, Ng et al. (Ng, Ang, Ling, & Liew, 2006) et Ang et al. (Ang, Ng, Ling, Por, & Liew, 2009) ont pris en considération la bande passante du réseau pour améliorer les performances de l'algorithme d'ordonnancement en utilisant le clustering des jobs. Liu et Liao (Liu & Liao, 2009) ont proposé un algorithme d'ordonnancement basé sur le clustering pour fusionner des tâches à granularité fine en fonction de la bande passante du réseau et de la capacité de traitement des ressources disponibles. Un clustering basé sur les labels (*label-based*) et sur les niveaux (*level-based*) a été développé par Singh (Singh, et al., 2008). Dans un clustering basé

sur les labels, les labels de tâches de l'utilisateur doivent être combinées. Ainsi, dans le clustering basé sur les niveaux, les tâches d'un même niveau de workflow sont combinées. La taille du cluster et le nombre de clusters doivent être attribués par l'utilisateur. Le temps d'exécution et la surcharge du système (overhead) doivent être prédits. Ferreira da Silva et al. (da Silva, Glatard, & Desprez, 2013; da Silva, Glatard, & Desprez, 2014) ont développé des algorithmes de dissociation et de regroupement des tâches pour contrôler la granularité des tâches de workflow dans un contexte en ligne et non clairvoyant, où aucune ou peu de caractéristiques de l'application ou des ressources sont connues à l'avance. Bien que ces méthodes réduisent considérablement l'impact de l'ordonnancement, elles ne résolvent pas le problème de tolérance aux fautes. Stergiou et al. (Stergiou, Psannis, Kim, & Gupta, 2018) ont présenté un clustering de tâches basé sur le temps d'exécution et le temps de transfert de données. Les tâches ayant un temps d'exécution moyen et une bonne variation, ne seront pas requises pour la clusterisation. Une tâche avec un temps d'exécution et une variation inacceptables doit être clusterisée. Un job avec plusieurs tâches provoque plus de chancre d'augmenter le taux de défaillance des tâches. Ce qui est l'effet secondaire du clustering de tâches. La défaillance affecte directement les performances du workflow de sorte que le modèle de défaillance des tâches se concentre sur les problèmes de performances dans le clustering. Chen et al. (Chen, da Silva, Deelman, & Fahringer, 2016) ont proposé différentes stratégies de clustering pour la tolérance aux fautes. Ils ont analysé ces stratégies pour cinq applications scientifiques réalistes, notamment Montage, Cybershake, LIGO, SIPHT et Epigenomics, dans lesquelles le temps d'exécution est capturé à partir de traces d'exécution réelles. Ils ont pris en compte des facteurs, tels que le taux de défaillance des tâches, pour trouver la fiabilité de leurs techniques de clustering. Ils ont utilisé la simulation basée sur les traces pour contrôler les paramètres de temps d'exécution des tâches et le temps d'arrivée des fautes pour améliorer les performances des algorithmes tolérants aux fautes proposés. Sauf que ces méthodes ne traitent pas le problème d'approvisionnement de ressources et du coût d'exécution des workflows dans l'environnement Cloud Computing. Dans (Dharwadkar, Poojara, & Kadam, 2018), les auteurs se sont concentrés sur le problème de l'ordonnancement des workflows scientifiques sur un ensemble de VMs avec une tolérance aux faute améliorée. Les stratégies proposées réduisent le temps d'exécution du workflow et le coût total. Ils ont appliqué des techniques de réplication et de points de reprise (*checkpoints*) ainsi que l'exécution de tâches parallèles. La stratégie de Reclustering Horizontal (HR) est appliquée pour réduire l'effet de faute et limiter la surcharge (overhead) de l'ordonnancement. La méthode de checkpointing ajoute des points de reprise dans le calcul des tâches échouées identifiées. L'algorithme HR regroupe uniquement les tâches ayant échoué au même niveau horizontal dans un nouveau job en cluster et recalcule les tâches échouées pour réduire la surcharge (overhead) de l'ordonnancement du workflow. En outre, Shojafar et al. (Shojafar, Javanmardi, Abolfazli, & Cordeschi, 2015) ont développé une nouvelle méthode d'ordonnancement des jobs appelée FUGE qui se hybride sur la théorie de la logique floue avec les algorithmes génétiques qui s'attendent à réduire l'équilibrage de charge en tenant compte du makespan et du coût lors de l'ordonnancement des jobs. Une hypothèse floue est appliquée pour analyser l'estimation de l'aptitude des chromosomes et pour l'opération de croisement. Elle donne une amélioration d'environ 50% en termes de temps d'exécution total et d'environ 45% en termes de coût d'exécution.

Plusieurs travaux ont abordé le problème d'ordonnancement des workflows en considérant l'heuristique de graphe acyclique dirigé (DAG) (Topcuoglu, Hariri, & Wu, 2002; Blythe, et al., 2005; Kalayci, Dasgupta, Fong, Ezenwoye, & Sadjadi, 2010). Dans (Kalayci, Dasgupta, Fong, Ezenwoye, & Sadjadi, 2010), les auteurs ont utilisé la méthode de mise en correspondance (*matchmaking*) pour mapper les tâches afin de calculer les ressources. Une méthode similaire a été adoptée par les auteurs (Chen, da Silva, Deelman, & Fahringer, 2016) pour déjouer l'ordonnancement du workflow et calculer les nœuds avec un taux de défaillance élevé. Dans ce travail, nous nous concentrons sur le gain de performances obtenu en ajustant la granularité du clustering pour réduire le coût d'exécution et réduire le makespan du workflow.

4.3 Workflow scientifique

Le concept de workflow trouve ses racines dans les entreprises commerciales en tant qu'outil de modélisation des processus métier. Ces workflows commerciaux visent à automatiser et optimiser les processus d'une organisation, considérés comme une séquence ordonnée d'activités, et constituent un domaine de recherche mature (Yildiz, Guabtni, & Ngu, 2009) dirigé par Workflow Management Coalition[4] (WfMC), fondée en 1993. Cette notion de workflow s'est étendue vers la communauté scientifique sous forme des workflows scientifiques utilisés pour soutenir des processus scientifiques complexes. Ils sont conçus pour mener des expériences et prouver des hypothèses scientifiques en gérant, analysant, simulant et visualisant des données scientifiques (Barker & Van Hemert, 2008). Par conséquent, même si les workflows commerciaux et scientifiques partagent le même concept de base, les deux ont des exigences spécifiques et doivent donc être examinés séparément.

Un workflow est défini par un ensemble de tâches de calcul avec des dépendances entre elles comme l'illustre la Figure 4.1. Dans les applications scientifiques, il est courant que les dépendances représentent un flux de données d'une tâche à l'autre ; les données de sortie générées par une tâche deviennent les données d'entrée de la tâche suivante. Ces applications peuvent être à forte intensité de CPU, de mémoire ou d'E/S (ou une combinaison de celles-ci), selon la nature du problème à résoudre. Dans un workflow à forte intensité de CPU, la plupart des tâches prennent beaucoup de temps d'exécution. Dans un workflow lié à la mémoire, la majorité des tâches nécessitent une utilisation élevée de la mémoire physique. Les workflows à forte intensité d'E/S sont composés de tâches qui nécessitent et produisent de grands volumes de données et passent donc la plupart de leur temps à effectuer des opérations d'E/S (Juve, et al., 2013).

Les workflows scientifiques sont gérés par différentes institutions ou personnes dans différents domaines, ce qui signifie qu'ils ont des exigences différentes en termes de logiciels nécessaires à l'exécution des tâches. Ces caractéristiques en font d'excellents candidats pour exploiter les capacités offertes par le Cloud Computing. Les scientifiques peuvent configurer des images de machine virtuelle (VM) pour répondre aux besoins logiciels d'un workflow spécifique et à l'aide des algorithmes d'ordonnancement et de systèmes de gestion de workflow, ils peuvent exécuter

[4] http://www.wfmc.org/

efficacement leurs applications sur une gamme de ressources Cloud pour obtenir des résultats dans un délai raisonnable. De cette manière, en fournissant un moyen simple et rentable d'exécuter des applications scientifiques accessibles à tous, le Cloud Computing révolutionne la science électronique (e-science) (Rodriguez & Buyya, 2017).

La portée de ce travail est limitée aux workflows modélisés sous forme de graphs acycliques dirigés (DAGs) qui, par définition, n'ont pas de cycles ou de dépendances conditionnelles. Bien qu'il existe d'autres modèles de calcul qui pourraient être utilisés pour exprimer et traiter les workflows scientifiques tels que les pipelines d'effort optimal, superscalaire et de streaming. Cette étude se concentre sur les DAGs car ils sont couramment utilisés par la communauté scientifique et de la recherche. Par exemple, les systèmes de gestion de workflow tels que Pegasus (Deelman, et al., 2015), Cloudbus WfMS (Pandey, Karunamoorthy, & Buyya, 2011), ASKALON (Fahringer, et al., 2007) et DAGMan (Kalayci, Dasgupta, Fong, Ezenwoye, & Sadjadi, 2010) prennent en charge l'exécution de workflows modélisés sous forme de DAG.

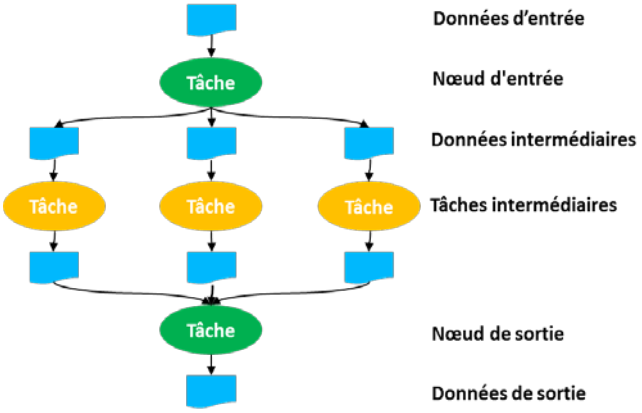


FIGURE 4.1 – Un exemple de workflow, représentant les tâches, les données et leurs dépendances

4.3.1 Applications de workflow

De nombreux domaines scientifiques ont adopté les workflows comme moyen d'exprimer des problèmes de calcul complexes qui peuvent être traités efficacement dans des environnements distribués. Nous allons décrire dans ce qui suit brièvement certaines applications réelles de workflows scientifiques les plus caractérisées.

Montage : Montage (Berriman, et al., 2004) a été créé par la *NASA/IPAC Infrared Science Archive* en tant que *Toolkit Open Source* qui peut être utilisé pour générer des mosaïques personnalisées du ciel en utilisant des images d'entrée au format FITS (*Flexible Image Transport System*). Lors de la production de la mosaïque finale, la géométrie de l'image de sortie est calculée à partir des images d'entrée. Les entrées sont ensuite re-projetées pour avoir la même échelle spatiale et la même rotation. Les émissions de fond dans les images sont corrigées pour avoir un niveau uniforme, et les images corrigées et re-projetées sont co-ajoutées

pour former la mosaïque de sortie. L'application Montage a été représentée comme un workflow qui peut être exécuté dans des environnements de grille tels que TeraGrid[5].

CyberShake : Le workflow CyberShake (Maechling, et al., 2007) est utilisé par SCEC[6] (*Southern California Earthquake Center*) pour caractériser les risques sismiques à l'aide de la technique PSHA (*Probabilistic Seismic Hazard Analysis*). Étant donné une région d'intérêt, une simulation à différence finie basée sur MPI (*Message Passing Interface*) est effectuée pour générer des SGTs (*Strain Green Tensors*). À partir des données SGT, des sismogrammes synthétiques sont calculés pour chacune d'une série de ruptures prévues. Une fois cela fait, les courbes d'accélération spectrale et d'aléa probabiliste sont calculées à partir des sismogrammes pour caractériser l'aléa sismique. Des workflows CyberShake composés de plus de 800 000 tâches ont été exécutés à l'aide du système de gestion de workflow Pegasus (Pegasus-WMS) sur le TeraGrid (Callaghan, et al., 2008).

Epigenomics : Le *USC Epigenome Center*[7] est actuellement impliqué dans le mappage de l'état épigénétique des cellules humaines à l'échelle du génome. Le workflow Epigenomics est essentiellement un pipeline de traitement de données qui utilise le système de gestion de workflow Pegasus pour automatiser l'exécution des différentes opérations de séquençage du génome. Les données de séquence d'ADN générées par le système d'analyseur génétique Illumina-Solexa[8] sont divisées en plusieurs morceaux qui peuvent être opérés en parallèle. Les données de chaque morceau sont converties dans un format de fichier qui peut être utilisé par le logiciel Maq[9] qui mappe les lectures de séquençage d'ADN courtes (Li, Ruan, & Durbin, 2008). A partir de là, les séquences sont filtrées pour éliminer les segments bruyants et contaminants et sont mappées à l'emplacement correct dans un génome de référence. Enfin, une carte globale des séquences alignées est générée et la densité de séquences à chaque position dans le génome est calculée. Ce workflow est utilisé par le Centre Epigenome dans le traitement des données de méthylation de l'ADN de production et de modification des histones.

LIGO Inspiral Analysis : Le LIGO[10] (*Laser Interferometer Gravitational Wave Observatory*) (Abramovici, et al., 1992) tente de détecter les ondes gravitationnelles produites par divers événements dans l'univers conformément à la théorie de la relativité générale d'Einstein. Le workflow *LIGO Inspiral Analysis* (Brown, et al., 2007) est utilisé pour analyser les données obtenues à partir de la coalescence de systèmes binaires compacts tels que les étoiles à neutrons binaires et les trous noirs. Les données temps-fréquence de chacun des trois détecteurs LIGO sont divisées en blocs plus petits pour l'analyse. Pour chaque bloc, le workflow génère un sous-ensemble de formes d'onde appartenant à l'espace des paramètres et calcule la sortie du filtre correspondant. Si un vrai *inspiral* a été détecté, un déclencheur est généré qui peut être vérifié

[5] The TeraGrid Project. http://teragrid.org/
[6] SCEC project, Southern California Earthquake Center. http://www.scec.org/
[7] USC Epigenome Center. http://epigenome.usc.edu
[8] Illumina. http://www.illumina.com
[9] Maq: mapping and assembly with qualities. http://www.maq.sourceforge.net
[10] LIGO project, LIGO—laser interferometer gravitational wave observatory. http://www.ligo.caltech.edu/

avec des déclencheurs des autres détecteurs. Plusieurs tests de cohérence supplémentaires peuvent également être ajoutés au workflow.

SIPHT : Le projet de bioinformatique de l'université de Harvard mène une vaste recherche sur les petits ARNs non traduits (*sRNAs*) qui régulent des processus tels que la sécrétion et la virulence chez les bactéries. Le programme SIPHT (*sRNA Identification Protocol using High-throughput Technology*) (Livny, Teonadi, Livny, & Waldor, 2008) utilise un workflow pour automatiser la recherche des gènes codant les ARNs de toutes les réplicons bactériens dans la base de données du NCBI (*National Center for Biotechnology Information*). La prédiction et l'annotation à l'échelle du royaume des gènes codant pour le sRNA impliquent une variété de programmes qui sont exécutés dans l'ordre approprié en utilisant les capacités de *Condor DAGMan*[11]. Celles-ci impliquent la prédiction des terminateurs transcriptionnels indépendants de Rho, des comparaisons BLAST (*Basic Local Alignment Search Tools*) des régions intergénétiques de différents réplicons et les annotations de tous les *sRNAs* trouvés.

Les applications sus mentionnées sont des bonnes représentations des workflows scientifiques car elles proviennent de différents domaines et donnent ensemble un large aperçu de l'utilisation des technologies de workflow pour gérer des analyses complexes. Chacun des workflows a des structures topologiques différentes, toutes communes dans les workflows scientifiques tels que les pipelines, la distribution et l'agrégation des données (Bharathi, et al., 2008). Ils ont également des caractéristiques de données et de calcul variées, notamment des tâches intensives en CPU, en E/S et en mémoire. La Figure 4.2 montre la structure de ces cinq workflows scientifiques et leurs caractérisations complètes sont présentées par Juve et al. (Juve, et al., 2013).

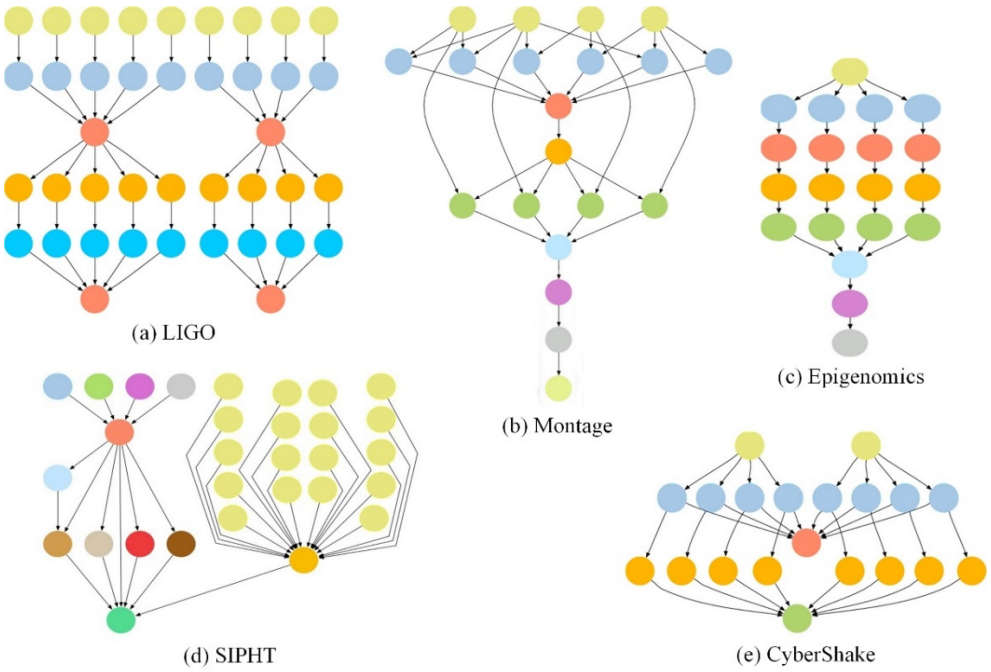


FIGURE 4.2 – Visualisation simplifiée de cinq différents workflows scientifiques réalistes

[11] DAGMan (directed acyclic graph manager). http://www.cs.wisc.edu/condor/dagman/.

4.3.2 Systèmes de gestion de workflow

Les systèmes de gestion de workflow (Workflow Management Systems (WFMS)) permettent une exécution automatisée et transparente des workflows. Ils permettent aux utilisateurs de définir et de modéliser les workflows, de définir leurs deadlines et leurs limites budgétaires, ainsi que les environnements dans lesquels ils souhaitent s'exécuter. Le WFMS évalue ensuite ces entrées et les exécutent dans les contraintes définies. Les principaux composants d'un WFMS typique sont donnés dans la Figure 4.3 (Chen, da Silva, Deelman, & Fahringer, 2016) :

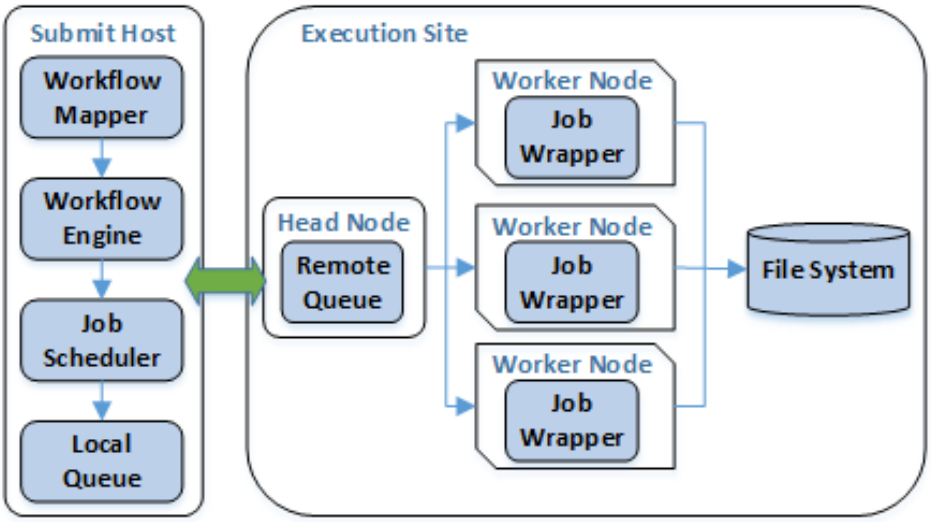


FIGURE 4.3 – Présentation du système de gestion de workflow WFMS

Mappeur de workflow (*Workflow Mapper*) : importe un workflow abstrait au format standard et génère un workflow exécutable en créant la liste des jobs et d'autres informations requises (Deelman, et al., 2015). Le workflow sera restructuré pour optimiser les performances et pour la gestion des données. Le *Workflow Mapper* est également utilisé pour regrouper les tâches dans un seul job en cluster.

Moteur de workflow (*Workflow Engine*) : exécute les jobs en fonction des dépendances entre les tâches de workflow. Seules les tâches dont toutes les tâches précédentes ont été exécutées avec succès sont soumises à l'ordonnanceur de job (*Job Scheduler*). Le *Workflow Engine* dépend des critères de ressources (traitement, stockage et réseau) fournis avec le workflow exécutable pour effectuer les calculs. L'intervalle de temps entre la réalisation du job et sa soumission au *Job Scheduler* est indiqué comme le délai de moteur de workflow (*Workflow Engine delay*).

File d'attente locale et ordonnanceur de job (*Local Queue and Job Scheduler*) : supervise l'exécution des jobs individuels de workflow sur des ressources distantes et locales. L'intervalle de temps entre la soumission du job à l'ordonnanceur et le premier traitement sur un *Worker Node* est indiqué comme le délai de la file d'attente (*Queue Delay*). La disponibilité des ressources et l'efficacité de l'ordonnanceur sont prises en compte par le délai d'attente (*Queue Delay*).

Emballeur de job (*Job Wrapper*) : exécute les tâches extraites des jobs clusterisés sur les *Worker Nodes*. La durée nécessaire pour le processus d'extraction est indiquée comme le délai de clustering (*Clustering Delay*).

4.3.3 Ordonnancement de workflows

L'ordonnancement mappe les tâches de workflow aux ressources distribuées de sorte que les dépendances ne soient pas violées. L'ordonnancement des workflows est un problème NP-Complet bien connu (Brandic, Music, & Dustdar, 2009).

L'architecture d'ordonnancement des workflows spécifie le placement de l'ordonnanceur dans un WFMS (*Workflow Management Systems*) et peut être classée en trois grandes catégories, comme l'illustre la Figure 4.4 : *centralisé*, *hiérarchique* et *décentralisé* (Yu & Buyya, 2005). Dans l'approche *centralisée,* un ordonnanceur centralisé prend toutes les décisions d'ordonnancement pour l'ensemble du workflow. L'inconvénient de cette approche est qu'elle n'est pas évolutive ; cependant, elle peut produire des ordonnancements efficaces car *l'ordonnanceur* centralisé dispose de toutes les informations nécessaires. Dans l'ordonnancement *hiérarchique,* il existe un gestionnaire central chargé de contrôler l'exécution du workflow et d'attribuer les sous-workflows aux ordonnanceurs de bas niveau. Les ordonnanceurs de bas niveau mappent les tâches des sous-workflows attribués par le gestionnaire central. En revanche, l'ordonnancement décentralisé n'a pas de contrôleur central. Il permet aux tâches d'être ordonnancées par plusieurs ordonnanceurs, chacun communique avec les autres et ordonnance un sous-workflow ou une tâche (Yu & Buyya, 2005).

La planification de l'ordonnancement de workflow pour ces applications, également appelée schéma de planification, est de deux types : *statique* (*hors ligne*) et *dynamique* (*en ligne*). Le schéma *statique* mappe les tâches aux ressources au moment de la compilation. Ces algorithmes nécessitent la connaissance préalable des tâches de workflow et des caractéristiques des ressources. Par contre, le schéma *dynamique* peut faire peu d'hypothèses avant l'exécution et prendre une décision d'ordonnancement juste à temps (Kwok & Ahmad, 1999). Ici, des informations dynamiques et statiques sur l'environnement sont utilisées dans les décisions d'ordonnancement.

De plus, les techniques d'ordonnancement de workflow sont les approches ou les méthodologies utilisées pour mapper les tâches de workflow aux ressources, et elles peuvent être classées en deux types : *heuristiques* et *méta-heuristiques*. Les solutions *heuristiques* exploitent les informations dépendantes du problème pour fournir une solution approximative qui négocie l'optimalité, l'exhaustivité, la précision et/ou la vitesse de traitement. D'autre part, les *méta-heuristiques* sont des procédures plus abstraites qui peuvent être appliquées à une variété de problèmes. Une approche méta-heuristique est indépendante des problèmes et les traite comme des boîtes noires. Parmi les principales approches méta-heuristiques, citons les algorithmes génétiques, l'optimisation des essaims de particules, le recuit simulé et l'optimisation des colonies de fourmis (Yu, Buyya, & Ramamohanarao, 2008).

Chaque algorithme d'ordonnancement pour n'importe quel workflow doit tracer un ou plusieurs objectifs. Les stratégies ou les objectifs les plus utilisés sont indiqués dans la Figure 4.4. Le temps, le coût, l'énergie et la qualité de service (QoS) sont les objectifs les plus couramment utilisés pour un algorithme d'ordonnancement de workflow. Les algorithmes peuvent avoir un seul objectif ou plusieurs objectifs en fonction du scénario et de l'énoncé du problème.

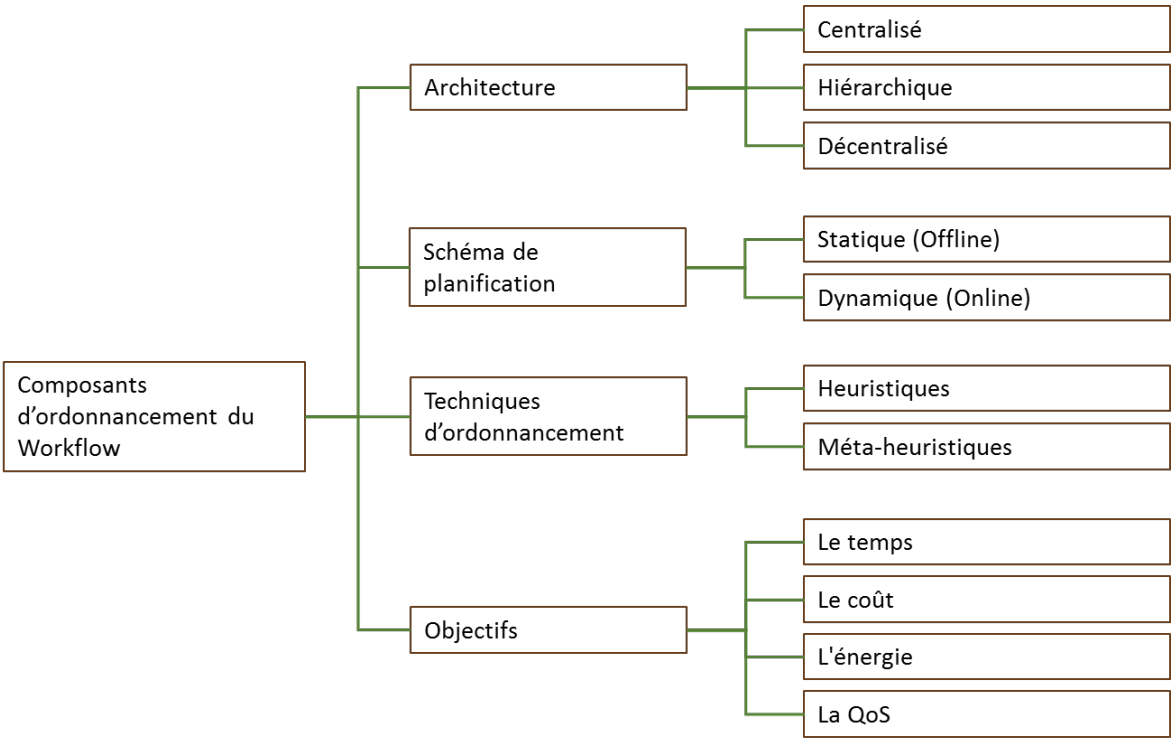


FIGURE 4.4 – Composants d'ordonnancement des workflows

4.3.4 Tolérance aux fautes dans les systèmes distribués

Les workflows sont généralement composés de milliers de tâches, avec des dépendances complexes entre les tâches. Par exemple, certains workflows importants (comme le montre la Figure 4.2) largement considérés sont Montage, CyberShake, Epigenomics, LIGO Inspiral Analysis et SIPHT, qui sont des workflows scientifiques complexes provenant de différents domaines tels que l'astronomie, les sciences de la vie, la physique et la biologie. Ces workflows sont composés de milliers de tâches avec des dépendances complexes qui nécessitent des temps d'exécutions importants.

Les tâches de workflow sont souvent exécutées sur des ressources distribuées de nature hétérogène. Les WFMSs qui allouent ces workflows utilisent des outils middleware qui nécessitent de fonctionner de manière conviviale dans un environnement distribué. Cette nature très complexe et compliquée des WFMSs et de leur environnement suscite de nombreuses incertitudes et risques des défaillances à différents niveaux.

En particulier, dans les workflows à forte intensité de données (*data-intensive*) qui traitent continuellement des données, une panne de machine est inévitable. Ainsi, la défaillance est une préoccupation majeure lors de l'exécution de frameworks de workflows à forte intensité de données, tels que MapReduce et Dryad (Isard, Budiu, Yu, Birrell, & Fetterly, 2007). Des défaillances transitoires et permanentes peuvent survenir dans les workflows à forte intensité de données (Ko, Hoque, Cho, & Gupta, 2010). Par exemple, Google a signalé en moyenne 5 défaillances permanentes sous forme de pannes de machines par workflow MapReduce en mars 2006 (Dean, 2006) et au moins une panne de disque à chaque exécution de workflow MapReduce avec 4000 tâches.

La nécessité d'une tolérance aux fautes découle de cette nature même de l'application et de l'environnement. Les workflows sont des applications qui sont le plus souvent utilisées dans un environnement collaboratif et sont réparties à travers la géographie impliquant diverses personnes de différents domaines (Juve, et al., 2013). Tant de diversités sont des causes potentielles d'adversités. Par conséquent, pour fournir une expérience transparente sur un environnement distribué à plusieurs utilisateurs d'une application complexe, la tolérance aux fautes est une exigence primordiale de tout WFMS.

4.4 Modèle proposé

4.4.1 Système de workflow

Le modèle proposé tente de tolérer les fautes d'un système de workflow scientifique dans un environnement Cloud Computing (Khaldi, Mohammed, Meftah, & Smail, 2019).

Nous modélisons un workflow scientifique par un graphe acyclique dirigé (DAG), un DAG est défini par $G = \{\mathbf{T}, \mathbf{E}\}$, où $\mathbf{T} = \{t_1, t_2, \ldots, t_n\}$ est un ensemble de nœuds et chaque nœud représente une tâche dépendante qui est censée être non préemptive, et **E** est un ensemble d'arêtes dirigés qui représente les dépendances entre les tâches. Un $e_{ij} = \{t_i, t_j\}$ dans **E** indique que la tâche t_j dépend des données générées par la tâche t_i pour son exécution, donc, t_j ne peut pas commencer à s'exécuter avant que t_i ne soit terminé, et les données fournies par t_i ont été soumises à l'emplacement où t_j s'exécutera. La tâche t_i est un parent de t_j et t_j est un enfant de t_i. Pour chaque tâche $t_i \in \mathbf{T}$, nous utilisons $P(t_i)$ et $C(t_i)$ pour indiquer respectivement l'ensemble de ses parents et l'ensemble de ses enfants. $P(t_i) = \emptyset$ si t_i n'a pas de parents, et $C(t_i) = \emptyset$ si t_i n'a pas d'enfants. Chaque workflow scientifique G a un temps d'arrivée $a(G)$ et un deadline $d(G)$. Pour une tâche $t_i \in \mathbf{T}$, elle peut être modélisée comme t_i (s_i, a_i, d_i) où s_i, a_i et d_i représentent respectivement la taille, le temps d'arrivée et le deadline de t_i. Chaque deadline de tâche dans un workflow scientifique peut être déduit en fonction du deadline du workflow. Chaque tâche représente un programme de calcul et un ensemble de paramètres nécessaires à son exécution. Ce modèle est utilisé dans plusieurs WFMS, comme Askalon (Fahringer, et al., 2007), Taverna (Oinn, et al., 2004) et Pegasus (Deelman, et al., 2015).

Makespan, $m(G)$: est le temps global nécessaire à l'exécution complète du workflow. Le deadline $d(G)$ étant considéré comme une contrainte, alors que le makepan doit être inférieur au deadline ($m(G) \leq d(G)$). Le makespan du workflow est calculé comme suit : [m(G) =

complete$_{tn}$ – *ST*], où *ST* est le temps de soumission et *complete*$_{tn}$ est le temps complet du nœud de sortie du workflow.

***Chemin critique* (*Critical Path,* *CP*)** : est le chemin le plus long entre le nœud d'entrée et le nœud de sortie du workflow. Le makespan du workflow est déterminée par le chemin critique. Le chemin critique est calculé par la méthode « en largeur d'abord » en déterminant les poids de chaque nœud. Le poids du nœud est considéré comme le temps maximum de transfert de données et le temps écoulé des prédécesseurs estimé par l'équation 3.1 donnée par Topcuoglu et al. (Topcuoglu, Hariri, & Wu, 1999),

$$weight\ (t_i) = \max_{t_p \in\ pred\ (t_i)} \{weight\ (t_p) + \ w_p + \ c_{p,i}\} \qquad (4.1)$$

où *pred* (t_i) est l'ensemble les nœuds prédécesseurs de t_i, w_i est le temps d'exécution du nœud t_i sur un type de ressource sélectionné par l'algorithme. $c_{p,i}$ est le temps de transfert de données du nœud t_i à t_p. Le temps du chemin critique est le poids maximum parmi les nœuds de sortie. Lorsqu'un nœud termine son exécution, son poids et son temps de transfert de données vers tous ses nœuds successeurs sont rendus zéro et le chemin critique est recalculé.

Dans ce travail, nous ordonnançons les tâches de workflow scientifique sur un site d'exécution avec plusieurs ressources de calcul, telles que les instances VM sur des plates-formes Cloud (voir Figure 4.5).

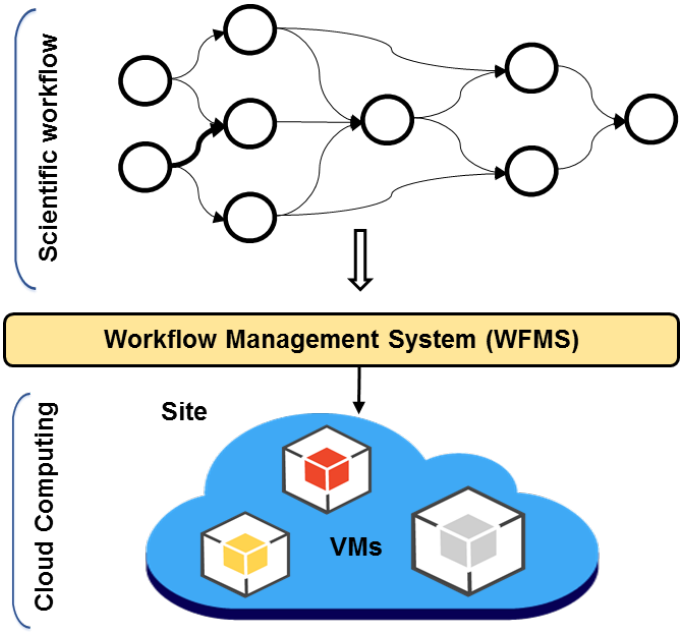


FIGURE 4.5 – Ordonnancement d'un workflow scientifique sur un environnement Cloud

La Figure 4.3 illustre un environnement d'exécution de workflow général qui cible des nœuds de calcul homogènes (par exemple, une machine virtuelle sur le Cloud). L'hôte de soumission (*Submit Host*) prépare le workflow pour l'exécution (par exemple, ordonnancement, clustering, etc.). Les jobs sont exécutés à distance sur les nœuds travailleurs (*Worker Node*).

4.4.2 Modèle de Cloud Computing

Chaque ressource des plates-formes Cloud peut être virtualisée vers un ensemble de machines virtuelles (VMs) via une technologie de virtualisation. Par conséquent, la VM est considérée comme une unité de traitement de base dans le système Cloud au lieu du serveur. Les plates-formes Cloud fournissent une compilation des ressources VMs dénotées par $VM = \{VM_1, \ldots, VM_i, \ldots, VM_k\}$ aux utilisateurs sur la base du modèle de paiement à l'utilisation (pay-per-use). Par exemple, Amazon EC2 propose différents types de VM de la série m4, qui sont indiqués dans le Tableau 4.1 (Amazon EC2, 2019). Ainsi, une instance VM_k est principalement définie par la capacité de traitement P_k (en unité de calcul) et le coût de location C_k (par heure). Amazon EC2 facture aux utilisateurs selon différents modèles de tarification, par exemple dans le modèle de tarification horaire, les utilisateurs doivent payer pour toute l'heure louée même si l'instance VM ne loue qu'une minute (Rodriguez & Buyya, 2014; Li, et al., 2018). La technologie de virtualisation permet d'accéder à un nombre infini de VMs dans la plate-forme Cloud afin que les utilisateurs puissent louer un nombre arbitraire de VMs. De plus, comme toutes les VMs sont situées dans le même centre de données, la bande passante entre les VMs est supposée être égale (Li, et al., 2016; Yao, Ding, & Hao, 2017).

VM type	Compute Unit	Cost per Hour ($)
m4.large	2	0.1
m4.xlarge	4	0.2
m4.2large	8	0.4
m4.4large	16	0.8
m4.10large	40	2
m4.16large	64	3.2

TABLEAU 4.1 – 4 séries de VM dans Amazon EC2

4.4.3 Modèle de faute

Réduire le makespan et le coût d'exécution est l'objectif principal de notre modèle de tolérance aux fautes. Le temps d'exécution requis pour le clustering des tâches et la tolérance des défaillances du job détermine le makespan. Par conséquent, nous considérons les fautes internes et externes dans un problème d'ordonnancement de workflow scientifique sur un environnement Cloud. Les défaillances de l'hôte, y compris les tâches de workflow et les instances VM, sont considérées comme des fautes internes. Ainsi, un détecteur de faute fiable est supposé détecter les défaillances de l'hôte (Manimaran & Murthy, 1998). De plus, les fautes de l'hôte peuvent être permanentes ou transitoires. Nous supposons que la probabilité que deux hôtes tombent en panne simultanément est faible, ce qui signifie qu'au maximum un hôte tombe en panne à la fois (Zhu, et al., 2016).

4.5 Clustering tolérant aux fautes

Le makespan et le coût d'exécution du workflow peuvent être affectés négativement par un clustering de tâches inadéquat dans des environnements Cloud défaillants. Pour améliorer notre

approche de tolérance aux fautes, nous proposons deux méthodes de clustering de tâches tolérantes aux fautes : le Clustering Critique (CC) et le Clustering Horizontal (HC).

4.5.1 Technique de clustering

L'objectif principal du clustering de tâches est de fusionner des tâches de petite taille du workflow en un job de taille relativement importante dans le but de réduire le makespan du workflow en minimisant la surcharge (overhead) du système. Par conséquent, le temps de la file d'attente diminuera par rapport aux tâches à granularité fine. Dans ce document, nous appliquons d'abord un clustering critique (CC) basé sur le chemin critique, dans lequel les tâches du même pipeline (tâches critiques) du chemin critique peuvent être regroupées pour former un cluster. Pour le reste des tâches (tâches non critiques), nous appliquons un clustering basé sur les niveaux, également appelé clustering horizontal (HC). Dans cette technique de clustering, les tâches au même niveau de workflow sont regroupées. Ensuite, nous fusionnons les tâches à différents niveaux en respectant le deadline des tâches (voir Figure 4.6).

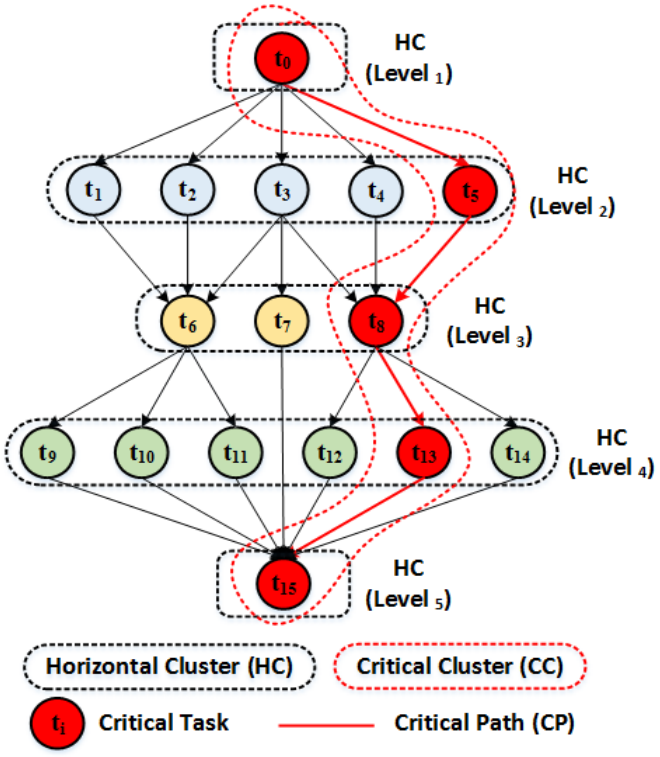


FIGURE 4.6 – Exemple de workflow en cluster

4.5.2 Tolérance aux fautes

Nous utilisons la méthode de réplication pour assurer la tolérance aux fautes et garantir l'achèvement de toutes les tâches avant leur deadline. Nous appliquons une réplication active pour les tâches du chemin critique et une réplication passive pour les tâches non critiques. Ainsi, nous exploitons le temps d'inactivité des VMs pour la réplication.

4.5.2.1 Réplication active

La réplication active est la procédure qui consiste à créer la même copie des données existantes. Dans notre cas, nous créons de multiples répliques des tâches critiques afin que les répliques soient utilisées pour poursuivre l'exécution lorsqu'une défaillance se produit. Chaque copie est allouée à une unité de traitement (VM) différente (voir Tableau 4.2, Figure 4.7).

Tâche	Taille	Début	Fin	#VM
t_0 , t_0'	0	0	0	VM_1, VM_2
t_5 , t_5'	90	0	90	VM_1, VM_2
t_8 , t_8'	40	90	130	VM_1, VM_2
t_{13} , t_{13}'	70	130	20	VM_1, VM_2
t_{15} , t_{15}'	0	200	200	VM_1, VM_2

TABLEAU 4.2 – Temps de calcul des tâches critiques et de leurs répliques

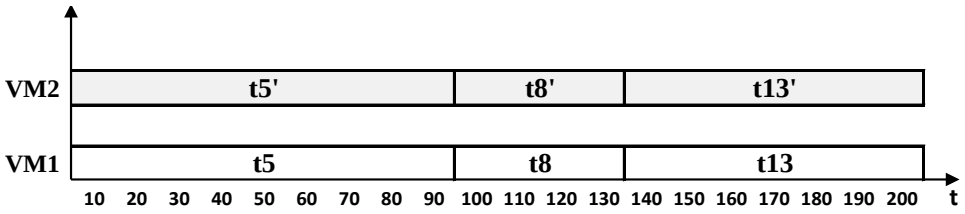


FIGURE 4.7 – Ordonnancement des tâches critiques et de leurs répliques sur différentes VMs

4.5.2.2 Réplication passive

Dans cette stratégie de tolérance aux fautes, nous utilisons un modèle de sauvegarde primaire (PB, en bref). Avec le modèle PB, une tâche non critique n'obtient que deux copies : copie principale (PR) et copie de sauvegarde (BK), dans lesquelles chaque copie est allouée à une unité de traitement (VM) différente. Si la copie principale (PR) échoue, la copie de sauvegarde (BK) prend le relais, ce qui peut entraîner une certaine interruption de service. Ce type de réplication est basé sur la sauvegarde de l'état interne de la tâche, donc un point de reprise (checkpointing) supplémentaire est donc utilisé (voir Tableau 4.3, Figure 4.8).

Tâche	Taille	Début	Fin	#VM
t_1 (PR), t_1 (BK)	50	0	50	VM_3,VM_4
t_2 (PR), t_2 (BK)	20	0	20	VM_7,VM_8
t_3 (PR), t_3 (BK)	40	0	40	VM_5,VM_6
t_4 (PR), t_4 (BK)	30	20	50	VM_8,VM_7
t_6 (PR), t_6 (BK)	60	50	110	VM_4,VM_3
t_7 (PR), t_7 (BK)	80	40	120	VM_6,VM_5
t_9 (PR), t_9 (BK)	80	110	190	VM_3,VM_4
t_{10} (PR), t_{10} (BK)	10	170	180	VM_6,VM_5
t_{11} (PR), t_{11} (BK)	30	160	190	VM_8,VM_7
t_{12} (PR), t_{12} (BK)	30	130	160	VM_7,VM_8
t_{14} (PR), t_{14} (BK)	40	130	170	VM_5,VM_6

TABLEAU 4.3 – Temps de calcul des tâches non critiques et de leurs copies

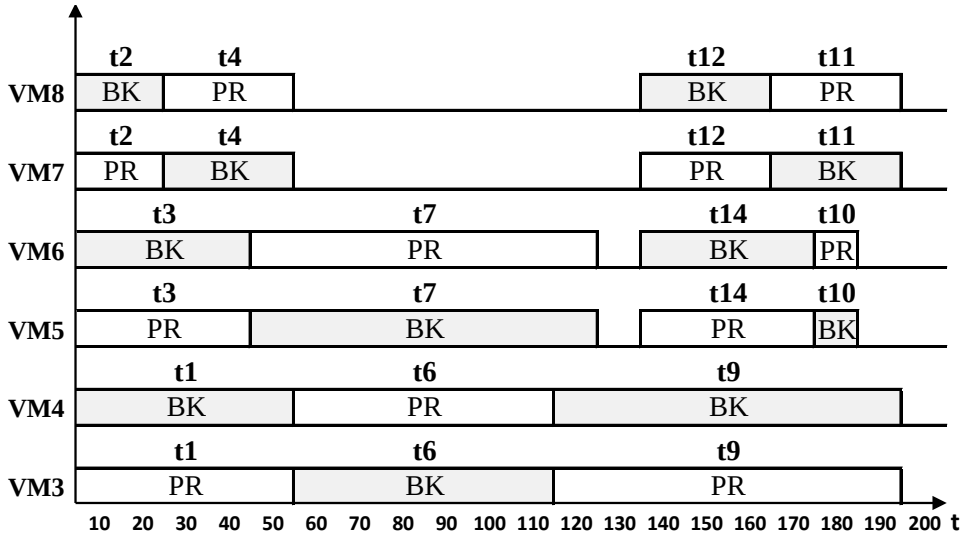


FIGURE 4.8 – Ordonnancement des tâches non critiques avec une copie primaire (pr) et une copie de sauvegarde (bk) sur différentes VMs

4.5.3 Implémentation de l'algorithme

Nous présentons un algorithme d'ordonnancement basé sur un clustering tolérant au fautes FT-HCC (*Fault Tolerant-Horizontal and Critical Clustering*) pour un workflow donné. L'algorithme 4.1 montre le pseudo-code de la méthode FT-HCC. Les notations utilisées dans l'algorithme sont énumérées dans le Tableau 4.4. Nous donnons en détail l'implémentation de l'algorithmes proposé comme suit. La procédure de CLUSTERING est citée dans le processus de clustering de tâches initial, elle permet de calculer le chemin critique (CP) pour définir le cluster critique (CC) et fusionner les tâches à différents niveaux horizontaux pour créer des clusters horizontaux (HC). Cette procédure crée une réplication pour chaque type de cluster (actif pour CC et passif pour HC) et prépare une liste d'ordonnancement (SL) en fonction du deadline des tâches. La procédure MAPPING est utilisée pour mapper les clusters sur différentes VMs respectivement pour exécution. La procédure RÉPLICATION permet de trouver les tâches échouées parmi toutes les tâches du workflow. Une fois qu'une tâche critique échouée a été détectée, la procédure prend une réplique de cette tâche et l'exécute jusqu'à ce qu'elle soit terminée avec succès (réplication active). Si une tâche non critique échouée a été détectée, la procédure utilise une copie de sauvegarde pour accomplir la tâche (réplication passive). La procédure est répétée jusqu'à ce que toutes les tâches du workflow soient achevées.

Algorithme 4.1 – FT-HCC Algorithm

Input: *W*: Workflow; *T*: Number of tasks per workflow; C_{size}: Clusters size; C_{num}: Clusters number; VM_{num}: Number of VMs

1: **Procedure** CLUSTERING (W; T)
2: Compute *CP* using Equation (1);
3: *CT* ← Tasks t_i in *CP;*
4: *TL*, *TL'* ← {};
5: **for all** Task t_i in *CT* **do**

```
6:      TL.add (t_i);
7:          t_i' ← Replica (t_i);
8:      TL'.add (t_i');
9:  end for
10: CC ← (TL, SL(d(t_i)));
11:     CC' ← (TL', SL(d(t_i')));
12: TL ← {};
13: for (i=1; i ≤ depth (W); i++) do
14:     for all Task t_ij at level_i and t_ij not in CC do
15:         TL.add(pr(t_ij));
16:         TL.add(bk(t_ij));
17:     end for
18: end for
19:     Add Tasks in TL to C_i by respecting the C_size and C_num;
20:         CL_i ← Merge (Ci);
21: HC ← (CL_i, SL (d(t_ij)));
22: end Procedure
23: Procedure MAPPING (CC; CC'; VM_num)
24:     while CC is unscheduled do
25:       Assign CC to VM;
26:       Assign CC' to VM';
27:     end while
28:     VM_idle ← VM_num -2;
39: end Procedure
30: Procedure MAPPING (HC; VM_idle)
31:     while HC is unscheduled do
32:         SELECT CL_i from HC
33:         for (i=1; i ≤ VM_idle; i++) do
34:             Assign CL_i to VM_i
35:         end for
36:     end while
37: end procedure
38: Procedure REPLICATION (HC; CC; SL)
39:     if task t_i fails then
40:         if t_i in CC then
41:             Take replica of failed Task t_i and execute t_i
42:             until it successfully
43:         else
44:             Take backup-copy of failed Task t_i and
45:             execute t_i it successfully
46:         end if
47:     end if
48: end Procedure
```

Parameters	Definition
CP	Critical Path
CT	Critical Tasks List
TL	Task List
t_i	Task identifier

t_i'	Replica of t_i
t_{ij}	Tasks at $level_i$
$pr(t_{ij})$	Primary Copy of t_{ij}
$bk(t_{ij})$	Backup Copy of t_{ij}
$d(t_{ij})$	Deadline of t_{ij}
CC	Critical Clusters
HC	Horizontal Clusters
C_i	Cluster identifier
CL_i	Cluster List
SL	Scheduling List
VM_{idle}	Idle VM

TABLEAU 4.4 – Les notations utilisées dans la conception de l'algorithme

La complexité temporelle pour calculer le chemin critique pour toutes les tâches prêtes est $O(n^2)$, où n est le nombre de tâches. Ainsi, la complexité de l'algorithme de mappage (ordonnancement) de chaque tâche sur différentes VMs est $O(n)$. Par conséquent, la complexité temporelle de l'algorithme est $O(n^2)$.

Différentes méthodes de clustering pour la tolérance aux fautes sont disponibles pour réduire l'impact des défaillances. Pour l'étude, nous considérons quelques méthodes issues de systèmes existants qui fonctionnent plus particulièrement comme suit (voir Tableau 4.5).

Algorithme	Description
Selective Reclustering (SR) (Chen, da Silva, Deelman, & Fahringer, 2016)	- Basé sur le clustering horizontal (Singh, et al., 2008) - Fusionne les tâches défaillantes au sein d'un job clusterisé en un nouveau job clusterisé pour le relancer. - Ne pas analyser la taille du clustering
Dynamic Reclustering (DR) (Chen, da Silva, Deelman, & Fahringer, 2016)	- Basé sur le clustering horizontal - Ajuste la taille du clustering des jobs pour réduire l'impact des défaillances de tâches - Fusionne les tâches défaillantes dans de nouveaux jobs clusterisés en définissant la taille du clustering et des relances
Vertical Reclustering (VR) (Chen, da Silva, Deelman, & Fahringer, 2016)	- Basé sur le clustering horizontal - Relance les tâches défaillantes ou non terminées - Ne pas analyser la taille du clustering
Horizontal Reclustering (HR) (Dharwadkar, Poojara, & Kadam, 2018)	- Combinaison de HR et de DR. - Fusionne les tâches défaillantes de même niveau horizontal en un nouveau job clusterisé pour le relancer. - Ajuste la taille du clustering de manière dynamique

TABLEAU 4.5 – Comparaison de divers algorithmes de clustering de tâches tolérants aux fautes existants

4.5.4 Analyse

L'objectif de ce travail est de réduire le makespan et le coût d'exécution du workflow tout en respectant la contrainte du deadline même en cas de défaillance. Dans ce cas, considérons que $replica(t_i)$ est le nombre de réplications (actives et passives) de la tâche t_i. Par conséquent, le temps d'exécution et le coût d'exécution de la tâche t_i sont calculés par :

$$T_{exec}(t_i) = T_{exec}(t_i, VM_k) + T_{exec}(replica(t_i), VM_k) \quad (4.2)$$

$$C_{exec}(t_i) = C_{exec}(t_i, VM_k) + C_{exec}(replica(t_i), VM_k) \quad (4.3)$$

Où $T_{exec}(t_i, VM_k)$ et $C_{exec}(t_i, VM_k)$ sont respectivement le temps d'exécution et le coût d'exécution d'une seule copie de la tâche t_i exécutée sur VM_k, $T_{exec}(replica(t_i), VM_k)$ et $C_{exec}(replica(t_i), VM_k)$ sont le temps d'exécution et le coût d'exécution des répliques de tâches t_i exécutées sur VM_k.

Ainsi, le makespan (temps total d'exécution) et le coût total d'exécution du workflow (W) peuvent être calculés par :

$$T_{exec}(W) = \sum_{t_i \in T} T_{exec}(t_i) \quad (4.4)$$

$$C_{exec}(W) = \sum_{t_i \in T} C_{exec}(t_i) \quad (4.5)$$

Finalement, le problème d'ordonnancement du workflow peut être formellement décrit comme suit : trouver un schéma d'ordonnancement pour réduire le temps total d'exécution et le coût total d'exécution, et le *makepan* (m) est égal ou inférieur au *deadline* (d), qui est indiquée comme suit :

$$\text{Minimiser : } T_{exec}(W), C_{exec}(W) \quad (4.6)$$

$$\text{Contrainte: } m(W) \leq d(W) \quad (4.7)$$

4.6 Expérimentations de la simulation

Dans cette section, des expérimentations ont été adoptées pour évaluer l'approche de clustering de tâches tolérant aux fautes FT-HCC avec cinq applications de workflow scientifique, qui sont LIGO (détection des ondes gravitationnelles), Montage (production de mosaïques du ciel), Epigenome (analyse des données épigénomiques humaines), SIPHT (bioinformatique) et Cybershake (caractérisation des risques sismiques) décrites précédemment (Bharathi, et al., 2008; Juve, et al., 2013). Ces applications ont un modèle de workflow et des caractéristiques de calcul différents. Nous pratiquons une approche basée sur la simulation en faisant varier les paramètres du système afin d'évaluer les performances de notre technique de clustering.

4.6.1 Conditions d'expérimentation

Les évaluations des performances de l'approche FT-HCC et la comparaison avec d'autres algorithmes existants ont été implémentées sur WorkflowSim (Chen & Deelman, 2012). Nous étendons le simulateur WorkflowSim avec notre technique de clustering et d'autres méthodes de clustering tolérantes aux fautes pour simuler un environnement Cloud vérifié. Les paramètres du système (c.-à-d., Le taux de défaillance des tâches) et du workflow (c.-à-d., La taille des tâches) sont variables et les performances de notre approche de clustering de tâches tolérant aux fautes sont entièrement explorées. WorkflowSim est un simulateur de workflow open source étendu à partir de CloudSim (Calheiros, Ranjan, Beloglazov, De Rose, & Buyya, 2011) en fournissant des fonctionnalités supplémentaires de prise en charge de l'approvisionnement des ressources, de l'ordonnancement des tâches et du clustering des tâches au niveau du workflow. Il a été intégré dans divers domaines d'étude des workflows scientifiques (Chen & Deelman, 2012).

Dans cette étape, nous définissons les paramètres des cloudlets (tâches) et des VMs et sélectionnons les cloudlets pour le mappage des instances VM conformément aux méthodes comparées. La plate-forme simulée dispose d'un Datacenter contenant 20 VMs de ressources homogènes uniques (worker nodes), qui modélise certains environnements Cloud typiques tels qu'Amazon EC2. Chaque VM simulée dispose de 512 Mo de mémoire (RAM), d'une capacité de traitement de 1 000 millions d'instructions par seconde (MIPS), et la bande passante réseau par défaut est de 15 Mo selon notre environnement d'exécution (Chen, da Silva, Deelman, & Fahringer, 2016). Pour chaque tâche du workflow, la bande passante réseau et la charge de travail (workload) sont représentées par la vitesse de transfert de données maximale autorisée entre une paire de VMs par fichier. Les paramètres pertinents pour les expérimentations sont présentés dans le Tableau 4.6.

Paramètres	Valeurs
Nombre de Datacenters	1
Nombre d'hôtes	2
Nombre total de VMs	20
Fréquence VM	1,000 MIPS
Mémoire VM (RAM)	512 Mo
Bande passante VM	15
Nombre de PEs requis	2
Coût par heure ($)	0.1

TABLEAU 4.6 – Paramètres des expérimentations

Workload dataset. Nous avons collecté des traces d'exécution de workflows à partir d'exécutions réelles des cinq applications de workflow scientifique en temps réel précédemment rapportées. Ces traces sont exploitées pour créer des workflows synthétiques à l'aide de toolkit Workflow Generator (da Silva, Chen, Juve, Vahi, & Deelman, 2014). Le Toolkit Workflow Generator traite les données statistiques collectées à partir des traces des workflows scientifiques courants pour générer des applications synthétiques qui simulent les workflows réels. La structure du workflow synthétique est déterminée par le nombre d'entrées, la taille des tâches et leurs fonctionnalités. Les caractéristiques du workflow, les traces, les

modèles et les données de profil sont disponibles gratuitement et en ligne pour la communauté (Workflow Generator, 2020). Pour nos expérimentations, nous générons un workflow synthétique pour chaque application de workflow en fonction des caractéristiques présentées dans le Tableau 4.7.

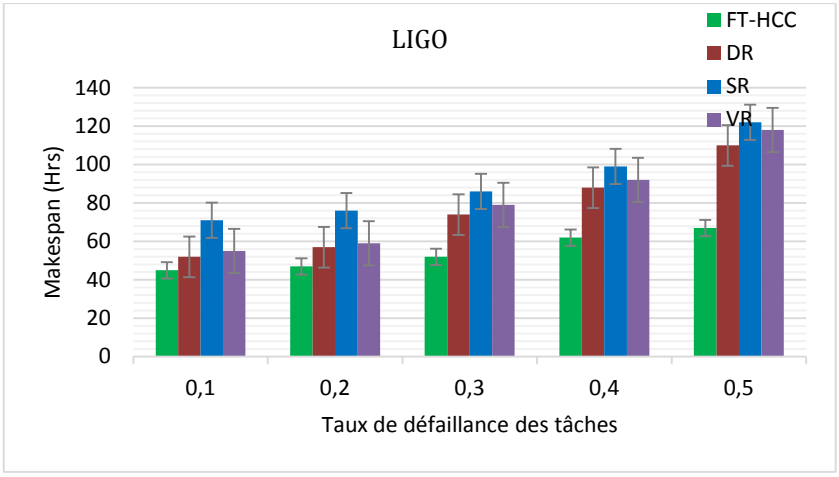
	# Tâches	Taille de la tâche (moy)
LIGO	800	228 s
Montage	300	11 s
Epigenomics	165	2952 s
SIPHT	1,000	180 s
CyberShake	700	23 s

TABLEAU 4.7 – Caractéristiques principales du workflow

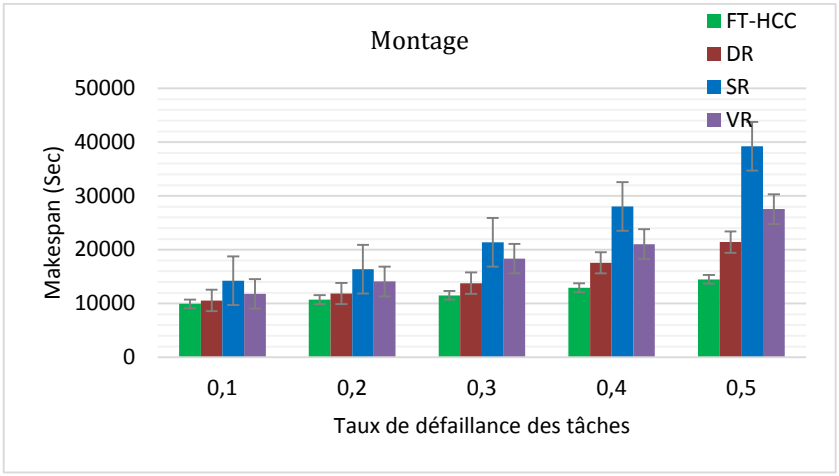
4.6.2 Résultats et discussion

Nous évaluons les performances et la fiabilité de notre algorithme FT-HCC de clustering de tâches tolérant aux fautes par rapport à des méthodes similaires existantes (Dynamic Reclustering (DR), Vertical Reclustering (VR) et Selective Reclustering (SR)) (Chen, da Silva, Deelman, & Fahringer, 2016). À ce propos, nous conduisons une série d'expérimentations pour identifier les conditions dans lesquelles chaque approche donne les meilleurs résultats. Cependant, nous évaluons l'amélioration des performances pour différents paramètres de taille et de taux de défaillance des tâches. Les mesures observées sont le makespan et le coût obtenus lors de l'exécution des workflows.

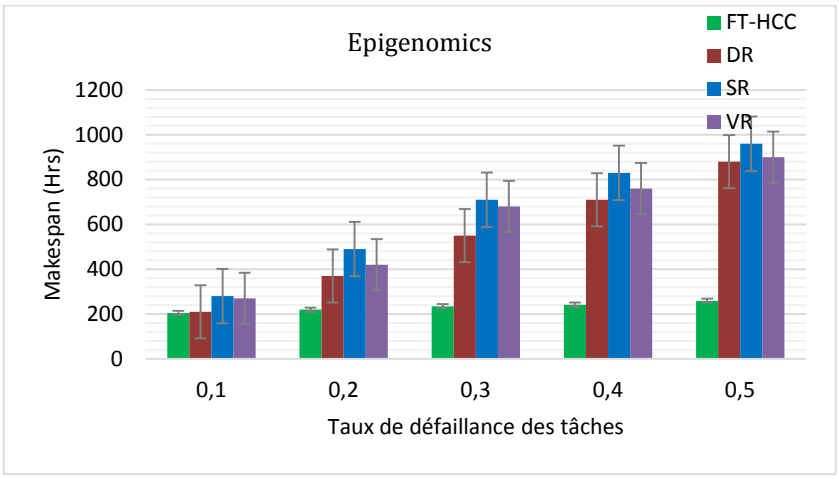
Expérimentation 1 (évaluation de makespan) : Le makespan encouru pour les cinq workflows scientifiques réalistes avec différentes méthodes de clustering tolérant au fautes est illustrée dans la Figure 4.9. D'après ce graphe, il est clairement vu que le makespan augmente avec l'augmentation du taux de défaillance des tâches dans les méthodes FT-HCC, DR, SR et VR. Parmi tous les algorithmes, la méthode FT-HCC donne les meilleures performances avec moins de makespan en faisant varier le taux de défaillance des tâches. Par conséquent, la méthode proposée utilise la réplication active pour les clusters critiques ainsi elle exploite le temps d'inactivité des VMs pour ordonnancer les tâches défaillantes, ce qui permet de consommer moins de temps pour compléter le workflow et permet de mieux respecter le deadline par rapport aux algorithmes DR, SR et VR.



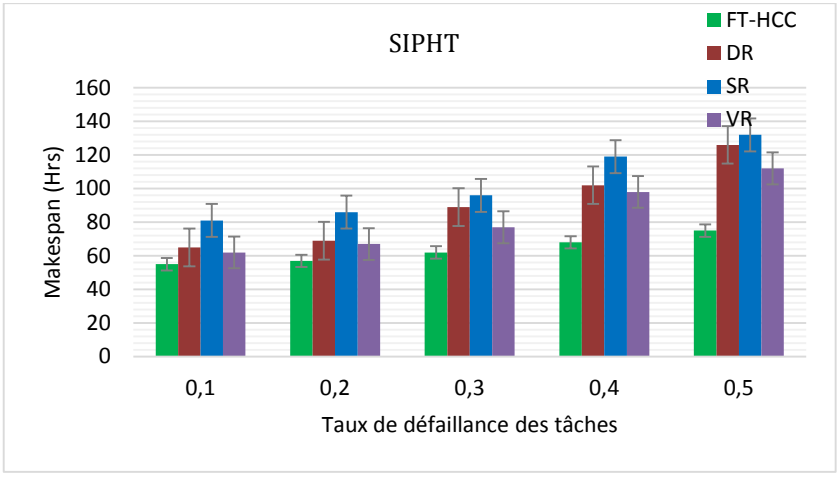
(a) Workflow LIGO



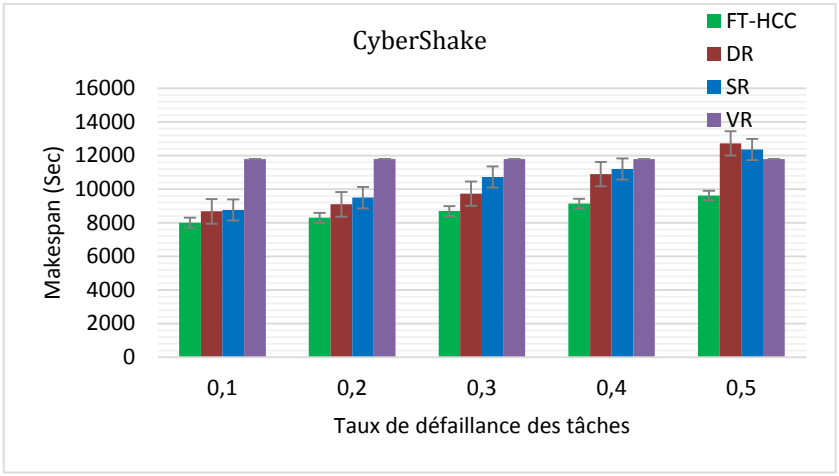
(b) Workflow Montage



(c) Workflow Epigenomics



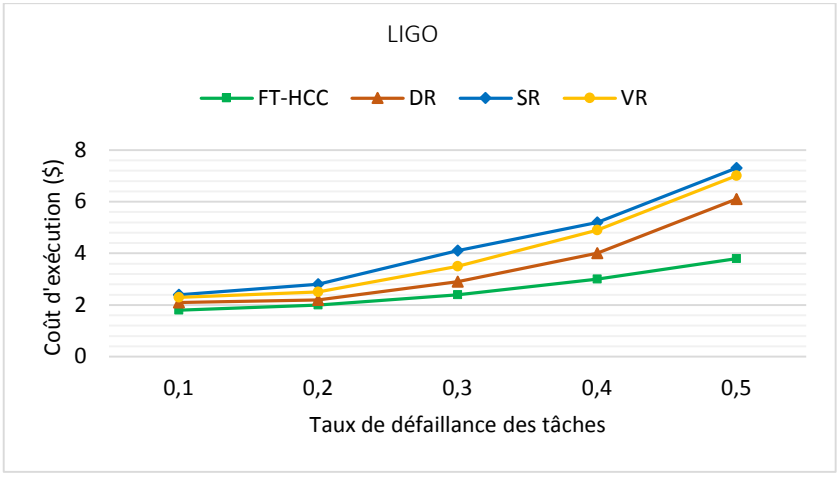
(d) Workflow SIPHT



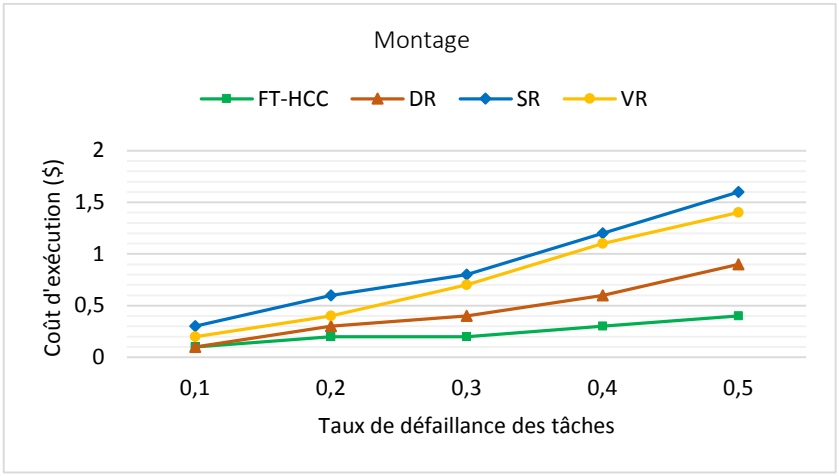
(e) Workflow CyberShake

FIGURE 4.9 – Évaluation du makespan pour différentes valeurs du taux de défaillance des tâches

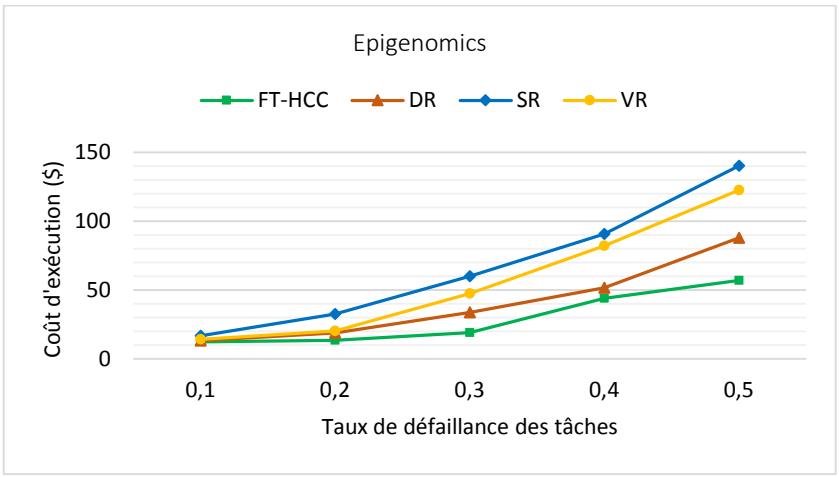
Expérimentation 2 (évaluation du coût d'exécution) : la Figure 4.10 montre le coût d'exécution du calcul pour les cinq workflows scientifiques. Le coût d'exécution inclut toutes les répliques pour tolérer les tâches défaillantes ainsi que les tâches terminées avec succès. Le coût d'exécution de tous les algorithmes augmente avec l'augmentation du taux de défaillance des tâches, sauf que le coût d'exécution obtenu par l'approche FT-HCC proposée augmente beaucoup plus lentement que les heuristiques DR, SR et VR. Comme nous utilisons une réplication passive pour les tâches les plus défaillantes, qui est moins coûteuse rapport à la technique de retrying des tâches, l'algorithme FT-HCC considère le temps d'inactivité des VMs pour la réplication afin de réduire le coût d'exécution du workflow.



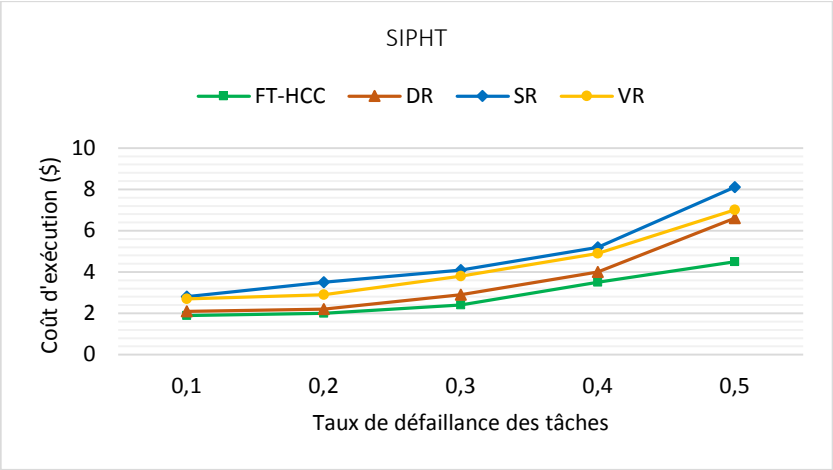
(a) Workflow LIGO



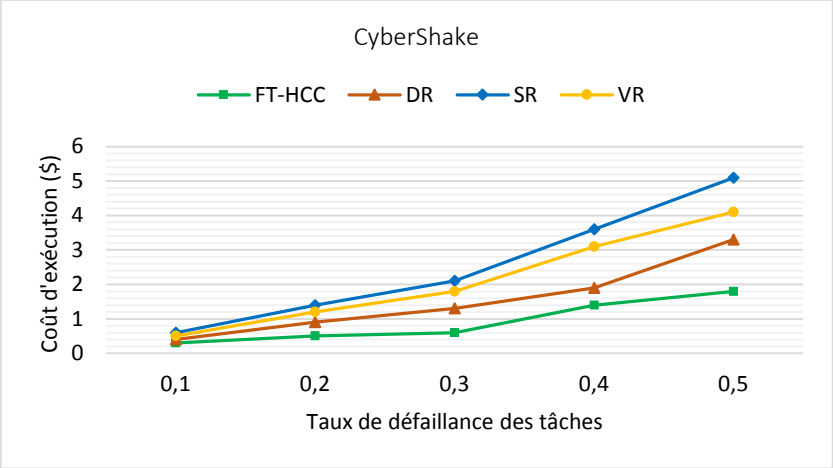
(b) Workflow Montage



(c) Workflow Epigenomics



(d) Workflow SIPHT



(e) Workflow CyberShake

FIGURE 4.10 – Évaluation des coûts d'exécution pour différentes valeurs du taux de défaillance des tâches

Les résultats de ces simulations concluent qu'il y a un gain substantiel de makepan ainsi qu'une réduction considérable des coûts d'exécution pendant l'exécution des workflows en utilisant notre algorithme FT-HCC par rapport aux méthodes DR, SR et VR.

4.7 Conclusion

Dans ce travail, nous avons proposé un nouvel algorithme FT-HCC pour améliorer l'ordonnancement tolérant aux fautes des tâches de workflow et nous l'avons appliqué sur cinq workflows scientifiques en temps réel. Les résultats de ces expérimentations concluent que l'algorithme proposé améliore considérablement le makespan ainsi que le coût d'exécution du workflow par rapport à des méthodes de clustering de tâches tolérant aux fautes antérieures utilisées dans les WFMS et ordonnancées sur un environnement de ressources distribuées de Cloud. La technique de réplication appliquée, ainsi que le temps d'inactivité des VMs exploité pour ordonnancer les tâches défaillantes sont considérés pour économiser le makespan et le

coût lors de l'exécution du workflow. Nous limitons notre modèle de défaillance pour ne prendre en compte que les défaillances des VMs (worker nodes) et les défaillances de tâches. La cause des fautes pourrait cependant être envisagée dans un modèle de fiabilité plus sophistiqué. De plus, nous supposons le cas où toutes les VMs sont dans le même datacenter, connectées par des liens à large bande passante et à faible latence. Ainsi, notre méthode ne prend pas en compte le temps de réplication qui dépend des VMs entre lesquelles la réplication est effectuée. Par conséquent, nous nous concentrons dans ce travail sur l'évaluation des performances de l'ordonnancement tolérant aux fautes des méthodes de clustering des tâches sur l'approvisionnement des ressources homogènes. Dans le futur, nous prévoyons de combiner notre contribution avec un ordonnancement tolérant aux fautes pour des applications de workflow dans des environnements distribués hétérogènes, et avoir un algorithme d'ordonnancement qui évite de mapper des tâches clusterisées à des ressources instables avec des modèles de prédiction de fautes. Nous avons également l'intention de considérer différents modèles de réseau pour analyser leur impact sur notre stratégie de clustering de tâches tolérant aux fautes. Les travaux futurs porteront également sur d'autres facteurs tels que la consommation d'énergie et l'efficacité énergétique, qui pourraient améliorer la précision du modèle proposé.

Conclusion et Perspectives

Les travaux présentés dans cette thèse portent sur la tolérance aux fautes à base de clusterisation dans les systèmes distribués à large échelle. P2P, grilles de calcul et Cloud Computing représentent des infrastructures très intéressantes pour répondre à la demande croissante de puissance de calcul, en ce qui concerne des applications de calcul de haute performance. Ces infrastructures pourraient résoudre des problèmes scientifiques à grande échelle en utilisant des ressources hétérogènes distribuées géographiquement, en mettant en place une puissance de calcul impossible à obtenir de manière individuelle. Les principales difficultés proviennent de la nature même de ces systèmes, qui sont des architectures très hétérogènes et fortement dynamiques. De nombreux travaux de recherche ont proposé des solutions permettant de faire face à une partie de ces difficultés. En ce qui nous concerne, nous nous sommes intéressés au problème de tolérance aux fautes dans les grilles de calcul et le Cloud Computing. Dans ces architectures, 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. Il est donc nécessaire de gérer ce risque élevé de défaillances, en proposant des techniques de tolérance aux fautes basé sur la clusterisation qui puissent prendre en compte cet aspect dans la gestion globale des grilles de calcul ainsi que dans le Cloud Computing, tant au niveau infrastructure qu'au niveau applicatifs. Vu l'importance et l'intérêt réel d'une telle problématique, de nombreux travaux ont été menés sur le thème général de sûreté de fonctionnement dans les systèmes distribués à large échelle, et en particulier les grilles de calcul et le Cloud Computing. En ce qui concerne, nos travaux, nous nous sommes orientés vers deux axes complémentaires pour la gestion de la tolérance aux fautes dans les grilles d'abord puis dans le Cloud Computing, pour satisfaire aux exigences de qualité de service (QoS) et pour augmenter les performances. Cette gestion sera d'autant plus facile, si cette tolérance aux fautes se base sur des modèles théoriques représentant ces architectures. Pour cela, nous avons concentré nos réflexions sur deux types de modèles.

1. Les modèles de tolérance aux fautes basés sur la clusterisation des ressources dans des environnements de grille de calcul
2. Les modèles de tolérance aux fautes basés sur la clusterisation des tâches d'un workflow scientifique dans des environnements de Cloud Computing.

Contributions

Les travaux que nous avons menés dans ce contexte, ainsi que les résultats que nous avons obtenus, permettent de situer nos contributions comme suit :

– Comme première contribution, nous nous sommes intéressés aux grilles de calcul, où nous avons proposé un modèle de tolérance aux faute basé sur la clusterisation des ressources de la grille. Ce modèle exploite les graphes colorés dynamiques pour capter les trois caractéristiques d'une grille, à savoir l'hétérogénéité, la dynamicité et le passage à l'échelle. La richesse de ces graphes, notamment grâce aux couleurs, nous a permis

de mettre en place deux techniques très intéressantes de tolérance aux fautes, la première se base sur un algorithme de clustering à *1*-saut appelé HCC pour transformer la grille en un ensemble de clusters interconnectés, et la deuxième se base sur une fonction de *scoring* pour calculer le niveau de performance de chaque nœud en fonction de leur caractéristiques physiques et logiques. Le modèle proposé montre l'efficacité de la méthode de *scoring* et le gain obtenu en utilisant le niveau de performance des nœuds de la grille pour trouver des substituts qui remplacent le nœud défaillant en commençant la recherche dans leur cluster puis les plus proches substituts dans les clusters voisins.

- La deuxième contribution investit dans l'augmentation des performances de l'exécution d'un système de workflow scientifique dans un environnement de Cloud Computing, comme un critère de base pour la tolérance aux fautes. Elle tente de réduire de temps d'exécution (*makespan*) et le coût d'exécution du workflow en appliquant des techniques de clustering des tâches basées sur le chemin critique et sur les niveaux horizontaux du workflow, ainsi que des mécanismes de réplication active pour les tâches appartiennes au chemin critique et de réplication passive pour le reste des tâches.

L'ensemble des propositions qui ont été faites dans cette thèse ont été validées expérimentalement soit à partir de simulateurs existants, soit à partir de simulateurs que nous avons développés pour nos propres besoins.

Perspectives

Les résultats théoriques et pratiques que nous avons obtenus dans le cadre de cette thèse, permettent d'envisager un certain nombre de perspectives de recherche fort intéressantes.

Plusieurs perspectives nous semblent très pertinentes: (i) l'adaptation des modèles proposés dans cette thèse, à des environnements Internet des Objets (IoT) par l'intégration des systèmes de type Fog Computing et Edge Computing qui sont considérés, dans certains cas, comme des extensions des systèmes Cloud; (ii) combiner notre contribution avec un ordonnancement tolérant aux fautes pour un ensemble de workflows dans des environnements distribués hétérogènes, et proposer une technique d'ordonnancement qui évite de mapper des tâches clusterisées à des ressources instables avec des modèles de prédiction de fautes; (iii) étendre notre approche par l'intégration de d'autres techniques de tolérance aux fautes tel que les points de reprises (*checkpoonting*); (iv) considérer d'autres facteurs tels que la consommation d'énergie et l'efficacité énergétique, qui pourraient améliorer la précision du modèle proposé.

Publications de l'auteur

1. Journal international

- **Khaldi, M**., Rebbah, M., Meftah, B., & Smail, O. (2019). Fault tolerance for a scientific workflow system in a Cloud computing environment. *International Journal of Computers and Applications*. DOI: 10.1080/1206212X.2019.1647651

2. Conférences internationales avec actes et comité de lecture

- **Khaldi, M**., Rebbah, M., & Meftah, B. (2017): Fault Tolerance in Grid Computing by Resource Clustering, *3rd International Conference on Networking and Advanced Systems (ICNAS'2017)*, Annaba, Algeria, 13-14 December.

- **Khaldi, M**., Rebbah, M., & Meftah, B. (2018): Fault tolerance for a scientific workflow system in a Cloud computing environment, *3rd International Conference on multimedia information processing (CITIM'2018)*, Mascara, Algeria, 09-10 October.

- Rebbah, M., Yermes, M., **Khaldi, M.**, & Debakla, M. (2015). Hybrid Distribution for Association Rules Extraction on Grid Computing. *Proceedings of 2015 International Conference on Image Processing, Production and Computer Science (ICIPCS'2015)*, (pp. 14-22), Istanbul, Turkey, 3-4 June.

- Rebbah, M., Mokhtari, C., **Khaldi, M.**, Bourasi, M. F., & Smail, O. (2010). Hierarchical model for fault tolerant grid computing over Globus Toolkit. International Congress on Models, Optimization and Security of Systems (ICMOSS), Tiaret, Algeria.

Bibliographie

Abbes, H., & Crin, C. (2010). A decentralized and fault-tolerant desktop grid system for distributed applications. *Concurrency and Computation: Practice and Experience, 22*(3), 261–277.

Abramovici, A., Althouse, W., Drever, R., Gürsel, Y., Kawamura, S., Raab, F., . . . Zucker, M. (1992). LIGO: the laser interferometer gravitational-wave observatory. *Science, 256*(5055), 325–333.

Ahmed, W., & Wu, Y. W. (2013). A survey on reliability in distributed systems. *Journal of Computer and System Sciences, 78*(8), 1243–1255.

Ahn, J. (2007). Efficient failure detection and recovery scheme for hierarchical distributed monitoring, future generation communication and networking. *In Proceedings of the Future Generation Communication and Networking Conference (FGCN'2007),* (pp. 510–515). Jeju, South Korea: IEEE.

Aliaa, A. A., Atef, Z. G., & Mohammed, E. E. (2010). An efficient decentralized grid service advertisement approach using multi-agent system. *Computer and Information Science, 3*(2), 220–228.

Almond, J., & Snelling, D. F. (1999). UNICORE : uniform access to supercomputing as an element of electronic commerce. *Future Generation Computer Systems, 15*(5-6), 539–548.

Amazon EC2. (2019). Retrieved from http://aws.amazon.com/ec2

Amis, A. d., & Prakash, R. (2000). Load-balancing clusters in wireless ad hoc networks. *In Proceedings 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET)*, (pp. 25–32). Richardson, TX, USA.

Amis, A. D., Prakash, R., Vuong, T. H., & Huynh, D. T. (2000). Max-min d-cluster formation in wireless ad hoc networks. *In Proceedings IEEE INFOCOM 2000*, (pp. 32–41).

Amoon, M., El-Bahnasawy, N., Sadi, S., & Wagdi, M. (2019). On the design of reactive approach with flexible checkpoint interval to tolerate faults in cloud computing systems. *Journal of Ambient Intelligence and Humanized Computing, 10*, 4567–4577.

Anderson, D. P. (2004). BOINC: a system for public-resource computing and storage. *In 5th IEEE/ACM International Workshop on Grid Computing (GRID'04)*, (pp. 4–10). Pittsburgh, PA, USA.

Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., & Werthimer, D. (2002). SETI@home: an experiment in public-resource computing. *Communications of the ACM, 45*(11), 56–61.

Ang, T., Ng, W., Ling, T., Por, L., & Liew, C. (2009). A bandwidth-aware job grouping-based scheduling on grid environment. *Information Technology Journal, 8*(3), 372–377.

Antoniu, G., Hatcher, P., Jan, M., & Noblet, D. A. (2005). Performance evaluation of jxta communication layers. *In IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, *1*, pp. 251–258. Cardiff, Wales, UK.

Ardagna, D., Trubianb, M., & Zhangc, L. (2007). SLA based resource allocation policies in autonomic environments. *Journal of Parallel and Distributed Computing, 67*(3), 259–270.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., & Katz, R. (2009). *Above the Clouds: A Berkeley View of Cloud Computing.* Technical Report No. UCB/EECS-2009-28, EECS Department, University of California at Berkeley.

Avizienis, A., Laprie, J. C., & Randell, B. (2004). Dependability and Its Threats: A Taxonomy. *In Building the Information Society, IFIP 18th World Computer Congress*, (pp. 91–120). Toulouse, France.

Baker, M., Buyya, R., & Laforenza, D. (2002). Grids and grid technologies for wide area distributed computing. *Software: Practice and Experience, 32*(15), 1437–1466.

Banerjee, S., & Khuller, S. (2001). A clustering scheme for hierarchical control in multi-hop wireless networks. *In Proceedings IEEE INFOCOM 2001.*, *2*, pp. 1028–1037. Anchorage, AK, USA.

Barker, A., & Van Hemert, J. (2008). Scientific Workflow: A Survey and Research Directions. *In International Conference on Parallel Processing and Applied Mathematics (PPAM'07),* (pp. 746–753). Gdansk, Poland: Springer, Berlin, Heidelberg.

Beaumont, O., Bonichon, N., Duchon, P., Eyraud-Dubois, L., & Larchevêque, H. (2008). A distributed algorithm for resource clustering in large scale platforms. *In International Conference On Principles Of Distributed Systems (OPODIS'08),* (pp. 564–567). Luxor, Egypt: Springer, Berlin, Heidelberg.

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

Berge, C. (1976). *Graphs and Hypergraphs* (2nd edition ed.). North-Holland.

Berman, F., Fox, G., & Hey, A. J. (2003). *Grid Computing : Making the Global Infrastructure a Reality.* Wiley.

Berriman, G. B., Deelman, E., Good, J., Jacob, J., Kat, D. S., Kesselman, C., . . . Su, M. H. (2004). Montage: A grid enabled engine for delivering custom science-grade mosaics on demand. *In Proceedings of SPIE - The International Society for Optical Engineering*, *5493*, pp. 221–232. Glasgow, United Kingdom.

Berriman, G. B., Juve, G., Deelman, E., Regelson, M., & Plavchan, P. (2010). The application of cloud computing to astronomy: A study of cost and performance. *In 2010 Sixth IEEE International Conference on e-Science Workshops*, (pp. 1–7). Brisbane, QLD, Australia.

Bertier, M., Marin, O., & Sens, P. (2003). Performance analysis of a hierarchical failure detector. *In International Conference on Dependable Systems and Networks (DSN'03)*, (pp. 635–644). San Francisco, CA.

Bharathi, S., Chervenak, A., Deelman, E., Mehta, G., Su, M. H., & Vahi, K. (2008). Characterization of scientific workflows. *In 2008 Third Workshop on Workflows in Support of Large-Scale Science,* (pp. 1–10). Austin, TX, USA: IEEE.

Blythe, J., Jain, S., Deelman, E., Gil, Y., Vahi, K., Mandal, A., & Kennedy, K. (2005). Task scheduling strategies for workflow-based applications in grids. *In 5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, *2*, pp. 759–767. Cardiff, Wales, UK, United Kingdom.

Brandic, I., Music, D., & Dustdar, S. (2009). Service mediation and negotiation bootstrapping as first achievements towards self-adaptable grid and cloud services. *In Proceedings of the 6th international conference industry session on Grids meets autonomic computing (GMAC'09)*, (pp. 1–8). New York, NY, USA.

Bresnahan, J., Freeman, T., La Bissoniere, D., & Keahey, K. (2011). Managing appliance launches in infrastructure clouds. *In Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery (TG'11)*, *12*, pp. 1–7. Salt Lake City Utah, USA.

Brown, D. A., Brady, P. R., Dietz, A., Cao, J., Johnson, B., & McNabb, J. (2007). A Case Study on the Use of Workflow Technologies for Scientific Analysis: Gravitational Wave Data Analysis. In I. J. Taylor, E. Deelman, D. B. Gannon, & M. Shields (Eds.), *Workflows for e-Science* (pp. 39–59). Springer, London.

Brun, Y., Bang, J., Edwards, G., & Medvidovic, N. (2015). Self-Adapting Reliability in Distributed Software Systems. *IEEE Transactions on Software Engineering, 41*(8), 764–780.

Buyya, B., & Murshed, M. (2002). Gridsim : A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation : Practice and Experience (CCPE), 14*(13), 1175–1220.

Buyya, R., Pandey, S., & Vecchiola, C. (2009). Cloudbus toolkit for market-oriented cloud computing. *In IEEE International Conference on Cloud Computing (CloudCom'09)*, (pp. 3–27). Beijing, China.

Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., & Brandic, I. (2009). Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems, 25*(6), 599–616.

Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A., & Buyya, R. (2011). CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience, 41*(1), 23–50.

Callaghan, S., Maechling, P., Deelman, E., Vahi, K., Mehta, G., Juve, G., . . . Jordan, T. (2008). Reducing time-to-solution using distributed high-throughput mega-workflows: experiences from SCEC CyberShake. *In 2008 IEEE Fourth International Conference on eScience (e-Science'08)*, (pp. 151–158). Indianapolis, IN, USA.

Cappello, F., Primet, P., Richard, O., Cérin, C., & Sens, P. (2005). Data gridexplorer : une plate-forme d'émulation de grilles. *ACI Masse de Données/DGDX Paristic*, 506–520.

Carr, N. (2009). *The Big Switch: Rewiring the World, from Edison to Google.* W. W. Norton & Co.

Casanova, H. (2001). Simgrid : A toolkit for the simulation of application scheduling. *In Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'01)*, (pp. 430–437). Brisbane, Queensland, Australia.

Chandra, T. D., & Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM, 43*(2), 225–267.

Chandrasekaran, S., & Vaideeswaran, J. (2014). Meta-scheduler using agents for fault tolerance in computational grid. *International Journal of Computational Vision and Robotics, 4*(4), 294–329.

Chandy, K. M., & Lamport, L. (1985). Distributed snapshots : determining global states of distributed systems. *ACM Transactions on Computer Systems, 3*(1), 63–75.

Chatterjee, M., Das, S., & Turgut, D. (2002). WCA: a weighted clustering algorithm for mobile ad hoc networks. *Cluster Computing Journal, 5*(2), 193–204.

Chauhan, P., & Nitin. (2012). Fault tolerant decentralized scheduling algorithm for P2P grid. *In 2nd International Conference on Communication, Computing & Security (ICCCS'12)*, *6*, pp. 698–707.

Chen, W., & Deelman, E. (2012). Fault tolerant clustering in scientific workflows. *In 2012 IEEE Eighth World Congress on Services*, (pp. 9–16). Honolulu, HI, USA.

Chen, W., & Deelman, E. (2012). Integration of workflow partitioning and resource provisioning. *In 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'12)*, (pp. 764–768). Ottawa, ON, Canada.

Chen, W., & Deelman, E. (2012). WorkflowSim: A toolkit for simulating scientific workflows in distributed environments. *In 2012 IEEE 8th International Conference on e-Science (e-Science'12)*, (pp. 1–8). Chicago, IL, USA.

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

Chervenak, A. L., Schuler, R., Ripeanu, M., Amer, M. A., Bharathi, S., Foster, I., . . . Kesselman, C. (2009). The Globus replica location service: design and experience. *Transactions on Parallel and Distributed Systems, 20*(9), 1260–1272.

Cohen, I., Goldszmidt, M., Kelly, T., Symons, J., & Chase, J. S. (2004). Correlating instrumentation data to system states: a building block for automated diagnosis and control. *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation (OSDI'04)*, *6*, pp. 16–32. San Francisco, CA, USA.

Cook, K., Robinson, M., Ferrag, M. A., Maglaras, L. A., & Helge Janicke, H. (2018). Internet of Cloud: Security and Privacy Issues. In B. Mishra, H. Das, S. Dehuri, & A. Jagadev (Eds.), *Cloud Computing for Optimization: Foundations, Applications, and Challenges. Studies in Big Data* (Vol. 39, pp. 271–301). Springer, Cham.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to Algorithms* (2nd edition ed.). The MIT Press.

Crosby, P. (2003). *Edgsim. a simulation of the european data grid project*. Retrieved from http ://www.hep.ucl.ac.uk/ pac/EDGSim/

da Silva, R. F., Chen, W., Juve, G., Vahi, K., & Deelman, E. (2014). Community Resources for Enabling Research in Distributed Scientific Workflows. *In 2014 IEEE 10th International Conference on e-Science (e-Science'14)*, (pp. 177–184). Sao Paulo, Brazil.

da Silva, R. F., Glatard, T., & Desprez, F. (2013). On-line, non-clairvoyant optimization of workflow activity granularity on grids. *In European Conference on Parallel Processing (Euro-Par'13)*, (pp. 255–266). Aachen, Germany.

da Silva, R. F., Glatard, T., & Desprez, F. (2014). Controlling fairness and task granularity in distributed, online, non-clairvoyant workflow executions. *Concurrency and Computation: Practice and Experience, 26*(14), 2347–2366.

da Silva, R., & Glatard, T. (2013). A Science-Gateway Workload Archive to Study Pilot Jobs, User Activity, Bag of Tasks, Task Sub-steps, and Workflow Executions. *In European Conference on Parallel Processing Workshops (Euro-Par'13)*, (pp. 79–88). Aachen, Germany.

Dai, Y. S., Yang, B., Dongarra, J., & Zhang, G. (2009). Cloud Service Reliability: Modeling and Analysis. *In 15th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'09)*, (pp. 1–17). Shanghai, China.

Darte, A., Robert, Y., & Vivien, F. (2000). *Scheduling and Automatic Parallelization.* Birkhäuser, Boston, MA, USA.

Dean, J. (2006). Experiences with mapreduce, an abstraction for large-scale computation. *In Proceedings of the 15th international conference on Parallel architectures and compilation techniques (PACT'06)*, (p. 1). Seattle Washington, USA .

Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Patil, S., . . . Livny, M. (2004). Pegasus: Mapping scientific workflows onto the grid. *In Second European AcrossGrids Conference (AxGrids'04)*, (pp. 11–20). Nicosia, Cyprus.

Deelman, E., Singh, G., Livny, M., Berriman, B., & Good, J. (2008). The cost of doing science on the cloud: The montage example. *In Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC'08)*, (pp. 1–12). Austin, TX, USA.

Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P. J., . . . Wenger, K. (2015). Pegasus, a workflow management system for science automation. *Future Generation Computer Systems, 46*, 17–35.

Dharwadkar, N. V., Poojara, S. R., & Kadam, P. M. (2018). Fault Tolerant and Optimal Task Clustering for Scientific Workflow in Cloud. *International Journal of Cloud Applications and Computing, 8*(3), 1–19.

Díaz, D., Pardo, X. C., Martín, M. J., & González, P. (2008). Application-level fault-tolerance solutions for grid computing. *In 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'08)*, (pp. 554–559). Lyon, France.

Du, S., PalChaudhuri, A. K., Post, A., Saha, A. K., Druschel, P., Johnson, D. B., & Riedi, R. (2008). Safari: A self-organizing, hierarchical architecture for scalable ad hoc networking. *Ad Hoc Networks, 6*(4), 485–507.

Duan, R., Prodan, R., & Fahringer, T. (2006). Run-time optimisation of grid workflow applications. *In 2006 7th IEEE/ACM International Conference on Grid Computing*, (pp. 33–40). Barcelona, Spain.

Dumitrescu, C. L., & Foster, I. (2005). Gangsim : a simulator for grid scheduling studies. *IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, *2*, pp. 1151–1158. Cardiff, Wales, UK, United Kingdom.

Elnozahy, E. N., Alvisi, L., Wang, Y. M., & Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR), 34*(3), 375–408.

El-Rewini, H., Lewis, T. G., & Ali, H. H. (1994). *Task Scheduling in Parallel and Distributed Systems.* Prentice-Hall, Inc.

Ephremides, A., Wieselthier, J. E., & Baker, D. J. (1987). A design concept for reliable mobile radio networks with frequency hopping signaling. *Proceedings of the IEEE, 75*(1), 56–73.

Fahringer, T., Prodan, R., Duan, R., Hofer, J., Nadeem, F., Nerieri, F., . . . Wieczorek, M. (2007). ASKALON: A development and grid computing environment for scientific workflows. In I. J. Taylor, E. Deelman, D. B. Gannon, & M. Shields (Eds.), *Workflows for e-Science* (pp. 450–471). Springer, London.

Felber, P., Défago, X., Eugster, P., & Schiper, A. (1999). Replicating CORBA Objects : a marriage between active and passive replication. *In IFIP TC6 WG6.1 Second International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99)*, (pp. 375–387). Helsinki, Finland.

Ferreira, L., Berstis, V., Armstrong, J., Kendzierski, M., & al. (2003). *Introduction to grid computing with globus.* IBM Corp.

Florin, G. (1996). La tolérance aux pannes dans les systèmes répartis. Technical report, Laboratoire CEDRIC/CNAM.

Foster, I., & Kesselman, C. (1997). Globus: a Metacomputing Infrastructure Toolkit . *The International Journal of Supercomputer Applications and High Performance Computing, 11*(2), 115–128.

Foster, I., & Kesselman, C. (2003). *The Grid 2 : Blueprint for a New Computing Infrastructure.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Foster, I., Kesselman, C., & Tuecke, S. (2001). The anatomy of the grid : Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications, 15*(3), 200–222.

Foster, I., Kesselman, C., Nick, J., & Tuecke, S. (2002). *The physiology of the grid : An open grid services architecture for distributed systems integration.* Retrieved from http ://www.globus.org/alliance/publications/papers/ogsa.pdf

Garcés-Erice, L., Biersack, E. W., Felber, P. A., Ross, K. W., & Urvoy-Keller, G. (2003). Hierarchical Peer-to-Peer Systems. *In European Conference on Parallel Processing (Euro-Par'03)*, (pp. 1230–1239). Klagenfurt, Austria.

Gärtner, F. C. (1999). Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments. *ACM Computing Surveys, 31*(1), 1–26.

Gerasoulis, A., & Yang, T. (1992). A comparison of clustering heuristics for scheduling DAGs on multiprocessors. *Journal of Parallel and Distributed Computing, 16*(4), 276–291.

Gerla, M., & Tsai, J. T.-C. (1995). Multicluster, mobile, multimedia radio network. *Journal of Wireless Networks, 1*, 255–265.

Gil, J.-M., Cho, Y.-H., & Hong, S.-H. (2015). Result verification scheme with resource clustering in desktop grids. In J. Park, H. C. Chao, H. Arabnia, & N. Yen (Eds.), *Advanced Multimedia and Ubiquitous Engineering. Lecture Notes in Electrical Engineering* (Vol. 352, pp. 177–182). Springer-Verlag Berlin Heidelberg.

Gil, J.-M., Kim, S., & Lee, J. (2014). Task scheduling scheme based on resource clustering in desktop grids. *International Journal of Communication Systems, 27*(6), 918–930.

Girault, A., Lavarenne, C., Sighireanu, M., & Sorel, Y. (2001). Fault-tolerant static scheduling for real-time distributed embedded systems. *In Proceedings 21st International Conference on Distributed Computing Systems (ICDCS).* (pp. 695–698). Mesa, AZ, USA: IEEE.

GraphStream A Dynamic Graph Library. (2020). Retrieved from http://graphstream-project.org/

Grimshaw, A. S., Wulf, W. A., & The Legion Team. (1997). The legion vision of a worldwide virtual computer. *Communications of the ACM, 40*(1), 39–45.

Guerraoui, R., & Schiper, A. (1996). Fault-tolerance by replication in distributed systems. *In International Conference on Reliable Software Technologies (Ada-Europe'96)*, (pp. 38-57). Montreux, Switzerland.

Hayes, B. (2008). Cloud computing. *Communications of the ACM, 51*(7), 9–11.

Huedo, E., Montero, R. S., & Llorente, I. M. (2006). Evaluating the reliability of computational grids from the end user's point of view. *Journal of Systems Architecture, 52*(12), 727–736.

Immonen, A., & Niemelä, E. (2008). Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software & Systems Modeling, 7*(1), 49–65.

Isard, M., Budiu, M., Yu, Y., Birrell, A., & Fetterly, D. (2007). Dryad: Distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review, 41*(3), 59–72.

Jaeger, P. T., Lin, J., Grimes, J. M., & Simmons, S. N. (2009). Where is the cloud? Geography, economics, environment, and jurisdiction in cloud computing. *First Monday, 14*(5).

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

Jhawar, R., Piuri, V., & Santambrogio, M. (2012). A comprehensive conceptual system-level approach to fault tolerance in Cloud Computing. *In 2012 IEEE International Systems Conference (SysCon'12)*, (pp. 1–5). Vancouver, BC, Canada.

Jin, H., Zou, D., Chen, H., Sun, J., & Wu, S. (2003). Fault-tolerant grid architecture and practice. *Journal of Computer Science and Technology, 18*(4), 423–433.

Joshy, J., & Fellenstein, C. (2003). *Grid Computing.* Prentice Hall PTR.

Juve, G., Chervenak, A., Deelman, E., Bharathi, S., Mehta, G., & Vahi, K. (2013). Characterizing and profiling scientific workflows. *Future Generation Computer Systems, 29*(3), 682–692.

Juve, G., Deelman, E., Vahi, K., & Mehta, G. (2010). Experiences with Resource Provisioning for Scientific Workflows Using Corral. *Scientific Programming, 18*(2), 77–92.

Kahanwal, B., & Singh, T. P. (2012). The distributed computing paradigms: P2P, grid, cluster, cloud and jungle. *International Journal of Latest Research in Science and Technology, 1*(2), 183–187.

Kalakech, A. (2005). *Etalonnage de la sûreté de fonctionnement des systèmes d'exploitation – Spécifications et mise en oeuvre.* PhD thesis, Institut National Polytechnique de Toulouse - INPT.

Kalayci, S., Dasgupta, G., Fong, L., Ezenwoye, O., & Sadjadi, S. M. (2010). Distributed and adaptive execution of condor DAGMan workflows. *In Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'10)*, (pp. 587–590). Redwood City, San Francisco Bay, CA, USA,.

Kalla, H. (2004). *Génération automatique de distributions/ordonnancements temps réel, fiables et tolérants aux fautes.* PhD thesis, Institut National Polytechnique de Grenoble - INPG.

Kandaswamy, G., Mandal, A., & Reed, D. (2008). Fault tolerance and recovery of scientific workflows on computational grids. *In 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'08)*, (pp. 777–782). Lyon, France.

Katz, D. S., Anagnostou, N., Berriman, G. B., Deelman, E., Good, J. C., Jacob, J. C., . . . Williams, R. (2006). Astronomical Image Mosaicking on a Grid: Initial Experiences. In B. D. Martino, J. Dongarra, A. Hoisie, L. T. Yang, & H. Zima (Eds.), *Engineering The Grid - Status and Perspectives.* American Scientific Publishers.

Kaur, E. R. (2015). A review of computing technologies: distributed, utility, cluster, grid and cloud computing. *International Journal of Advanced Research in Computer Science and Software Engineering, 5*(2), 144–148.

Kaur, K., & Rai, A. K. (2014). A comparative analysis: grid, cluster and cloud computing. *International Journal of Advanced Research in Computer and Communication Engineering, 3*(3), 5730–5734.

Kee, Y. S., Logothetis, D., Huang, R., Casanova, H., & Chien, A. A. (2005). Efficient resource description and high quality selection for virtual grids. *In IEEE International Symposium of Cluster Computing and the Grids (CCGrid'05)*, (pp. 598–606). Cardiff, Wales, UK, United Kingdom.

Khaldi, M., Mohammed, R., Meftah, B., & Smail, O. (2019). Fault tolerance for a scientific workflow system in a Cloud computing environment. *International Journal of Computers and Applications*. doi:DOI: 10.1080/1206212X.2019.1647651

Ko, S. Y., Hoque, I., Cho, B., & Gupta, I. (2010). Making cloud intermediate data fault-tolerant. *In Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*, (pp. 181–192). Indianapolis, Indiana, USA.

Krishna, P., Vaidya, N. H., Chatterjee, M., & Pradhan, D. K. (1997). A cluster-based approach for routing in dynamic networks. *ACM SIGCOMM Computer Communication Review, 27*(2), 49–65.

Kulatunga, H., Argent-Katwala, A., & Knottenbelt, W. (2007). Cluster grid based response-time analysis module for the PIPE tool. *In Fourth International Conference on the Quantitative Evaluation of Systems (QEST'07),* (pp. 51–52). Edinburgh, UK: IEEE.

Kwok, Y., & Ahmad, I. (1999). Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys, 31*(4), 406–471.

Kwon, G., & Ryu, K. D. (2003). An efficient peer-to-peer file sharing exploiting hierarchy and asymmetry. *In Proceedings of the Symposium on Applications and the Internet Conference (SAINT'03),* (pp. 226–233). Orlando, FL, USA: IEEE.

Laprie, J. -C. (2004). Dependable computing : Concepts, challenges, directions. *In Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04). 1*, p. 242. Hong Kong, China: IEEE.

Lathers, A., Su, M.-H., Kulungowski, A., Lin, A. W., Mehta, G., Peltier, S. T., . . . Ellisman, M. H. (2006). Enabling parallel scientific applications with workflow tools. *In 2006 IEEE Challenges of Large Applications in Distributed Environments*, (pp. 55–60). Paris, France.

Li, H., Ruan, J., & Durbin, R. (2008). Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research, 18*(11), 1851–1858.

Li, Y., Lao, L., & Cui, J. H. (2006). SDC: A Distributed Clustering Protocol for Peer-to-Peer Network. In F. Boavida, T. Plagemann, B. Stiller, C. Westphal, & E. Monteiro (Eds.), *NETWORKING 2006. Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communications Systems. Lecture Notes in Computer Science* (Vol. 3976). Springer, Berlin, Heidelberg.

Li, Y., Lao, L., & Cui, J. H. (2011). SDC: a distributed clustering protocol. *International Journal of Computer Networks (IJCN), 2*(6), 205–226.

Li, Z., Ge, J., Hu, H., Song, W., Hu, H., & Luo, B. (2018). Cost and Energy Aware Scheduling Algorithm for Scientific Workflows with Deadline Constraint in Clouds. *IEEE Transactions on Services Computing, 11*(4), 713–726.

Li, Z., Ge, J., Yang, H., Huang, L., Hu, H. Y., Hu, H., & Luo, B. (2016). A security and cost aware scheduling algorithm for heterogeneous tasks of scientific workflow in clouds. *Future Generation Computer Systems, 65*, 140–152.

Liben-Nowell, D., Balakrishnan, H., & Karger, D. (2002). Observations on the dynamic evolution of peer-to-peer networks. *In International Workshop on Peer-to-Peer Systems (IPTPS'02),* (pp. 22–33). Cambridge, MA, USA: Springer, Berlin, Heidelberg.

Litke, A., Skoutas, D., Tserpes, K., & Varvarigou, T. (2007). Efficient task replication and management for adaptive fault tolerance in Mobile Grid environments. *Future Generation Computer Systems, 23*(2), 163–178.

Liu, Q., & Liao, Y. (2009). Grouping-based fine-grained job scheduling in grid computing. *In 2009 First International Workshop on Education Technology and Computer Science,* (pp. 556–559). Wuhan, Hubei, China: IEEE.

Livny, J., Teonadi, H., Livny, M., & Waldor, M. (2008). High-Throughput, Kingdom-Wide Prediction and Annotation of Bacterial Non-Coding RNAs. *PLoS ONE, 3*(9), e3197.

Maechling, P., Deelman, E., Zhao, L., Graves, R., Mehta, G., Gupta, N., . . . Field, E. (2007). SCEC CyberShake Workflows—Automating Probabilistic Seismic Hazard Analysis Calculations. In I. J. Taylor, E. Deelman, D. B. Gannon, & M. Shields (Eds.), *Workflows for e-Science* (pp. 143–163). Springer, London.

Maheshwari, K., Espinosa, A., Wilde, M., Zhang, Z., Foster, I., Callaghan, S., & Maechling, P. (2012). Job and data clustering for aggregate use of multiple production cyberinfrastructures. *In Proceedings of the Fifth International Workshop on Data-Intensive Distributed Computing Date (DIDC'12),* (pp. 3–12). Delft The Netherlands.

Malathy, G., Somasundaram, R., & Duraiswamy, K. (2013). Performance improvement in cloud computing using resource clustering. *Journal of Computer Science, 9*(6), 671–677.

Manimaran, G., & Murthy, C. S. (1998). A Fault-tolerant Dynamic Scheduling Algorithm for Multiprocessor Real-time Systems and its Analysis. *IEEE Transactions on Parallel and Distributed Systems, 9*(11), 1137–1152.

Mario, A. N. (2003). Peer-to-peer : harnessing the power of disruptive technologies. *ACM SIGMOD Record, 32*(2), 57–58.

Massie, M. L., Chun, B. N., & Culler, D. E. (2004). The ganglia distributed monitoring system : design, implementation, and experience. *Parallel Computing, 30*(7), 817–840.

Mell, P., & Grance, T. (2011). *The NIST Definition of Cloud Computing.SP 800-145.* Technical Report, National Institute of Standards & Technology Gaithersburg MD United States.

Mirarchi, D., Canino, G., Vizza, P., Veltri, P., Cuomo, S., Petrolo, C., & Chiarella, G. (2017). Data mining techniques for vestibular data classification. *International Journal of Internet Technology and Secured Transactions (IJITST), 7*(1), 51–70.

Mittal, G., Kesswani, N., & Goswami, K. (2013). A survey of current trends in distributed, grid and cloud computing. *International Journal of Advanced Studies in Computers, Science & Engineering (IJASCSE), 2*(3), 1–6.

Mohsin, A. (2011). Conception d'une architecture journalisée tolérante aux fautes pour un processeur de données. PhD thesis, Université Paul Verlaine, Metz.

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

Montagnat, J., Glatard, T., Reimert, D., Maheshwari, K., Caron, E., & Desprez, F. (2010). Workflow-based comparison of two distributed computing infrastructures. *In The 5th Workshop on Workflows in Support of Large-Scale Science (WORKS),* (pp. 1–10). New Orleans, LA, USA: IEEE.

Morin, C. (1998). *Architectures et systèmes distribués tolérants aux fautes*. Habilitation á diriger des recherches, IRISA - Institut de Recherche en Informatique et Systèmes Aléatoires, INRIA Rennes.

Muthuvelu, N., Chai, I., & Eswaran, C. (2008). An adaptive and parameterized job grouping algorithm for scheduling grid jobs. *In 2008 10th International Conference on Advanced Communication Technology (ICACT'08),* (pp. 975–980). Gangwon-Do, South Korea: IEEE.

Muthuvelu, N., Chai, I., Chikkannan, E., & Buyya, R. (2010). On-line task granularity adaptation for dynamic grid applications. *In Algorithms and Architectures for Parallel Processing, 10th International Conference (ICA3PP'10)*, (pp. 266–277). Busan, Korea.

Muthuvelu, N., Liu, J., Soe, N. L., Venugopal, S., Sulistio, A., & Buyya, R. (2005). A dynamic job grouping-based scheduling for deploying applications with fine grained tasks on global grids. *Australasian Workshop on Grid Computing and e-Research (AusGrid'05)*, (pp. 41–48). Newcastle, NSW, Australia.

Nerurkar, P., Shirke, A., Chandane, M., & Bhirud, S. (2018). Empirical analysis of data clustering algorithms. *Procedia Computer Science, 125*(1), 770–779.

Ng, W. K., Ang, T., Ling, T., & Liew, C. (2006). Scheduling framework for bandwidth-aware job grouping-based scheduling in grid computing. *Malaysian Journal of Computer Science, 19*(2), 117–126.

Nikaein, N., Labiod, H., & Bonnet, C. (2000). DDR-distributed dynamic routing algorithm for mobile ad hoc networks. *In 2000 First Annual Workshop on Mobile and Ad Hoc Networking and Computing. MobiHOC (Cat. No.00EX444),* (pp. 19–27). Boston, MA, USA: IEEE.

Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., & Zagorodnov, D. (2009). The Eucalyptus Open-Source Cloud-Computing System. *In 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'09)*, (pp. 124–131). Shanghai, China.

Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., . . . Li, P. (2004). Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics, 20*(17), 3045–3054.

Olivier, S. (2005). *Stockage dans les systèmes pair à pair.* PhD thesis, Université de Picardie Jules Verne.

Oluwatosin, H. S. (2014). Client-server model. *IOSR Journal of Computer Engineering (IOSR-JCE), 16*(1), 67–71.

Padmakumari, P., & Umamakeswari, A. (2019). Development of cognitive fault tolerant model for scientific workflows by integrating overlapped migration and check-pointing approach. *Journal of Ambient Intelligence and Humanized Computing*. doi:https://doi.org/10.1007/s12652-019-01174-9

Pandey, S., Karunamoorthy, D., & Buyya, R. (2011). Workflow engine for clouds. In *Cloud Computing: Principles and Paradigms* (pp. 321–344). Wiley.

Pankaj, J. (1994). *Fault tolerance in distributed systems.* Prentice-Hall, Inc.

Pezoa, J. E., & Hayat, M. M. (2012). Performance and Reliability of Non-Markovian Heterogeneous Distributed Computing Systems. *IEEE Transactions on Parallel and Distributed Systems, 23*(7), 1288–1301.

Plankensteiner, K., Prodan, R., & Fahringer, T. (2009). A new fault tolerance heuristic for scientific workflows in highly distributed environments based on resubmission impact. *2009 Fifth IEEE International Conference on e-Science (e-Science'09)*, (pp. 313–320). Oxford, UK.

Plankensteiner, K., Prodan, R., Fahringer, T., Kertész, A., & Kacsuk, P. (2008). Fault Detection, Prevention and Recovery in Current Grid Workflow Systems. *In Grid and Services Evolution, Proceedings of the 3rd CoreGRID Workshop on Grid Middleware,* (pp. 1–13). Barcelona, Spain: Springer, Boston, MA.

Poola, D., Ramamohanarao, K., & Buyya, R. (2014). Fault-tolerant Workflow Scheduling using Spot Instances on Clouds. *Procedia Computer Science, 29*, 523–533.

Raghavendra, C. S., & Makam, S. V. (1986). Reliability Modeling and Analysis of Computer Networks. *IEEE Transactions on Reliability, 32*(2), 156–160.

Rajkumar, K., & Swaminathan, P. (2016). Fault tolerance in distributed systems using fused data structures with the help of LT codes. *International Journal of Advanced Intelligence Paradigms (IJAIP), 8*(2), 183–190.

Ramalingam, A., Subramani, S., & Perumalsamy, K. (2002). Associativity-based cluster formation and cluster management in ad hoc networks. *In 9th International conference on high performance computing (HiPC'02).* Bangalore, India.

Ramaswamy, L., Gedik, B., & Liu, L. (2005). A distributed approach to node clustering in decentralized peer-to-peer networks. *IEEE Transactions on Parallel and Distributed Systems, 16*(9), 814–829.

Randell, B. (1975). System structure for software fault tolerance. *EEE Transactions on Software Engineering, SE-1*(2), 220–232.

Rebbah, M. (2015). Tolérance aux fautes dans les grilles de calcul. PhD thesis, Université des Sciences et de la Technologie d'Oran.

Rebbah, M., Mokhtari, C., Khaldi, M., Bourasi, M. F., & Smail, O. (2010). Hierarchical model for fault tolerant grid computing over Globus Toolkit. *In International Congress on Models, Optimization and Security of Systems (ICMOSS).* Tiaret, Algeria.

Rebbah, M., Slimani, Y., & Benyettou, A. (2014). A decentralized fault tolerant model for grid computing. *International Journal of Computer Science Issues (IJCSI), 11*(1), 123–130.

Rebbah, M., Slimani, Y., Benyettou, A., & Brunie, L. (2016). A decentralized fault tolerance model based on level of performance for Grid environment. *Cluster Computing Journal, 9*(1), 13–27.

Rebbah, M., Slimani, Y., Benyettou, A., Meftah, B., & Brunie, L. (2015). On advantages of dynamic colored graph for fault tolerance in grid computing. *Global Journal on Technology, 8*(8), 84–97.

Rimal, B., Choi, E., & Lumb, I. (2009). A taxonomy and survey of cloud computing systems. *In 2009 Fifth International Joint Conference on INC, IMS and IDC (NCM'09),* (pp. 44–51). Seoul, South Korea: IEEE.

Rodriguez, M. A., & Buyya, R. (2014). Deadline based Resource Provisioning and Scheduling Algorithm for Scientific Workflows on Clouds. *IEEE Transactions on Cloud Computing, 2*(2), 222–235.

Rodriguez, M. A., & Buyya, R. (2017). A taxonomy and survey on scheduling algorithms for scientific workflows in IaaS cloud computing environments. *Concurrency and Computation Practice and Experience, 29*(8).

Rowstron, A., & Druschel, P. (2001). Pastry : Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *In Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms,* (pp. 329–350). Heidelberg, Germany: Springer, Berlin, Heidelberg.

Samak, T., Gunter, D., Goode, M., Deelman, E., Mehta, G., Silva, F., & Vahi, K. (2011). Failure prediction and localization in large scientific workflows. *In Proceedings of the 6th workshop on Workflows in support of large-scale science (WORKS'11)*, (pp. 107–116). Seattle Washington USA.

Sarkar, V. (1989). *Partitionning and Scheduling Parallel Programs for Execution on Multiprocessors.* MIT Press.

Saroiu, S., Gummadi, P. K., & Gribble, S. D. (2003). Measuring and analyzing the characteristics of napster and gnutella hosts. *Multimedia Systems, 9*(2), 170–184.

Schmidt, D., Stal, M., Rohnert, H., & Buschmann, F. (2000). *Pattern-Oriented Software Architecture, Volume 2 : Patterns for Concurrent and Networked Objects.* Wiley.

Schollmeier, R. (2001). A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. *In Proceedings First International Conference on Peer-to-Peer Computing (P2P'01),* (pp. 101–102). Linkoping, Sweden: IEEE.

Schroeder, B., & Gibson, G. (2010). A large-scale study of failures in highperformance computing systems. *IEEE Transactions on Dependable and Secure Computing, 7*(4), 337–350.

Schroeder, B., & Gibson, G. A. (2006). A large-scale study of failures in high-performance computing systems. *In International Conference on Dependable Systems and Networks (DSN'06),* (pp. 249–258). Philadelphia, PA, USA: IEEE.

Shen, X., Yu, H., Buford, J., & Akon, M. (2009). *Handbook of Peer-to-Peer Networking.* Springer US.

Shojafar, M., Javanmardi, S., Abolfazli, S., & Cordeschi, N. (2015). FUGE: A joint meta-heuristic approach to cloud job scheduling algorithm using fuzzy theory and a genetic method. *Cluster Computing, 18*(2), 829–844.

Singh, G., Su, M., Vahi, K., Deelman, E., Berriman, B., Good, J., . . . Mehta, G. (2008). Workflow task clustering for best effort systems with pegasus. *In Proceedings of 15th Mardi Gras Conference (MG'08)*, (pp. 1–8). Baton Rouge Louisiana USA.

Sinnen, O. (2007). *Task Scheduling for Parallel Systems.* Wiley.

Sinnen, O., & Sousa, L. (1999). *A Classification of Graph Theoretic Models for Parallel Computing.* Technical Report RT/005/99, INESC-ID, Instituto Superior Técnico, Technical University of Lisbon, Portugal.

Sirohi, S., & Yadav, M. (2016). A survey of clustering scheme for MANETS. *IOSR Journal of Computer Engineering (IOSR-JCE), 18*(6), 38–47.

Smail, O., Cousin, B., & Snoussaoui, I. (2017). Energy-aware and stable cluster-based multipathrouting protocol for wireless ad hoc networks. *International Journal of Networking and Virtual Organisations (IJNVO), 17*(2/3), 229–251.

Sonna Momo, L. (2001). *Réplication et durabilité dans les systèmes répartis.* Technical report, École Polytechnique Fédérale de Lausanne.

Sood, D., Kour, H., & Kumar, S. (2016). Survey of computing technologies: distributed, utility, cluster, grid and cloud computing. *Journal of Network Communications and Emerging Technologies (JNCET), 6*(5), 99–102.

Sotomayor, B., Montero, R. S., Llorente, I. M., & Foster, I. (2009). Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing, 13*(5), 14–22.

Stergiou, C., Psannis, K. E., Kim, B. -G., & Gupta, B. (2018). Secure integration of IoT and Cloud Computing. *Future Generation Computer Systems, 78*(3), 964–975.

Strom, R. E., & Yemini, S. (1985). Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems, 3*(3), 204–226.

Tanenbaum, A. (1999). *Systèmes d'exploitation. Systèmes centralisés, systèmes distribués.* Dunod.

Taylor, I. J., Deelman, E., Gannon, D. B., & Shields, M. (2014). *Workflows for e-Science: Scientific Workflows for Grids.* Springer-Verlag London.

Topcuoglu, H., Hariri, S., & Wu, M. Y. (1999). Task scheduling algorithms for heterogeneous processors. *In Proceedings. Eighth Heterogeneous Computing Workshop (HCW'99),* (pp. 3–14). San Juan, Puerto Rico, USA: IEEE.

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

Tronel, F. (2003). *Application des problèmes d'accord à la tolérance aux défaillances dans les systèmes distribués asynchrones.* PhD thesis, Rennes 1.

Tsai, C. W., Kang, C. T., Hu, K. C., & Chiang, M. C. (2016). A quantum-inspired evolutionary clustering algorithm for the lifetime problem of wireless sensor network. *International Journal of Internet Technology and Secured Transactions (IJITST), 6*(4), 259–290.

Vaquero, L. M., Rodero-Merino, L., Caceres, J., & Lindner, M. (2009). A break in the clouds: Towards a cloud definition. *SIGCOMM Computer Communications Review, 39*(1), 50–55.

Vinay, K., Kumar, S. M., Raghavendra, S., & Venugopal, K. R. (2018). Cost and fault-tolerant aware resource management for scientific workflows using hybrid instances on clouds. *Multimedia Tools and Applications, 77*, 10171–10193.

Voelter, M., Kircher, M., & Uwe, Z. (2004). *Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware.* Wiley.

Vu, Q. H., Lupu, M., & Ooi., B. C. (2010). *Peer-to-Peer Computing - Principles and Applications.* Springer-Verlag Berlin Heidelberg.

Wan, H., Huang, H., Yang, J., & Chen, Y. (2011). Reliability model of distributed simulation system. *2011 International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering (ICQR2MSE),* (pp. 99–104). Xi'an, China: IEEE.

Wang, Y., & Bao, F. S. (2007). An entropy-based weighted clustering algorithm and its optimization for ad hoc networks. *In Third IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob'07)*, (pp. 56–56). White Plains, NY, USA.

Workflow Generator. (2020). Retrieved from https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator

Xia, Y., Jiang, C., Sun, T., & Yang, R. (2011). A novel failure detection algorithm for reliable distributed systems. *Journal of Computers, 6*(10), 2013–2020.

Yao, G., Ding, Y., & Hao, K. (2017). Using Imbalance Characteristic for Fault-Tolerant Workflow Scheduling in Cloud Systems. *IEEE Transactions on Parallel and Distributed Systems, 28*(12), 3671–3683.

Yao, L., Li, L., & Lei, C. (2017). Fault tolerant control for a class of nonlinear non-Gaussian singular stochastic distribution systems. *International Journal of Modelling, Identification and Control (IJMIC), 27*(2), 104–113.

Yildiz, U., Guabtni, A., & Ngu, A. H. (2009). Business versus scientific workflows: A comparative study. *2009 Congress on Services - I,* (pp. 340–343). Los Angeles, CA, USA: IEEE.

Ying, C., Yu, J., & He, J. (2018). Towards fault tolerance optimization based on checkpoints of in-memory framework spark. *Journal of Ambient Intelligence and Humanized Computing*. doi:https://doi.org/10.1007/s12652-018-1018-6

Youseff, L., Butrico, M., & Da Silva, D. (2008). Toward a unified ontology of cloud computing. *In 2008 Grid Computing Environments Workshop (GCE'08),* (pp. 1–10). Austin, TX, USA: IEEE.

Yu, J. Y., & Chong, P. H. (2003). 3hBAC (3-hop between adjacent clusterheads): a novel non-overlapping clustering algorithm for mobile ad hoc networks. *In 2003 IEEE Pacific Rim Conference on Communications Computers and Signal Processing (PACRIM'03) (Cat. No.03CH37490)*, *1*, pp. 318–321. Victoria, BC, Canada.

Yu, J. Y., & Chong, P. H. (2005). A survey of clustering schemes for mobile ad hoc networks. *IEEE Communications Surveys and Tutorials, 7*(1), 32–48.

Yu, J., & Buyya, R. (2005, September). A taxonomy of scientific workflow systems for grid computing. *SIGMOD Record, 34*(3), 44–49.

Yu, J., Buyya, R., & Ramamohanarao, K. (2008). Workflow Scheduling Algorithms for Grid Computing. In F. Xhafa, & A. Abraham (Eds.), *Metaheuristics for Scheduling in Distributed Computing Environments. Studies in Computational Intelligence* (Vol. 146, pp. 173-214). Springer, Berlin, Heidelberg.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., . . . Stoica, I. (2012). Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. *In Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12),* (p. 2). San Jose, CA: USENIX Association.

Zhang, Q., Yang, J., Gu, N., Zong, Y., Ding, Z., & Zhang, S. (2006). Dynamic replica location service supporting data Grid systems. *In The Sixth IEEE International Conference on Computer and Information Technology (CIT'06)*, (p. 61). Seoul, Korea.

Zhang, X., & Pham, H. (2000). An analysis of factors affecting software reliability. *The Journal of Systems and Software, 50*(1), 43–56.

Zhang, Y., Mandal, A., Koelbel, C., & Cooper, K. (2009). Combined fault tolerance and scheduling techniques for workflow applications on computational grids. *In 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'09)*, (pp. 244–251). Shanghai, China.

Zhang, Y., Squillante, M. S., Sivasubramaniam, A., & Sahoo, R. K. (2004). Performance implications of failures in large-scale cluster scheduling. *In Proceedings of the 10th international conference*

on Job Scheduling Strategies for Parallel Processing (JSSPP'04), (pp. 233–252). New York, NY, USA: Springer, Berlin, Heidelberg.

Zhao, W., Melliar-Smith, P. M., & Moser, L. E. (2010). Fault Tolerance Middleware for Cloud Computing. *In 2010 IEEE 3rd International Conference on Cloud Computing*, (pp. 67–74). Miami, FL, USA.

Zhu, X., Wang, J., Guo, H., Zhu, D., Yang, L. T., & Liu, L. (2016). Fault-Tolerant Scheduling for Real-Time Scientific Workflows with Elastic Resource Provisioning in Virtualized Clouds. *IEEE Transactions on Parallel and Distributed Systems, 27*(12), 3501–3517.

ملخص

توفر بيئات الأنظمة الموزعة على نطاق واسع للعملاء قاعدة موارد غير محدودة لحساب البيانات وتخزينها. يمكن تمييز هذه البيئات في شكل شبكة حاسوبية أو الحوسبة السحابية أو أنظمة نظير إلى نظير. موارد هذه الأنظمة عرضة للفشل بسبب الأعطال من جميع الأنواع (الانهيار، الانفصال، البيزنطية الخ). الهدف الرئيسي من هذه الأطروحة هو (أ) اقتراح تقنيات للتسامح مع الخطأ على أساس تجميع الموارد في بيئات الشبكة الحاسوبية و (ب) تجميع مهام نظام سير العمل العلمي في بيئات الحوسبة السحابية. تسمح هذه التقنيات بتعميم الإشكاليات المثارة عن طريق تطوير إثنين من المناهج التجريبية المتسامحة مع الأخطاء وكذا تنفيذها وتقييمها عن طريق المحاكاة. كإسهام أول لهذه الأطروحة، نقترح نموذج للتسامح مع الخطأ FT-GRC الذي يسعى إلى البحث عن البديل الأنسب للعقدة الفاشلة بواسطة تجميع موارد الحوسبة دون أي نسخ متماثل. يستند هذا النموذج على الرسوم البيانية الملونة الديناميكية التي يمكن أن تأخذ في الاعتبار الخصائص الأساسية الثلاث للحوسبة الشبكية، وهي عدم التجانس والديناميكية والقابلية للتطور. تستخدم آلية التسامح مع الخطأ المقترحة خوارزمية تجميع من نوع 1-قفزة تسمى HCC لتحويل الشبكة إلى مجموعة من الكتل المترابطة ومن خلال إدخال دالة تقييم تتيح حساب مستوى أداء كل عقدة في الشبكة وفقًا لخصائصها المادية والمنطقية. وقد أتاح لنا الجمع بين هذه التقنيات تحديد اختيار البدائل على النحو الأمثل من خلال البحث عن بدائل في نفس المجموعة (التسامح مع الخطأ داخل الكتلة) ثم من خلال البحث عن أقرب البدائل (التسامح مع أخطاء بين الكتل). وبالتالي، استخدمنا الترحيل الدوري للمهام من العقدة الفاشلة إلى البدائل المناسبة. في الإسهام الثاني، نقترح نموذجًا جديدًا للتسامح مع الخطأ يسمى FT-HCC لنظام سير العمل العلمي الذي يتكون من مجموعة من المهام المجمعة والتي يتم تنفيذها في بيئة الحوسبة السحابية. قمنا بتطبيق هذا النهج على خمسة مسارات سير عمل علمي واسعة الاستخدام مع نموذج سير عمل مختلف وخصائص حسابية مختلفة. تقليل وقت التنفيذ وكذا تكلفة التنفيذ هو الهدف الرئيسي لنموذج التسامح مع الخطأ المقترح من أجل زيادة أداء سير العمل. لقد أخذنا في الاعتبار الأخطاء الداخلية مثل فشل الحاسب المضيف، بما في ذلك مهام سير العمل والأجهزة الافتراضية (VM). من وجهة نظر التجميع، اقترحنا استراتيجيتين لتجميع المهام قابلتين للتسامح مع الخطأ لتحسين أداء تنفيذ سير العمل: مجموعة حاسمة (CC) تعتمد على المسار الحاسم ومجموعة أفقية (HC) تستند على المستويات الأفقية لسير العمل. تقنية التسامح مع الخطأ المقترحة تعمل على تطبيق النسخ المتماثل النشط لمهام المسار الحاسم، والنسخ المتماثل السلبي للمهام غير الحاسمة. كذلك، يتم استغلال وقت الخمول للأجهزة الافتراضية في النسخ المتماثل السلبي.

الكلمات المفتاحية:

الشبكات الحاسوبية، الحوسبة السحابية، التسامح مع الخطأ، التجميع، مسارات سير العمل العلمي، مستوى الأداء.

Résumé

Les environnements de systèmes distribués à large échelle offrent aux clients un parc de ressources illimitées pour le calcul et le stockage des données. Ces environnements peuvent être structurés sous forme de grille de calcul, de Cloud Computing ou de systèmes pair à pair. Les ressources de tels systèmes sont sujettes à des pannes de toutes nature (crash, déconnexion, byzantine etc.). L'objectif principal de cette thèse est (i) la proposition des techniques de tolérance aux fautes basé sur la clusterisation des ressources dans des environnements de grille de calcul et (ii) une clusterisation des tâches d'un système de workflow scientifique dans des environnements de Cloud Computing. Elles généralisent la problématique soulevée par le développement de deux heuristiques de clustering tolérant aux fautes, leurs implémentations et leurs évaluations par des simulations. Comme première contribution, nous proposons FT-GRC un modèle de tolérance aux fautes qui cherche à trouver le substitut le plus adéquat au nœud défaillant par la clusterisation des ressources de calcul sans aucune réplication. Ce modèle est basé sur des graphes colorés dynamiques qui puissent tenir compte des trois caractéristiques fondamentales des grilles, à savoir l'hétérogénéité, la dynamicité et le passage à l'échelle. Le mécanisme de tolérance aux fautes proposé utilise un algorithme de clustering à 1 saut (*1*-hop) appelé HCC pour transformer la grille en un ensemble de clusters interconnectés et par l'introduction d'une fonction de *scoring* qui permettes de calculer le niveau de performance de chaque nœud de la grille en fonction de leurs caractéristiques physiques et logiques. La combinaison de ces techniques, nous a permis de déterminer le choix des substituts de manière optimale en cherchant les substituts dans le même cluster (tolérance aux fautes local intra-cluster) puis par la recherche des plus proches substituts (tolérance aux fautes inter-cluster). Ainsi, nous avons utilisé une migration périodique des jobs du nœud défaillant vers les substituts appropriés. Dans la deuxième contribution, nous proposons un nouveau modèle de tolérance aux fautes appelée FT-HCC pour un système de workflow scientifique composé d'un ensemble de tâches clusterisées soumises dans un environnement de Cloud Computing. Nous avons appliqué cette approche à cinq workflows scientifiques en temps réel avec un modèle de workflow et des caractéristiques de calcul différents. Réduire le temps d'exécution (makespan) et le coût d'exécution est l'objectif principal de notre modèle de tolérance aux fautes afin d'augmenter les performances du workflow. Nous avons considéré les fautes internes tels que les défaillances de l'hôte, y compris les tâches de workflow et les instances VM. Du point de vue clustering, nous avons proposé deux stratégies de clustering de tâches tolérantes aux fautes pour améliorer les performances d'exécution de workflow : un Clustering Critique (CC) basé sur le chemin critique et un Clustering Horizontal (HC) basé sur les niveaux du workflow. La technique de tolérance aux fautes proposée applique une réplication active pour les tâches du chemin critique, et une réplication passive pour les tâches non critiques. Ainsi, nous exploitons le temps d'inactivité des VMs pour la réplication passive.

Mots clés : Grille de calcul, Cloud Computing, Tolérance aux fautes, Clustering, Workflow scientifique, Niveau de performance.

Abstract

Large-scale distributed systems environments provide customers with an unlimited pool of resources for calculation and data storage. These environments can be characterized in the form of Grid Computing, Cloud Computing or peer-to-peer (P2P) systems. The resources of such systems are subject to failures of all kinds (crash, disconnection, byzantine etc.). The main objective of this thesis is (i) the proposal of fault tolerance techniques based on the resource clustering in grid computing environments and (ii) a task clustering of a scientific workflow system in Cloud Computing environments. They generalize the problem raised by the development of two fault-tolerant clustering heuristics, their implementations and their evaluations by simulations. As a first contribution, we propose FT-GRC a fault tolerance model that seeks to find the most suitable substitute for the failed node by the clustering of the grid resources. This model is based on dynamic colored graphs which can take into account the three basic characteristics of grid computing, such as dynamicity, heterogeneity and scalability. The proposed fault tolerance mechanism uses a *1*-hop clustering algorithm called HCC to transform the grid into a set of interconnected clusters and by the introduction of a scoring function that calculates the level of performance of each node of the grid according to their physical and logical characteristics. The combination of these techniques allowed us to determine the choice of substitutes in an optimal way by looking for substitutes in the same cluster (intra-cluster fault tolerance) then by the search for the closest substitutes (inter-cluster fault tolerance). Thus, we used a periodic migration of jobs from the failed node to the appropriate substitutes. In the second contribution, we propose a new fault tolerance model called FT-HCC for a scientific workflow system composed of a set of clustered tasks submitted in a Cloud Computing environment. We applied this approach to five real-time scientific workflows with different workflow models and computational characteristics. Reducing execution time (makespan) and execution cost is the main objective of our fault-tolerance model in order to increase workflow performance. We considered internal faults such as host failures, including workflow tasks and VM instances. From a clustering point of view, we proposed two fault tolerant task clustering strategies to improve workflow execution performance: a Critical Clustering (CC) based on the critical path and a Horizontal Clustering (HC) based on the workflow levels. The proposed fault tolerance technique applies active replication for critical path tasks, and passive replication for non-critical tasks. Thus, we exploit the idle time of VMs for passive replication.

Keywords: Grid Computing, Cloud Computing, Fault tolerance, Clustering, Scientific workflow, Performance level.