Déploiement auto-adaptatif d'intergiciel sur plate-forme élastique

Maurice-Djibril Faye

THÈSE

en vue de l'obtention du grade de

Docteur de l'Université de Lyon, délivré par l'École Normale Supérieure de Lyon

En cotutelle avec l'Université Gaston Berger de St-Louis (Sénégal)

Discipline : Informatique

Laboratoire de l'Informatique du Parallélisme (LIP)

École Doctorale INFORMATIQUE ET MATHÉMATIQUES DE LYON

présentée et soutenue publiquement le **10 Novembre 2015**

par Monsieur Maurice Djibril FAYE

Déploiement auto-adaptatif d'intergiciel sur plate-forme élastique

---

Devant le jury composé de :

M. Eddy CARON, ENS de Lyon (LIP), Directeur de thèse

M. Noel DE PALMA, Université Joseph Fourier à Grenoble, Examinateur

M. Olivier FLAUZAC, Université de Reims Champagne-Ardenne, Rapporteur

M. Franck PETIT, Université Pierre et Marie Curie Paris 6 (LIP6), Rapporteur

M. Ousmane THIARÉ, Université Gaston Berger de St-Louis (LANI), Co-tuteur de thèse

M. Cédric TÉDESCHI, Université de Rennes (IRISA), Examinateur

Remerciements

Je tiens à remercier très sincèrement mes directeurs de thèse, Eddy CARON et Ousmane THIARÉ, qui m'ont fait confiance en acceptant d'encadrer la thèse, pour les conseils, les suggestions, les critiques, la recherche de financement, mais aussi et surtout pour avoir été à la fois patients et exigeants, malgré ces longues années et les difficultés de toutes sortes, finalement surmontées. Je vous remercie aussi pour les encouragements à l'autonomie dans le travail de recherche, tout en étant capable d'intégrer les critiques.

Je remercie tous les membres du jury pour l'honneur qu'ils m'ont fait en acceptant d'en faire partie. Des remerciements spéciaux à M. Olivier FLAUZAC et M. Franck PETIT pour avoir accepté d'être les rapporteurs de la thèse. En plus, vos commentaires et remarques m'ont permis d'améliorer le manuscrit.

Mes remerciements à l'UFR SAT de l'UGB, sa section informatique, la région Rhône-Alpes (à travers le CMIRA), l'ENS-Lyon pour avoir participé au financement de certains séjours à Lyon et d'autres activités.

Les agents du secrétariat du LIP et du service mobilité internationale de l'ENS-Lyon pour leur professionnalisme. Mention spéciale à Evelyne BLESLE.

Les membres du LIP, en particulier ceux de l'équipe Avalon (doctorants, post-doctorants, ingénieurs et permanents), très accessibles, pour leurs efforts pour m'intégrer.

Remerciements à Waly Diouf, grâce à qui mes séjours à Lyon et Paris se sont bien passés.

Remerciements spéciaux à la famille (pensées à ceux et celles qui continuent leurs vies dans une autre vie), pour son soutien multiforme, indéfectible, constant et désintéressé ainsi que pour les encouragements lorsque la motivation n'était pas au mieux. Merci aussi pour les prières. Que ce travail serve de motivation aux plus jeunes, qu'il aiguise votre curiosité et suscite des vocations dans la recherche du savoir. C'est un défi que je vous lance et je suis sûr que vous le relèverez avec brio et que vous allez faire beaucoup mieux.

Je dédie ce travail à toutes les personnes qui ont payé de leurs vies, pour la liberté de l'être humain, pour la lumière de la raison, pour avoir combattu l'obscurantisme et l'intolérance.

Table des matières

Liste des figures

Liste des tableaux

Introduction

Les progrès dans la miniaturisation des composants électroniques, au niveau des réseaux informatiques, ont largement contribué à l'apparition de plusieurs systèmes distribués qui sont maintenant omniprésents (internet). Un système distribué est une collection d'entités de calcul, autonomes, interconnectées [1]. De tels systèmes permettent d'échanger des données (grâce aux réseaux WAN[1]), de partager des ressources (une imprimante dans un réseau LAN[2]), d'augmenter des performances et d'améliorer notre capacité de calcul, en distribuant les tâches entre plusieurs processus, etc.

Cependant, ils sont difficiles à concevoir, à contrôler, à maintenir car constitués d'une variété de composants (logiciels et physiques) complexes qui sont susceptibles de tomber en panne ou de subir des variations de leurs paramètres.

La nature des entités de calcul et la manière de les interconnecter laissent la place à une large gamme de matériels et de réseaux.

Les différents éléments d'un système distribué coopèrent pour atteindre un objectif. La manière d'établir cette coopération soulèvent des problèmes fondamentaux qui constituent aussi le champ d'étude de ce domaine.

Ces systèmes sont donc caractérisés par l'hétérogénéité des éléments qui les composent, une capacité de passage à l'échelle que ce soit par augmentation du nombre d'éléments, par répartition sur de vastes étendues géographiques. Ils se présentent sous diverses formes et certains sont très dynamiques quant au nombre d'éléments qui les composent (des éléments peuvent rejoindre ou quitter). Parmi les types de systèmes, les grilles [2] et les clouds [3]. Ces infrastructures offrent des services à la demande aux utilisateurs.

Pour supporter l'hétérogénéité du matériel et des réseaux tout en offrant une vue unique aux utilisateurs, ces systèmes sont généralement organisés au moyen d'une couche logicielle, appelée intergiciel (middleware) [4], qui est logiquement placée entre une couche de haut niveau (utilisateurs et applications) et une couche de bas niveau (systèmes d'exploitation, gestionnaires de ressources et autres protocoles de communication).

Ces intergiciels sont de différents types et rendent différents services [5–10]. Ils ne sont pas des blocs monolithiques mais se présentent sous la forme d'une intégration de collections hétérogènes de composants logiciels [11–13]. Un composant logiciel est défini dans [11] comme une unité de composition, qui implémente des fonctionnalités et qui a, par contrat, spécifié ses interfaces et ses dépendances de contexte. Il est caractérisé par les propriétés suivantes :

- c'est une unité de déploiement indépendant;

[1] Wide Area Network
[2] Local Area Network

- c'est une unité de composition par des tierces entités;
- il n'a pas d'état persistant.

Il existe différents modèles de composants logiciels [14] comme le modèle de composant CORBA (CCM)[3], FRACTAL [15], etc.

Un composant logiciel doit être instancié pendant la phase d'exécution. Ainsi, pendant son exécution, une application à base de composants est constituée d'un ensemble d'instances des composants qui la définissent, instances connectées selon les contrats et interfaces définis.

L'intergiciel DIET [16], sur lequel nous avons appliqué les travaux décrits dans la suite de ce manuscrit, est implémenté en CORBA. Son architecture est décrite au chapitre 1, section 1.1.1.

Avant de pouvoir bénéficier des services d'un intergiciel, il doit d'abord être déployé sur une infrastructure matérielle.

Déploiement de logiciel

Le déploiement de logiciel [17–20] est définit comme "un processus qui organise et orchestre un ensemble d'activités ayant pour but de rendre le logiciel disponible à l'utilisation et de le maintenir à jour et opérationnel" [20]. Il désigne l'ensemble des tâches à exécuter pour rendre un système logiciel fonctionnel. C'est une tâche complexe, surtout sur une infrastructure distribuée. Selon les auteurs, le processus peut être divisé en plusieurs sous-tâches. Pour [17, 19], le processus commence depuis la dernière phase de développement du logiciel, et regroupe les tâches suivantes :

- le dépôt : cette étape est une phase intermédiaire entre le développement du logiciel et le processus de déploiement. Elle couvre les activités qui rendent le logiciel prêt à être installé;
- l'installation : cette étape couvre les activités permettant de transférer les données vers les ressources cibles ainsi que les opérations de configuration nécessaires à l'activation du logiciel;
- l'activation : c'est le démarrage des composants exécutables du logiciel;
- la désactivation : c'est l'arrêt de composants en exécution;
- la mise à jour : lorsqu'une nouvelle version est disponible. Cette étape peut nécessiter l'arrêt du système en cours, sa mise à jour et sa réactivation. Cependant certains systèmes permettent la mise à jour d'une version sans arrêt du système en cours d'exécution. C'est le cas des logiciels développés avec le langage Erlang [21] que nous avons utilisé pour créer le simulateur décrit au chapitre 5;
- l'adaptation : cette étape couvre les activités qui ont cours pendant que le système est en cours d'exécution. Cela comprend la réaction de l'application déployée aux événements de son environnement;

[3] OMG : CORBA Component Model,v4.0 http://www.omg.org/spec/CCM/4.0/ 2016

- la désinstallation : la suppression de certains fichiers et composants préalablement installés, et éventuellement la reconfiguration d'autres composants affectés par cette suppression;

- le retrait : lorsque le logiciel est obsolète, il est retiré.

Dans la spécification pour le déploiement d'applications distribuées à base de composants [18], le processus de déploiement commence seulement après que le logiciel a été développé, assemblé et publié (rendu disponible par exemple sur un entrepôt accessible par internet ou à travers des disques de sauvegarde). A partir de ce moment, les différentes étapes suivantes sont considérées :

- l'installation : la définition d'installation est différente avec le cas précédent. Il s'agit de la mise à disposition des fichiers constituant le logiciel dans un entrepôt sous le contrôle de l'entité chargée de réaliser le déploiement (une personne si le déploiement sera manuel et/ou des outils spécialisés pour les déploiement automatiques [22–25]). L'endroit où les données sont stockées ne coïncide pas nécessairement avec les ressources sur lesquelles les instances vont s'exécuter;

- la configuration : il s'agit ici de la possibilité de prévoir des configurations par défaut; par exemple de fixer la couleur de l'arrière plan d'une fenêtre selon que l'on soit dans une situation ou une autre;

- le plan de déploiement : il s'agit de trouver comment et où les composants du logiciels devront être déployés, parmi les ressources de l'environnement cible. Il nécessite de prendre en compte les exigences des composants et les possibilités des ressources. Cette étape produit un plan de déploiement;

- la préparation : recouvre les tâches à accomplir sur les ressources (physiques, logicielles) de l'environnement cible afin que le logiciel soit prêt à s'exécuter. Il s'agit de transférer des fichiers exécutables et les données nécessaires sur les machines sur lesquelles les composants du logiciel vont effectivement s'exécuter;

- le lancement : couvre les tâches, après celles liées à la préparation, à la suite desquelles, le logiciel est dans un état "en exécution", et donc disponible pour être utilisé.

Se basant sur les subdivisions précédentes, [20] propose les étapes suivantes : le dépôt, l'installation, l'activation, la désactivation, la désinstallation, le retrait, la mise à jour, l'adaptation, la reconfiguration et la redistribution.

De ces différentes listes des étapes du processus de déploiement de logiciel, nous pouvons constater que globalement certaines étapes sont prises en compte par toutes les classifications et que les quelques différences interviennent dans la prise en compte ou non des étapes liées au dépôt (dernière étape de la phase de développement ou production) et au suivi du logiciel une fois disponible (adaptation, mise à jour).

Nous pouvons regrouper les différentes étapes en trois classes :

- une qui regroupe les actions qui ont lieu avant que le logiciel ne soit en cours d'exécution. Par exemple le transfert des fichiers sur les machines cibles, la configuration, l'activation. Nous appellerons cette étape phase de préparation;

- une qui regroupe les actions qui peuvent avoir lieu pendant que le logiciel est en cours d'exécution. Nous appellerons les événements qui ont lieu pendant cette phase adaptation. Cette adaptation peut se réaliser sous forme de modification de la structure du logiciel par modifications des liens entre les instances des composants pour diverses raisons (l'arrêt accidentel d'une instance, la migration d'une instance vers une autre ressource). Les actions effectuées dans la phase d'adaptation peuvent faire appel à des actions définies dans les étapes précédentes. Par exemple, un changement de topologie peut s'accompagner de la création d'une nouvelle instance, sur une ressource de l'infrastructure cible, création pendant laquelle il peut être fait appel aux opérations de transferts de fichiers, de configuration et toutes les autres opérations nécessaires pour activer la nouvelle instance. On peut même faire appel aux opérations de désinstallation sur la ressource où l'instance déplacée s'exécutait avant. Nous appellerons cette étape phase d'adaptation;

- une qui regroupe les actions pour mettre le logiciel à l'arrêt, désinstaller ce qui a été installé au début. Nous appellerons cette étape phase d'arrêt.

Ainsi, pour nous, le processus de déploiement se déroule en trois phases :

D'abord une phase de préparation qui regroupe les actions à la fin desquelles le logiciel est en cours d'exécution. Cette phase consiste entre autres activités à réaliser un plan de déploiement. Pour faire un plan de déploiement, il faut prendre en compte les caractéristiques et exigences du logiciel, celles de l'infrastructure cible et le résultat sera un plan de déploiement. Nous avons dans le chapitre 3, proposé un modèle pour décrire l'architecture d'un intergiciel hiérarchique, un modèle pour décrire une infrastructure matérielle, cible éventuelle d'un déploiement, et un modèle pour décrire un plan de déploiement et/ou un appariement (mapping) entre les instances de l'intergiciel hiérarchique et les ressources physiques sur lesquelles elles s'exécutent.

Ensuite une phase d'adaptation (éventuelle) qui regroupe les actions qui ont lieu pendant que le logiciel est en cours d'exécution et réagit aux événements de son environnement afin de continuer à fournir le service correspondant à sa spécification. L'algorithme auto-adaptatif décrit dans le chapitre 4 s'inscrit dans cadre. Il décrit des actions d'adaptation selon les événements détectés pendant que l'intergiciel est déployé et en cours d'exécution. Et c'est cet algorithme qui est ensuite simulé comme décrit dans le chapitre 5.

Enfin, la phase d'arrêt qui regroupe les actions qui, appliquées à un logiciel en exécution le mettent à l'arrêt.

Notre travail de thèse s'inscrit surtout dans les étapes préparation et adaptation.

Un déploiement qui, une fois réalisé, ne peut pas réagir aux variations de son environnement (logiciel et/ou matériel) d'exécution est dit statique. Un déploiement qui peut, de manière autonome (totalement ou en partie), réagir ou s'adapter aux variations de son environnement d'exécution, sera dit auto-adaptatif.

Les variations de l'environnement d'exécution sont très probables, surtout sur des plates-formes élastiques.

Plate-forme élastique

La notion de plate-forme élastique fait référence à la possibilité d'ajout ou de retrait de nœuds physiques à l'infrastructure physique au cours du temps, et pendant que les applications s'exécutent dessus [26, 27]. L'élasticité est une caractéristique importante

de beaucoup d'infrastructures distribuées modernes. C'est le cas dans les clouds, où en fonction des besoins, le nombre de machines virtuelles dédiées à une application peut être augmenté ou diminué.

Lorsque, dans une application distribuée, les processus peuvent disparaître et que de nouveaux processus peuvent rejoindre, cette application est qualifiée de dynamique. Lorsque le nombre de processus est fixe, on parle d'application statique.

L'élasticité d'une plate-forme peut être la cause de la disparition de processus car lorsqu'un nœud physique quitte la plate-forme (panne, rupture du lien de connexion), les processus qui s'exécutaient dessus peuvent se terminer ou bien ils se trouvent isolés et ne participent plus à l'algorithme distribué. Mais une application peut être dynamique, même avec une plate-forme avec un nombre fixe de nœuds. Ainsi, l'élasticité d'une plate-forme peut rendre une application dynamique, mais une application peut être intrinsèquement dynamique même sur une plate-forme non élastique.

Dans ce travail, nous avons considéré une application dynamique sur une plate-forme élastique. L'application est la simulation d'un intergiciel distribué. La plate-forme sur laquelle l'application est déployée est simulée comme une plate-forme élastique que les nœuds peuvent joindre ou quitter.

L'application est dynamique dans la mesure où, elle doit réagir aux conséquences de l'élasticité de la plate-forme (disparition de nœuds et donc de processus, ajout de nœuds donc possibilité de création de nouveaux processus, en cas de besoin). L'application est aussi dynamique pour des raisons autres que l'élasticité de la plate-forme. En effet, des processus peuvent disparaître ou être crées dans une recherche d'une qualité de service.

Un déploiement statique d'une application, qui ne peut prendre en compte les variations de l'infrastructure sur laquelle l'application s'exécute, n'est pas une bonne solution car lorsque l'application, pour une raison liée aux ressources matérielles, aux liens réseaux, aux processus, ne parvient plus à assurer le service ou bien l'assure de manière dégradée, l'unique solution est de reprendre tout le processus de déploiement, une opération qui est coûteuse.

Une meilleure solution serait d'avoir des systèmes logiciels qui soient capables, de manière totalement ou partiellement autonome, de s'auto-adapter.

Informatique autonome

L'informatique autonome [28, 29] est un paradigme pour la conception de systèmes logiciels auto-adaptatifs [30, 31]. De tels systèmes ont la capacité de s'auto-adapter, en cours d'exécution, pour maintenir une qualité de service, une topologie ou, de manière générale, optimiser une fonction objective [32]. L'architecture de tels systèmes est en général basée sur un modèle de boucle de contrôle fermée, inspirée de l'automatique. Cette architecture, présentée dans [29], connue sous le nom de **MAPE-K** repose sur un module de surveillance (**M**onitoring) du système à gérer, un module d'analyse (**A**nalyze) des informations collectées, un module de planification (**P**lan) des actions à exécuter après la phase d'analyse et d'un module pour exécuter (**E**xecute) les décisions prises. Tous ces modules partagent un ensemble de connaissances (**K**nowledge) qui peuvent être liées au système logiciel à gérer et aussi aux ressources sur lesquelles le système s'exécute.

Les systèmes auto-adaptatifs offrent un ensemble de capacités d'auto-gestion, parmi lesquels l'auto-réparation [33–35] en cas de défaillance. L'auto-réparation, qui est aussi un moyen d'assurer une tolérance aux pannes (ou fautes), peut être obtenue par des

algorithmes auto-stabilisants [36], dont l'exécution permettra au système de retrouver un état correct, dans un temps fini. C'est dans ce sillage que s'inscrit l'essentiel de cette thèse.

Problématique

Notre objectif est d'ajouter des capacités d'auto-adaptation à un intergiciel existant (qui n'a pas été conçu dans une perspective d'informatique autonome) afin que son déploiement soit auto-adaptatif. Le déploiement auto-adaptatif est constitué globalement de plusieurs aspects comme le résume la Figure 1.

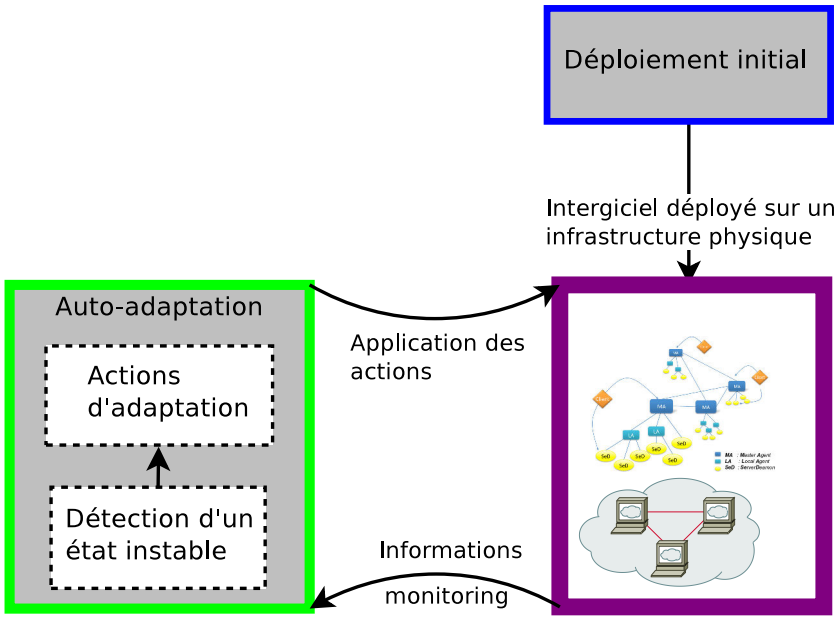

Figure 1: Vue générale d'un déploiement auto-adaptatif

L'un des aspects concerne le **déploiement initial**, ensemble de tâches (description des ressources, de l'application, algorithmes de planification, transfert des fichiers, configurer des machines, installer des librairies,...), qui utilise plusieurs informations d'entrée et outils de déploiement, et qui, en fin de compte, permet de déployer l'application sur une infrastructure physique, rendant ainsi l'application disponible aux utilisateurs.

L'autre aspect concerne l'**auto-adaptation**. Pour ce dernier aspect (l'essentiel de cette thèse a été consacré à cet aspect), une fois que l'intergiciel est déployé et en cours d'utilisation, il faut savoir détecter les situations qui nécessitent une adaptation, ensuite il faut écrire des algorithmes dont l'exécution (une réaction à la variation du contexte) aura pour effet une adaptation. Si les événements qui déclenchent les adaptations sont liés à une qualité de service, une idée serait de chercher à prédire ces événements et à agir de manière proactive et éviter ainsi d'être dans une position défensive, réactive. Par contre, si les événements, à la détection desquels le système doit réagir sont imprévisibles (cas réaliste car une panne survient en général de manière non prévue), le système sera amené à réagir.

Nous serons donc amené à définir ce qu'est un déploiement "stable" qui correspond à une situation dans laquelle l'intergiciel déployé peut fournir le service pour lequel il

est déployé. Cette situation de déploiement stable à lieu lorsqu'un certain nombre de conditions sont toutes vraies. Par conséquent, on sera dans une situation "instable" à chaque fois qu'une au moins des conditions de stabilité est fausse.

A chaque fois que le déploiement sera instable, cet état sera détecté et les mécanismes d'auto-adaptation devront s'activer pour qu'en fin de compte le déploiement retrouve un état stable.

Le mécanisme d'auto-adaptation (algorithme d'auto-adaptation), peut être centralisé ou distribué.

Si l'algorithme est centralisé, un processus qui a une vision globale du déploiement utilise les informations issues de la surveillance du système déployé pour évaluer l'état du déploiement. Si cet état est instable, le processus exécute l'algorithme centralisé visant à faire retrouver au déploiement un état stable.

Si l'algorithme est distribué (notre cas), chaque processus se basent sur les informations conservées dans son état interne (ces informations provenant totalement où en partie de la surveillance du système déployé) pour évaluer l'état du déploiement. Si cet état est instable, le processus exécute l'algorithme distribué (chaque processus exécute le code de l'algorithme) visant à faire retrouver au déploiement un état stable.

Contributions

Les principales contributions de cette thèse peuvent être regroupées en trois points :

La proposition d'un algorithme distribué d'auto-adaptation, permettant à un déploiement dont l'état est instable, de réagir pour retrouver un état stable. Cet algorithme est auto-stabilisant, ce qui fait qu'un déploiement instable, exécutant cet algorithme, retrouvera un état stable, dans un temps fini. Une preuve du caractère auto-stabilisant de l'algorithme est donnée.

Une autre contribution de cette thèse est la conception et la programmation d'un simulateur pour valider l'algorithme.

La thèse s'intéresse aussi au déploiement initial, dans la proposition de formalismes/descriptions de certaines des entrées nécessaires pour le déploiement initial, à savoir l'infrastructure cible sur laquelle une application est susceptible d'être déployée ainsi que l'application elle-même.

Plan

La suite de ce manuscrit est organisée comme suit :

Le **Chapitre 1** est consacré aux généralités sur les systèmes distribués. Certains concepts de base relatifs aux systèmes distribués et qui sont utilisés dans le reste du manuscrit y sont expliqués.L'intergiciel DIET sur lequel nous avons appliqué nos travaux y est aussi décrit.

Le **Chapitre 2** est consacré à l'état de l'art sur les domaines de recherche en lien avec notre travail.

Dans le **Chapitre 3**, nous décrivons l'architecture proposée, qui met en perspective nos contributions avec les travaux antérieurs sur l'intergiciel DIET. Nous y décrivons aussi nos contributions pour le déploiement initial; à savoir, les descriptions de certaines entrées nécessaires pour le déploiement initial. Il s'agit de la description d'une infrastructure physique sur laquelle on peut déployer une application et de la description d'un intergiciel hiérarchique de type DIET.

Dans le **Chapitre 4**, nous avons décrit un algorithme distribué dont le but est de ramener (après chaque perturbation) le déploiement à un état stable.

Le **Chapitre 5** est consacré à la description du simulateur que nous avons conçu pour valider l'algorithme décrit dans le chapitre 4. Les simulations effectuées y sont aussi décrites.

Ce dernier chapitre est suivi par la conclusion et les perspectives.

CHAPITRE 1

Systèmes Distribués

Sommaire

1.1 Généralités

Un système distribué est défini par Tanenbaum et al. [37] comme une collection d'ordinateurs indépendants qui apparaît à ses utilisateurs comme un système unique et cohérent. Cependant, un système distribué n'est pas toujours qu'une collection d'ordinateurs mais peut aussi être une collection de processus, de processeurs, et plus généralement d'entités autonomes. Ces entités autonomes et interconnectées coopèrent pour la réalisation d'un objectif. Si les entités autonomes sont, par exemple, des processus, l'objectif est l'exécution d'un algorithme distribué dont chaque processus exécute le code.

Les entités qui composent ces systèmes partagent un certain nombre de caractéristiques de base [38] :

- elles sont autonomes et sont ainsi en mesure d'exécuter des tâches de manière indépendante;

- elles sont interconnectées : directement où indirectement, ces entités doivent pouvoir communiquer, selon un modèle de communication;

- elles disposent d'un mécanisme de coordination leur permettant de coopérer pour atteindre un objectif.

Ces entités peuvent être homogènes (dans ce cas, elles sont identiques) ou hétérogènes. Elles sont réparties géographiquement, sont concurrentes et asynchrones (il n'existe pas un temps global pour tout le système et chaque entité à son horloge locale).

Un des avantages majeurs des systèmes distribués est le partage de ressources (matérielles et/ou logicielles). Ceci permet aux utilisateurs d'accéder à des ressources distantes, d'avoir accès à des services qu'une seule entité ne pourrait offrir (augmentation de performance par une parallélisation par exemple). Contrairement à un système centralisé, caractérisé par un élément central qui rend le système indisponible en cas de panne, et qui constitue un goulet d'étranglement, un système distribué n'a pas un élément central et peut être tolérant aux pannes (continuer à fonctionner après la défaillance d'une partie des entités), en répliquant par exemple les ressources et les calculs sur différents sites, augmentant ainsi la fiabilité du système.

Cependant, ces systèmes présentent plusieurs points de défaillance possibles (puisque chaque entité est autonome et peut tomber en panne indépendamment des autres) et sont difficiles à gérer. De même, leur sécurité est plus complexe à assurer puisque les entités peuvent être réparties géographiquement.

Les systèmes informatiques sont passés d'une époque où ils étaient chers, centralisés, larges, isolés (1945-1985) à une autre époque (1985+) marquée par deux avancées majeures que sont le développement de puissants microprocesseurs et l'avènement de réseaux informatiques de plus en plus performants. L'arrivée de ces microprocesseurs qui avaient la puissance des gros systèmes a réduit la taille et les coûts de ces matériels. Quant aux réseaux, ils ont permis d'interconnecter des machines proches (LAN) ou lointaines (WAN) afin qu'elles puissent s'échanger des informations. Contrairement aux précédents, ces systèmes n'étaient plus centralisés mais distribués.

1.1.1 Exemples de systèmes distribués contemporains

Différents types de plates-formes distribuées sont apparues au début des années 2000 grâce à l'exploitation des travaux antérieurs et aux avancées technologiques. Parmi ces plates-formes, on peut citer les grilles (grid), les clouds, les réseaux Ad hoc. Les réseaux Ad hoc sont des réseaux sans infrastructure [39], dans lesquels il n'existe pas une entité centrale qui coordonne les communications comme c'est le cas dans les réseaux avec infrastructure. Ils sont constitués d'un ensemble de nœuds, dotés de capacités de communication sans fil, qui participent eux mêmes au routage des messages en transmettant ceux qui ne leur sont pas destinés pour qu'ils atteignent leur destination. Les entités qui les constituent peuvent être mobiles (se déplacer de manière indépendante le cas échéant) avec pour conséquence une topologie du réseau qui change continuellement. Chaque entité peut communiquer avec celles qui sont dans sa portée radio. Si tous les nœuds sont mobiles on les appelle MANET (Mobile Ad hoc NETwork). Les nœuds peuvent être des téléphones, des ordinateurs portables, tablettes, des véhicules, etc. Les réseaux de capteurs sans fil [40] sont un cas particulier des réseaux ad hoc. Les nœuds sont des capteurs, disposant d'interfaces de communication sans fil. Les données obtenues par les capteurs sont transmises à un élément central en les faisant transiter éventuellement par d'autres nœuds.

Les grilles informatiques

Une grille informatique (grid computing) est un type de plate-forme distribuée introduit à la fin des années 90 par Ian Foster et Carl Kesselman [41, 42] qui le définissaient comme une "infrastructure matérielle et logicielle qui fournit un accès sûr (fiable), accessible et bon marché à de grandes capacités de calcul". La grille signifiait alors une infrastructure de calcul distribué pour la science de pointe avec des applications très gourmandes en puissance de calcul (simulations de physique nucléaire, prédictions météorologiques,...). Le terme de grille a été choisi par analogie avec le réseau électrique (appelé power grid). Cela signifie que la fourniture des services informatiques devrait avoir des caractéristiques semblables à la distribution de l'électricité : disponible partout, simple et facile d'accès à travers une interface standard (prise électrique normalisée), utilisation à la demande et en fonction des moyens de l'utilisateur (pas forcément informaticien) sans avoir à se préoccuper des aspects techniques de production (types de machines, moyens de transport, provenance, etc.). Le concept a été popularisé au début des années 2000 même si plusieurs travaux antérieurs permettant sa mise en production existaient bien avant sans porter le nom de grille [43].

Les grilles sont organisées dans une architecture en couche : entre la couche physique (ou fabrique) et la couche application se trouve une couche intermédiaire appelé intergiciel (middleware) qui offre divers services aux applications et aux utilisateurs (découverte et allocation de ressources par exemple). Parmi ces intergiciels on peut citer Globus [2, 44], Unicore [45], DIET [16].

Les grilles ont évolué en trois phases : les premières grilles étaient axées d'abord sur le partage de la puissance de calcul entre centres informatiques, le partage des données a suivi. Elles utilisaient des solutions sur mesure, pour des besoins spécifiques (première version de Globus). La deuxième génération se caractérise par l'utilisation des intergiciels permettant d'intégrer des technologies de grille différentes. La troisième génération correspond à l'intégration des technologies web dans les intergiciels, qui avec les techniques de virtualisation rendent la complexité de l'infrastructure presque invisible. Ils ont été ensuite enrichis par ajout d'une couche de sémantique, les rendant plus "intelligents" et autonomes. Cependant, ces grilles ne prenaient pas en compte les nouveaux paramètres comme la généralisation des appareils mobiles, les réseaux sans fil, etc. De nouveaux projets de grille ont émergés en mettant l'accent dès leur conception sur des problèmes liés à des notions comme l'ubiquité ("pervasiveness") et l'auto-gestion ("self-management"). Les grilles jusqu'à la troisième génération sont qualifiées de grilles traditionnelles et les autres de grilles émergentes [46].

Des infrastructures de grilles sont aujourd'hui en production à travers le monde comme grid'5000 [47], EGI [48], OSG[1], etc.

L'OGF (Open Grid Forum)[2] coordonne les efforts de standardisation dans le domaine.

Les Clouds

Le Cloud [3, 49] est une évolution du concept de grille. Il désigne un ensemble de technologies et systèmes permettant de fournir divers types de ressources (calcul, stockage, logiciels, etc.) à la demande, à travers internet et généralement payant en fonction de l'utilisation. Les ressources sont fournies de manière dynamique et peuvent ainsi s'adapter à la charge de l'utilisateur. Cela permet au fournisseur d'exploiter son infrastructure de

[1] http://www.opensciencegrid.org/
[2] www.ogf.org

manière optimale. Le cloud est en général la propriété d'une seule organisation, peut être privé, public, hybride, communautaire. Une caractéristique du cloud est l'élasticité, permettant au fournisseur d'être en mesure d'augmenter ou de réduire les ressources offertes en fonction de la variation des besoins des utilisateurs.

Plusieurs types de solutions de cloud sont disponibles [50, 51] comme : OpenStack[3], OpenNebula[4], Eucalyptus[5].

L'intergiciel de grille et cloud DIET

L'intergiciel DIET [16] nous sert de cas d'utilisation, et les travaux décrits dans ce manuscrit lui sont appliqués.

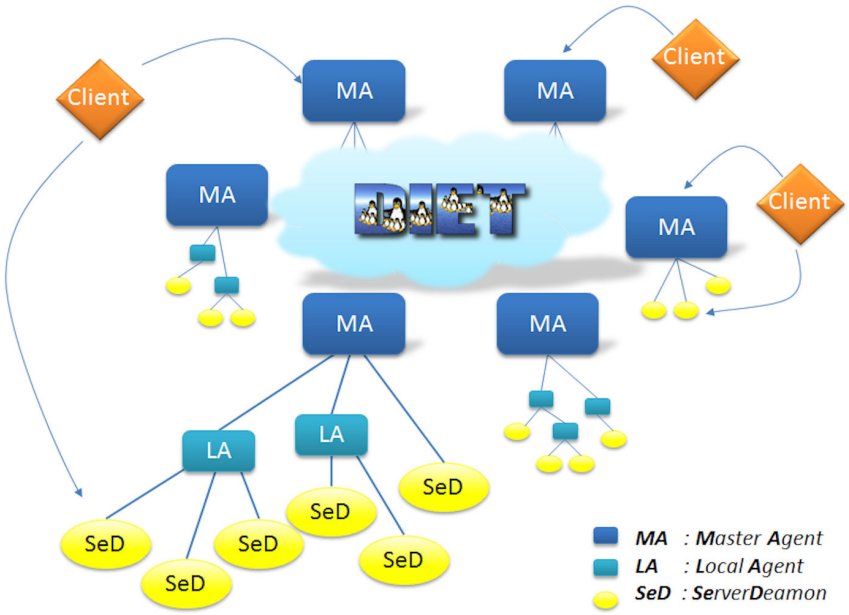

Figure 1.1: Hierarchie multi-MA de DIET.

DIET est un intergiciel GridRPC [52]. Un des objectifs de l'API GridRPC [53] est de définir clairement une syntaxe et une sémantique pour les GridRPC qui sont une extension des Remote Procedure Call (RPC) [54] appliquée au domaine des grilles de calcul. Le modèle de programmation RPC est l'un des premiers modèles permettant d'exécuter des applications sur un environnement distribué. Les applications client et serveur des utilisateurs finaux doivent être décrites dans le modèle de programmation fourni. L'architecture par composant de DIET est structurée de manière hiérarchique pour améliorer le passage à l'échelle comme illustrée à la Fig. 1.1 . La boite à outils DIET est implémentée en CORBA [55]. Il bénéficie par conséquent des mises à jour des services standardisés et stables d'implémentation à haute performance et librement disponibles de CORBA. DIET est constitué de plusieurs types de composants. Un **Client** est une application qui utilise l'infrastructure DIET pour résoudre un problème en utilisant une approche GridRPC. Un SED (Server Daemon) joue le rôle de fournisseur de services. Il exporte ses fonctionnalités via une interface de service de calcul standardisée. Un

[3] www.openstack.org
[4] http://opennebula.org/
[5] https://github.com/eucalyptus/eucalyptus/wiki

seul SeD peut offrir plusieurs services de calcul. Le troisième composant de DIET, les **agents**, facilitent la localisation et l'invocation des services et donc l'interaction entre les clients et les SeDs. La hiérarchie des agents fournit des services de haut niveaux comme l'ordonnancement et la gestion des données. Ces services permettent un passage à l'échelle grâce à leur distribution dans la hiérarchie des agents composés d'un agent maître (**Master Agent ou MA**) et de plusieurs agents locaux (**Local Agents ou LA**). Plusieurs hiérarchies peuvent être inter-connectées pour former une plateforme multi-MA.

Une inter-action typique de DIET se déroule selon le scénario suivant :

(1) D'abord un **Client** se connecte à la hiérarchie et envoie un message de découverte en fonction du type de service qu'il souhaite utilisé. Le message est envoyé au **MA** auquel le client est connecté;

(2) ensuite, le message est propagé dans la hiérarchie du **MA** vers les SeDs à travers les **LA**;

(3) Les SeDs qui ont reçu le message répondent avec un **vecteur d'estimation** : un ensemble de valeurs qui décrit la disposition du SeD à traiter la requête. En fonction de l'implémentation du service, le **vecteur d'estimation** peut contenir des informations comme le puissance de calcul, la quantité de RAM, le temps estimé pour exécuter la requête, le nombre de requête en file d'attente, etc.;

(4) À chaque niveau de la hiérarchie des agents, les **vecteurs d'estimation** sont agrégés de sorte que le **MA** ne va recevoir qu'un nombre réduit de vecteurs;

(5) Enfin, un ou plusieurs vecteurs sont retournés au **Client** qui avait lancé la requête. Ce dernier choisit le SeD qui lui convient;

(6) La requête et les données nécessaires pour résoudre le problème sont envoyées par le **Client** au SeD choisi.

1.1.2 Tâches classiques

Bien que les systèmes distribués se présentent sous différentes facettes, un certain nombre de problèmes fondamentaux leur sont communs et servent de base au domaine. Un concepteur d'une application distribuée peut être amené à trouver une solution ou à utiliser les algorithmes existants concernant un ou plusieurs de ces problèmes. En plus de la tolérance aux pannes, un certain nombre de ces problèmes sont décrits ci-dessous.

Élection de leader

Plusieurs applications distribuées reposent sur l'existence d'un processus leader. L'élection d'un leader [56] consiste à distinguer un seul processus qui sera appelé leader avec un statut particulier à partir de tous les processus (ou d'un groupe de processus) candidats. Les autres processus sont dans un autre état différent de celui du leader. Lorsque le leader meurt ou devient injoignable, un autre processus est élu parmi les processus qui sont dans un état correct. Lorsque le graphe des processus n'est pas connexe, un leader est élu pour chaque composante connexe, et lorsque le graphe redevient connexe, un seul leader reste.

Exclusion mutuelle

Dans un système distribué, les processus s'exécutent ensemble de manière simultanée et coopèrent pour atteindre un objectif. Ils peuvent donc chercher à avoir accès à une même ressource partagée. L'objectif des algorithmes d'exclusion mutuelle [57] est de garantir qu'au plus, un seul processus peut entrer en section critique d'une ressource partagée à un moment donné. Lorsqu'un processus entre en section critique, les autres requêtes devront attendre sa sortie pour qu'un autre processus puisse avoir l'accès à la ressource. Certaines ressources ne peuvent être utilisées que par un seul processus à la fois (exemple de l'imprimante) à un moment donné.

L'exclusion mutuelle de groupe [58, 59] est une généralisation de l'exclusion mutuelle dans laquelle plusieurs ressources sont partagées entre les processus et plusieurs processus appartenant au même groupe peuvent accéder simultanément à la même ressource partagée. Cependant, des processus de groupes différents doivent accéder aux ressources partagées de manière exclusive.

Détection de propriété globale

L'état global d'un système distribué (ou une configuration du système) est constitué de l'ensemble des états des processus du système à un moment donné. Il est parfois nécessaire de déterminer si cet état global satisfait à un ou plusieurs critères ("stabilité", terminaison, ...).

La détection de la terminaison [60] d'un calcul distribué est un problème dans lequel on cherche à déterminer si tous les processus du système ont terminé un calcul. Cette détection de la terminaison peut être effectuée par une entité centrale qui a une vue globale du système. Si par contre, chaque processus doit détecter lui même la terminaison du calcul global, on dit que c'est une détection distribuée de la terminaison, qui est un problème fondamental dans les systèmes distribués. Depuis son introduction au début des années 80 [61, 62], la détection distribuée de la terminaison d'un algorithme a été bien étudiée [63–68].

Algorithmes à vagues

Les algorithmes à vagues [69–73] sont un type d'algorithme distribué classique. Ils sont utilisés, entre autres cas, pour diffuser une information dans un réseau, collecter des valeurs, synchroniser, etc. Un algorithme à vagues peut être constitué d'une ou de plusieurs vagues successives. Dans un algorithme à vagues, un nœud initie une vague en diffusant une information (jeton, requête,...) qui est propagée dans le réseau, ensuite les réponses sont remontées vers l'initiateur qui prend une "décision" et peut lancer une autre vague si nécessaire.

Les algorithmes d'écho [69], de collecte et d'agrégation de données dans un réseau [74] sont des types d'algorithmes à vagues.

Les algorithmes d'écho [69], utilisent une technique de diffusion permettant à un nœud de transmettre une information à un autre nœud. L'information est transmise par chaque nœud à ses voisins jusqu'à ce qu'elle atteigne le destinataire. L'inconvénient de cette méthode est le coût élevé en nombre de messages qu'elle induit. Ses avantages sont sa simplicité et sa facilité de mise en œuvre.

Les algorithmes distribués d'agrégation de données [74] sont des algorithmes à vagues, qui permettent de diffuser une requête dans un réseau et de collecter et d'agréger les

réponses vers le nœud source qui avait émis la requête.

Ce type d'algorithme agit en deux phases : une phase durant laquelle un nœud source diffuse une requête qui est propagée par chaque nœud à ses voisins jusqu'à ce que la requête atteigne les feuilles; et une deuxième phase durant laquelle chaque nœud, en commençant par les feuilles, renvoie sa réponse à son parent et ces réponses sont agrégées au fur et à mesure que l'information remonte vers le nœud source qui avait émis la requête.

Les algorithmes à vagues se divisent en deux familles : celle des algorithmes qui utilisent une circulation d'un jeton et celle des algorithmes qui utilisent une propagation d'information avec retour ou PIF (Propagation of Information with Feedback).

Un déploiement de DIET fonctionne sous la forme d'un PIF. Lorsqu'un client se connecte sur un master, ce dernier diffuse une requête dans le réseau pour trouver le meilleur SeD (les SeDs sont au niveau des feuilles) pour satisfaire le client. Les réponses des SeDs sont agrégées au fur et à mesure qu'elles remontent vers le master qui avait émis la requête.

1.1.3 Tolérance aux fautes

La tolérance aux pannes vise à masquer les effets d'une défaillance ou à restaurer un comportement conforme à sa spécification pour un système qui a dévié de sa spécification à cause d'une faute [73].

Plus généralement, l'objectif est de gérer les pannes qui peuvent survenir pendant une exécution comme l'arrêt brutal (crash) d'un processus, une rupture d'un lien de communication entre deux nœuds. Un système distribué peut être complexe, impliquant divers types de ressources autonomes (pouvant défaillir localement et de manière indépendante), géographiquement réparties, raison pour laquelle les fautes et les défaillances sont plus courantes que dans les systèmes centralisés. Une panne peut être locale et affecter le comportement d'une partie des autres nœuds du système sans affecter une autre partie.

En plus de la possibilité de pouvoir partager des ressources, avoir des systèmes en mesure de continuer à fonctionner (même si ce n 'est pas de manière optimale) même lorsque des défaillances touchent une partie des éléments qui le composent est un objectif majeur dans la conception des systèmes distribués que l'on cherche à rendre fiable, disponible, sûr et maintenable [37].

Un système distribué est en panne lorsqu'il ne se comporte plus conformément à sa fonction (ce pour quoi il a été prévu) [75]. Une erreur est une partie de l'état du système qui peut causé une panne. Une faute est ce qui cause une erreur. Être capable de détecter les fautes est donc d'une grande importance.

Ainsi, un système est tolérant aux pannes s'il peut continuer à fournir le service pour lequel il est prévu, et ceci même en présence de fautes.

Types de pannes

Différents modèles de fautes sont considérés dans les systèmes distribués. Lorsqu'on a une vue du système distribué de niveau processus, on peut distinguer les différents types de fautes au niveau processus [37, 73, 75].

- les arrêts : un processus à l'arrêt cesse d'exécuter ses actions (interne, de communication, de lecture et d'écriture). L'arrêt peut être définitif ("crash stop") ou temporaire ("crash recovery");

- les omissions : elles modélisent les fautes qui peuvent conduire à la perte de messages. Ce type de faute peut affecter les canaux de communication et se manifester sous la forme d'une rupture du lien (du à une problème au niveau du réseau physique sous-jacent par exemple) rendant certaines communications impossibles. Les fautes au niveau des canaux peuvent aussi provoquer la perte, la duplication, la transmission hors délais des messages. Un canal qui peut perdre des messages peut être modélisé en considérant qu'un des processus au bout du canal échoue à transmettre ou à recevoir certains messages qu'il devait envoyer ou recevoir. Un autre moyen de modéliser les pertes de messages dans un système synchrone avec passage de messages est de permettre la perte d'au plus un certain nombre de messages à chaque round, mais les canaux sur lesquels ces pertes apparaissent peuvent changer d'un round en un autre;

- les pannes temporelles : elles sont dues à un délai non respecté, par exemple dans un système temps réel où on exige que les actions soient terminées dans un intervalle de temps donné.

Les différents types de pannes peuvent être classés dans des catégories de plus haut niveau :

- les pannes transitoires : une panne transitoire peut perturber l'état d'un processus d'une manière arbitraire. Elles capturent les effets de l'environnement, dont la durée est limitée. L'élément responsable de la panne peut n'être actif que pendant un temps limité, mais l'effet produit sur l'état global du système reste. Les omissions sont un cas de panne transitoire, lorsque l'état d'un canal est perturbé;

- les pannes byzantines : elles modélisent un comportement arbitraire des processus. Ce dernier modèle est utile pour simuler des attaques et situations dans lesquelles les fautes sont difficiles à caractériser. Un algorithme dans le modèle avec fautes byzantines doit donc fonctionner correctement (atteindre son but) quel que soit le comportement des processus.

Techniques de tolérance aux pannes

Pour assurer une gestion des pannes qui peuvent éventuellement survenir dans un système distribué, il faut d'abord être en mesure de détecter ces événements, c'est-à-dire, être en mesure de détecter que le système ne se comporte plus de manière conforme à sa fonction. La détection d'une panne n'est pas toujours possible (par exemple dans un système asynchrone où le temps d'exécution des actions n'est pas borné). Mais lorsque la détection est possible, elle peut se faire lorsqu'on détecte un signal ou message d'erreur. Une erreur latente est une erreur présente mais non détectée. Une fois la faute détectée, il faut la gérer. Les techniques utilisées pour assurer la tolérance aux pannes peuvent être regroupées en deux catégories selon que les pannes sont masquées ou non masquées.

Pannes masquées

Lorsqu'une panne est masquée, son occurrence n'a pas d'impact sur le système. Cette catégorie de techniques adoptent une vision pessimiste de la tolérance aux pannes. Ces algorithmes tolèrent des dysfonctionnements continus touchant le système. Ces techniques sont nécessaires dans les systèmes critiques (mettant en général la vie des personnes en

danger en cas de défaillance total : un avion doit pouvoir continuer à voler même si un de ses appareils ne fonctionne pas parfaitement). Elles sont cependant difficiles à mettre en œuvre et ne tolèrent qu'un nombre restreint de dysfonctionnements. Les techniques de réplication utilisées pour assurer la tolérance aux pannes font partie de cette catégorie.

Dans les techniques de réplication, les données et/ou les programmes sont répliqués ce qui permet au système de pouvoir continuer à fonctionner même en présence de pannes [76].

Pannes non masquées

Dans cette catégorie, les pannes peuvent affecter temporairement le comportement du système, moment pendant lequel il ne se comporte plus exactement comme spécifié. Cependant, une restauration du comportement conforme à la spécification aura lieu. Parmi les techniques utilisées pour assurer la restauration d'un comportement correct on retrouve la reprise sur panne [77–79], et les algorithmes auto-stabilisants [36].

La reprise sur panne repose sur un historique, un enregistrement périodique des états des processus au cours de leur exécution, sauvegardé dans une mémoire stable. Lorsqu'une panne est détectée, le système est restauré à partir des derniers états sauvegardés. L'état retrouvé n'est pas forcément l'état avant la panne, mais un état correct.

Algorithmes de consensus

Il existe des situations dans lesquelles des processus distribués doivent trouver un accord, prendre la même décision,... C'est le cas, par exemple, dans un système de transaction où tous les processus qui participent doivent tomber d'accord sur l'opération à exécuter et l'appliquer : soit sauvegarder les résultats de la transaction, soit les annuler. Dans tous les cas, la décision prise doit être la même pour tous les processus qui participent. Ils vont donc appliquer la même opération. Ce problème, connu sous le nom de consensus [73, 80, 81], implique un ensemble de processus distribués, dont certains peuvent ne pas être fiables. Chaque processus choisit une valeur initiale, à partir d'un ensemble commun à tous les processus. Le problème consiste, pour les processus fiables, à trouver un consensus, c'est à dire choisir, de manière irrévocable, la même valeur finale, parmi celles proposées; en respectant les conditions suivantes :

- tout processus fiable finira par décider, c'est à dire choisir une valeur finale (terminaison);
- la valeur finale choisie doit être identique pour tous les processus fiables (accord);
- la valeur finale choisie doit avoir été proposée (validité). Ainsi, si tous les processus fiables avaient choisi la valeur initiale v, alors la valeur finale doit être v.

Il existe des variantes du problème dans lesquelles, on exige plus que tous les processus choisissent la même valeur, mais que le cardinal de l'ensemble des valeurs choisies soit au plus égal à un entier k ("k-set consensus"). Dans ce cas, le consensus devient un cas particulier lorsque $k = 1$ [82].

Dans un système où le réseau et les processus sont complètement fiables, le problème peut trouver une solution triviale. Par exemple, les processus peuvent s'échanger les valeurs et choisir une valeur finale de manière déterministe en appliquant la même fonction (le maximum/minimum par exemple) à l'ensemble (des valeurs initiales) reçu. Le même ensemble sera reçu par tous puisqu'il n'y a pas de pannes.

Cependant, les systèmes réels sont en général sujets à des pannes, soit des liens de communication, soit des processus qui peuvent se terminer ou se comporter de manière arbitraire.

Le protocole de consensus décrit dans [81, 83] se base sur un modèle de faute de type “crash recovery”, dans lequel un processus peut se terminer à tout instant mais peut également redémarrer.

La recherche d’un algorithme de consensus peut se révéler plus difficile, voire impossible, en fonction des hypothèses et modèles considérés. Ficher et al. ont montré [84] que : même en excluant les pannes byzantines, et en considérant comme type de panne que le “crash” (terminaison permanente du processus) et en supposant un environnement dans lequel l’envoi et la réception des messages sont fiables (un message envoyé arrive à sa destination et n’est pas dupliqué), aucun protocole de consensus, complètement asynchrone, ne peut tolérer le crash, ne serait ce que d’un seul des processus. Il faut souligner que dans le modèle asynchrone, le “crash d’un processus” ne peut pas être détecté de manière fiable car il est difficile de faire la distinction entre un processus très lent dans l’exécution des instructions et un processus qui s’est terminé. Ce résultat (connu sous le nom de FLP) a suscité un grand nombre de travaux qui l’ont étendu en utilisant d’autres hypothèses ou modèles [85–87].

La recherche d’un consensus, dans un modèle de fautes byzantines, est connue sous le nom de “problème des généraux byzantins” [88].

1.2 Modèles

Les systèmes distribués sont implémentés de diverses manières. Lorsqu’on les étudie, on se base généralement sur des modèles [1, 89] permettant de décrire leurs caractéristiques et de faire abstraction des détails sur le réseau physique sous-jacent par exemple. Un modèle peut capturer les caractéristiques essentielles d’une grande variétés de systèmes réels. L’avantage d’une telle démarche est de pouvoir réfléchir à partir des modèles et non des systèmes réels. Plusieurs modèles ont été proposés pour les systèmes distribués. L’existence de ces différents modèles a suscité des réflexions sur les relations entre les différents modèles, la nature des problèmes qui peuvent être résolus (ou qui ne peuvent pas l’être) dans un modèle donné, le modèle qui permet de résoudre le plus de problèmes,...

Les systèmes distribués sont modélisés de manière générale sous la forme d’un graphe. Les communications entre les processus formant le système distribué sont décrites selon un modèle à mémoire partagée ou à passage de messages.

Un algorithme distribué est décrit par un ensemble de règles avec des gardes. chaque processus exécute un programme séquentiel constitué d’un certain nombre de règles. L’exécution d’un algorithme distribué est généralement décrit par un modèle de transition.

1.2.1 Systèmes distribués

Un système distribué peut être modélisé par un graphe $G = (V, E)$ où V représente l’ensemble des sommets (appelés aussi nœuds) du graphe et E l’ensemble des arêtes. Chaque $v \in V$ représente une des entités constituant le système et chaque arête $e \in E$ (e est un couple (u, v) avec $u \in V, v \in V$) représente une relation (généralement un lien de communication) entre les deux entités u et v. Un graphe peut être orienté ou non orienté. Pour un graphe orienté, toute arête $e = (u, v)$ est orientée de u vers v, est sortante pour

u et entrante pour v. La paire (u, v) est ordonnée, u est appelé prédécesseur de v et v est appelé successeur de u.

Dans la suite, nous considérons uniquement les graphes non orientés et donnons la définition des quelques concepts sur les graphes [90].

Si $e = (u, v) \in E$, on dit que le nœud u est **adjacents** à v, que les nœuds u et v sont **voisins**. On définit et note le **voisinage** (l'ensemble des voisins) d'un nœud u par $\mathcal{N}_u = \{v \in V, (u, v) \quad ou \quad (v, u) \in E\}$.

Le **degré** d'un nœud u est défini et noté par $D(v) = |\mathcal{N}_u|$.

Un nœud u est dit **isolé** lorsque $D(u) = 0$, c'est à dire un nœud sans voisin.

Deux arêtes $e_1,\ e_2 \in E$ sont dites adjacentes lorsque $e_1 \neq_2$ et $e_1 \bigcap_2 \neq \emptyset$ (les deux arêtes sont distinctes et ont un sommet en commun).

Un chemin entre v_1 et v_k est une séquence d'arêtes adjacentes (v_1, v_2), (v_2, v_3),..., (v_{k-1}, v_k) plus simplement noté $(v_1, v_2, ..., v_{k-1}, v_k)$ dans lequel tous les v_i sont distincts à l'exception éventuellement de v_1 et v_k qui peuvent être égaux. Lorsque $v_1 = v_k$, le chemin est appelé circuit ou cycle. Un graphe est dit acyclique s'il ne comporte pas de cycle. Deux sommets sont dits connectés lorsqu'il existe un chemin entre eux.

Un sous graphe du graphe $G = (V, E)$ est un graphe $G1 = (V1, E1)$ tel que $V1 \subseteq V$ et $E1 \subseteq E$. On dit que le sous graphe $G1$ est induit par $V1 \subseteq V$ lorsque $G1$ est obtenu en supprimant de G, l'ensemble des sommets dans $V - V1$ et les arêtes qui leur sont incidentes.

Soit la relation R sur l'ensemble V des sommets du graphe non orienté G tel : pour une paire de sommets (u, v), uRv signifie que u et v sont connectés; autrement dit, qu'il existe un chemin entre u et v. Cette relation R, "est connecté", est une relation d'équivalence. Chaque classe d'équivalence de sommets de V induit un sous-graphe, appelé composante connexe.

Un graphe est dit connexe s'il est constitué d'une seule composante connexe; et non-connexe dans le cas où il est constitué de plusieurs composantes connexes. Ainsi, dans un graphe connexe, il existe un chemin entre chaque paire de sommets alors que dans un graphe non connexe, il existe au moins une paire de sommets non connectés. Lorsqu'une composante connexe est constituée d'un unique sommet, ce sommet est dit isolé.

Un graphe acyclique et non-connexe est appelé une forêt.

Un graphe $G = (V, E)$ est dit complet si $\forall v \in V, \mathcal{N}_v = V - \{v\}$, c'est à dire que chaque sommet est connecté à tous les autres sommets. Dans ce cas, si $|V|=n$, $|E|=n(n-1)/2$.

1.2.2 Modèle de communication

Les modèles de communication pour les systèmes distribués sont de deux types : le modèle à mémoire partagée (dans lequel on retrouve le modèle à état et le modèle à registre) et le modèle à passage de messages.

Dans le modèle à mémoire partagée, les nœuds communiquent en écrivant et lisant des mémoires partagées. On retrouve dans cette classe, le modèle à état et le modèle à registre.

Dans le modèle à état, si deux nœuds sont voisins, alors chacun peut modifier son état (lui seul peut le faire) et lire l'état de son voisin.

Dans le modèle à registre, si deux nœud N_i et N_j sont voisins, alors il existe deux registres R_i (pour N_i) et R_j (pour N_j) entre eux. Pour communiquer, N_i écrit dans R_i et lit R_j tandis que N_j écrit dans R_j et lit R_i.

La simulation d'un modèle A par un modèle B permet, lorsque cela est possible, d'adapter tout algorithme conçu pour le modèle A en un algorithme pour le modèle B [91, 92].

Nous détaillons dans la section suivante le modèle à passage de messages, modèle sur lequel est basé le travail présenté dans ce manuscrit.

Modèle à passage de messages

Dans ce modèle, les nœuds (processus) communiquent uniquement par échange de messages qui transitent par des canaux de communication [1]. Chaque nœud dispose de son propre espace d'adressage.

Les communications peuvent suivre un modèle synchrone ou asynchrone [72, 73].

Dans un modèle asynchrone, l'envoi et la réception d'un message sont des événements indépendants. Ainsi, l'envoi d'un message i n'est pas conditionné par la réception d'un message précédent $(i-1)$. Il n'y a pas de temps global et chaque nœud possède sa propre horloge. Le temps que prend un message émis pour atteindre son destinataire est fini mais non borné. Ce délai peut donc être arbitrairement long.

Dans un modèle synchrone, l'envoi et la réception d'un message sont coordonnés pour former un seul événement. Ainsi, un message n'est envoyé que lorsque son destinataire est prêt pour le recevoir. Les horloges des différents nœuds sont synchronisées et marquent le même temps, ou bien le décalage est borné. Les nœuds exécutent leurs actions par cycle (ou round). Pour chaque cycle, chaque nœud exécute un ensemble prédéfini d'actions, et aucun nœud ne débute le cycle i que lorsque tous les processus ont terminé le cycle $(i-1)$.

Dans un modèle à passage de messages, des hypothèses sont faites sur les propriétés des canaux de communications par où transitent les messages.

Ainsi, les canaux peuvent être fiables ou non fiables, avec des capacités (tailles) finies (bornées ou non bornées) ou infinies. Dans un canal fiable, les pertes de messages sont inexistantes contrairement à un canal non fiable où cette possibilité est prise en compte.

Lorsque la taille d'un canal est bornée, et lorsque la file d'attente est pleine, le processus qui transmet est bloqué, où une erreur est générée ou un message est supprimé de la file. L'ordre dans lequel les messages sont transmis et reçus peut être modélisé par des files de type FIFO (les messages échangés entre deux processus sont reçus dans l'ordre où ils sont été envoyés) ou non.

1.2.3 Modèle d'exécution

Dans la suite, on s'intéresse aux systèmes distribués avec un modèle de communication par passage de messages.

Dans un algorithme centralisé, les processus ont une vision globale du système (topologie, les autres processus, etc.) et peuvent prendre une décision optimale.

Un algorithme distribué est exécuté par un ensemble de processus formant un système distribué qui coopèrent pour atteindre un objectif. Lorsque tous les processus exécutent le même programme, on dit que le système est uniforme. Sinon, l'algorithme est dit non-uniforme. Dans un algorithme distribué, les processus ont une vision réduite du système (en général, ils ont connaissance d'une partie ou de l'ensemble de leurs voisins, mais ils n'ont pas de connaissance globale de la topologie du réseau ou du nombre total de processus).

Dans un modèle à passage de messages, l'exécution d'un algorithme distribué et son évolution dans le temps sont modélisés par un système de transition [1, 38, 72].

L'algorithme est exécuté par un ensemble de processus séquentiels, qui communiquent entre eux par échange de messages pour atteindre un objectif commun. Un processus séquentiel exécute un ensemble d'instructions de manière séquentielle.

Définition 1 (Algorithme distribué)**.** *Un algorithme distribué A peut être modélisé par un ensemble de règles A :: R_1 | R_2 |.....| R_n où chaque R_i est une règle de la forme :*

si *$Guard_i$* ***alors*** *$Action_i$*

Où

$Guard_i$ est un prédicat booléen, fonction des variables locales du nœud (toutes ou une partie) et éventuellement d'information externe au nœud (provenant d'oracle, d'un serveur, des voisins);

$Action_i$ est un ensemble d'instructions exécutées par le nœud et qui peuvent modifier les valeurs de ses variables locales. Ces instructions ne sont exécutées que lorsque $Guard_i$ est vrai, et dans ce cas on dit que le nœud est activable ou déclenchable ("enabled").

L'ensemble des actions que peut exécuter un processus peuvent être regroupées dans les catégories suivantes :

- action interne : toute exécution dans l'espace d'adressage du processus produisant la modification de la valeur d'une ou de plusieurs de ses variables locales (état interne);
- action de communication : tout envoi ou réception d'un message à destination ou en provenance d'un autre processus du système (ensemble des processus formant le système distribué);
- action de lecture : toute lecture de données en provenance d'un élément externe au système. Les données lues peuvent avoir une influence sur l'état interne du processus;
- action de d'écriture : toute écriture de données vers un élément extérieur au système.

Chaque processus a un état e_i.

Définition 2 (État d'un nœud)**.** *L'état d'un nœud est défini à partir de ses variables locales (toutes ou une partie).*

On note E l'ensemble des états possibles d'un processus. Un processus peut changer d'état à la suite d'une action interne.

Définition 3 (État d'un canal de communication)**.** *L'état d'un canal de communication est défini par l'ensemble des messages qui y circulent au moment considéré. L'état d'un canal est donc soit vide soit non vide.*

Définition 4 (État global ou configuration)**.** *Un état global d'un algorithme distribué, appelé aussi une configuration (notons le c) est un vecteur des états de chaque processus, de la forme $c=[e_1, e_2, ..., e_n]$ où e_i est l'état courant du processus i. Dans le cas d'un système asynchrone, il faut en plus du vecteur des états, prendre en compte le vecteur constitué de l'état de chaque canal de communication.*

La configuration d'un algorithme distribué évolue à la suite d'une action interne à un processus. L'évolution de l'état global, appelée transition, se fait donc de manière discrète.

Définition 5 (Système de transition)**.** *Un système de transition est un triplet* $\mathcal{T}=(\mathcal{C}, \rightarrow, \mathcal{I})$ *où :*

- $\mathcal{C}$ *est l'ensemble de toutes les configurations possibles;*
- $\rightarrow$ *est une relation binaire dans* $\mathcal{C}$*;*
- $\mathcal{I} \subseteq \mathcal{C}$ *est l'ensemble des configurations initiales.*

Une transition t_i est donc un couple $(c_i, c_{i+1}) \in \mathcal{C} \times \mathcal{C}$ tel que $c_i \rightarrow c_{i+1}$.

Puisque plusieurs nœuds peuvent être simultanément activables, et afin de modéliser le comportement d'un nœud activable, on utilise un ordonnanceur, appelé *démon* (deamon) ou *adversaire* [93]. A chaque pas de calcul (transition), il choisit les nœuds qui vont exécuter leurs actions parmi les nœuds activables. Un *démon* est dit :

- *central* ou *séquentiel* s'il n'active qu'un seul nœud activable;
- *distribué* s'il peut activer un sous ensemble de nœuds (de cardinal au moins égal à deux) parmi les nœuds activables. Les nœuds activés exécutent leurs actions de manière synchrone;
- *synchrone* ou *parallèle* s'il doit activer tous les nœuds activables. Les nœuds activés exécutent leurs actions de manière asynchrone.

Pour modéliser les choix du *démon*, la notion d'équité ("fairness") est utilisée. Le *démon* est dit [94] :

- *faiblement équitable ("unfair")* s'il doit ultimement activer tout nœud continûment et infiniment activable;
- *fortement équitable ("fair")* s'il doit ultimement activer tout nœud infiniment activable;
- *inéquitable* s'il n'est pas équitable (ni fortement, ni faiblement).

Définition 6 (Configuration terminale)**.** *Une configuration* c_t *est dite terminale s'il n'existe aucune autre configuration* $c \in \mathcal{C}$ *telle que* (c_t, c) *soit une transition.*

Dans un système de transition, l'évolution au cours du temps de l'état global est une suite de transitions causées par des événements internes au processus. Cette évolution est capturée par la notion d'exécution.

Définition 7 (Exécution)**.** *Une exécution* $\mathcal{E}$ *est une séquence* $(c_0, c_1, c_2, ...)$ *de configurations telle que* $c_0 \in \mathcal{I}$*, et* $c_i \rightarrow c_{i+1}$ *pour tout* $i \geq 0$*. Une exécution est finie si elle se termine par une configuration terminale, sinon elle est infinie.*

1.2.4 Auto-stabilisation

L'apparition de pannes dans un système distribué, constitué d'un grand nombre de processus et de liens de communication, est un événement courant et pas exceptionnel. Certains de ces systèmes sont dynamiques, permettant ainsi l'ajout et le retrait de processus en cours d'exécution. Ils sont confrontés à des changements de topologies, et à diverses autres perturbations.

Il est donc nécessaire de proposer des moyens de gérer ces pannes. Cependant, vue la taille et la complexité de ces systèmes, une gestion manuelle des pannes serait inefficace, voire impossible. Les techniques de tolérance aux pannes permettent une gestion automatisée des pannes.

Ces techniques se divisent en deux classes : celle dans laquelle on masque les pannes (les effets des pannes sont invisibles à l'application) et celle dans laquelle les pannes ne sont pas masquées. L'auto-stabilisation fait partie de cette dernière classe.

Le concept d'auto-stabilisation dans les systèmes distribués a été introduit en 1974 par E. W. Dijkstra [36].

Un système auto-stabilisant doit tolérer les pannes transitoires (des processus et des liens). Une panne transitoire peut corrompre les données en mémoire des processus (variables, pointeur de programme), les canaux de communication, mais sans corrompre le code qui est exécuté. L'exécution du code de l'algorithme auto-stabilisant devra permettre de retrouver un état correct, à partir de n'importe quel état, atteint à cause des pannes transitoires ou de valeurs initiales arbitraires.

Définition 8 (Algorithme auto-stabilisant)**.** *Un algorithme est dit auto-stabilisant si quel que soit son état initial, il atteindra un état correct (légitime), après un nombre fini d'étapes.*

Intuitivement, un algorithme est auto-stabilisant s'il est capable de retrouver un comportement correct à partir d'un état global initial arbitraire [93]. L'état initial arbitraire permet de prendre en compte l'effet des fautes sur le système.

Les algorithmes auto-stabilisants sont utilisés comme un moyen d'assurer une tolérance aux pannes parce qu'après une perturbation imprévue, ils offrent la garanti de retrouver un état correct sans intervention extérieure.

Pour valider le caractère auto-stabilisant d'un algorithme, il faut montrer que les propriétés de **convergence** et de **clôture** [95] sont vérifiées.

Définition 9 (Convergence)**.** *La propriété de convergence stipule que quelque soit l'état initial, un système exécutant un algorithme auto-stabilisant va atteindre un état légal au bout d'un nombre fini de transitions.*

On doit souligner ici que la convergence commence après que la dernière action de ce qui constitue la panne a été appliquée. Si le système est perpétuellement perturbé, sa convergence ne pourra pas être prouvée. Donc, il faut au moins qu' un délai suffisamment long pour permettre une convergence existe entre deux pannes. En d'autres termes, le temps moyen entre les pannes doit être plus grand que le temps de réparation (c'est à dire d'exécution de l'algorithme auto-stabilisant).

Définition 10 (Clôture)**.** *La propriété de clôture stipule qu'une fois un système auto-stabilisant a atteint un état légal, et en l'absence de fautes, les transitions le laisseront dans un état légal.*

La capacité d'un système à pouvoir se réajuster ou retrouver un état correct, après des perturbations, et sans intervention externe, est une propriété utile dans les systèmes distribués. C'est la raison pour laquelle, l'auto-stabilisation, en tant qu'une des techniques de tolérance aux pannes à suscité beaucoup d'intérêt.

CHAPITRE 2

État de l'art

Sommaire

Le déploiement de logiciel [17–20] est défini dans [19] comme le processus, constitué d'un ensemble d'activités liées, entre l'acquisition et l'exécution du logiciel. Ce processus a pour objectif de rendre opérationnel une application, qui peut ainsi être utilisée par les utilisateurs. Une fois l'application déployée, le processus de déploiement continue par les mécanismes d'adaptation du logiciel afin de chercher à atteindre une qualité de service. En effet, les infrastructures modernes sur lesquelles on déploie des applications sont caractérisées par de fréquentes variations de leur environnement.

Cependant, l'objectif d'un déploiement de logiciel, en plus de ceux déjà notés dans ces définitions, peut être de maintenir une qualité de service autre que la seule disponibilité, et que la non-atteinte de cette qualité, provoque une stratégie de redéploiement. Cette qualité de service peut être qualitative (par exemple maintenir une topologie particulière) ou bien quantitative (par exemple le logiciel devra être capable de réaliser certaines tâches dans un temps inférieur à une valeur donnée).

Ainsi, s'appuyant sur les définitions précédentes, on peut définir le déploiement de logiciel comme un processus consistant en un ensemble d'activités reliées et ayant pour but de rendre le logiciel disponible à l'utilisation, à jour et en état d'assurer une qualité de service prédéfinie.

Le processus de déploiement suppose au moins l'existence d'un logiciel qu'on veut déployer, d'une infrastructure cible, constituée de ressources informatiques (ordinateurs,

clusters, téléphones,...) interconnectées, sur laquelle le logiciel sera déployé, et, éventuellement, d'outils permettant d'automatiser le déploiement (sinon l'opération sera effectuée manuellement).

2.1 Outils et frameworks de déploiement

Des standards et spécifications du domaine, on peut citer la spécification de l'OMG (Object Management Group) pour le déploiement d'applications distribuées à base de composants [18], l'OSGi (Open Services Gateway initiative) [96].

OSGi fournit un environnement d'exécution, basé sur la technologie Java. Le processus de déploiement inclut les activités suivantes : l'installation, la mise à jour, la désinstallation. Il fournit un cadre qui permet le déploiement d'applications Java, extensibles et téléchargeables (appelées "bundle"). Un "bundle" est constitué de classes Java et d'autres ressources (librairies, fichiers, etc.), l'ensemble pouvant fournir un ou plusieurs services aux utilisateurs. Ils sont déployés sous la forme d'archives JAR (Java ARchive). Les "bundles" sont les seules entités utilisées pour le déploiement d'applications. Les appareils OSGi compatibles peuvent télécharger, installer, supprimer les "bundles". L'installation et la mise à jour se font de manière dynamique, en gérant les dépendances entre les "bundles" et les services. Les limitations de ce modèle sont liées au fait qu'il est spécifique à l'environnement Java et à des applications non distribuées.

Quant à la spécification de l'OMG, elle a pour objectif de fournir un modèle de données et d'exécution permettant de gérer le développement, le packaging, le déploiement et la configuration d'applications à base de composants. La spécification est décrite à travers une entité appelée "Platform-Independent Model" (PIM), composée d'un ensemble de modèles UML[1] et de règles sémantiques associées. Le PIM est indépendant de tout modèle de composant particulier. Pour utiliser cette spécification avec un modèle particulier de composant, il faut créer une entité appelée "Platform-Specific Mapping" (PSM). Le PSM est un ensemble de règles qui transforme les modèles UML du PIM en données et modèles d'exécution, dans un format approprié pour le déploiement du modèle de composant cible. La spécification n'a pour l'instant standardisée que le PSM pour le modèle de composant corba[2], dans lequel les modèles de données et d'exécution sont transformés en deux formats : XML schema pour le stockage sur disque et l'échange entre outils, et IDL (Interface Definition Language) pour la représentation du modèle d'exécution et des communications entre les entités du déploiement.

Des outils de déploiement [22, 23, 97–99] et de gestion de configuration comme Chef [24], Puppet [25], Ansible [100], TakTuk [98] permettent un certain niveau d'automatisation du processus de déploiement. Les outils de configuration récents offrent la possibilité de "programmer" la manière dont les ressources seront configurées. Une description de l'état désiré des ressources considérées est donnée (un modèle, au format YAML pour Ansible par exemple), et ces outils transforment le modèle en un ensemble de commandes dont l'exécution permettra d'avoir l'état désiré.

Une étude des outils et techniques de déploiement d'applications a été faite dans [101] en se basant sur une division du processus de déploiement en dix (10) étapes basées en partie sur les étapes discutées dans [17, 19].

Parmi les différentes approches de déploiement citées, le déploiement dirigé par la

[1] Unified Modeling Language

[2] OMG : CORBA Component Model, v4.0 http://www.omg.org/spec/CCM/4.0/ 2016

qualité de service, dans lequel une application est déployée et éventuellement redéployée pour atteindre une certaine qualité de service comme la tolérance aux pannes. Une autre des approches de déploiement utilisée pour l'étude est le "déploiement à chaud" (hot deployment) concernant les techniques permettant à une application en cours d'exécution de s'adapter. Cependant, les exemples cités sont des cadres pour concevoir des applications capables de s'adapter en cours d'exécution et pas des applications existantes.

Une étude plus récente [102] se base sur les critères suivants : l'unité de déploiement, le domaine de déploiement, l'expression des propriétés, l'expertise du concepteur du déploiement, les activités de déploiement, le contrôle du déploiement et la nature du bootstrap. Le bootstrap signifiant ici un programme d'amorce qui doit être opérationnel sur les appareils cibles avant le déploiement.

Nous avons supposé l'existence de tels outils (l'intergiciel DIET dispose d'un outil de ce genre, GoDIET [22]) et dans cette thèse nous n'avons pas cherché à réaliser des outils de déploiement. Dans la partie dans laquelle nous simulons l'algorithme distribué que nous avons proposé, le déploiement est simulé par la création de hiérarchie des processus (une hiérarchie DIET) sur un ensemble de machines virtuelles interconnectées. Cette opération aurait nécessité un outil de déploiement dans un environnement réel. On s'intéresse à la phase où l'application est déjà déployée (on suppose avec des outils d'automatisation existants), et à comment gérer la détection d'un état de l'application qui n'est plus conforme à sa spécification?

TUNe [103], une évolution de **Jade** [104], propose un cadre (framework) pour encapsuler des logiciels patrimoniaux (développés dans/avec un paradigme autre que celui dans lequel on cherche à les utiliser) dans un modèle de composant FRACTAL [15], afin de leur assurer une gestion autonome. Le cadre propose un langage de description d'architecture permettant de décrire l'application à déployer, la possibilité d'implanter des politiques de reconfiguration de l'application à base de composant obtenue.

Ce type d'approche, pour rendre autonome l'administration de logiciels, est aussi proposé par **Rainbow** [105], un des premiers cadres destinés à la conception d'applications autonomes basées sur le modèle de système autonome introduit par IBM (MAPE-K) [29]. **Rainbow** utilise un modèle pour surveiller les propriétés d'un système en exécution, évalue le modèle pour détecter des violations de contraintes et si nécessaire applique des actions d'adaptation sur le système en exécution. **Rainbow** doit être utilisé pendant la phase de conception et développement de l'application et n'est donc pas destiné aux applications patrimoniales.

CASA (Contract-based Adaptive Software Architecture) [106] est un cadre qui permet l'adaptation dynamique des applications. Il inclut différents mécanismes d'adaptation dont la recomposition dynamique des composants d'une application.

Ils existent d'autres cadres basés sur d'autres approches telles que Model@run.time [107]. C'est une approche de génie logiciel dont l'objectif est de proposer des outils et méthodologies adaptés à la conception de logiciels complexes, en se basant sur des modèles. Ces modèles permettent de décrire les fonctionnalités avec un haut niveau d'abstraction et d'utiliser des outils de transformation de modèles pour obtenir des implémentations (en partie ou entièrement) des logiciels dans des langages spécifiques. L'utilisation de cette approche pour le développement de systèmes auto-adaptatifs [108–110] considère les modèles comme les éléments de base à partir desquels les applications seront modifiées en temps réel. Dans cette approche, le système en exécution est représenté sous forme d'un modèle (une réification du système réel) et des actions peuvent être appliquées au niveau du modèle (pour créer des adaptations par exemple), ce qui va se répercuter sur le système

réel car il y'a une relation causale entre le système réel et le modèle. Cette connexion crée une synchronisation entre le modèle et le système en exécution [111].

ACTRESS [112] est un cadre basé sur l'ingénierie des modèles et qui fournit des outils pour concevoir et intégrer des mécanismes d'adaptation dans une application, sous la forme de boucle de contrôle.

EUREMA (ExecUtable RuntimE MegAmodels) [113] est aussi une approche basée sur l'ingénierie des modèles. **EUREMA** propose un langage de modélisation et un environnement permettant la spécification et l'exécution de mécanismes d'adaptation constitués d'une ou de plusieurs boucles de contrôle. L'approche repose sur une architecture en couches, dans laquelle le système auto-adaptatif est séparé logiquement en deux parties : la couche métier qui fournit les fonctionnalités, et au dessus, la couche qui gère le mécanisme d'adaptation (de la couche métier), sous la forme de boucles de contrôle.

Tous ces cadres sont destinés au développement, à la conception de systèmes auto-adaptatifs.

Le système que nous simulons repose sur un modèle de composant CORBA [55] et est patrimonial. Cependant notre approche a des points communs avec ce qui est proposé dans **TUNe** : nous avons proposé un formalisme pour décrire un intergiciel hiérarchique à base de composants en ne faisant pas d'hypothèses sur le modèle de composant. De la même manière, nous avons défini des politiques de reconfiguration qui sont spécifiques à l'intergiciel cible.

De même, pour simuler les politiques d'auto-adaptation que nous avons proposées, on utilise une solution proche des Model@run.time car le système réel en exécution (les processus) est abstrait sous la forme d'une structure de donnée (un graphe), et il y' a un lien causal entre cette structure de donnée, qu'on utilise pour raisonner et prendre des décisions, et le système réel.

L'approche décrite dans [114] utilise une structure de graphe pour modéliser de manière formelle différentes entités qui interviennent dans le processus de déploiement comme l'application à base de composant qui doit être déployée, l'infrastructure cible sur lequel l'application sera déployée, la recherche d'un plan de déploiement dirigée par une qualité de service. Les algorithmes présentés sont assez général et peuvent être adaptés à d'autres types de qualité de service. Des étapes du déploiement sont ramenés, après la formalisation, à des problèmes de la théorie des graphes (comme la recherche d'un arbre couvrant minimum). L' aspect adaptation n'a cependant pas été pris en compte, mais plutôt la préparation d'un déploiement initial. Nous avons aussi utilisé une structure de graphe pour représenter une partie des entités du déploiement de manière moins formelle mais plus focalisée sur l'aspect adaptation.

Disnix [8] est un outil de déploiement automatique avec une approche qui partage des points communs avec notre travail. C'est le cas dans l'utilisation d'une description de l'infrastructure cible, de l'application à déployer. Cependant, **Disnix** reste l'outil qui fait le déploiement et non une application à déployer. Or, dans notre travail, l'outil de déploiement est important car c'est lui qui lance le déploiement, qui participe aux tâches de ré-déploiement, mais l'accent est surtout mis sur l'application déployée et sa manière de réagir aux événements de son environnement d'exécution.

SHMF (Scalable Hierarchical Management Framework) [6] est une approche hiérarchique de gestion d'une hiérarchie de type arbre, dans laquelle des contraintes sont fixées, comme une limite pour le nombre de fils d'un nœud par exemple. Le point commun avec notre travail est qu'ici, c'est l'application en exécution qui gère l'auto-adaptation comme c'est le cas pour nous où se sont les instances formant la hiérarchie elles mêmes qui gèrent

l'adaptation.

ADAGE [115] permet de déployer des applications décrites en utilisant un formalisme générique (GADe) sur une grille. Le formalisme est indépendant des technologies de composant. Le déploiement réalisé est statique, dans la mesure où une fois réalisé, de futures modifications ne sont plus possibles. Cependant, utilisé avec CoRDAGe [116], il permet un déploiement dynamique.

LE-DAnCE (Locality-Enabled Deployment and Configuration Engine) [117] est un outil de déploiement pour applications distribuées et hétérogènes. L'outil lui même s'adapte face aux variations de l'environnement sur lequel il déploie des applications, et aussi en fonction des contraintes de l'application qui est déployée. En plus, il implémente la spécification D&C de l'OMG [18].

L'approche présentée dans [97] est comparable avec ce que nous avons simulé à la différence que notre objectif n'est pas quantitatif mais qualitatif. L'architecture en trois couches : une couche décrit les processus et le matériel sur lequel ils s'exécutent, avec des sondes qui surveillent des paramètres et reportent les données à une couche au dessus qui crée une image du système en exécution. Cette couche correspond dans notre cas au serveur de déploiement, c'est une couche de réification du systèmes réel. Une troisième couche de haut niveau pour décrire les préférences des utilisateurs, ce qui correspond dans notre cas à l'obtention d'un déploiement stable.

2.2 Description d'architecture logicielle

Comment décrire la structure d'une application à base de composant, les interactions et les dépendances entre ses composants. Des langages de description d'architecture permettent de saisir ces relations. Nous avons proposé une grammaire, sous forme d'un schéma XML, qui permet de décrire l'architecture d'un intergiciel hiérarchique du type de DIET [5].

OVF (Open Virtual Format) [118] est une spécification permettant de décrire la structure d'un déploiement de machines virtuelles.

DADL (Distributed Application Description Language) [119] est un moyen de décrire des applications distribuées. Son infrastructure cible est le cloud.

Fractal Deployment Framework (FDF) [99] intègre un langage de description de déploiement qui permet de décrire les relations entre composants constituant une application répartie. Certains parmi les outils cités dans la section 2.1 disposent de ce moyen de décrire la structure et des contraintes sur les applications qu'ils vont déployer : **TUNe** [103], **Disnix** [8], **ADAGE** [115] en disposent chacun.

DELADAS (DEclarative LAnguage for Describing Autonomic Systems) [120], MuScADeL [20], j-ASD [121] permettent aussi de décrire la structure des différentes entités qui forment un système logiciel.

2.3 Description d'infrastructure distribuée

Nous avons proposé une description d'une infrastructure distribuée comme grid'5000[3] ou un cloud ou bien juste un cluster sur lequel on peut déployer une application. Parmi les travaux de ce sous domaine, hwloc [122], qui en plus de la description de la structure

[3] https://www.grid5000.fr/

jusqu'à des niveaux de détails très fins (cache, mémoire, core), fournit une API pour faire des requêtes sur les données de la ressource.

ADAGE [115] et **Disnix** [8] proposent des moyens de décrire l'infrastructure sur laquelle les applications doivent être déployées. En général, cette description est utilisée pour calculer un plan de déploiement initial.

2.4 Algorithmes auto-stabilisants

Nous avons proposé un algorithme auto-stabilisant dont l'objectif est qualitatif, à savoir : assurer l'existence d'un déploiement "toujours vivant" (cf. chapitre 4) et maintenir le déploiement existant dans un état stable. A chaque instant, un déploiement existe (sauf le cas extrême où on tuerait tous les processus simultanément), soit dans un état stable ou instable. Et si le déploiement est dans un état instable, il retrouvera un état stable après un nombre fini d'étapes, suite à l'exécution de l'algorithme.

Pour atteindre cet objectif, l'algorithme réagit à la détection d'un état instable (dû aux pannes transitoires) pour retrouver un état correct, état dans lequel le déploiement peut exécuter les tâches correspondant à sa spécification. Pour cela, l'algorithme cherche à maintenir une topologie de graphe correspondant à un état correct du déploiement. L'algorithme est basé sur un modèle de passage de messages, asynchrone, non-uniforme (les nœuds de même type exécutent le même code, mais il y a plusieurs types de nœuds).

L'auto-stabilisation est une propriété de certains algorithmes qui leur assure une tolérance à des pannes transitoires. Depuis son introduction par E. W. Dijkstra [36] en 1974, un grand nombre de travaux ont été réalisés, dans divers contextes et avec diverses hypothèses [123–134].

Les systèmes sur lesquels on utilise des algorithmes auto-stabilisants sont généralement dynamiques. Ces systèmes sont sujets à des pannes transitoires. Ces pannes sont liées, entre autres, aux fréquents changements de la topologie, qui sont une conséquence de la possibilité qu'ont les processus de quitter ou de rejoindre de tels systèmes. Dans cette situation on cherche à créer ou maintenir une topologie pour assurer au système certaines propriétés. Dans notre cas, on cherche à maintenir une topologie de type graphe qui est une condition nécessaire (mais pas suffisante) de la stabilité du système. Ainsi, dans certains travaux, on cherche à construire et/ou à maintenir une topologie de type arbre [123], de type graphe [124, 125, 132]. Une revue de la littérature sur les algorithmes auto-stabilisants de construction d'un arbre couvrant est disponible dans [94, 135, 136].

Des algorithmes auto-stabilisants sont aussi proposés pour le partitionnement en clusters [128–130, 133].

2.4.1 Algorithmes auto-stabilisants à vagues

Des algorithmes auto-stabilisants, à vagues, de type propagation d'information avec retour ou PIF (Propagation of Information with Feedback) ont été proposés. Une version instantanément stabilisante du PIF est proposée dans [137].

Certains travaux supposent la construction ou l'existence d'un arbre couvrant pour résoudre des problèmes du type synchronisation [138], élection de leader [139], ré-initialisation après faute [140]. L'algorithme décrit dans [141] est aussi de type PIF mais pour des réseaux quelconques et sans l'hypothèse d'un arbre couvrant. Un algorithme auto-stabilisant, à vagues, avec circulation de jeton est décrit dans [131].

CHAPITRE 3

Déploiement initial

Sommaire

3.1 Introduction

Nous avons présenté l'architecture de l'intergiciel DIET (cf. Chapitre 1, Section 1.1.1) sur lequel on a appliqué notre travail. Cet intergiciel, qui fournit des services de calcul haute-performance, doit d'abord être déployé sur une infrastructure cible. En plus, une fois déployé, on souhaite qu'il puisse réagir de manière autonome, lorsqu'il se trouve dans un état instable (un déploiement dans cet état est considéré comme non efficace), pour retrouver un état stable (état défini au Chapitre 4, Section 4.3). On a donc deux problématiques :

- réaliser un déploiement initial;
- gérer l'adaptation du déploiement obtenu.

Nous allons, dans la suite de ce chapitre, présenté l'architecture générale proposée. Ensuite, nous présenterons nos contributions pour la première problématique, à savoir la réalisation d'un déploiement initial. Les contributions pour la deuxième problématique (l'adaptation d'un déploiement en cours) seront présentées dans les deux chapitres suivants.

3.2 Architecture proposée

Dans cette section, nous décrivons l'architecture proposée pour le déploiement auto-adaptatif d'une application à base de composants (en l'occurrence l'intergiciel DIET). Cette architecture est une adaptation du modèle de boucle de contrôle **MAPE-K** défini dans [29]. Nous commentons les différents modules qui sont numérotés, comme indiqué sur la Figure 3.1.

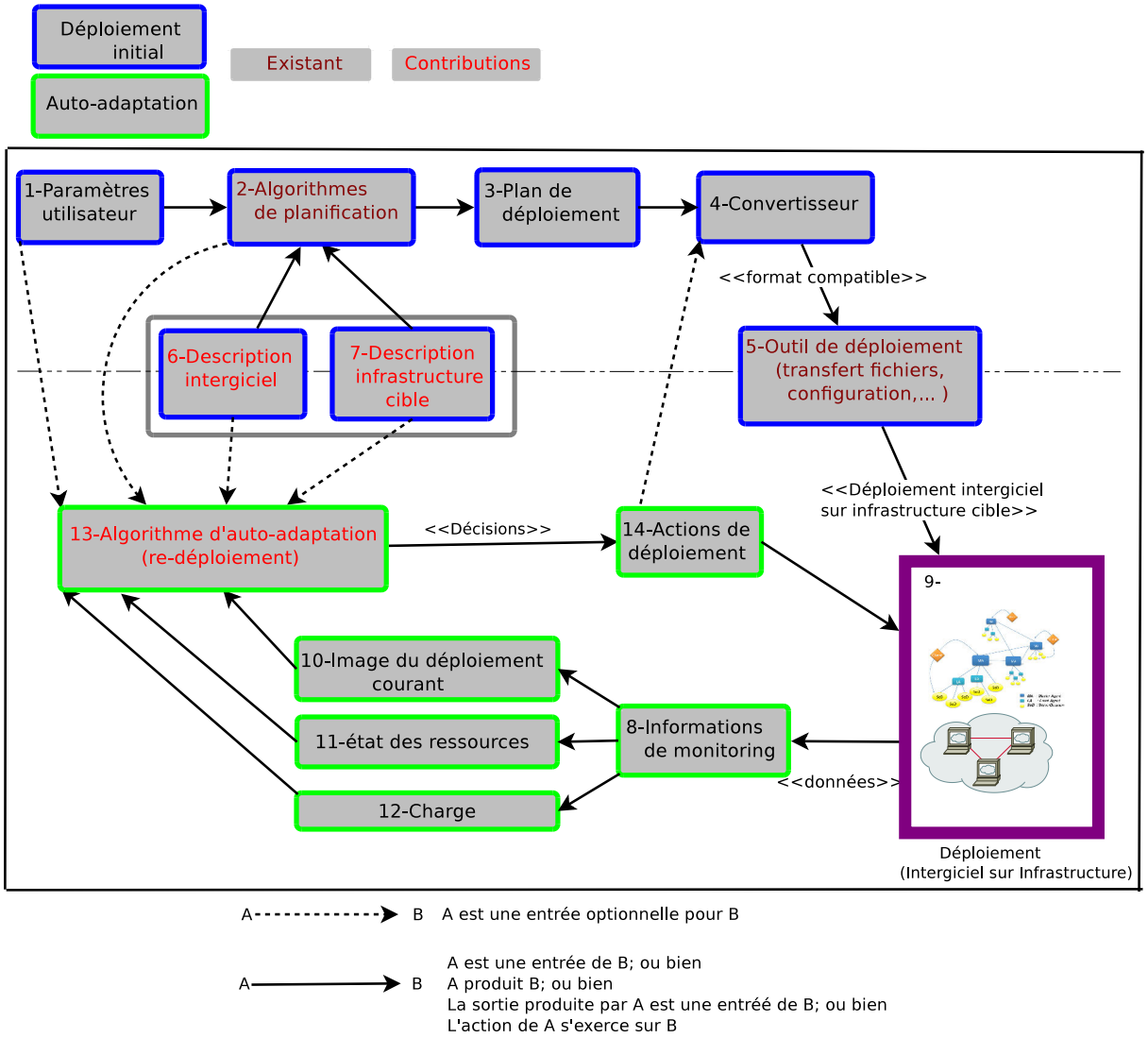

Figure 3.1: Architecture pour le déploiement auto-adaptatif d'intergiciel

Algorithmes de planification (2) : la fonction d'un algorithme de planification consiste à répartir les composants de l'intergiciel sur les ressources de la plate-forme qui satisfont leurs besoins (si possible) et de sorte que les objectifs prédéfinis par l'utilisateur soient atteints. Pour ce faire, les algorithmes ont besoin de connaître :

- **les paramètres de l'utilisateur (1)** qui expriment les préférences de l'utilisateur (contraintes sur le déploiement, qualité de service,...);
- **une description de l'intergiciel (6)** qui doit être déployé;

- **une description de l'infrastructure (7)** sur laquelle l'intergiciel sera déployé. La description concerne les ressources et leurs relations.

La sortie des **Algorithmes de planification (2)** est un **plan de déploiement (3)**, exprimé dans un format donné (en XML par exemple). Il précise pour chaque instance (d'un composant de base de l'intergiciel) qui sera déployée, les ressources qui lui sont allouées.

Le convertisseur (4) : ce module convertit le fichier de déploiement (exprimé dans un format générique) en un fichier au format compris par l'outil de déploiement particulier utilisé (5). Il faut prendre de (3) les informations pertinentes et créer l'entrée de l'outil de déploiement (5) qui exécute les opérations de bas niveau du processus de déploiement [17, 18] comme le transfert de fichiers, la configuration des ressources ciblées, l'activation des processus, etc. Après les actions de (5), nous obtenons une hiérarchie d'instances de composants de l'intergiciel , en cours d'exécution sur les ressources de l'infrastructure physique qui leur ont été allouées par les algorithmes de planification.

À partir de ce moment, nous avons un déploiement initial, avec un intergiciel qui est disponible à l'utilisation.

Informations de surveillance (8) : elles sont recueillies à travers des sondes et concernent aussi bien l'état des processus que des ressources physiques sur lesquelles s'exécutent les processus.

À partir des informations recueillies (8), on peut analyser l'état du déploiement, créer une image du déploiement courant (représentation formelle du déploiement courant sous la forme d'un graphe par exemple) (10), connaître l'état des ressources physiques (11), leurs charges (12).

Ces informations, déduites des données issues du monitoring du système déployé, permettent d'évaluer si le déploiement est stable ou instable. Si le déploiement est instable, l'**algorithme d'auto-adaptation** (13), dont l'objectif est d'amener un déploiement instable vers un état stable, s'exécute. Cette exécution peut comporter des actions qui fassent appel aux outils de déploiement et d'autres actions exécutées directement par les processus.

L'algorithme d'auto-adaptation peut utiliser (ou non) la totalité ou une partie des algorithmes de planification (2). La sortie l'algorithme d'auto-adaptation est constituée des actions de re-déploiement (14). Certaines de ces actions, pour être exécutées, peuvent nécessiter un appel aux outils de déploiement. Elles sont fournies dans le format d'un fichier de déploiement générique et peuvent avoir besoin d'être traduites par un convertisseur pour un outil de déploiement particulier. Les actions qui n'ont pas besoin d'un outil particulier seront exécutées directement.

Cette phase d'analyse de l'état du déploiement et d'exécution de l'algorithme d'auto-adaptation est effectuée par une entité centralisée qui a une vision globale du déploiement dans le cadre d'un algorithme centralisé. Dans le cadre d'un algorithme distribué, cette phase est effectuée par chaque instance de composant déployée (un processus), en fonction de son état interne qui est mis à jour grâce aux informations de surveillance et des échanges de messages avec les autres instances (ses voisins dans la hiérarchie déployée).

Dans la suite de ce chapitre, nous allons décrire les modules (6) et (7). La partie auto-adaptation sera décrite en détails dans les chapitres 4 et 5.

3.2.1 Travaux antérieurs

Plusieurs travaux liés à l'intergiciel DIET ont été réalisés. Parmi eux, nous résumons ceux qui ont un lien avec l'architecture proposée.

Des algorithmes de planning ont été proposés dans [142] pour trouver un déploiement optimal de DIET sur un environnement homogène (les nœuds de calcul et les liens ont les mêmes caractéristiques : puissance de calcul et bande passante); ou bien pour trouver la meilleure hiérarchie dans un environnement hétérogène (le problème du planning étant NP-complet dans ce cas [143]).

Des heuristiques sont proposées dans [144], qui permettent de déterminer, de manière automatique, un plan de déploiement de DIET satisfaisant à un critère fixé (maximiser le nombre de tâches exécutées par unité de temps, lorsque plusieurs applications s'exécutent simultanément par l'intermédiaire de l'intergiciel). Ces heuristiques sont proposées en fonction des hypothèses faites sur la nature des infrastructures cibles, qui peuvent être homogènes ou hétérogènes.

Ces travaux proposent ainsi des moyens d'obtenir un plan de déploiement initial pour l'intergiciel DIET.

GoDIET [22] est un outil de déploiement spécifique à DIET. Il prend en entrée un fichier XML décrivant le déploiement (mise en correspondance entre les composants de l'intergiciel et les ressources sur lesquelles ils seront instanciés). Cependant, d'autres outils de déploiement non spécifiques à DIET comme ADAGE [115] et TUNe [103] peuvent être utilisés (et ont été utilisés [144]) pour le déployer.

3.3 Contribution pour le déploiement initial

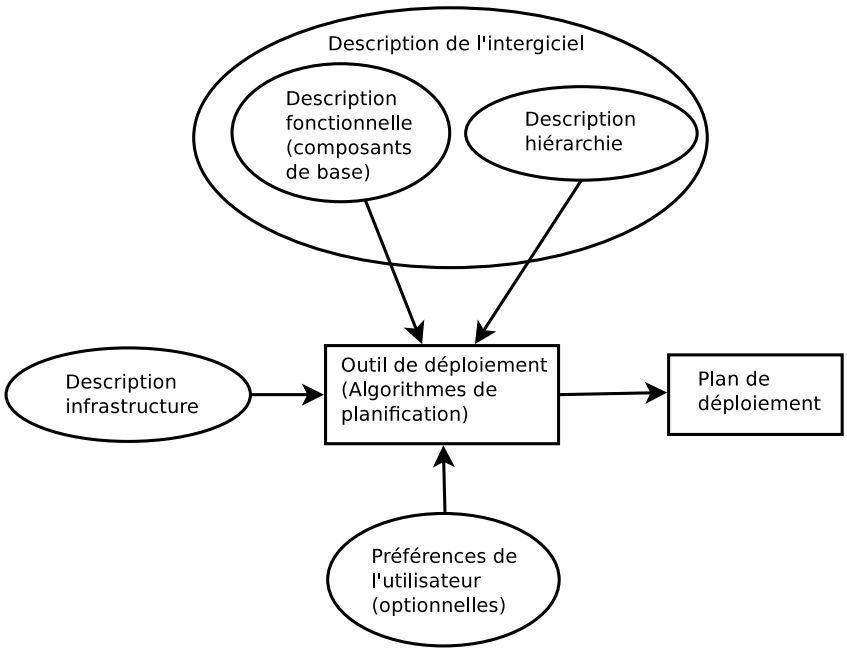

Figure 3.2: Déploiement initial de DIET

Les besoins pour un déploiement initial de DIET, de manière générale, sont résumés par la Figure 3.2. Les algorithmes de planification utilisent comme entrées les descriptions

de l'intergiciel et de l'infrastructure cible et éventuellement, des paramètres de l'utilisateur. Ils fournissent en sortie un plan de déploiement dans un format donné. Nous avons proposé une formalisme que nous voulons générique pour décrire l'intergiciel DIET et les infrastructures sur lesquelles il est susceptible d'être déployé.

Ces description sont nécessaires pour réaliser un déploiement initial; et correspondent aux modules (6) et (7) de l'architecture proposée. Nous présentons les descriptions proposées sous forme de diagrammes de classes UML modélisant les différentes entités.

3.3.1 Description de l'infrastructure

La description de l'infrastructure sur laquelle l'application sera déployée est une entrée des algorithmes de planification (Figure 3.3).

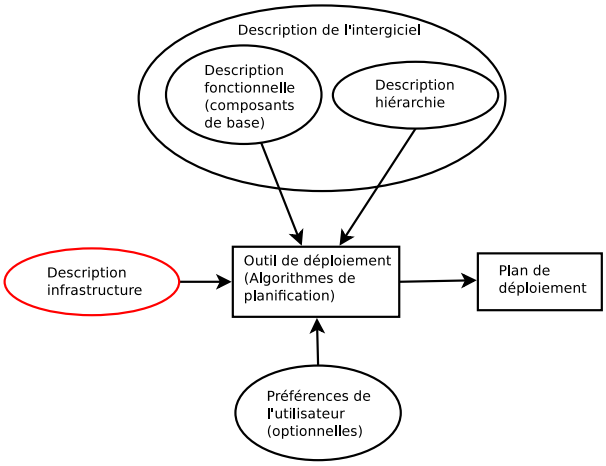

Figure 3.3: Description de l'infrastructure comme entrée pour créer un déploiement initial

Le type d'infrastructure visé est un système distribué tel qu'une grille, une fédération de clusters ou de Clouds. Notre but est de fournir un modèle qui puisse représenter ces différents types de systèmes où l'intergiciel peut être déployé, avec un accent sur une fédération de sites hébergeant des clusters (Figure 3.4) comme le cas de Grid'5000[1].

La Figure 3.5 représente le diagramme des classes modélisant une infrastructure. Il se compose de plusieurs éléments :

Plateform : cet élément représente la plate-forme (infrastructure). La plate-forme est composée d'un ensemble de ressources et de liens. Elle a un nom et une propriété "variation" ("dynamicity"), définie par les administrateurs de la plate-forme et dont le but est de capturer à quel point les paramètres considérés de la plate-forme sont variables (peu fréquente, fréquente, très fréquente). Cette valeur peut être calculée (ou estimée) en prenant en compte la "variation" de chaque ressource composant la plate-forme ou en analysant l'historique de la plate-forme (combien de fois une machine est tombée en panne dans un intervalle de temps donné par exemple). Dans le cas des stratégies auto-adaptatives de redéploiement, il est utile de pouvoir quantifier les variations de la plate-forme. En effet, certaines stratégies de redéploiement peuvent être efficaces avec une plate-forme très dynamique et l'être moins avec une plate-forme peu dynamique.

[1] https://www.grid5000.fr/

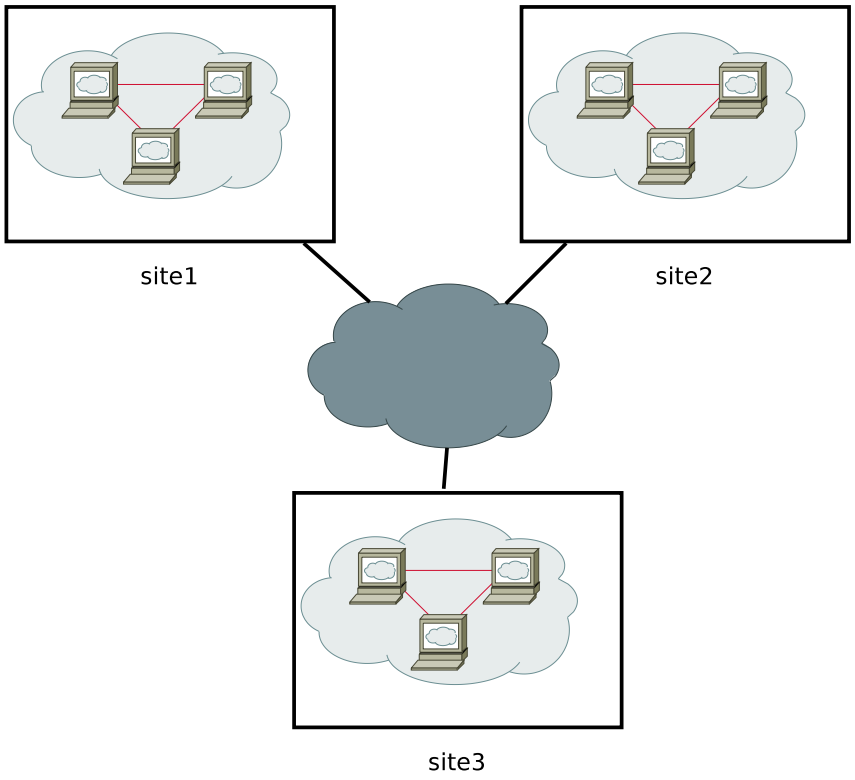

Figure 3.4: Exemple d'infrastructure

Ressource : les ressources peuvent être de différents types : Cluster, Nœud de Calcul (**Node**) et Site. Une ressource a un identifiant et dispose de zéro ou plusieurs localisations (élément **Location**). Chaque localisation est une pair (clef, valeur) qui spécifie un groupe auquel appartient la ressource. Chaque pair capture la notation d'appartenance/localisation d'une ressource. Par exemple, une ressource appartient à un site (site, nomSite), à un réseau local (reseauLocal, adresse IP) et à une ville (ville, nomVille). Certaines localisations peuvent être géographiques et d'autres liées au réseau. La localisation et l'appartenance d'une ressources à des groupes sont nécessaires pour certaines décisions de redéploiement, quand il est nécessaire de déplacer un composant ou des données mais aussi pour des raisons de sécurité. Une ressource est aussi décrite par un ensemble de liens (**Link**).

Link : cet élément décrit les liens de communications entre les ressources (nœud, cluster, site). Il est rattachée à deux ressources (endpoint1 et endpoint2). L'attribut **linklevel** spécifie la nature et le niveau dans la hiérarchie du lien. **linklevel** peut avoir les valeurs suivantes : "intraCluster", "intraSite", "interSite", "interCluster" et "interNode". Le lien est aussi composé d'un ensemble de **Capacité** comme la bande passante et la latence.

Capacity : une capacité est décrite par un nom (CPU, diskSpace, numberOfCore, etc.), une capacityUnit qui spécifie dans quelle unité la valeur de la capacité est exprimée, une capacityFlavor qui définit si la valeur de la capacité est représenté comme une valeur simple, un intervalle ou une liste et capacityValue qui contient la ou les valeurs.

Node : un nœud est défini comme un ensemble d'éléments **Capacity**. Il a un identifiant qui est unique au sein du cluster auquel il appartient.

Cluster : un Cluster est un ensemble de nœuds dont un (appelé frontEnd) a une fonc-

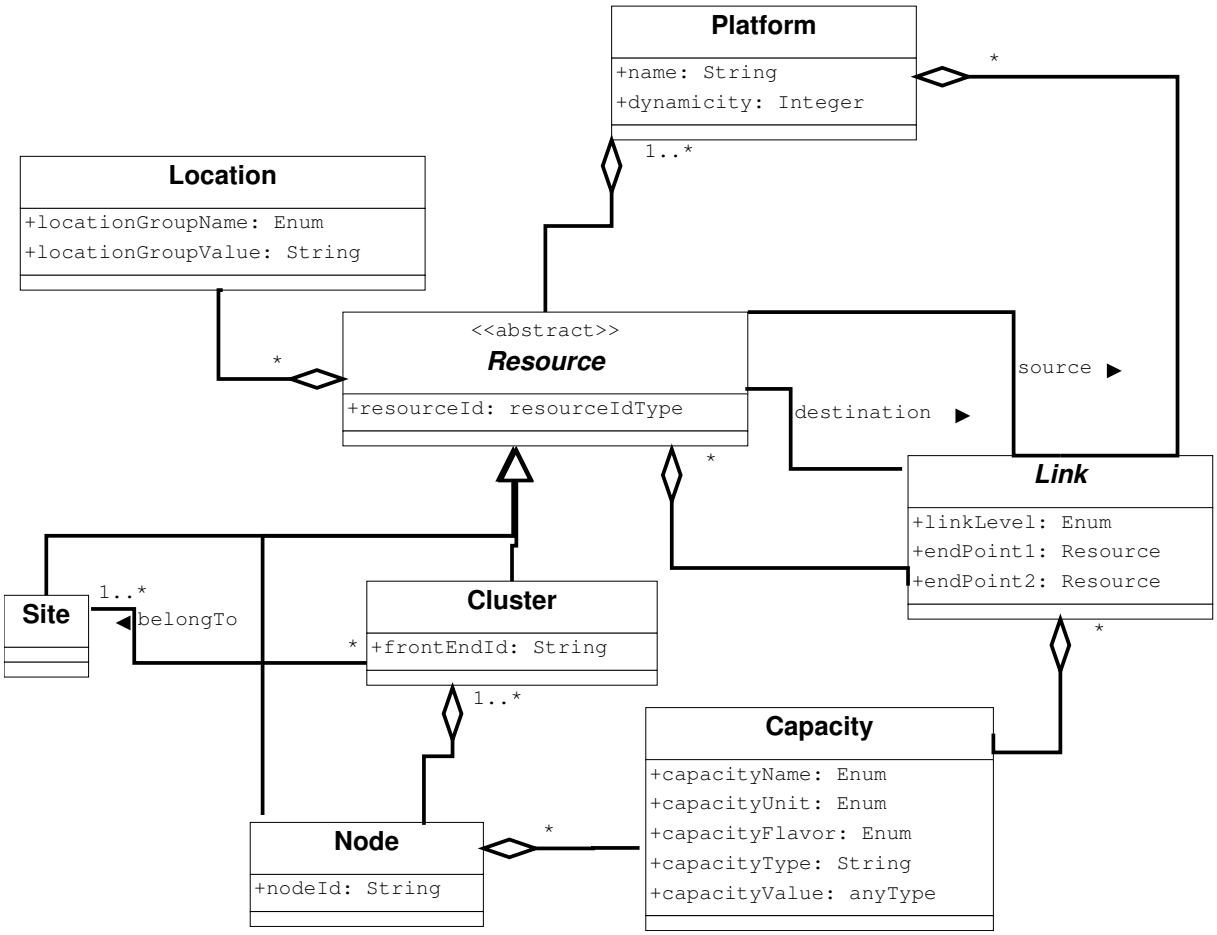

Figure 3.5: Modèle d'infrastructure

tion spéciale. Un cluster virtuel tournant dans un Cloud est représenté comme un cluster avec un ensemble de machines virtuelles (décrites comme des nœuds) et un ensemble de liens de communications. Si un cluster appartient à un ou plusieurs sites, un ou plusieurs éléments **Location** sont utilisés.

3.3.2 Description de l'intergiciel

Nous avons séparé la description de l'intergiciel en deux parties complémentaires : la description fonctionnelle qui capture les caractéristiques de l'intergiciel (les composants de base) qui sont fixées durant la phase de développement et la description d'une hiérarchie DIET qui capture les relations et les exigences entre les instances des composants de base. Certaines de ces relations et exigences sont fixes et d'autres peuvent varier d'un déploiement à un autre.

Description fonctionnelle de l'intergiciel

La Figure 3.8 représente la description fonctionnelle d'un intergiciel comme DIET, constitué d'un ensemble fini de composants de base, dont les fonctions et les possibilités de communication entre les instances de ces composants de base sont fixées à la conception.

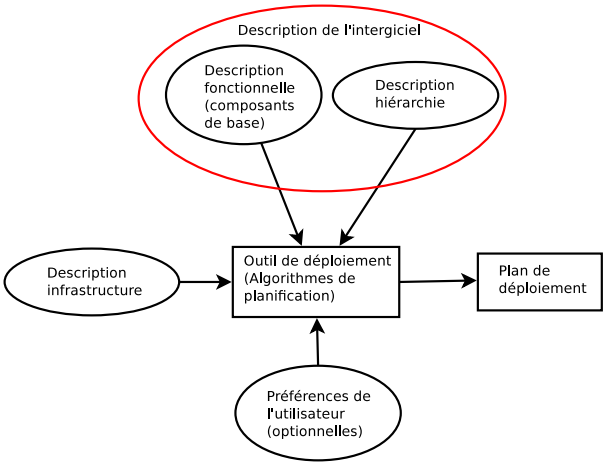

Figure 3.6: Description de l'intergiciel comme entrée pour créer un déploiement initial

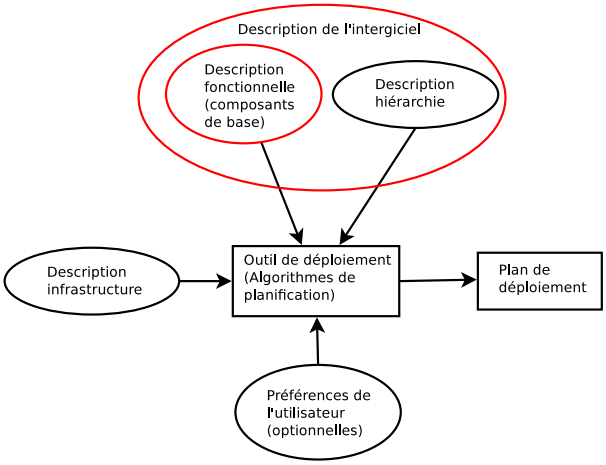

Figure 3.7: Description fonctionnelle de l'intergiciel comme entrée pour créer un déploiement initial

Cette description est composée de plusieurs éléments :

Middleware : il représente l'intergiciel et dispose d'attributs comme un nom et une version. Il est constitué d'un ensemble fini de composants de base. Ce sont des instances des composants de base qui s'exécutent effectivement sur les machines physiques ou virtuelles.

MiddBaseComponent : un composant de base de l'intergiciel contient un identifiant et contient les sections suivantes :

- **SoftwareInfo** : cet élément décrit les informations logicielles du composant qui sont fixées durant la conception comme les binaires;
- **CommunicationInfo** : décrit pour un composant de base, les autres composants avec qui les instances peuvent communiquer et comment est faite cette communication. La communication est décrite comme un ensemble d'exigences. Une exigence est exprimée comme une capacité (ce qu'elle exige doit faire partie des capacités d'une ressource). Un attribut spécifie si cette exigence est stricte (doit forcément être satisfaite) ou non (politique de "best effort");

- **SoftwareRequirement** : il représente les exigences logicielles d'un composant sur une ressource sur laquelle il peut être exécuté. Par exemple, un composant peut exiger un système d'exploitation spécifique. Il est composé d'un ensemble de capacités exigées;

- **HardwareRequirement** : comme pour le SoftwareRequirement, cette section exprime un ensemble de capacités exigées mais celles-ci sont matérielles. Par exemple, le composant peut exiger un type particulier d'architecture processeur ou une quantité minimum d'espace disque;

- **LocalityRequirement** : cette section décrit les exigences liées à la localisation du composant. Certains composant peuvent souhaiter partager la même localisation pour différentes raisons comme la rapidité des communications;

- **SecurityRequirement** : cette section définit le niveau de sécurité qu'exige le composant. Cette valeur (basse, moyenne, haute) est interprétée par l'algorithme de déploiement. Elle est spécifique à chaque intergiciel.

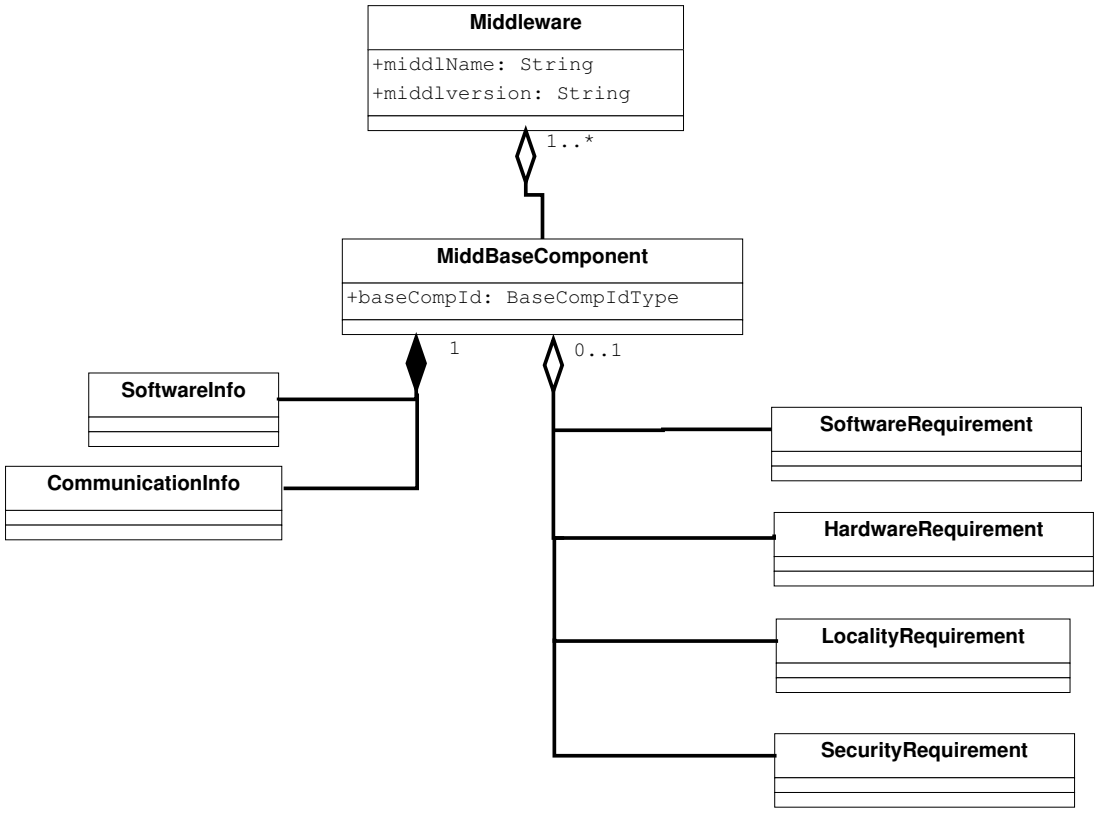

Figure 3.8: Modèle de middleware

Description d'une hiérarchie

Nous cherchons à capturer à travers cette description une hiérarchie d'instances des composants de base de l'intergiciel en exécution sur des ressources physiques. En d'autres

termes, on cherche à capturer la correspondance (mapping) entre les instances des composants de l'intergiciel et les ressources physiques (Cluster, Nœud,...) sur lesquelles elles sont en train de s'exécuter ou bien sur lesquelles elles pourraient être instanciées plus tard. Cette description peut capturer aussi une hiérarchie sans précision des ressources qui leur sont affectées.

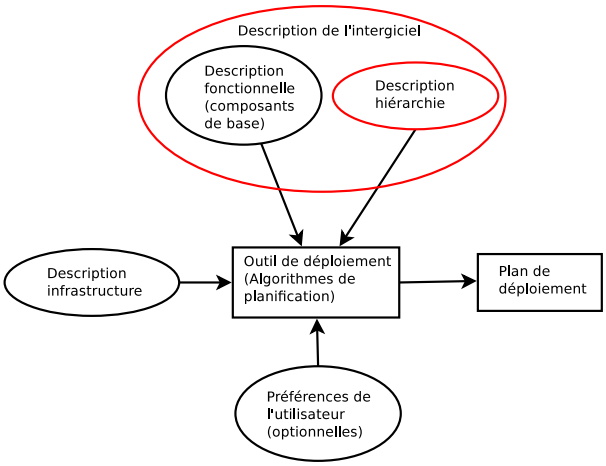

Figure 3.9: Description de la hiérarchie de l'intergiciel comme entrée optionnelle pour créer un déploiement initial

Cette description est optionnelle comme entrée pour les algorithmes de planification (dans ce cas, il y'aura uniquement la description fonctionnelle de l'intergiciel). Cette description peut être utilisée de plusieurs manières. En effet, un utilisateur peut fixer sa propre hiérarchie qu'il veut déployer en précisant les ressources sur lesquelles les instances seront déployées ou en ne les précisant pas. Dans le cas où les ressources matérielles sont précisées (cela suppose que l'utilisateur a une connaissance approfondie de son infrastructure physique), l'outil de déploiement ne cherchera pas un plan de déploiement (avec des algorithmes de planification), mais utilisera ses fonctionnalités et outils (de transfert de fichiers, de configuration,...) pour transformer la hiérarchie en déploiement en exécution (instancier les composants sur des ressources matérielles) telle que c'est décrit. Lorsqu'on connaît la hiérarchie à déployer et aussi les ressources sur lesquelles les instances seront déployées, le déploiement peut être fait de manière manuelle aussi, puisqu'on dispose de toutes les informations nécessaires; même si la manière automatisée est préférable (moins d'erreur dans l'exécution des commandes). Dans le cas où les ressources matérielles ne sont pas précisées, l'outil de déploiement (ses algorithmes de planification) devra trouver, si possible, les ressources permettant de déployer la hiérarchie décrite et ensuite réaliser le déploiement effectif (ses outils de transferts de fichiers, de configuration,...)

Cette description peut aussi servir à décrire un plan de déploiement (hiérarchie avec précision des ressources matérielles sur lesquelles les instances seront déployées). Cependant, tous les outils de déploiement n'utilisent pas le même formalisme/description pour les plans de déploiement; ce qui fait que la description proposée ici est assez spécifique à l'environnement de DIET.

La Figure 3.10 représente le modèle d'une hiérarchie de DIET. Il est décrit par les éléments suivants :

Hierarchy : la hiérarchie est un ensemble d'instances (**DeployedInstance**) de composants de base et de leurs liens de communication. Chaque instance contient les in-

formations et exigences nécessaires pour l'algorithme de planification et pour l'outil de déploiement.

DeployInstanceInfo : cet élément contient les informations relatives à l'instance. Il contient un ensemble d'exigences présentées ci-dessous :

- **LocalityRequirement** : il spécifie qu'une instance doit appartenir à une localisation (réseau ou géographique). Cette exigence peut être stricte ou flexible;
- **ColocationRequirement** : il spécifie si l'instance doit avoir la même localisation que d'autres instances (par exemple même site, même cluster, même Nœud, même réseau local, ...);
- **SecurityRequirement** : cet élément est utilisé pour spécifier l'exigence de sécurité de l'instance du composant;
- **LinkRequirement** : chaque instance déployée a un ensemble d'éléments **LinkTo** qui spécifient les exigences de communication. L'élément **LinkTo** décrit les connexions entre deux instances. Dans le cas d'un intergiciel hiérarchique, une instance est au moins connecté à une autre. Cet élément peut aussi contenir un ensemble d'exigences sur le lien de communication qui guidera l'algorithme de planification.

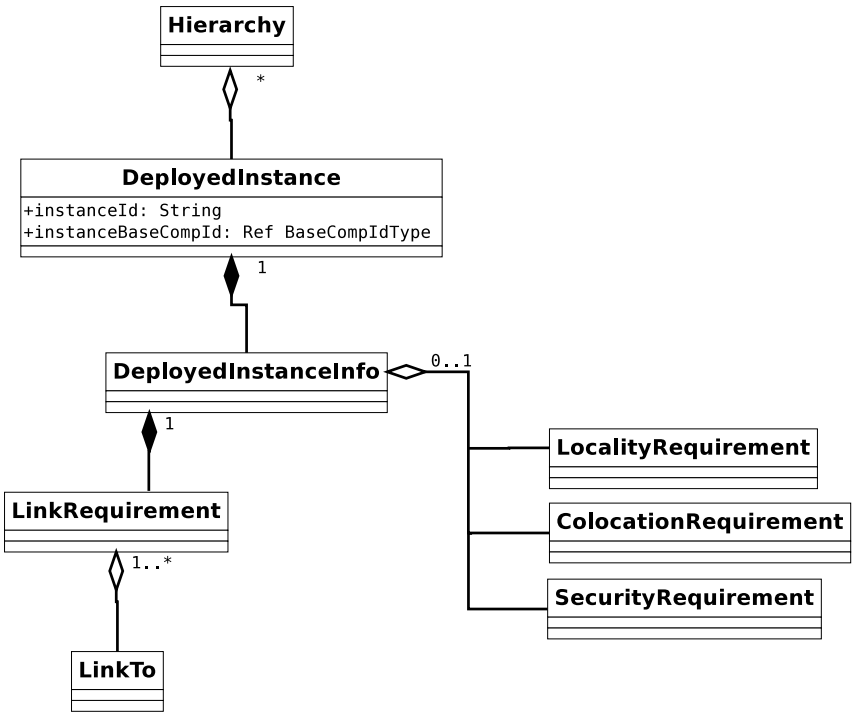

Figure 3.10: Modèle de déploiement

3.4 Conclusion

Notre objectif est de réaliser un déploiement auto-adaptatif d'un intergiciel. Pour cela, il faut d'abord un déploiement initial de l'intergiciel. Une fois un déploiement initial obtenu, les mécanismes d'auto-adaptation interviennent pour permettre au système déployé de réagir aux événements.

Pour réaliser un déploiement initial de l'intergiciel, des algorithmes de planning utilisent comme entrées, les descriptions de l'intergiciel à déployer, de l'infrastructure sur laquelle l'intergiciel sera déployé et éventuellement des paramètres de l'utilisateur. Ces descriptions peuvent aussi être utilisées par les mécanismes d'auto-adaptation, en plus d'informations issues de la surveillance du système déployé et de l'infrastructure, lorsqu'ils définissent les stratégies d'adaptation.

Dans ce chapitre, nous avons présenté les descriptions que nous proposons pour l'intergiciel et l'infrastructure, et qui peuvent servir comme information d'entrée à des algorithmes de planning, en vue de produire un plan de déploiement initial.

CHAPITRE 4

Algorithmes

Sommaire

4.1 Résumé du chapitre

Ce chapitre décrit un travail dont l'objectif est d'ajouter des capacités d'auto-adaptation à un intergiciel existant; lesquelles capacités devront permettre au déploiement de l'intergiciel de s'auto-adapter lorsque certains événements sont détectés. Le chapitre inclut la description d'un algorithme auto-adaptatif, de la preuve de son caractère auto-stabilisant.

4.2 Motivation

Les systèmes distribués améliorent notre capacité de calcul et à échanger des informations. Cependant, ils sont difficiles à concevoir, à contrôler, à maintenir car constitués d'une variété de composants (logiciels et physiques) complexes qui sont susceptibles de tomber en pannes ou de subir des variations de leur paramètres.

Dans certains cas, comme celui des Clouds, l'élasticité du nombre de ressources est une caractéristique majeure du système. L'accès aux ressources de ces systèmes se fait généralement à travers un intergiciel, et ce dernier doit d'abord être déployé.

Comme l'environnement évolue, que se passe t-il par exemple si une partie des processus qui constituent l'intergiciel cesse de fonctionner pour une raison quelconque ?

Si le déploiement est statique, alors le seul moyen de réagir aux événements dont les effets peuvent dégrader la qualité du service fourni est de refaire tout le processus de déploiement. Cette opération est cependant assez coûteuse.

Une meilleure solution consisterait à faire de sorte que le déploiement puisse s'auto-adapter et éviter autant que possible de reprendre tout le processus de déploiement. Il s'agit donc de concevoir un système auto-adaptatif [29, 31]. De tels systèmes ont la capacité de modifier en temps réel leurs comportements de manière autonome (totalement ou en partie) pour s'adapter aux variations de leurs environnements.

L'algorithme décrit dans ce chapitre a pour objectif, de permettre à un déploiement de l'intergiciel DIET, de retrouver un état légitime ou stable dans un nombre fini d'étapes, à chaque fois qu'un état instable est détecté.

4.3 Définitions et Notations

Avant de décrire l'algorithme, nous allons fournir un ensemble de définitions qui clarifient ce que signifie dans notre cas un nœud stable et un déploiement stable. Rappelons que ces définitions sont liées à l'intergiciel DIET (cf. chapitre 1, section 1.1.1) qui nous sert de cas d'utilisation.

Définition 11 (Nœud stable)**.** *Un nœud est stable si son état est légitime (correct). La signification exacte de ce qui est jugé légitime ou pas dépend de la nature du problème à résoudre (du code qui est exécuté). Dans la suite du document, un état stable ou légitime ou correct pour un nœud correspond à la situation où le nœud exécute autre chose qu'une des règles de l'algorithme; puisque l'exécution des actions associées à une règle a lieu à la suite de la détection d'un événement qui rend le nœud instable, donc le déploiement.*

Définition 12 (Déploiement stable)**.** *Un déploiement est une hiérarchie de nœuds interconnectés qui a une structure arborescente.*

Un déploiement (ou état) stable de l'intergiciel est un déploiement efficace qui a les caractéristiques suivantes :

- *il respecte les règles de hiérarchie des composants de l'intergiciel. Ces règles de hiérarchie imposent qu'un MA peut être le père d'un MA, d'un LA, d'un* SeD*; q'un LA peut être le père d'un LA, d'un* SeD*; qu'un* SeD *ne peut avoir de fils (c'est une feuille dans la structure arborescente de la hiérarchie); qu'un Client se connecte à un MA ou* SeD*;*
- *tous les éléments sont connectés entre eux (le déploiement a une structure de graphe et il y'a une seule composante connexe, pour un déploiement efficace);*
- *il n'y a pas de chaîne d'agents (pour des raisons d'efficacité);*
- *aucun agent n'est en surcharge (pour des raisons d'efficacité également). La surcharge est mesurée par rapport au nombre d'enfants de l'agent concerné (un seuil est fixé).*

Lorsqu'un déploiement satisfait à ces critères, alors chacun de ses nœuds est stable et l'état global du déploiement est aussi stable.

4.3.1 Modèle d'un déploiement

L'intergiciel, une fois déployé, peut être modélisé par un graphe non orienté $G = (V, E)$ où V désigne l'ensemble des processus et E l'ensemble des liens entre processus. Une arête $(u, v) \in E$ si et seulement si il existe un lien entre u et v. Dans ce modèle, l'existence d'un lien entre deux processus signifie que les deux processus sont voisins. Ce lien implique aussi que, si l'un des deux processus se termine (terminaison normale ou anormale), l'autre processus détectera cet événement. Les processus communiquent uniquement par échange de messages. Un processus peut envoyer un message à un autre s'il connaît son adresse. Chaque processus a un identifiant unique et conserve une liste des adresses de ses voisins qu'il met à jour en fonction des messages reçus et des tests effectués.

Les processus sont indépendants dans l'exécution des actions dès lors que les gardes sont vraies (*démon synchrone*). Autrement dit, l'exécution des actions au niveau d'un processus ne dépend pas d'un autre processus. Les règles sont définies pour chaque type de composant. L'algorithme n'est pas uniforme car tous les processus n'exécutent pas le même code.

4.4 Algorithme auto-adaptatif

Une fois qu'un intergiciel est déployé, des événements susceptibles de modifier les critères attendus peuvent survenir. Ces événements peuvent provenir de l'intergiciel lui même (arrêt d'un processus) ou de l'infrastructure sur laquelle l'intergiciel est déployé (panne d'une ressource physique, problème de réseau).

L'objectif de cet algorithme est de permettre à un intergiciel déployé de pouvoir s'auto-adapter lorsque certains événements sont détectés. Ce processus d'adaptation se fait par exécution de règles incorporées dans les composants. C'est cet ensemble de règles qui constituent l'algorithme.

Le type d'événements susceptibles de modifier le fonctionnement de l'intergiciel couvre un large spectre. Nous avons donc considéré uniquement un ensemble restreint d'événements auxquels les composants qui constituent l'intergiciel devront réagir. Ces événements sont essentiellement liés à l'intergiciel et correspondent à des situations dans lesquelles le déploiement n'est pas stable (Définition 12). Pour chaque type de composant de DIET, les événements gérés correspondent à la partie condition des règles qui sont définies pour ce composant.

Intuitivement, cet algorithme a pour objectif de maintenir un déploiement "toujours vivant" et stable.

4.4.1 Spécification de l'algorithme

L'algorithme est constitué des règles suivantes, regroupées en fonction du type de composant. Toutes les instances d'un composant exécutent le même programme (les règles définies pour ce type de composant). Chaque règle est constituée d'une partie condition qui exprime la détection d'un événement, et d'une partie action correspondant aux instructions d'auto-adaptation à exécuter lorsque l'événement est détecté. Dans certains cas, une instance a besoin, en plus de son état interne (ses variables locales) d'une information externe. Nous supposons donc l'existence d'un oracle capable de fournir ce type d'information et qui joue le rôle d'un service de découverte de ressources. On suppose

donc que la fonction découverte de ressources est assuré par un autre système extérieur à l'algorithme mais que ce dernier peut interroger.

Règles définies pour les instances de type Client

3 règles sont définies pour le composant client :

La règle R1 définit comment un client réagit lorsqu'il détecte la perte d'une connexion avec un MA et que le client a l'information qu'il existe au moins un autre MA dans le déploiement. Dans ce cas, le client se connecte à un autre MA, sélectionné de manière aléatoire, parmi ceux dont il a connaissance.

Client règle 1: R1

1 **if** $Client \wedge (MA\_lost == Vrai) \wedge (\#MA > 0)$ **then**
2 | sélectionner un MA et se connecter;
3 **end**

La règle R2 définit comment un client réagit lorsqu'il détecte la perte d'une connexion avec un MA et que le client a l'information qu'il n'existe plus de MA dans le déploiement. Dans ce cas, le client crée un fils MA.

Client règle 2: R2

1 **if** $Client \wedge (MA\_lost == Vrai) \wedge (\#MA == 0)$ **then**
2 | créer un fils MA ;
3 **end**

La règle R3 définit comment un client réagit lorsqu'il détecte la perte d'une connexion avec un SeD. Dans ce cas, il soumet sa requête de nouveau.

Client règle 3: R3

1 **if** $Client \wedge (SeD\_lost == Vrai)$ **then**
2 | soumettre de nouveau la requête ;
3 **end**

Règles définies pour les instances de type MA

5 règles sont définies pour le MA :

La règle R4 définit comment réagit un MA lorsqu'il détecte qu'il n'a pas de fils et qu'il a l'information qu'il existe au moins un autre MA que lui même dans le déploiement. Dans ce cas, il se suicide.

MA règle 4: R4

1 **if** $MA \wedge (\#MA\_children == 0) \wedge (\#MA > 1)$ **then**
2 | $\#MA = \#MA - 1$;
3 **end**

MA règle 5: R5

1 **if** $MA \wedge (\#MA\_children == 0) \wedge (\#MA == 1)$ **then**
2 | créer un fils de type SED;
3 **end**

La règle R5 définit comment réagit un MA lorsqu'il détecte qu'il n'a pas de fils et qu'il a l'information qu'il est l'unique MA du déploiement. Dans ce cas, il crée un fils de type SED.

La règle R6 définit comment réagit un MA réagit lorsqu'il détecte qu'il a un fils unique de type MA ou LA (une chaîne d'agents). Dans ce cas, il exécute la fonction $Fusionner(MA, MA\_child)$. La fonction $Fusionner(x, y)$ (ligne 2, R6) connecte les fils de y comme fils de x et supprime y.

MA règle 6: R6

1 **if** $MA \wedge (\#MA\_children == 1) \wedge (MA\_child\_type == (MA \vee LA))$ **then**
2 | $Fusionner(MA, MA\_child)$;
3 **end**

La figure 4.1 montre un exemple d'application de la règle MA R6.

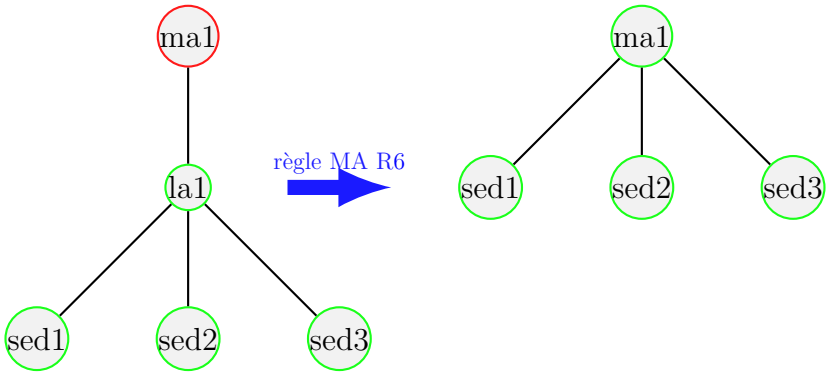

Figure 4.1: Exemple d'application de la règle MA R6 lorsque l'unique fils du MA est de type LA. Cela aurait été pareil si l'unique fils était un MA. En rouge, l'instance instable qui a détecté et exécuté la règle. En vert, les instances stables.

La règle R7 définit comment réagit un MA réagit lorsqu'il détecte qu'il est la racine d'un sous arbre du déploiement (cela signifie qu'il n'a pas de père de type MA même si un client peut être connecté sur lui ou pas) et qu'il a l'information qu'il existe au moins un autre sous-arbre qui a une racine de type MA et que les deux sous-arbres sont déconnectés.

Dans ce cas, le MA qui a détecté l'événement se connecte en tant que fils à l'un des MA, racine d'un des autres sous-arbres. Ainsi le nombre de sous-arbres sera réduit de un (1).

MA règle 7: R7

if $MA \wedge (\{\#père \ / \ TypeDuPère = MA\} == 0) \wedge (\{\#sous-arbre \ / \ TypeRacine = MA\} > 1)$ **then**
  sélectionner une des racines de type MA comme père;
end

La règle R8 définit comment réagit un MA lorsqu'il détecte qu'il est surchargé.

MA règle 8: R8

if $MA \wedge (MA\_charge \geq MA\_seuil\_charge)$ **then**
  Partitionner l'ensemble de ses fils en deux sous-ensembles A et B tels que : ;
  $\mid card(A) - card(B) \mid \leq 3$;
  créer un agent comme père de tous les éléments pour chaque sous-ensemble ;
  les racines (2 agents) des sous arbres nouvellement créés deviennent les fils du MA ;
end

La surcharge est simulée en fixant un seuil pour le nombre de fils que peut avoir une instance. On a surcharge lorsque le seuil est dépassé. Dans ce cas, le MA surchargé réduit sa charge (donc le nombre de ses fils) en créant deux agents qui deviendront ses deux seuls fils et en distribuant l'ancienne charge (ses anciens fils) entre ses deux nouveaux fils. Ainsi, après l'opération, ses anciens fils qui étaient à l'origine de la surcharge, se retrouvent comme ses petits fils et le MA n'a plus que deux (nouveaux) fils. Dans les détails, la réduction de la surcharge se fait de la manière suivante : rappelons nous que Le MA ne peut avoir que des fils des trois types suivants : MA, LA, SED. Soient $\mathcal{F}_{ma}$, $\mathcal{F}_{la}$, $\mathcal{F}_{sed}$, les trois ensembles qui désignent respectivement les fils du MA qui sont de type MA, ceux de type LA et enfin ceux de type SED. Chacun de ses ensembles peut être vide.

Chacun de ses ensembles est divisé en deux sous ensembles qui ont le même cardinal (si le cardinal de l'ensemble divisé est pair) ou bien dont la différence des cardinaux est égal à un (1) (si le cardinal de l'ensemble divisé est impair) :

$\mathcal{F}_{ma} = F1_{ma} \bigcup F2_{ma}$ avec $\mid card(F1_{ma}) - card(F2_{ma}) \mid \leq 1$.

$\mathcal{F}_{la} = F1_{la} \bigcup F2_{la}$ avec $\mid card(F1_{la}) - card(F2_{la}) \mid \leq 1$.

$\mathcal{F}_{sed} = F1_{sed} \bigcup F2_{sed}$ avec $\mid card(F1_{sed}) - card(F2_{sed}) \mid \leq 1$.

Le MA partitionne l'ensemble de ses fils en deux ensembles $\mathcal{A}$ et $\mathcal{B}$ obtenus de la manière suivante :

$\mathcal{A} = F1_{ma} \bigcup F1_{la} \bigcup F1_{sed}$

$\mathcal{B} = F2_{ma} \bigcup F2_{la} \bigcup F2_{sed}$

Ainsi les deux ensembles sont tels que : $\mid card(\mathcal{A}) - card(\mathcal{B}) \mid \leq 3$. Une fois ces deux ensembles créés, le MA surchargé va procéder comme suit : Il crée un agent comme père de tous les éléments de l'ensemble $\mathcal{A}$ (ses anciens fils) et fait la même chose pour les éléments de $\mathcal{B}$.

Les deux agents nouvellement créés deviennent les deux fils du MA et ses anciens fils deviennent ses petit-fils. L' agent nouvellement créé comme père des éléments dans $\mathcal{A}$ est de type LA s'il n'existe pas de MA dans $\mathcal{A}$ (un LA ne pouvant pas être père d'un MA), et de type MA sinon. Cette dernière remarque est valable aussi pour le type de l'agent créé pour les éléments dans $\mathcal{B}$.

En fin de compte, si tout le processus s'est bien déroulé, le MA qui était surchargé se retrouve avec uniquement deux fils. Ces deux fils (de type agent) peuvent être surchargés juste après leur création, et dans ce cas, ils exécuteront la même règle (si ce sont des MA) sinon la règle correspondante pour un LA.

Règles définies pour les instances de type LA

Six (6) règles sont définies pour le LA. Les règles définies pour le LA sont presque les mêmes que celles définies pour le MA puisque tous les deux sont des agents et jouent presque le même rôle. Le LA a six (6) règles au lieu de cinq comme pour le MA. Cette règle supplémentaire, R13, gère le cas où un LA détecte qu'il n'a pas de père et qu'il a l'information qu'il n'existe aucun MA dans le déploiement. Les autres règles du LA, à savoir, R9, R10, R11, R12, R14 peuvent être respectivement interprétées de la même manière que les règles suivantes du MA, R4, R5, R6, R7, R8 en remplaçant MA par LA et agent (qui peut être un MA ou un LA) par LA.

La règle R9 (similaire à MA R4) définit comment réagit un LA lorsqu'il détecte qu'il n'a pas de fils et qu'il a l'information qu'il existe au moins un autre LA que lui même dans le déploiement. Dans ce cas, il se suicide.

LA règle 9: R9

1 **if** $LA \wedge (\#LA\_children == 0) \wedge (\#LA > 1)$ **then**
2 $\quad \#LA = \#LA - 1$;
3 **end**

La règle R10 (similaire à MA R5) définit comment réagit un LA lorsqu'il détecte qu'il n'a pas de fils et qu'il a l'information qu'il est l'unique LA du déploiement. Dans ce cas, il crée un fils de type SED.

LA règle 10: R10

1 **if** $LA \wedge (\#LA\_children == 0) \wedge (\#LA == 1)$ **then**
2 $\quad$ créer un fils de type SED;
3 **end**

La règle R11 (similaire à MA R6) définit comment réagit un LA lorsqu'il détecte qu'il a un fils unique de type LA (une chaîne de LA). Dans ce cas, il exécute la fonction $Fusionner(LA, LA\_child)$. La fonction $Fusionner(x, y)$ (ligne 2, R11) connecte les fils de y comme fils de x et supprime y.

La règle R12 (similaire à MA R7) définit comment réagit un LA lorsqu'il détecte qu'il est la racine d'un sous arbre du déploiement (ce qui signifie qu'il n'a pas de père) et qu'il

LA règle 11: R11

1 **if** $LA \wedge (\#LA\_children == 1) \wedge (LA\_child\_type == LA)$ **then**
2 | $Fusionner(LA, LA\_child)$;
3 **end**

a l'information qu'il existe au moins un autre sous-arbre qui a une racine de type MA ou LA et que les deux sous-arbres sont déconnectés. Dans ce cas, le LA qui a détecté l'événement se connecte en tant que fils à l'un des agents (MA ou LA), racine d'un des autres sous-arbres. Ainsi le nombre de sous-arbres sera réduit de un (1).

LA règle 12: R12

1 **if** $LA \wedge (\#père == 0) \wedge (\#\{sous-arbre : TypeRacine = (LA \vee MA)\} > 1)$ **then**
2 | sélectionner une des racines (de type MA ou LA) comme père;
3 **end**

La règle R13 définit comment réagit un LA lorsqu'il détecte qu'il n'a pas de père et qu'il a l'information qu'il est l'unique agent dans le déploiement. Dans ce cas, il crée un MA comme père.

Cette règle est spécifique au LA, car la même situation pour un MA est normale et signifie juste qu'il n y a pas de client connecté. En d'autres termes, un MA peut ne pas avoir de père car pouvant être la racine d'une hiérarchie stable alors qu'un LA doit avoir un père car il ne peut être la racine d'une hiérarchie stable.

LA règle 13: R13

1 **if**
$LA \wedge (\#père == 0) \wedge (\#\{sous-arbre : TypeRacine = (LA \vee MA)\} == 1)$ **then**
2 | créer un MA comme père ;
3 **end**

La règle R14 (similaire à MA R8) définit comment réagit un LA lorsqu'il détecte qu'il est surchargé. La surcharge est simulée en fixant un seuil pour le nombre de fils que peut avoir une instance. On a surcharge lorsque le seuil est dépassé. Dans ce cas, le LA surchargé réduit le nombre de ses fils de la manière décrite au niveau de la règle MA R8 en prenant en compte le fait qu'un LA ne peut avoir que des fils de deux types : LA et SED, contrairement au MA qui peut en avoir de trois types. En plus, les agents nouvellement créés sont tous de type LA alors que pour le MA ils pouvaient être de type LA ou MA.

En fin de compte, si le processus s'est bien déroulé, le LA qui était surchargé se retrouve avec deux fils de type LA et ses anciens fils deviennent ses petits-fils.

LA règle 14: R14

1 **if** $LA \wedge (LA\_seuil \geq LA\_seuil\_charge)$ **then**
2 | Partitionner l'ensemble de ses fils en deux sous-ensembles A et B tels que : ;
3 | $\mid card(A) - card(B) \mid \leq 3$;
4 | créer un LA comme père de tous les éléments pour chaque sous-ensemble ;
5 | les racines (2 LA) des sous arbres nouvellement créés deviennent les fils du LA ;
6 **end**

Règles définies pour les instances de type SeD

Trois règles sont définies pour le SeD:

La règle R15 illustre la réaction d'un SeD qui n'est pas en train d'exécuter une tâche (job), qui n'a pas de père et qui a l'information qu'il n' y a pas d'agent dans le déploiement. Dans ce cas, il crée un MA comme père.

SeD règle 15: R15

1 **if** SeD $\wedge$ $(\#père == 0) \wedge (exécute\ tâche == Faux) \wedge (\#\{MA, LA\} == 0)$ **then**
2 | créer un MA comme père ;
3 **end**

La règle R16 illustre la réaction d'un SeD qui n'est pas en train d'exécuter une tâche (job), qui n'a pas de père et qui a l'information qu'il existe au moins un agent dans le déploiement. Dans ce cas, il sélectionne un des agents comme père.

SeD règle 16: R16

1 **if** SeD $\wedge$ $(\#père == 0) \wedge (exécute\ tâche == Faux) \wedge (\#\{MA, LA\} > 0)$ **then**
2 | sélectionner un des agents (MA ou LA) comme père ;
3 **end**

La règle R17 illustre la réaction d'un SeD qui est en train d'exécuter une tâche et qui n'a pas de père. Dans ce cas, il continue l'exécution pendant au maximum un temps fini T, fixé par l'utilisateur. T peut représenter le temps estimé pour exécuter une tâche.

Résumé des effets des règles

Les effets de chacune des règles sont résumés dans le Tableau 4.1.

SeD règle 17: R17

```
1 if SeD ∧ (#père == 0) ∧ (exécute tâche == Vrai) then
2 |   continuer l'exécution pour un temps maximum de T unités de temps ;
3 |   après quoi, l'exécution de la tâche courante est supposée être terminée ;
  |   /* T est un paramètre définit par l'utilisateur.                    */
  |   /* L'exécution d'une tâche est supposée être finie au maximum dans
  |      un temps T. Il n'y a donc pas de calcul infini                   */
4 end
```

Tableau 4.1: Effets des règles

Élément	Id règle	Effet de la règle
Client	R1	#sous-arbre - 1
	R2	#MA + 1
	R3	re-soumettre requête
MA	R4	#MA - 1
	R5	#SeD + 1
	R6	#Agent - 1
	R7	#sous-arbre - 1
	R8	#Agent + 2
LA	R9	#LA - 1
	R10	#SeD + 1
	R11	#LA - 1
	R12	#sous-arbre - 1
	R13	#MA + 1
	R14	#LA + 2
SeD	R15	#MA + 1
	R16	#sous-arbre - 1
	R17	exécution pendant T unités de temps au maximum

4.4.2 Preuve d'auto-stabilisation de l'algorithme

Le modèle de Pannes

Un système auto-stabilisant doit tolérer les pannes transitoires (des processus et des liens). Une panne transitoire peut corrompre les données en mémoire des processus (variables, pointeur de programme), les canaux de communication, mais sans corrompre le code qui est exécuté. Les pannes considérées sont celles qui peuvent induire une modification de la topologie du réseau (nouveaux nœuds qui rejoignent le réseau ou bien disparition de nœuds), une corruption des variables des processus (par exemple la liste des voisins). Pour rendre le réseau instable, nous n'allons ajouter que des nœuds isolés, c'est à dire sans voisin. Or ce type de nœud est toujours instable et rend le réseau instable car le graphe correspondant au réseau obtenu n'est plus connexe. Or, la connexité du graphe est une condition nécessaire pour que le déploiement soit stable (voir définition 12). Ainsi le nœud isolé cherchera à se stabiliser par l'exécution des règles (qui dépendent de son type). La disparition provoquée d'un ou plusieurs nœuds peut aussi rendre le réseau instable avec une différence par rapport à l'ajout de nœuds isolés : la disparition d'un ou

plusieurs nœuds ne mènent pas forcément à un réseau instable, tout dépend de leur type, de leur position dans la hiérarchie, etc.

Chaque processus teste régulièrement ses liens avec ses voisins. Lorsque le test échoue, cela est aussi considéré comme une modification de la topologie (disparition d'une arête du graphe modélisant un déploiement). Dans ce cas, le processus met à jour ses variables internes en supprimant de la liste de ses voisins l'identifiant du processus avec qui le test a échoué.

L'ajout des nouveaux nœuds (processus) isolés, la suppression de nœuds existants, et le test des liens avec ses voisins constituent les différentes actions qui modifient la topologie d'un déploiement, et ces actions peuvent rendre actives toutes les règles qui ont été définies.

Nous considérons qu'une panne est constituée d'une opération unique (par exemple le fait de tuer une instance en un seule opération) ou de plusieurs opérations uniques (par exemple tuer 100 instances en répétant 100 fois l'opération tuer une instance).

Nous considérons qu'il n'y a plus d'événements externes après l'exécution de la dernière opération de la panne (qui est considérée comme un événement externe). En d'autres termes, après l'exécution de la dernière opération de la panne, les seuls événements qui ont cours sont ceux prévus par le programme exécuté par les instances, notamment les règles d'auto-adaptation. Si un déploiement est perturbé avec une fréquence qui ne laisse pas à l'algorithme auto-adaptatif le temps de s'exécuter, le système sera constamment instable et la convergence de l'algorithme ne pourra pas être vérifiée.

Propriétés d'auto-stabilisation

Considérant l'algorithme distribué, spécifié sous la formes des règles définies à la Section 4.4.1, nous allons fournir une esquisse de preuve (sketch of proof), montrant que l'algorithme est auto-stabilisant (Définition 8); ce qui signifie dans ce contexte qu'un déploiement de DIET, dont les instances exécutent les règles définies précédemment, soumis à des pannes transitoires, retrouvera un état stable après un temps fini.

Pour cela, nous allons montrer les deux propriétés de **convergence** (Définition 9) et de **clôture** (Définition 10) de l'algorithme.

Preuve d'auto-stabilisation

Preuve de la propriété de convergence

Pour prouver qu'un déploiement sujet à des pannes transitoires va retrouver un état stable dans un temps fini, il suffit de prouver les propriétés suivantes :

- $\mathcal{P}_1$: le nombre de sous-arbres diminue;
- $\mathcal{P}_2$: la création de nouvelles instances se termine;
- $\mathcal{P}_3$: la suppression d'instances se termine;
- $\mathcal{P}_4$: l'exécution d'une tâche se termine.

preuve de $\mathcal{P}_1$:

- on peut constater qu'aucune des règles (cf. Tableau 4.1) n'a pour effet d'augmenter le nombre de sous-arbres;

- au même moment, les règles suivantes ont pour effet de diminuer le nombre de sous-arbres : Client R1, MA R7, LA R12, SED R16;
- ainsi, le nombre de sous-arbres est constant (dans ce cas il est égal à 1) ou diminue. □.

preuve de $\mathcal{P}_2$:

- l'exécution de chacune de ces règles conduit à la création d'une ou de deux instances : Client R2, MA [R5, R8], LA [R10, R13, R14] et SED R15;
- aucune de ces règles ne peut créer un sous-arbre déconnecté de celui qui contient l'instance exécutant la règle; elles peuvent juste ajouter une ou deux instances à un sous arbre existant;
- lorsqu'un agent (MA ou LA) n'a pas de fils, il crée un fils de type SED. De ce fait, une fois que MA R5 ou LA R10 est exécutée, la situation qui nécessite son exécution disparaît. L'exécution de ces règles ne créent pas de chaîne d'agents/ LA ni d'agents sans fils;
- lorsqu'un client perd la connexion avec un MA, il crée un MA en exécutant la règle Client R2 une fois. Le MA créé par cette règle va exécuter la règle MA R5 une fois;
- lorsqu'un agent est surchargé, il crée deux nouveaux agents en exécutant une fois soit la règle MA R8 (si c'est un MA), soit la règle LA R14 (si c'est un LA). Chacun des agents nouvellement créés a un père et au moins un fils. Par conséquent, si un agent nouvellement créé (par une des règles ci-dessus) n'est pas surchargé, il ne va exécuter aucune règle qui a pour effet une création d'instance. Si par contre un agent nouvellement créé est surchargé, il va exécuter lui aussi les règles ci-dessus et ne sera plus surchargé. Ainsi, de manière générale, après un nombre fini d'étapes, on aura des agents nouvellement créés par une des règles MA R8 ou LA R14 qui ne seront pas surchargés;
- lorsqu'un SED est isolé et qu'il n'y a pas d'agent dans le déploiement, il exécute la règle SED R15. Après cette opération, $\#\{Agent\} \geq 1$ et cette règle ne s'exécutera plus parce qu'il y a au moins un agent;
- On peut donc dire que la création de nouvelles instances se termine car à chaque fois, l'exécution d'une règle qui a pour effet une création d'instances élimine la situation qui avait nécessité cette exécution. □.

preuve de $\mathcal{P}_3$:

- l'exécution de chacune des règles suivantes a pour effet la suppression d'une instance : MA R4, MA R6, LA R9, LA R11, LA R12;
- un agent sans fils est supprimé (MA R4, LA R9) sauf s'il est l'unique agent du déploiement (MA R5, LA R10);
- chaque agent créé par MA R8 ou LA R13 ou LA R14 a au moins un fils. Par conséquent, pour chacun de ces agents, les règles MA R4 ou LA R9 ne seront pas exécutées;

- un MA créé par Client R2 ne sera pas supprimé mais va exécuter MA R5;
- toute chaîne d'agents est supprimé (MA R6, LA R11). Le nombre d'agents dans un déploiement est fini, et donc la suppression des chaînes se termine en un temps fini;
- ainsi, on peut dire que la suppression d'instances se termine. □.

preuve de $\mathcal{P}_4$:

- la règle SED R17 montre que tout calcul par un SED se termine au bout d'un temps fini. □.

À partir de $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_4$, nous pouvons dire qu'une configuration correcte sera atteint par le déploiement quelque soit la configuration initiale. En effet, $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$ montrent qu'il arrivera un moment où le graphe qui modélise un déploiement sera constitué d'une seule composante connexe ($\mathcal{P}_1$), qu'il n'y a plus de création de nouvelles instances ($\mathcal{P}_2$), ni de suppression d'instances ($\mathcal{P}_3$). Cela signifie que le nombre d'instances devient constant. À partir de ce moment, les seules règles dont les gardes peuvent être vraies sont celles qui n'ont ni d'effet de création ni de suppression, à savoir les deux règles : Client R3 et SED R17. Or, $\mathcal{P}_4$ montre que l'exécution de SED R17 se termine au bout d'un temps fini. Quant à la règle Client R3, elle est exécutée une fois et l'instance qui l'exécute (re-soumettre une requête à un MA) recevra une réponse positive (adresse d'un SED) ou négative (si aucun SED ne peut exécuter sa requête).

Ainsi, au bout d'un temps fini, toutes les instances seront stables, le déploiement aussi. En conclusion, on peut dire qu'un déploiement de l'intergiciel, sujet à des pannes transitoires, retrouvera une configuration stable au bout d'un temps fini.

Preuve de la propriété de clôture

Pour rappel, la propriété de clôture dispose qu'un déploiement stable reste stable en l'absence de fautes transitoires.

Dans notre cas, un déploiement est stable lorsque toutes les instances sont stables. Une instance est stable lorsqu'il n'est pas en train d'exécuter une règle. Lorsqu'une instance est stable (ses voisins aussi sont stables), les seuls événements qui peuvent le rendre instable sont les fautes transitoires comme la perte d'un voisin (ce qui n'arrivera pas puisque les voisins sont stables), l'ajout de nouvelles instances (ce qui n'arrivera pas car lorsque le déploiement est stable, le nombre d'instances est constant et il n'y a pas de nouvelles créations), la perte de connexion avec un voisin, etc. Donc, en l'absence de fautes transitoires, une instance stable reste stable, et par conséquent le déploiement reste stable. □

4.5 Conclusion

Dans ce chapitre, nous avons présenté un algorithme dont l'objectif est de rendre le déploiement d'un intergiciel auto-adaptatif. Nous avons prouvé que l'algorithme proposé est auto-stabilisant. Ainsi, à partir d'un déploiement initial arbitraire, un déploiement stable sera atteint au bout d'un temps fini. Nous n'avons cependant pas une idée claire du temps de stabilisation. Pour cela, nous allons effectuer des simulations pour faire une évaluation expérimentale du comportement de l'algorithme. Ces résultats sont présentés dans le Chapitre 5.

CHAPITRE 5

Simulations

Sommaire

5.1 Résumé du chapitre

Dans ce chapitre, nous décrivons le simulateur conçu pour faire une évaluation de certaines propriétés de l'algorithme décrit dans le chapitre précédent (comme le temps de stabilisation). Il inclut aussi les simulations réalisées et les résultats des simulations.

5.2 Introduction

Nous avons décidé de simuler l'algorithme proposé afin d'avoir une appréciation de son comportement. Les simulations d'un algorithme ne permettent pas de prouver son caractère auto-stabilisant (preuve apportée au Chapitre 4) puisqu'on a pas ici une borne supérieure pour le temps de stabilisation (la définition garantit l'atteinte d'un état global correct dans un temps fini, mais ne précise pas en combien d'étapes). Cependant les simulations pourront révéler des cas de convergence. L'objectif des simulations est d'étudier de manière expérimentale certains aspects de l'algorithme.

On n'a pas aussi des topologies régulières (la topologie d'un déploiement est un graphe dans le cadre général et un arbre dans des cas particulier, voire une chaîne). Avec des topologies régulières, on pourrait chercher des bornes supérieures (complexité temporelle) pour le temps de stabilisation.

Pour ce faire, nous avons conçu un simulateur Ad hoc permettant de simuler l'algorithme décrit dans le Chapitre 4. Nous voulons tester la convergence quand le réseau est à nouveau stable. Nous supposons donc que dans tous les cas, le réseau sous-jacent est connecté et pas partitionné parce que comme l'algorithme est basé sur une communication entre les processus, si une déconnexion du réseau rend cette communication impossible, l'algorithme que nous voulons évaluer ne pourra pas s'exécuter correctement.

Nous décrivons dans la suite de ce chapitre l'implémentation du simulateur, les simulations réalisées et leurs résultats.

5.3 Simulateur

Le simulateur a été conçu pour nous permettre de valider l'aspect auto-adaptatif de l'algorithme. Il a été programmé avec le langage Erlang [21, 145].

Dans un système programmé avec Erlang, le "travail" est réalisé par les processus. Le processus est l'élément de base, qui exécute les tâches et qui utilise des fonctions regroupées dans des modules. Les processus communiquent entre eux par échange de messages. L'échange de messages peut se faire de manière synchrone ou asynchrone. Un processus reçoit les messages qui lui sont envoyés (on suppose que le réseau fonctionne correctement). Un processus peut fixer les types de messages qu'il est prêt à recevoir et les actions à exécuter lorsque ce type de message est reçu. Les messages sont rangés dans une file et consommés les uns après les autres (selon leur ordre d'arrivée ou bien selon des priorités).

Un processus peut superviser l'existence (l'état vivant) d'autres processus en établissant un lien avec eux (ce lien est symétrique). Lorsqu'un processus se termine, il envoie un signal *EXIT* à tous les processus avec qui il existe un lien. Le comportement par défaut d'un processus qui reçoit le signal *EXIT* est de se terminer lui aussi en propageant le signal aux processus avec lesquels il est lié. Un processus peut changer ce comportement

par défaut. Dans ce cas, la réception du signal est traité de la même manière que les autres types de messages, afin de prévoir les actions à exécuter lorsqu'on le reçoit.

5.4 Fonctionnalités du simulateur

Le simulateur peut réaliser plusieurs actions. Les plus importantes, et celles que nous avons utilisées le plus sont les suivantes : créer un déploiement (prédéfini ou aléatoire), créer un événement de simulation pour rendre instable un déploiement stable ou afficher l'état global du déploiement (stable ou instable).

5.4.1 Créer un déploiement

On peut créer un déploiement prédéfini ou un déploiement aléatoire.

Déploiement prédéfini

Pour créer un déploiement prédéfini, il faut décrire la hiérarchie souhaitée dans un fichier. La hiérarchie est décrite comme une liste (représentée par une paire de crochets [...]) d'éléments de la forme :

```
1 [ {{type,id,l1,l2,l3,l4,l5,l6},init|adapt},
2   {{.....},init|adapt},
3   {{.....},init|adapt},
4   ..........................................
5   {{.....},init|adapt}
6
7  ].
```

Listing 5.1: Description d'un déploiement prédéfini

La structure $\{\{type, id, l1, l2, l3, l4, l5, l6\}, init|adapt\}$ décrit chaque élément de cette liste. Les données qu'elle contient ont les significations suivantes :

- type : le type du composant de base à savoir Client ou MA ou LA ou SeD;
- id : identifiant pour distinguer des éléments de même type. Cette information est utilisée surtout, avec d'autres éléments, lors de la création du processus, pour créer des identifiants uniques et globaux (avec une fonction de hachage) pour les processus crées;
- l1, l2, l3 : l1 (successivement l2 et l3) désigne la liste contenant l'ancêtre de type Client (successivement de type MA et de type LA). Nous rappelons qu'un élément de DIET ne peut avoir un type d'ancêtre différent de ces trois possibilités. Toutes ces listes peuvent être vides, dans le cadre d'un élément à qui on a pas fixé un ancêtre, ou bien pour un élément de type Client. Cependant, lorsqu'elles ne sont pas toutes vides, une seule d'entre elles contient des valeurs, les deux autres étant vides car un élément de la hiérarchie ne peut avoir qu'un ancêtre;

- l4, l5, l6 : l4 (successivement l5 et l6) désigne la liste contenant les fils de type MA (successivement de type LA et de type SED). Nous rappelons qu'un élément de DIET ne peut avoir un type de fils différent de ces trois possibilités. Toutes ces listes peuvent être vides, dans le cadre d'un élément qui n'a aucun fils, ou bien pour un élément de type SED. Certains éléments peuvent avoir des fils de différents types, et dans ce cas, chaque type de liste contient les informations correspondant aux fils de même type;
- init ou adapt : Le paramètre *init* permet de distinguer les processus qui ont été créés avant la création des événements de simulation (événements perturbateurs) des processus créés pendant les phases d'auto-adaptation (après un événement perturbateur); pour ces derniers le paramètre est *adapt*. Ces deux informations servent à l'exploitation des données de la simulation.

Ce fichier sera lu et chaque élément sera transformé en processus (simulant une instance du composant de base, s'exécutant sur une machine choisie de manière aléatoire parmi les machines du réseau) qui établira les liens avec ses voisins (ancêtre et fils). Un exemple de tel fichier pour définir un déploiement prédéfini est proposé dans le Listing 5.2.

```
1  [
2  {{ma,ma1,[],[],[],[],[],[{sed,sed1,nil},{sed,sed2,nil},
3          {sed,sed3,nil},{sed,sed4,nil},{sed,sed5,nil}]},init},
4
5  {{sed,sed1,[],[{ma,ma1,nil}],[],[],[],[]},init},
6  {{sed,sed2,[],[{ma,ma1,nil}],[],[],[],[]},init},
7  {{sed,sed3,[],[{ma,ma1,nil}],[],[],[],[]},init},
8  {{sed,sed4,[],[{ma,ma1,nil}],[],[],[],[]},init},
9  {{sed,sed5,[],[{ma,ma1,nil}],[],[],[],[]},init}
10 ].
```

Listing 5.2: Exemple de description d'un déploiement prédéfini

Il s'agit ici de décrire une hiérarchie qui sera composée d'un élément racine (*ma1*) (cas particulier où la hiérarchie est un arbre) de type MA, qui a cinq fils tous de type SED (*sed1* à *sed5*). Le paramètre *nil* sert à initialiser. Il précise qu'il faudra créer une référence lors de la création du processus. Cette référence sera combinée avec le paramètre id pour créer un nom global pour chaque processus. Ces deux informations (id, référence) permettront aussi, à tout processus qui les connaît de pouvoir retrouver l'adresse du processus correspondant pour lui envoyer un message.

L'élément *ma1* (décrit lignes 2 et 3), qui n'a pas de père ici (les listes pour ancêtres sont vides) est suivi des listes des fils (comme il n'a ni fils de type MA, ni fils de type LA, les listes prévues pour ses éléments sont vides) de type SED (seule liste de fils non vide).

Pour chacun des SED, seul la liste prévue pour les ancêtres de type MA contient leurs ancêtres (ici *ma1*) et comme le SED ne peut avoir de fils, les listes pour fils sont vides.

Le déploiement créé à partir de ce fichier ressemblera à la structure de la Figure 5.1. Les instances seront créées sur des machines choisies de manière aléatoire parmi celles constituant le réseau de nœuds Erlang (des machines virtuelles sur lesquelles s'exécutent les processus).

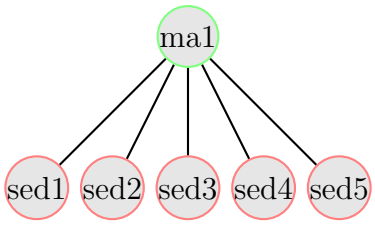

Figure 5.1: Un déploiement de 1 MA + 5 SeDs

Déploiement aléatoire

Pour un déploiement aléatoire, il suffit de lancer un nombre quelconque d'instances isolées (de même type ou de type différent), les laisser s'auto-adapter, et au bout d'un nombre fini de transitions, on obtient un déploiement aléatoire stable.

5.4.2 Créer un événement de simulation

Le simulateur permet de générer des événements qui peuvent rendre un déploiement instable et activer l'exécution de l'algorithme auto-adaptatif. En d'autres termes, ces événements peuvent rendre vraies les gardes des règles composant l'algorithme et permettre l'exécution des actions associées. Ces événements simulent des pannes transitoires. Les principaux événements générés sont l'ajout (création) d'un ensemble d'instances isolées (de même type ou de type différent) qui vont rejoindre un déploiement existant ou bien tuer un certain nombre d'instances (en les choisissant de manière aléatoire ou connaissant leurs identifiants) existant dans un déploiement. L'action qui consiste à tuer un nombre d'instances d'un déploiement existant se déroule de la façon suivante : à partir du déploiement de départ sur lequel on veut appliquer l'événement de simulation, et connaissant le nombre n d'instances qu'on veut tuer (n donné ou calculé si c'est un pourcentage), on exécute n fois l'action suivante : choisir de manière aléatoire une instance répondant aux critères (type d'instance par exemple) et lui envoyer un signal (message) pour qu'elle se termine.

5.4.3 Afficher l'état global d'un déploiement

On peut afficher l'état global du déploiement de manière périodique (la période peut être fixée par la personne qui conduit les simulations), c'est-à-dire si le déploiement est stable ou instable, avec le nombre d'instances stables, le nombre d'instances instables et le nombre total d'instances du déploiement. Cet affichage nous permet par la suite de compter le nombre d'unité de temps (appelé aussi hop) que prend un déploiement pour redevenir stable après un événement de simulation.

5.5 Description du simulateur

Le simulateur se compose de trois parties principales :

Un serveur de déploiement

Cet élément centralisé a une vue globale du déploiement. C'est l'élément qui joue le rôle d'oracle et qui répond aux requêtes relatives à la découverte de ressources. En effet,

la découverte de ressources n'a pas été intégrée dans le simulateur. Elle est donc simulée par cet élément.

Son état interne est représenté par une structure de donnée dynamique (graphe), qui représente une image temps réel du déploiement. Chaque sommet du graphe représente un processus simulant une instance de l'intergiciel. Chaque arête représente un lien entre un processus et un de ses voisins. Cette structure de donnée est mise à jour après chaque création d'une nouvelle instance, après chaque création ou suppression d'un lien, par chaque instance qui se termine. Ce serveur doit être déployé avant toute création d'instance.

Un serveur de détection de la stabilité du déploiement

Il sert à détecter l'état global d'un déploiement. En effet, pour les besoins des simulations, nous voulons pouvoir détecter si l'algorithme (qui doit permettre au système perturbé de retrouver un état correct et qui se déclenche lorsque l'état du système est instable) est en cours d'exécution ou s'il s'est terminé. Si l'algorithme (les règles) est en cours d'exécution, cela signifie que le déploiement est instable. Si l'algorithme ne "s'exécute plus", c'est qu'aucune règle n'est active, et donc que le déploiement est stable. Le fait que l'algorithme d'auto-stabilisation ne soit pas en train de s'exécuter, ne signifie pas qu'aucun autre algorithme n'est en train d'être exécuté. Cela signifie, ici, que le système est stable et qu'il est en train d'exécuter son algorithme de base, c'est à dire, qu'il est en train de fournir les fonctionnalités attendues. Dans la simulation, nous n'avons pas pris en compte ce que fait le système une fois qu'il est stable. Son algorithme n'a pas été programmé, les processus se contentent lorsqu'ils sont stables, de tester de manière périodique leurs états et leurs liens avec leurs voisins. Ce qui a été programmé, c'est l'algorithme qui permet au système de retrouver un état stable une fois qu'il a été perturbé.

Ce serveur nous sert donc à traiter les informations relatives à l'état global du déploiement et à afficher ses informations. Son état interne est constitué de deux variables entières positives ou nulles. Une des variables contient le nombre d'instances stables, l'autre le nombre d'instances instables. La somme des deux représente le nombre total d'instances d'un déploiement. La manière dont ces variables sont mises à jour est décrite à la Section 5.5.4. C'est une version centralisée de détection de la terminaison d'un algorithme [61, 62, 67, 68]. Ce serveur doit aussi être déployé avant toute création d'instance.

Un déploiement

Un déploiement est une hiérarchie d'instances qui a une structure de graphe. Chaque instance se comporte comme un automate à états finis. Les instances ne peuvent communiquer entre elles que par passage de messages.

5.5.1 Représentation des composants de l'intergiciel

Comme décrit dans la Section 1.1.1, l'intergiciel DIET est composé de quatre types de composant de base, dont les instances constituent un déploiement. Ces composants de base sont : Client, MA, LA, SeD.

Nous avons utilisé un Automate à États Finis (**AEF** dans la suite du document) [146] pour modéliser chacun de ces composants.

Un AEF est une machine abstraite qui permet de modéliser la dynamique d'une entité.

Il dispose d'un nombre fini d'états et réalise des transitions entre ses états en fonction des données en entrée. Ils sont utilisés pour modéliser divers phénomènes dans divers domaines comme l'apprentissage de la programmation par des étudiants [147], l'architecture [148], la biologie [149], les protocoles de communication [150], etc.

Définition 13 (Automate à états finis déterministe)**.** *Un automate à états finis déterministe est un quintuplet* $(E, \Sigma, \delta, e_0, F)$ *où*
E *désigne un ensemble fini d'états.*
Σ *représente un alphabet, un ensemble fini de symboles.*
δ *désigne une fonction de transitions entre états.* $\delta : E \times \Sigma \to E$
e_0 *désigne un état initial.*
F *est un sous-ensemble de E, appelé ensemble des états finaux (ou terminaux).*

Pour un AEF déterministe, $card(\delta(e, m)) = 1$, c'est-à-dire que l'entrée m lorsque l'automate est à l'état e produit la même cible.

Définition 14 (Automate à états finis Non déterministe)**.** *Un automate à états finis non déterministe est un quintuplet* $(E, \Sigma, \delta, I, F)$ *où*
E *désigne un ensemble fini d'états.*
Σ *représente un alphabet, un ensemble fini de symboles.*
δ *désigne une fonction de transitions entre états.* $\delta : E \times \Sigma \subseteq E$
I *représente un sous-ensemble de E, appelé ensemble des états initiaux.*
F *est un sous-ensemble de E, appelé ensemble des états finals (ou terminaux).*

Un AEF peut être non déterministe de trois manières :

- $card(I) > 1$: plusieurs états initiaux;
- $\delta(e, -) \neq \emptyset$: transition arbitraire sans se préoccuper de l'entrée;
- $card(\delta(e, m)) > 1$: plusieurs cibles pour la même entrée.

Nous utiliserons indifféremment les termes automate, AEF, instance, processus, nœud pour désigner un processus, qui simule un composant de base de l'intergiciel et qui agit comme un automate à états finis. Cela signifie que le processus a un nombre fini d'états et effectue des transitions entre les états. Il peut être de type Client, MA, LA, SED.

Dans notre cas, pour chaque élément $\in \{Client, MA, LA, \text{SED}\}$, l'automate qui le modélise est non déterministe dans le sens où il peut exister plusieurs cibles pour la même entrée, mais sans transitions arbitraires et avec un seul état initial. En plus, l'automate est tel que son nombre d'états, $card(E) = card(R) + 1$ avec R qui désigne l'ensemble des règles définies pour ce type de composant, et l'état supplémentaire représente l'état stable. Ainsi pour le composant Client, $card(E) = 3 + 1$ puisque trois règles ont été définies pour lui. Pour le MA, on a $card(E) = 5 + 1$, pour le LA on a $card(E) = 6 + 1$ et pour le SED on a $card(E) = 3 + 1$.

Aussi, Σ représente dans notre cas spécifique, l'ensemble des messages et événements dont la réception ou la détection peut changer l'état d'un AEF. Il contient un élément spécial (un message) qu'on appellera *calcul_état*. À la réception de ce message (dont l'envoi par l'automate à lui même peut être provoqué par divers événements), l'automate recalcule son état, et peut éventuellement faire une transition vers un autre état ou rester dans le même état qu'auparavant. L'automate peut recevoir divers autres messages, dont

le traitement peut provoquer l'envoi du message *calcul_état*. La réception d'un message autre que *calcul_état* ne peut pas directement générer une transition.

I est réduit à un seul état (l'état initial). Les processus partent de cet état juste après leur initialisation (cet état est pris une seule fois dans le cycle de vie d'un processus), et les transitions possibles se font entre les autres états.

La fonction de transition, δ, peut être caractérisée de la manière suivante :

$\forall e \in E, \forall m_i \in \Sigma \; tel \; que \; m_i \neq calcul\_état : \delta(e, m_i) = e$

$\forall e \in E : \delta(e, calcul\_état) \subseteq E \setminus I$, ce qui signifie que l'ensemble des transitions est constitué d'éléments de la forme

$(e_i, calcul\_état, e_j)$ avec $e_i \in E$ et $e_j \in E \setminus I$.

F est vide car il n'existe pas d'état terminal. Cela signifie qu'une fois l'état initial dépassé, quelque soit l'état dans lequel se trouve l'automate, les événements détectés et/ou les messages reçus et les traitements qu'ils provoquent, peuvent créer une transition vers un autre état. Il n'y a pas un état à partir duquel aucune transition n'est plus possible, tout dépend des événements détectés et des messages reçus.

Chaque AEF est à la fois un serveur et un client. Il est un serveur dans la mesure où il peut recevoir des requêtes (messages) auxquelles il répond mais aussi client car pouvant envoyer une requête à un autre AEF.

5.5.2 Gestion des états d'un AEF

Définition de l'état interne d'un AEF

L'état interne d'un AEF est constitué de l'ensemble de ses variables locales (leurs valeurs). La même structure de données (un enregistrement) est utilisée pour décrire cet état interne. Un sous-ensemble des champs de cette structure sert à conserver des données pour identifier l'élément, et un autre sous-ensemble des champs sert à conserver les identifiants des voisins (père et fils). Comme un élément ne peut avoir au plus que trois types de père et trois types de fils, six listes ont été prévues pour conserver ces informations sur les voisins. Cet état est mis à jour en fonction des messages reçus, des timeout, de la rupture de lien avec un voisin, etc. Mais seul l'instance propriétaire peut modifier son état interne.

Identification

Le sous-ensemble des variables locales qui conserve les données d'identification de l'élément est utilisé par une fonction de hachage qui crée, à partir de ces informations un nom unique (par rapport au cluster de machines virtuelles Erlang). Ce nom unique peut servir d'adresse au processus et on peut lui envoyer un message en donnant ce nom comme adresse. En plus, on peut retrouver l'identifiant du processus (PID) à partir de ce nom.

L'avantage de ce nom est la possibilité de l'utiliser pour personnaliser les informations sur les processus (en perspective du traitement des données à faire plus tard après la simulation). Par exemple on peut concaténer le type (ma, la, ...) du processus avec ce nom et on aura encore des noms uniques avec l'avantage d'avoir une information sur le type du processus. Par contre le PID offre moins de flexibilité car il ne permet pas de distinguer les types. Mais son avantage est que les communications sont plus rapides car il n'est pas nécessaire de faire des calculs pour trouver l'adresse, alors que cette étape est un préalable (fonction hachage) si on veut utiliser le nom unique de hachage. Dans le cas

de l'implémentation de l'intergiciel DIET, ce problème est résolu au travers des IOR de CORBA et du service de nommage (*Naming Services*).

Introspection

Un AEF à une capacité d'introspection, c'est-à-dire, qu'il a la capacité de lire son état interne (valeurs courantes des variables locales) à chaque fois que cela s'avère nécessaire et peut l'utiliser dans diverses opérations.

Calcul de l'état

L'état d'un AEF ne dépend que de son état interne. Dans son fonctionnement, un AEF peut envoyer et recevoir des messages, peut détecter certains événements de son environnement, peut exécuter des opérations périodiques.

À la suite de chacun de ces événements, un ensemble d'opérations prévues est exécuté en fonction du type du message, de l'événement détecté, du test effectué. Et à chaque fois que l'exécution de cet ensemble d'opérations est susceptible de modifier l'état interne de l'AEF, ce dernier calcule de nouveau son état pour le mettre à jour. Ainsi, le traitement associé à chaque événement pouvant modifier l'état interne, comporte une dernière opération qui demande de calculer à nouveau l'état et de le mettre à jour. Cette demande se fait par l'envoi d'un message particulier, *calcul_état* (cf. Section 5.5.1).

La réception de ce message spécial par un AEF provoque le calcul et la mise à jour de son état en fonction des valeurs courantes de ses variables locales. Ce message, ne peut être envoyé à un AEF que par lui même. Le résultat du calcul peut être une transition vers un autre état ou un *statu quo*.

Chaque instance vérifie de manière périodique ses liens avec ses voisins. Si après un certain nombre de tentatives, la vérification échoue, l'instance met à jour ses variables locales en supprimant de la liste des voisins l'instance avec qui la vérification a échoué, et recalcule son état.

De même, chaque instance lance de manière périodique, une opération de mise à jour de son état même si aucun message n'est reçu et traité, ni aucun événement détecté.

5.5.3 Définition d'un déploiement stable pour le simulateur

Un déploiement correspond à une hiérarchie d'instances, modélisées sous forme d'AEF. Chaque instance est déployée sur une machine virtuelle Erlang. Les instances ne peuvent communiquer que par envoi de messages.

Nous allons affiner la définition d'un déploiement stable. On l'a déjà défini dans le cadre général (Définition 12), on le redéfinit en tenant compte des particularités de la simulation.

Définition 15 (Un déploiement stable pendant une simulation)**.** *Pendant les simulations, un déploiement sera dit stable si :*

- *Chaque instance est dans son état stable (c'est-à-dire qu'elle n'est pas en train d'exécuter une règle). Si toutes les instances sont dans leur état stable, on a :*

  $$\#\{instances\ instables\} = 0$$

$\#\{instances\ stables\} = \#\{instances\ du\ déploiement\};$

- *Il y a au moins deux instances dans le déploiement (un déploiement avec une seule instance est toujours instable);*
- *Et le déploiement est stable (et le reste par la suite) pendant un nombre n d'unités de temps, et durant ces n unités de temps (et après), le nombre total d'instances reste constant en l'absence d'événements externes.*

L'idée derrière ce nombre n est qu'au bout de n unités de temps, tout message en transit devrait atteindre sa destination, et aussi qu'une instance nouvellement créée devrait terminer sa phase d'initialisation et se faire enregistrer.

Transition des états

Pour chaque type de composant $x \in \{Client, MA, LA, SeD\}$ nous **notons par** :
s : l'*état* d'une instance de type x.
ss : l'*état stable* d'une instance de type x.
us_i^x : l'*état instable* i d'une instance de type x.
$\mathcal{US}^x = \{us_1^x, ..., us_k^x\}$ l'*ensemble des états instables* d'une instance de type x. k dépend de x et est tel que $k = card(R)$, avec R qui représente l'ensemble des règles définies pour le type de composant x.
$\mathcal{AS}^x$: l'*ensemble des états possibles* d'une instance de type x :

$$\mathcal{AS}^x = \{ss, us_1^x, ..., us_k^x\} \ = \ \{ss\} \bigcup \mathcal{US}^x$$

La Figure 5.2 illustre le comportement générique d'un AEF et les types de transitions qu'il peut effectuer.

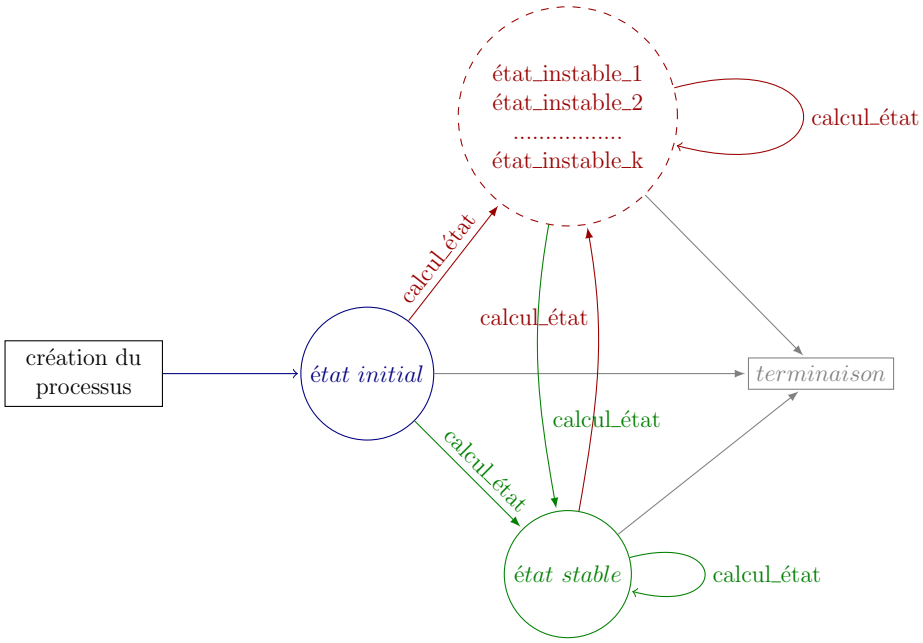

Figure 5.2: Transitions entre les états d'un AEF

Après la création du processus, l'AEF exécute des instructions d'initialisation et se met automatiquement dans l'état initial, qu'il occupera cette unique fois. Il s'auto-envoie un message pour calculer son état.

A un instant donné, l'état s d'un AEF "vivant"(déjà initialisé) de type x est tel :

$s \in \mathcal{AS}^x = \{ss, us_1^x, ..., us_k^x\}$; donc stable ou instable.

Si l'AEF est stable ($s = ss$), il exécute le code pour lequel il est spécifié. Il effectue aussi des vérifications périodiques et peut recevoir et gérer les messages qui lui sont envoyés. Après le traitement des événements, il calcule de nouveau son état et le met à jour. Le résultat de cette mise à jour est soit le statu quo (de stable à stable), soit une transition vers un des états instables $us_i^x \in \mathcal{US}^x, 1 \leq i \leq card(\mathcal{US}^x)$.

Si l'AEF est instable ($s \in \mathcal{US}^x = \{us_1^x, ..., us_k^x\}$), il peut faire une transition de son état instable courant vers le même état instable, ou bien vers un autre des états instables, ou bien vers l'état stable.

5.5.4 Détection d'un déploiement stable

Lorsqu'un déploiement est soumis à des pannes transitoires et devient instable, l'algorithme auto-adaptatif s'exécute pour que le système retrouve un état stable. Pour les besoins de la simulation, nous avons besoin de savoir, à un moment donné, si le déploiement est stable (l'algorithme auto-adaptatif ne s'exécute pas) ou instable (l'algorithme est en train de s'exécuter). Nous avons donc besoin de pouvoir déterminer l'état global d'un déploiement, et cela passe par la possibilité de détecter la terminaison (ou non) de l'exécution de l'algorithme auto-adaptatif.

L'état global d'un déploiement est constitué de l'état de chacun de ses nœuds et de l'état des canaux de communication (Définition 4). Détecter l'état global d'un système distribué où il n'existe pas de mémoire partagée, ni un temps global et où les délais des messages sont arbitraires n'est pas trivial. La détection d'un état global du déploiement peut être comprise comme un algorithme de détection de la terminaison [66], qui détermine si un algorithme distribué est terminé. La détection de la terminaison est un problème fondamental pour la programmation distribuée. S'inspirant des définitions dans [66], nous pouvons formuler notre problème (détecter si le déploiement est stable ou instable) comme un problème de détection de la terminaison d'un algorithme distribué.

Une partie des calculs et des messages envoyés et reçus constituent l'algorithme auto-adaptatif dont l'exécution a pour but de rendre le déploiement stable. C'est l'ensemble des règles définies pour les différents types de composant. Ces calculs sont appelés calculs de base et les messages produits dans ce cadre, des messages de base. Une partie supplémentaire est ajoutée et comprend les calculs et messages produits dans le but de détecter la terminaison. On les appelle calculs et messages de contrôle. Cette partie est indépendante de l'algorithme auto-adaptatif mais a été ajoutée pour détecter l'état global du déploiement, pour les besoins de la simulation. Ainsi, la partie contrôle est implémentée pour détecter la terminaison de l'algorithme de base.

Les messages de contrôle correspondent à ceux utilisés dans le Listing 5.3, ils sont envoyés par une instance à un observateur externe (serveur de détection de la stabilité du déploiement).

La terminaison peut être détectée de l'extérieur par un observateur externe (version centralisée que nous avons adoptée). Mais si ce sont les processus eux mêmes qui doivent détecter la terminaison, dans ce cas on a la détection distribuée de terminaison.

Une instance d'un déploiement est soit en exécution ("vivante"), soit elle est terminée.

Une instance en exécution est soit dans un état *stable* (actif), soit dans un état *instable* (passif). Lorsqu'une instance est dans un état *instable* (un des états de l'ensemble fini des états instables possibles pour ce type d'instance), elle exécute l'algorithme auto-adaptatif. Cela signifie qu'au moins une règle a une garde qui est vraie et les actions correspondantes sont en train d'être exécutées. Si l'état de l'instance n'est pas *instable*, alors il est *stable*.

Une instance terminée cesse d'exister et ne peut plus ni recevoir, ni envoyer de messages, ni exécuter aucune autre action. Une instance se termine à la réception d'un message particulier (message *exit* par exemple). Une instance se termine à la suite d'un ordre (message) de terminaison qu'il s'auto-envoie ou qui lui est envoyé par une autre instance.

Juste avant de cesser d'exister définitivement, une instance peut exécuter une dernière fonction (fonction terminer). Une instance peut utiliser cette fonction pour "avertir" les autres instances de son choix (notamment ses voisins) de sa terminaison. En plus, un message avertissant de la terminaison d'une instance est envoyé automatiquement à toutes les autres instances avec qui elle avait un lien. Donc, chaque fois qu'une instance se termine, ses voisins le sauront d'une manière ou d'une autre, même si c'est une terminaison brutale causée par une panne du matériel. Soit les voisins recevront un message les avertissant de la terminaison ou bien ils tenteront de vérifier le lien (ce qui se fait de manière périodique) et ce sera un échec. Dans ces différents cas, les voisins mettront à jour leurs données internes en supprimant l'instance terminée ou inaccessible de la liste de leurs voisins.

Un calcul distribué est considéré comme terminé lorsque tous ses processus en exécution sont dans l'état passive et qu'aucun message de base n'est en transit (tous les canaux de communication sont vides). Ceci est appelé la condition de terminaison distribuée [66].

Dans notre cas, cette condition correspond à la Définition 15. Les instructions et messages de contrôle pour détecter la terminaison de l'algorithme de base constituent l'algorithme de détection de la terminaison.

Détecter un déploiement stable est donc comparable à la détection d'une terminaison distribuée mais avec des hypothèses moins strictes. En effet, nous supposons que lorsque le déploiement est stable et reste stable pour un nombre fixé d'unités de temps, c'est qu'il n'y a pas de messages en transit, donc que les canaux sont vides. Les messages en transit incluent ceux qui sont en attente d'être consommés. Théoriquement, lorsque les communications sont asynchrones, les messages peuvent prendre un temps arbitraire mais fini. Notre hypothèse est de borner ce temps dans la mesure où nous effectuons les simulations dans un environnement stable où les messages sont transmis de manière spontanée.

Nous avons utilisé une méthode Ad hoc et centralisée pour détecter la terminaison. Les instances surveillent des variations particulières de leurs états et envoient des messages à un observateur extérieur qui se charge de détecter l'état global du déploiement. C'est le **serveur de détection de la stabilité du déploiement** (Section 5.5).

L'état interne de ce serveur est constitué de deux variables entières et positives :

StableSem pour compter le nombre courant d'instances stables et **UnstableSem** pour compter le nombre courant d'instances instables d'un déploiement. Ces deux variables sont initialisées à zéro lorsque le serveur est lancé (avant toute création d'instance) et sont mises à jour selon la méthode décrite dans le Listing 5.3.

Une instance qui vient d'être créée, après une phase d'initialisation se met dans l'état initial, qu'elle prend cette unique fois. Ensuite, au prochain calcul de son état, le résultat sera entre l'état stable ou un des états instables du type de l'instance. C'est à partir de

ce moment qu'elle peut s'enregistrer au niveau du serveur. Donc une instance qui n'est pas encore dans un état stable ou instable est inconnue de ce serveur.

Une instance connue du serveur (déjà enregistrée) est donc soit stable ou instable et peut se terminer à partir de cet état. Une instance qui a entamé une phase de terminaison ne peut plus changer d'état et son état est le dernier qu'elle a eu avant d'entamer la phase de terminaison (exécution de la fonction terminer).

À chaque fois qu'une instance veut calculer son état, elle sauvegarde son état précédent et le compare avec le nouvel état. Si certaines variations sont notées (une transition d'un état e_i à un état e_j tel $e_i \neq e_j$), l'instance envoie un message au serveur pour qu'il incrémente ou décrémente ses deux variables.

Ainsi une transition d'un état initial à un état stable incrémente **StableSem** et une transition d'un état initial à un état instable incrémente **UnstableSem**.

Une transition d'un état stable à un état instable incrémente **UnstableSem** et décrémente **StableSem** et une transition d'un état instable à un état stable incrémente **StableSem** et décrémente **UnstableSem**.

Lorsqu'une instance meurt (lorsque l'instance exécute la fonction terminer), elle envoie un message au serveur qui décrémente une des variables en fonction du dernier état de l'instance qui meurt.

```
1  case {EtatPrecedent, EtatCourant} of  /*case transition*/
2
3  /*transition de l'état initial vers un des états
4  instables. Premier calcul de son état après initialisation*/
5  {etat_initial, instable} : UnstableSem++;
6
7  /*transition de l'état initial vers l'état stable.
8  Premier calcul de son état après initialisation*/
9  {etat_initial, stable} : StableSem++;
10
11 /*transition de l'état stable vers un des états instables*/
12 {stable, instable} : UnstableSem++;
13                      StableSem--;
14
15 /*transition d'un des états instables vers l'état stable*/
16 {instable, stable} : StableSem++;
17                      UnstableSem--;
18
19 /*instance instable qui meurt (instance exécutant la
20 fonction terminer)*/
21 {instable, instance qui se termine} : UnstableSem--;
22
23 /*instance stable qui meurt (instance exécutant la
24 fonction terminer)*/
25 {stable, instance qui se termine} : StableSem--;
26
27  end;
```

Listing 5.3: Méthode de détection d'un déploiement stable

5.6 Configuration matérielle et logicielle

Toutes les simulations ont été réalisées sur une machine ayant les caractéristiques matérielles suivantes : Processeur Intel(R) Xeon(R) X5570 @ 2.93GHz avec 16 coeurs et 33 GB de RAM, avec le système d'exploitation Debian GNU/Linux 7 (wheezy), et la version Erlang R15B01 (erts-5.9.1).

5.7 Simulations et résultats

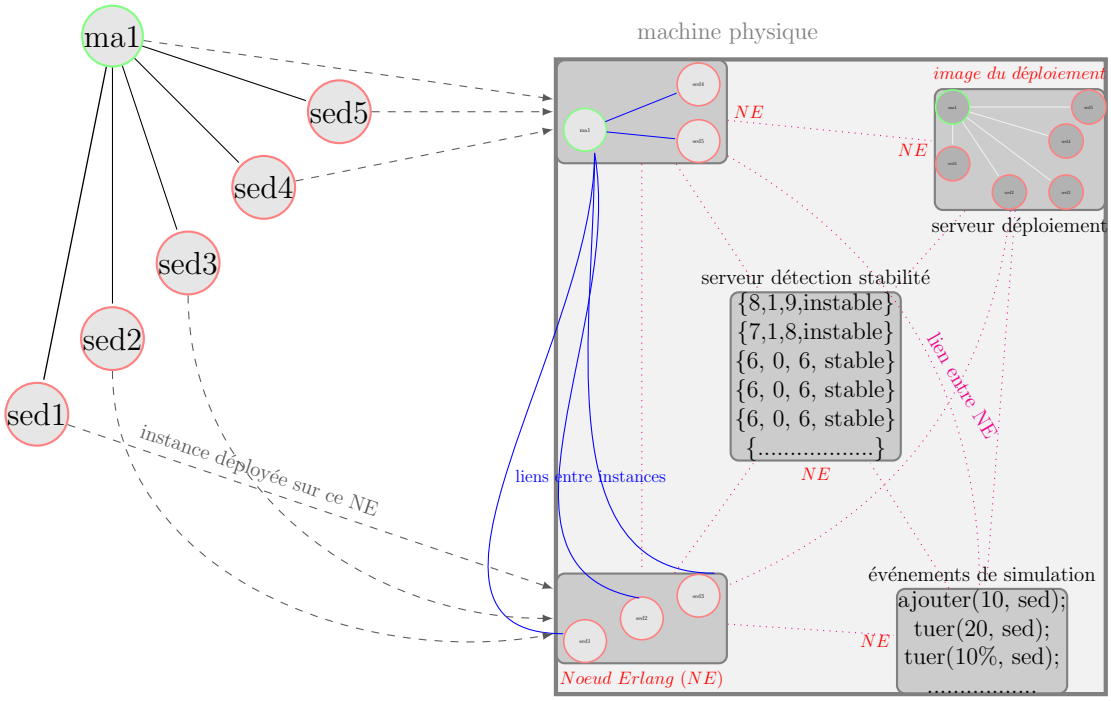

Figure 5.3: Un exemple de configuration de simulation

Pour toutes les simulations effectuées, nous avons utilisé cinq machines virtuelles Erlang (que nous appelons aussi nœuds erlang) déployées sur la même machine physique (Figure 5.3). Les nœuds erlang sont connectés entre eux sous la forme d'un graphe complet, formant ainsi une sorte de cluster. Ainsi, tout processus déployé sur un des nœuds peut communiquer avec un autre processus déployé sur le même nœud ou sur un autre nœud s'il connaît son adresse. De même, un processus peut exécuter une fonction sur un nœud autre que celui sur lequel il est déployé par un mécanisme d'appel de procédure à distance. Cette topologie n'est pas obligatoire mais elle est simple à mettre en œuvre. Deux nœuds sont utilisés pour le déploiement de la hiérarchie, un nœud pour le serveur de déploiement (contient une image du déploiement courant sous la forme d'une structure de données graphe), un œud pour le serveur de détection de la stabilité (affichage périodique du nombre d'instances stables, du nombre d'instances instables, du nombre total d'instances et de l'état global du déploiement), et un nœud pour lancer des événements de simulation (ajout de nouvelles instances, suppression d'instances).

Les instances de la hiérarchie sont déployées de manière aléatoire entre les deux nœuds prévus pour les recevoir.

Nous avons effectué quatre simulations que nous allons décrire dans la partie qui suit.

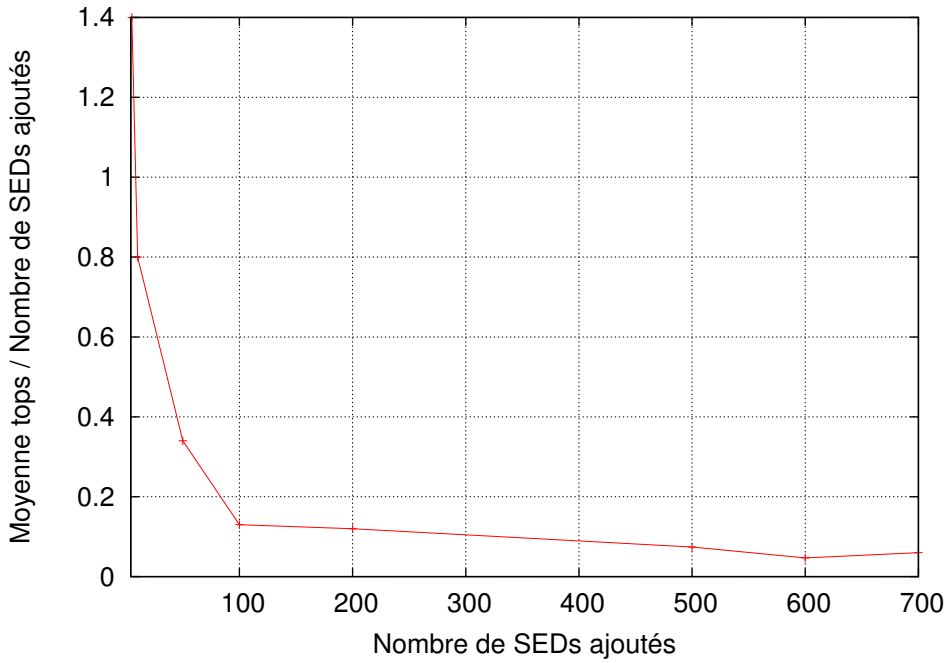

Figure 5.4: Ajout d'un nombre X (abscisse) de nouvelles instances (des SEDs isolés) à un déploiement stable. Pour chaque X, calculer la moyenne (sur cinq valeurs) des tops qu'il a fallu pour que le déploiement retrouve un état stable. Calculer le ratio entre cette moyenne et X. La courbe représente le ratio en fonction de X.

5.7.1 Effet d'un changement de topologie par ajout de nouvelles instances

L'idée de la première simulation est la suivante : créer un déploiement stable et simuler l'effet d'un changement de topologie dû à l'ajout de nouvelles instances. Pour réaliser cette simulation, nous partons d'un déploiement stable et simple constitué d'un (1) MA et de cinq SEDs, fils du MA. Et pour chaque $X \in \{5, 10, 50, 100, 200, 500, 600, 700\}$, nous ajoutons X nouveaux SEDs isolés (éléments instables) au déploiement de base et comptons le nombre de périodes de temps (500ms) que nous appelons tops que va prendre le déploiement pour redevenir stable. Pour chaque X, l'expérience est répétée cinq fois et on calcule la moyenne des tops pour les cinq expériences.

La Figure 5.4 montre le ratio entre la valeur moyenne des tops et le X correspondant.

5.7.2 Effet du changement de topologie par suppression d'instances

L'idée de ces simulations est d'étudier l'effet d'un changement de topologie dû à la suppression d'un certain nombre d'instances ou d'un pourcentage des instances.

Suppression d'un nombre d'instances

Pour cette simulation, nous créons un déploiement de départ stable avec un nombre assez important de SEDs. Pour ce faire, nous ajoutons 500 SEDs isolés à un déploiement de base constitué d'un MA et cinq SEDs et obtenons à la fin un déploiement stable avec un nombre important de SEDs.

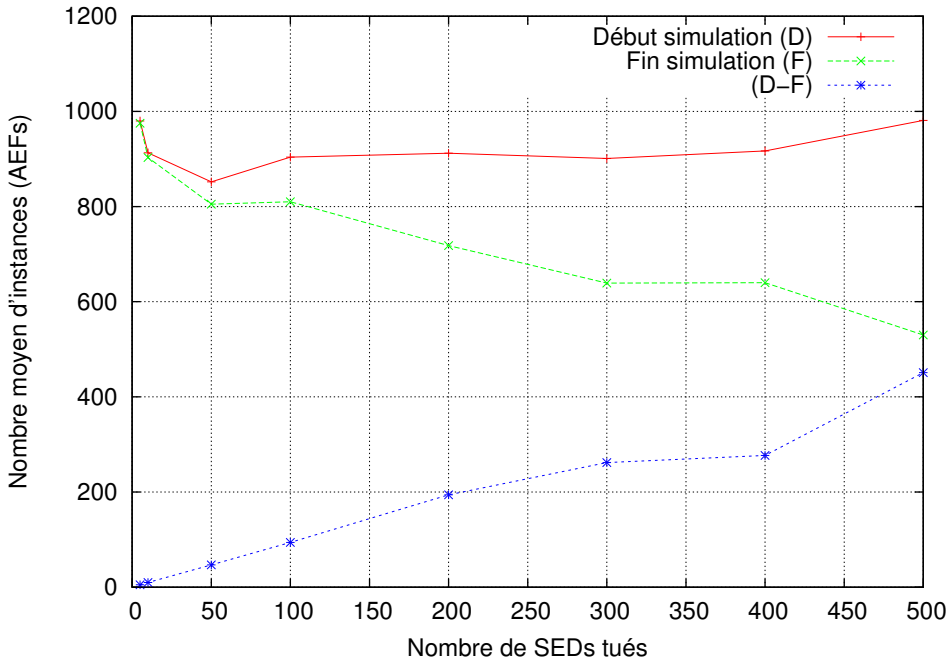

Figure 5.5: Suppression d'un nombre X (abscisse) d'instances (des SeDs) d'un déploiement stable. Pour chaque X, déterminer le nombre total d'instances de tout type (Client, MA, LA, SeD) avant et après la suppression, ainsi que la différence entre ces deux valeurs. La courbe représente ces trois valeurs en fonction de X.

Il est possible de faire autrement en créant directement un déploiement par ajout de 500 SeDs isolés et laisser faire le processus de stabilisation. Mais nous avons constaté que la première méthode prenait "moins de temps" que la seconde méthode pour que le déploiement se stabilise . Nous avons donc préféré la première méthode pour cette raison.

Les déploiements stables obtenus ne sont pas toujours les mêmes (ni en terme de structure ni en terme de nombre d'instances) même s'ils sont tous obtenus par les mêmes opérations. Cela est dû au fait que les décisions prises par les instances ne sont pas déterministes (nous rappelons que les processus sont modélisés par des AEF non déterministes).

Une fois ce déploiement stable obtenu avec un nombre assez important de SeDs, pour chaque $X \in \{5, 10, 50, 100, 200, 500\}$, on répète cinq fois les actions suivantes : X SeDs sont tués (choisis de manière aléatoire), on compte le nombre de tops d'horloge que le système prend pour redevenir stable, on sauvegarde le nombre total d'instances (AEF) avant l'événement de simulation ("tuer X SeDs") et aussi le nombre total d'instances (AEF) lorsque le déploiement redevient stable. Pour chaque X, on calcule les valeurs moyennes de ces paramètres (cinq valeurs pour chaque paramètre). Les Figures 5.5, 5.6 montrent les résultats de ces simulations.

La Figure 5.5 montre pour chaque X (nombre de SeDs tués) le nombre moyen d'AEF avant la simulation, le nombre moyen d'AEF après la simulation et la différence entre ces deux valeurs. La Figure 5.6, obtenue à partir des données du tableau 5.1, représente, pour chaque X, le ratio entre la moyenne des tops d'horloge (pour que le déploiement retrouve un état stable) et X. Pour cette expérience lorsqu'on tue 5 ou 10 ou 50 SeDs, le temps de stabilisation est presque instantané avec une moyenne du nombre de tops d'horloge pour

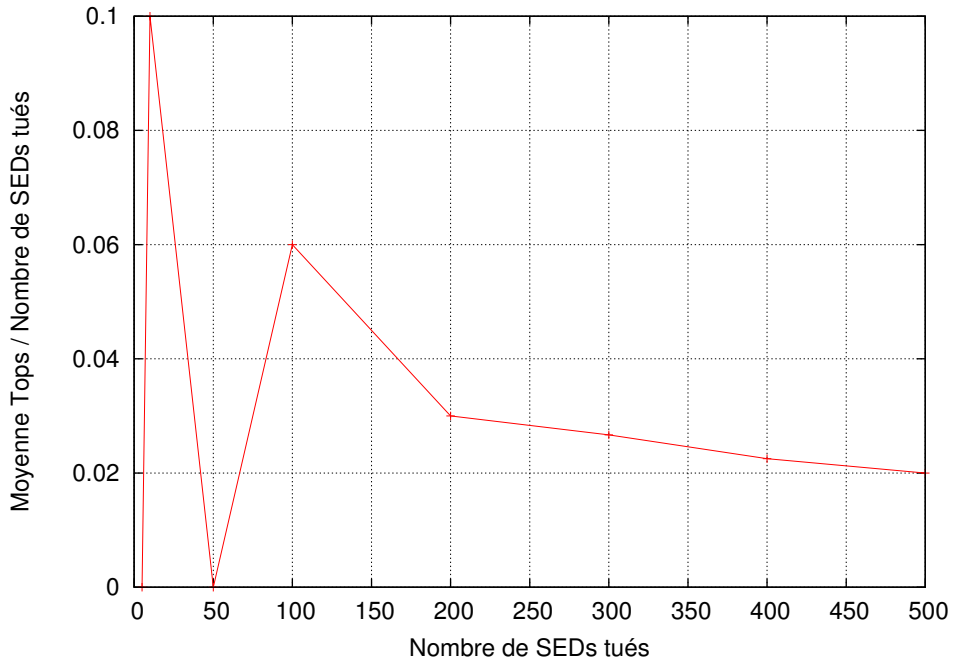

Figure 5.6: Suppression d'un nombre X (abscisse) d'instances (des SeDs) d'un déploiement stable. Pour chaque X, calculer la moyenne (sur cinq valeurs) des tops qu'il a fallu pour que le déploiement retrouve un état stable. Calculer le ratio entre cette moyenne et X. La courbe représente le ratio en fonction de X.

se stabiliser qui est égale respectivement à : $0,4$; $1,4$ et $0,2$. Cette moyenne passe à une valeur comprise entre 6 et $10,4$ lorsqu'on tue un nombre de SeDs compris 100 et 500. On peut dire que lorsqu'on tue un "petit" nombre de SeDs, l'effet sur le système est vite résorbé, si il n'est pas simplement négligeable. Ceci semble logique dans la mesure où, le fait de tuer un SeD, voire un ensemble de SeDs ne conduit pas forcément à un système instable.

Tableau 5.1: Données de la Figure 5.6

#SeDs tués	**moyenne tops d'horloge**	**(moyenne tops horloge) / (#SeDs)**
5	0,4	0,08
10	1,4	0,14
50	0,2	0,004
100	6	0,06
200	6,8	0,034
300	8	0,026
400	9,2	0,023
500	10,4	0,0208

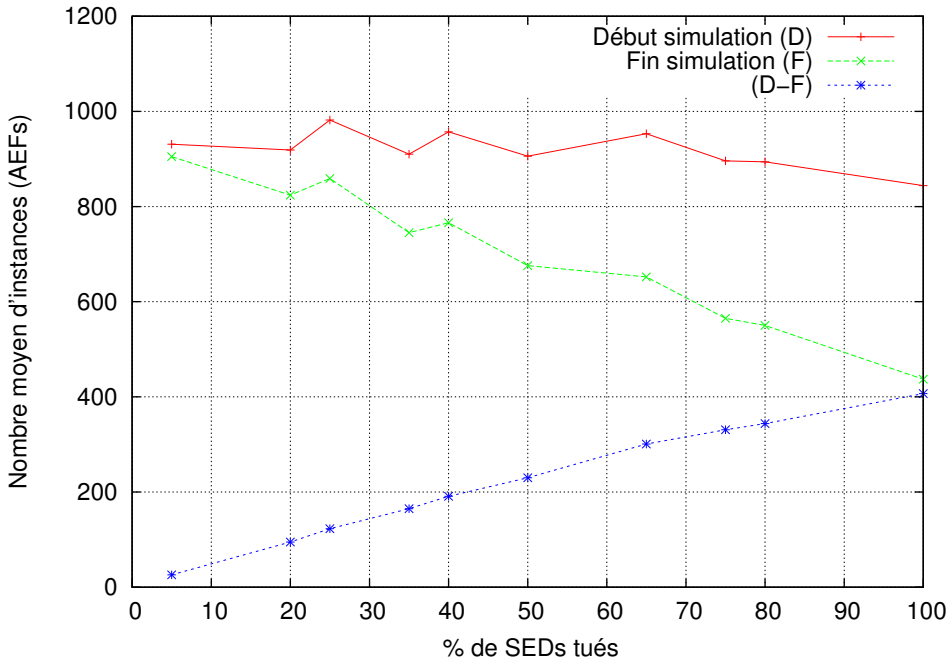

Figure 5.7: Suppression d'un pourcentage X (abscisse) d'instances (des SEDs) d'un déploiement stable. Pour chaque X, déterminer le nombre total d'instances de tout type (Client, MA, LA, SED) avant et après la suppression, ainsi que la différence entre ces deux valeurs. La courbe représente ces trois valeurs en fonction de X.

Suppression d'un pourcentage des instances

Cette simulation est comparable à la simulation précédente à la différence qu'au lieu de tuer X SEDs, ce sont $X\%$ des SEDs qui sont tués et $X \in \{5, 20, 25, 35, 40, 50, 65, 75, 80, 100\}$.

Ainsi donc, après avoir crée un déploiement stable avec un nombre "assez important" de SEDs, pour chaque $X \in \{5, 20, 25, 35, 40, 50, 65, 75, 80, 100\}$, $X\%$ des SEDs (choisis de manière aléatoire) et la simulation est répétée cinq fois et pour chaque fois les valeurs des paramètres suivants sont enregistrées:

- Le nombre de tops d'horloge que prend le système pour retrouver un état stable;
- Le nombre total d'instances (AEF) avant l'événement de simulation ("tuer $X\%$ des SEDs");
- Le nombre total d'instances (AEF) lorsque le déploiement redevient stable après application de l'événement de simulation ("tuer $X\%$ des SEDs").
- Le nombre total de SEDs avant l'événement de simulation ("tuer $X\%$ des SEDs");
- Le nombre total de SEDs lorsque le déploiement redevient stable après application de l'événement de simulation ("tuer $X\%$ des SEDs").

Pour chaque X, on calcule les valeurs moyennes de ces paramètres (cinq valeurs pour chaque paramètre).

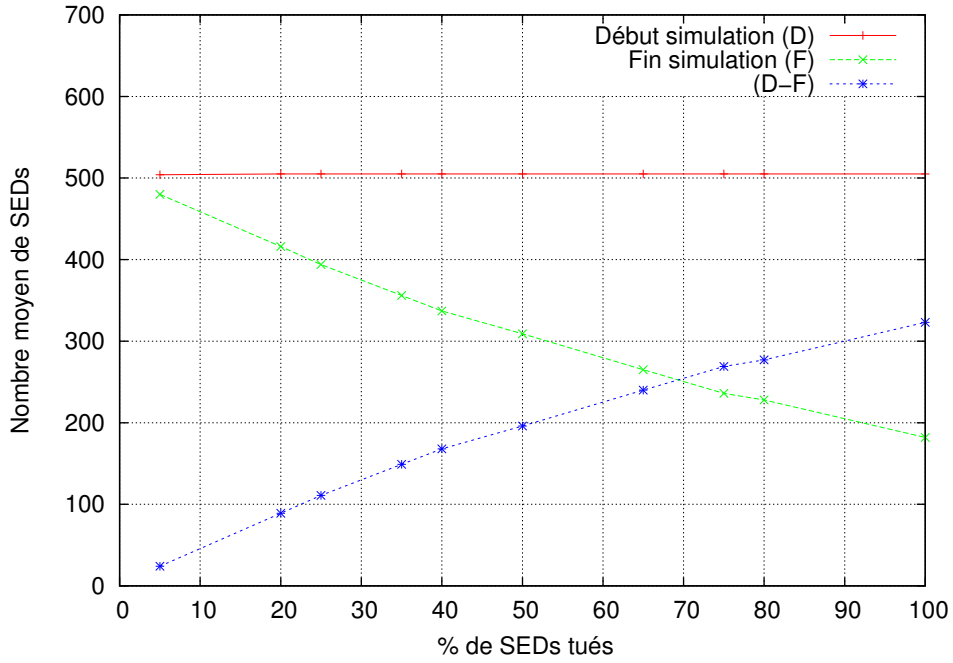

Figure 5.8: Suppression d'un pourcentage X (abscisse) d'instances (des SeDs) d'un déploiement stable. Pour chaque X, déterminer le nombre total d'instances de type SeD avant et après la suppression, ainsi que la différence entre ces deux valeurs. La courbe représente ces trois valeurs en fonction de X.

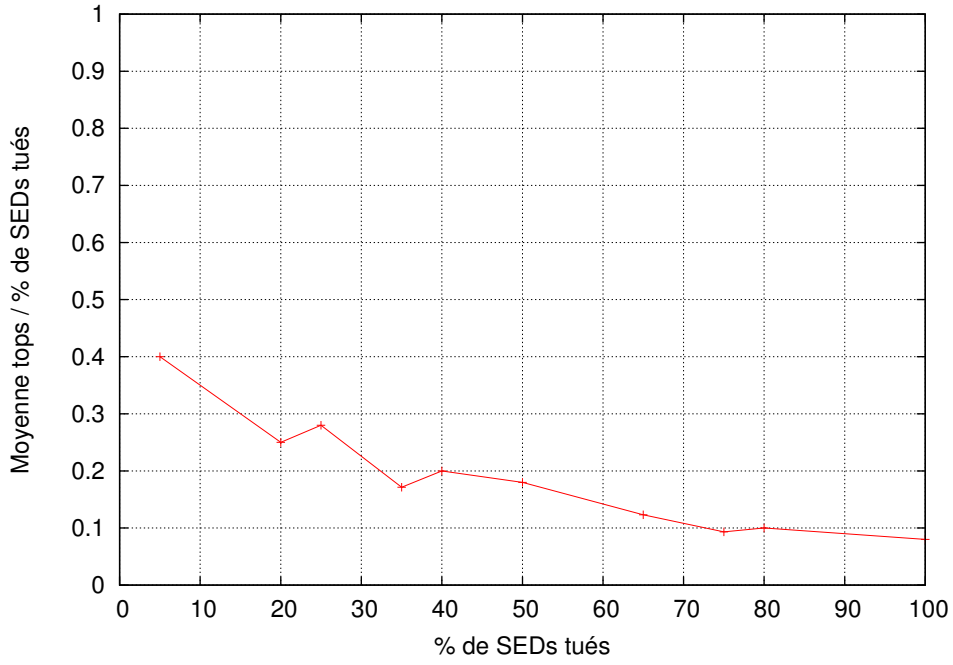

Figure 5.9: Suppression d'un pourcentage X (abscisse) d'instances (des SeDs) d'un déploiement stable. Pour chaque X, calculer la moyenne (sur cinq valeurs) des tops qu'il a fallu pour que le déploiement retrouve un état stable. Calculer le ratio entre cette moyenne et X. La courbe représente le ratio en fonction de X.

Les figures 5.7, 5.8, 5.9 montrent les résultats de cette simulation.

Les courbes sur la Figure 5.7 représente pour chaque X (le % de SEDs tués) le nombre moyen d'AEF avant l'événement de simulation (début simulation), après l'événement de simulation lorsque le déploiement est redevenu stable (fin simulation) et la différence entre ces deux valeurs moyennes.

Le nombre moyen d'AEF avant la simulation est indépendant de l'événement de simulation. Il varie entre 844 et 982 instances et est le résultat obtenu en créant un déploiement en ajoutant 500 SEDs isolés à un déploiement stable constitué d'un MA et cinq SEDs.

Le nombre moyen d'AEF à la fin de la simulation décroît lorsque le pourcentage de SEDs tués augmente.

La différence entre le nombre moyen d'AEF au début et à la fin de la simulation peut être intuitivement compris comme le nombre de SEDs tués. Cependant, le nombre totales d'instances tuées peut être supérieur ou inférieur au nombre de SEDs tués par l'événement de simulation car des instances peuvent être créées ou tuées durant la phase d'auto-adaptation après que l'événement de simulation a été déjà appliqué.

Les courbes sur la Figure 5.8 représentent pour chaque X (% de SEDs tués) le nombre moyen de SEDs au début de la simulation, à la fin de la simulation, et la différence entre ces deux valeurs. Comme dans le cas précédent, les valeurs moyennes avant le début de la simulation sont indépendantes de l'événement de simulation. Cette valeur est presque constant et varie entre 504 et 505 instances.

Le nombre moyen de SEDs à la fin de la simulation décroît lorsque le pourcentage de SEDs tués augmente mais est toujours supérieur ou égal à zéro même lorsque 100% des SEDs sont tués. En effet, des SEDs sont aussi créés quand c'est nécessaire par l'algorithme auto-adaptatif.

La différence entre les deux valeurs moyennes (au debut et à la fin) peut être intuitivement compris comme le nombre moyen de SEDs qui se sont finalement terminés.

La Figure 5.9 montre le ratio entre le nombre moyen de tops d'horloge pour que le déploiement recouvre un état stable (lorsque X% des SEDs sont tués) et X.

5.7.3 Effet du changement de topologie par alternance d'ajout et de suppression d'instances

L'idée de cette simulation est de partir d'un déploiement stable et d'alterner les ajouts de nouvelles instances aux suppressions d'instances. À chaque fois qu'un de ces événements est appliqué, on attend que le déploiement retrouve un état stable et on applique l'événement suivant.

Pour cela, nous créons d'abord un déploiement stable avec 408 instances, obtenu par ajout de 250 SEDs isolés à un déploiement stable constitué d'un (1) MA et de cinq SEDs.

A partir de ce déploiement stable de 408 instances, on tue 100 SEDs choisis de manière aléatoire, et on attend que le système retrouve un état stable. Une fois que le système est redevenu stable, on ajoute 100 SEDs et on attend encore que l'état stable soit atteint pour alterner ces deux opérations. Durant toute la simulation, le nombre total d'AEF stables et le nombre total d'AEF instables sont enregistrés de manière périodique. Pour dessiner les courbes, on a réduit les plages pendant lesquelles le système est stable. Donc, avant l'application de tout événement "ajout de 100 SEDs " ou "suppression de 100 SEDs ", le système a été stable pendant suffisamment longtemps, période qu'on a réduite pour mettre en exergue les moments pendant lesquels le système retrouve un état correct.

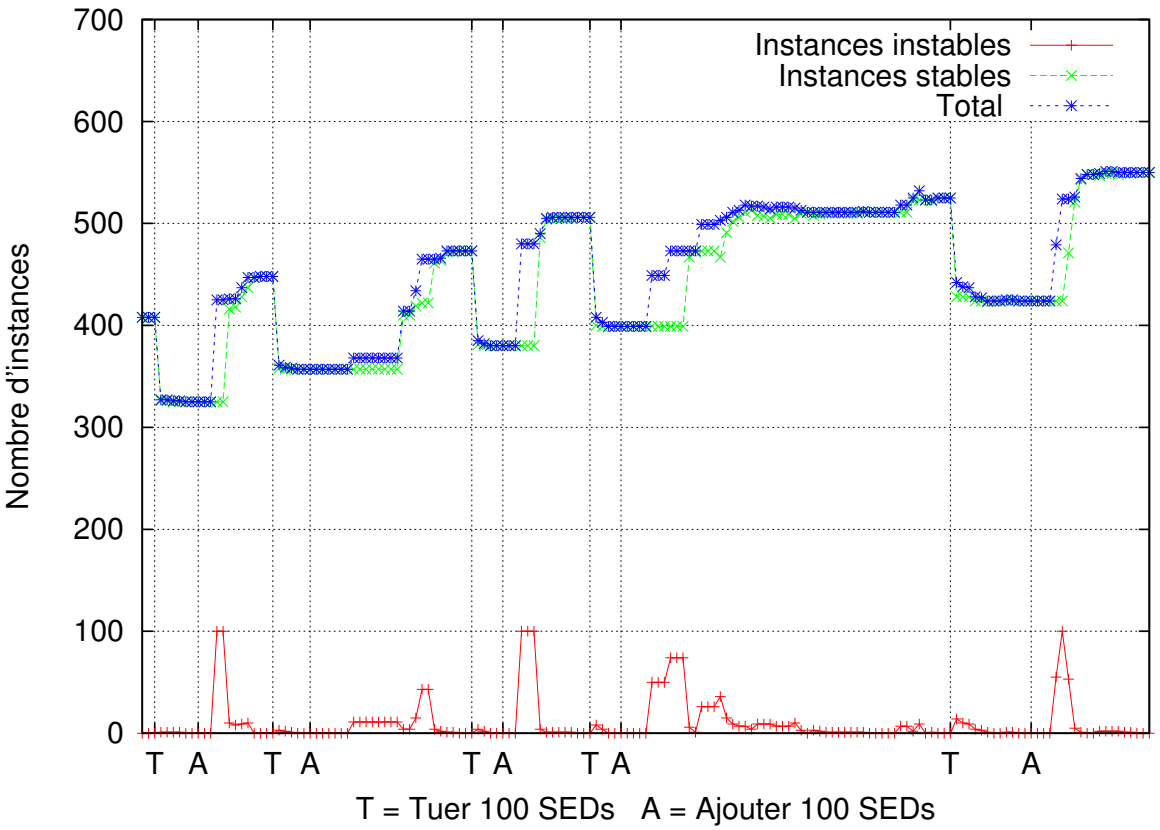

Figure 5.10: Alternance des événements “tuer 100 SeDs (T)” et “ajouter 100 SeDs (A)”. Après l'application d'un événement, attendre que le déploiement retrouve un état stable et appliquer l'événement suivant.

Les courbes sur les Figures 5.10 et 5.11 montrent les variations des valeurs enregistrées au cours de la simulation.

On peut faire quelques observations sur la courbe de la Figure 5.11. La première est qu'après l'application de chaque événement de simulation qui modifie la topologie (ajout ou suppression), le déploiement retrouve un état stable (nombre d'instances instables égal à zéro) au bout d'un certain temps.

On peut aussi observer que l'effet de l'événement de suppression d'instances est plus spontané que celui de l'ajout. Ceci peut s'expliquer par le fait qu'après une action d'ajout, les instances nouvellement créées ont besoin de s'initialiser avant que le processus d'auto-adaptatif ne commence. Or, pendant cette phase d'initialisation (le temps que cela prend peut varier d'une instance à une autre, en fonction de leur type et des données initiales, mais dans tous les cas, ce temps est non nul), les instances ne sont pas encore enregistrées au niveau du serveur qui détecte la stabilité même si l'instance est connue par le serveur de déploiement. C'est après la phase d'initialisation et une première mise à jour de son état que l'instance (maintenant dans un état stable ou instable) peut exécuter les instructions d'adaptation.

L'effet de l'action de suppression est plus spontané parce qu'une instance qui se termine exécute moins d'opérations en général qu'une instance qui s'initialise. Une instance qui se termine envoie des messages *exit* à ses voisins (avec qui elle a un lien) et un message

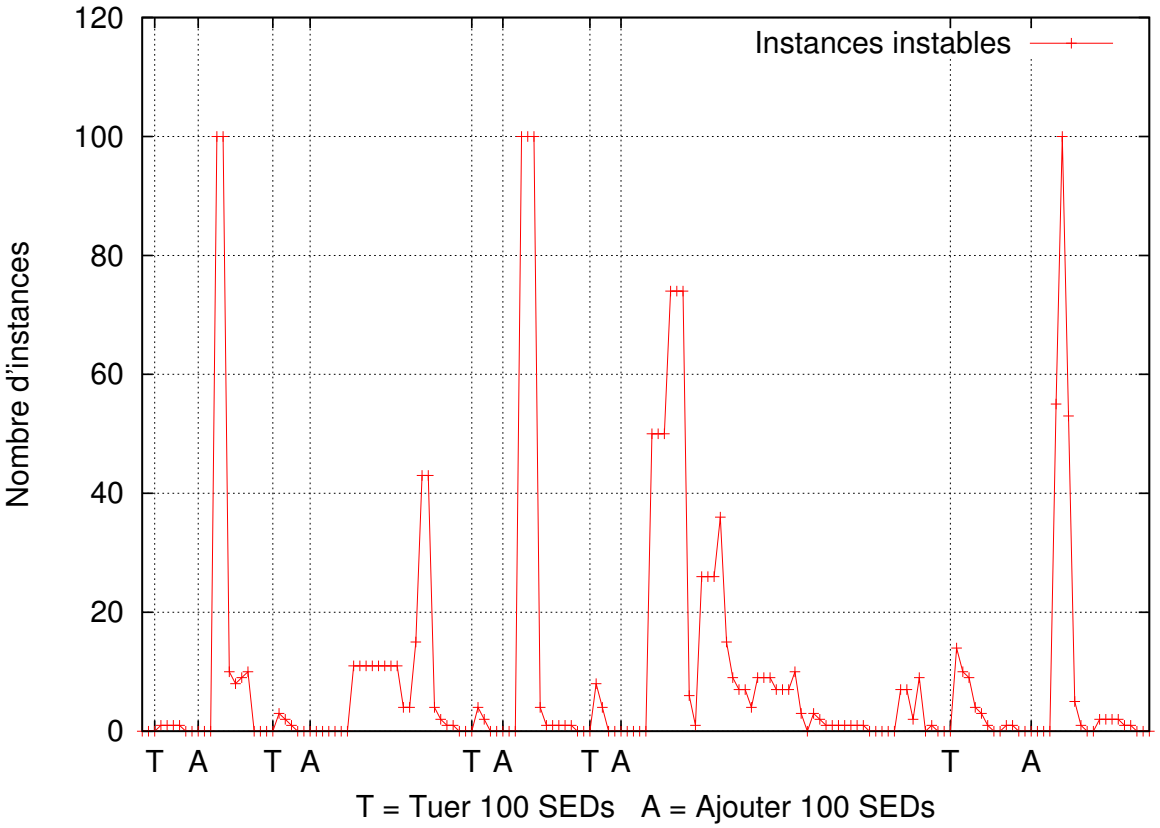

Figure 5.11: Alternance des événements “tuer 100 SEDs (T)” et “ajouter 100 SEDs (A)”. variation du nombre d'instances instables.

de mise à jour des variables au serveur de détection de la stabilité.

Dans certains cas, l'effet de l'action de suppression n'est pas très perceptible parce que supprimer des SEDs d'un déploiement stable ne rend pas forcément le déploiement instable, mais c'est uniquement le nombre d'instances qui diminue dans ce cas.

5.8 Conclusion

Dans ce chapitre, nous avons décrit le simulateur que nous avons conçu pour évaluer l'algorithme auto-adaptatif décrit dans le Chapitre 4. Les processus sont modélisés par un automate à états finis non déterministe. Nous avons décrit les simulations effectuées et présenter les résultats.

Conclusion

Dans cette thèse, nous avons étudié les moyens de rendre auto-adaptatif le déploiement d'un intergiciel hiérarchique. L'auto-adaptation a lieu lorsque l'intergiciel détecte certains événements liés à la plate-forme sur laquelle il est déployé ou bien aux processus de l'intergiciel en cours d'exécution. Ces événements peuvent être la variation du nombre de ressources de la plate-forme et/ou l'arrêt accidentel de certains processus de l'intergiciel, et qui rendent l'état du déploiement instable.

Pour ce faire, nous avons proposé des modèles pour décrire les entités qui interviennent dans ce processus, à savoir la plate-forme distribuée et élastique sur laquelle l'intergiciel est susceptibles d'être déployé, l'architecture de l'intergiciel ainsi qu'un déploiement en cours d'exécution.

Nous avons ensuite proposé un algorithme distribué permettant à l'intergiciel d'être tolérant à certains types de pannes, mais aussi de chercher à atteindre un objectif qualitatif qui a été traduit par la définition d'un déploiement stable. Nous avons prouvé que cet algorithme est auto-stabilisant.

Nous avons conçu un simulateur permettant d'exécuter un algorithme auto-adaptatif et d'étudier son comportement. Nous avons simulé l'algorithme proposé, pour étudier son comportement face à certaines modifications du contexte. Les résultats des simulations montrent que l'algorithme est auto-adaptatif. Les simulations montrent également que le temps de stabilisation (des simulations) est arbitraire mais fini.

Perspectives

Pour de futurs travaux, il serait intéressant de réfléchir sur les pistes ci-dessous :

- Intégrer les modèles définis au Chapitre 3 (dans le simulateur) pour les futurs simulations. En effet, la prise en compte des caractéristiques des ressources de l'infrastructure (mémoire, CPU, réseaux, etc.), des contraintes de haut niveau de l'intergiciel (préférence d'exécution sur les ressources d'un site A au lieu du site B) améliorent le réalisme des simulations. À l'heure actuelle, le choix des ressources sur lesquelles on déploie les processus est fait de manière aléatoire et les ressources sont considérées comme identiques;

- Il faudra une campagne de simulations avec des actions qui ne se limiteront plus aux SeD mais aussi aux autres types de composant de DIET pour comparer les effets;

- Il faudra aussi s'intéresser à d'autres modèles de pannes que celles transitoires;

- Une question intéressante est celle de savoir si l'algorithme est robuste ? Dans notre cas, cela signifie que l'algorithme est en mesure de fournir un service (même dégradé) pendant la phase même de stabilisation. Nous avons supposé dans ce travail que dès que le système est instable, il n'y a pas d'assurance qu'il puisse assurer sa spécification. Par principe de précaution, on suppose qu'il ne l'assure que lorsqu'il est stable et ne l'assure pas dans le cas contraire. Il serait intéressant d'étudier le comportement du déploiement pendant la phase même de stabilisation : est ce qu'il existe des "portions" (sous-arbres) de système stables pour exécuter des tâches même si le système global est instable ? Et si oui, de quel type, forme sont elles ? Le déploiement a une structure de graphe connexe lorsqu'il est stable. Mais il peut être une forêt pendant certaines phases de stabilisation avec plusieurs sous arbres non connectées. La question est de savoir si parmi ces sous-arbres, certains ne sont pas stables lorsqu'ils sont pris de manière isolée même si le déploiement global (ensemble de tous les sous-arbres) est instable;

- Nous avons utilisé des éléments centraux pour les besoins des simulations. Il serait intéressant de se passer des éléments centraux (serveur de déploiement qui joue le rôle d'oracle ou de service de découverte de ressources) en implémentant un mécanisme de découverte de ressources non centralisé pour que les nœuds ne se basent qu'exclusivement sur des informations locales. Une piste pourrait être l'utilisation d'algorithmes de type "gossip";

- Il serait intéressant aussi de réfléchir à l'adaptation du simulateur pour l'étendre à d'autres types d'intergiciels hiérarchiques ou non.

Annexes

Algorithme exécuté par toutes les instances de type Client

Algorithme 18: Client: MA_lost event

```
1  if CLIENT.lostMa == TRUE then
2  |   SetOfMa ← get the set of MA from the Oracle;
3  |   if Card(SetOfMa) > 0 then
4  |   |   selectOneMa();
5  |   |   connectToMa();
6  |   else
7  |   |   createOneMa();
8  |   |   connectToMa();
9  |   end
10 end
```

Algorithme 19: Client: SED _lost event

```
1 if CLIENT.lostSed == TRUE then
2 |   submitTheRequestAgain() ;
3 end
```

Algorithme exécuté par toutes les instances de type MA

Algorithme 20: MA: no child event

```
1 if MA.numberOfChildren == 0 then
2 |   SetOfMa ← get the set of MA from the Oracle;
3 |   if Card(SetOfMa) == 1 then
4 |   |   createSedAsChildWithBAsicService();
5 |   else
6 |   |   maSuicide();
7 |   end
8 end
```

Algorithme 21: MA: chain of Agent event

```
1 if (MA.numberOfChildren == 1) AND (MA.childType == MA OR LA) then
2 |   mergeMaAndChild() ;
3 end
```

Algorithme 22: MA: no MA father event

```
1 if MA did not have an father of type MA then
2 |   SetOfMaInOtherHierarchy ← get from the Oracle the set of MA in
  |   others hierarchy than the one which contains the MA executing this
  |   algorithm;
3 |   if Card(SetOfMaInOtherHierarchy) > 0 then
4 |   |   selectOneMAasFather();
5 |   |   connectToSelectedMa() ;
6 |   else
7 |   |   no forest;
8 |   end
9 end
```

Algorithme 23: MA: overloaded event

```
1 if MA.load ≥ MA.loadThreshold then
2 |   divideMaChildrenInTwoSet() ;
3 |   createOneAgentAsFatherForEachGroup() ;
4 |   theTwoNewlyCreatedAgentAsMaChildren() ;
5 end
```

Algorithme exécuté par toutes les instances de type LA

Algorithme 24: LA: no child event

```
1 if LA.numberOfChildren == 0 then
2 |   SetOfLa ← get the set of LA from the Oracle;
3 |   if Card(SetOfLa) == 1 then
4 |   |   createSedAsChildWithBAsicService();
5 |   else
6 |   |   laSuicide();
7 |   end
8 end
```

Algorithme 25: LA: chain of LA event

```
1 if (LA.numberOfChildren == 1) AND (LA.childType == LA) then
2 |   mergeLaAndChild() ;
3 end
```

Algorithme 26: LA: no father event

```
1 if LA.father == NULL then
2 |   SetOfAgentInOtherHierarchy ← get from the Oracle the set of Agent
  |   in others hierarchy than the one which contains the LA executing
  |   this algorithm;
3 |   if Card(SetOfAgentInOtherHierarchy) > 0 then
4 |   |   selectOneAgentAsFather() ;
5 |   |   connectToSelectedAgent() ;
6 |   else
7 |   |   createMa() ;
8 |   |   connectToMa() ;
9 |   end
10 end
```

Algorithme 27: LA: overloaded event

```
1 if LA.load ≥ LA.loadThreshold then
2 |   divideLaChildrenInTwoSet() ;
3 |   createOneLaAsFatherForEachGroup() ;
4 |   theTwoNewlyCreatedLaAsLaChildren() ;
5 end
```

Algorithme exécuté par toutes les instances de type SeD

Algorithme 28: SeD: no father event

```
1  if SED.father == NULL then
2      if SED.is_currently_computing == TRUE then
3          continue computing For T units of time.
4          T is supposed to be the Mean time a SeD took
5          to compute a job. After T units of time
6          the job is supposed to be finished ;
7      else
8          SetOfAgent ← get the set of Agents from the Oracle;
9          if Card(SetOfAgent) > 0 then
10             selectOneAgent();
11             connectToAgent();
12         else
13             createOneMa();
14             connectToMa();
15         end
16     end
17 end
```

Bibliographie

[1] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. Wiley-Interscience, 2004. 1, 18, 20

[2] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. 1, 11

[3] Peter Mell and Tim Grance. The nist definition of cloud computing. 2011. 1, 11

[4] Philip A Bernstein. Middleware: a model for distributed system services. *Communications of the ACM*, 39(2):86–98, 1996. 1

[5] Eddy Caron and Frédéric Desprez. DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006. 1, 29

[6] Hendrik Moens and Filip De Turck. A scalable approach for structuring large-scale hierarchical cloud management systems. In *CNSM*, pages 1–8. IEEE, 2013. 28

[7] Bernardetta Addis, Danilo Ardagna, Barbara Panicucci, Mark S. Squillante, and Li Zhang. A Hierarchical Approach for the Resource Management of Very Large Cloud Platforms. *IEEE Trans. Dependable Sec. Comput*, 10(5):253–272, 2013.

[8] Sander van der Burg and Eelco Dolstra. Disnix: A toolset for distributed deployment. *Science of Computer Programming*, 2012. 28, 29, 30

[9] Sander van der Burg and Eelco Dolstra. A self-adaptive deployment framework for service-oriented systems. In Holger Giese and Betty H. C. Cheng, editors, *2011 ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, Waikiki, Honolulu, HI, USA, May 23-24, 2011*, pages 208–217. ACM, 2011.

[10] Sander van der Burg and Eelco Dolstra. Automated deployment of a heterogeneous service-oriented system. In *Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on*, pages 183–190. IEEE, 2010. 1

[11] Clemens Szyperski. Components vs. objects vs. component objects. In *Proceedings of OOP*, volume 1999, 1999. 1

[12] Wolfgang Emmerich and Nima Kaveh. Component technologies: Java beans, com, corba, rmi, ejb and the corba component model. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 691–692. IEEE, 2002.

[13] Felix Bachmann, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. *Volume II: Technical concepts of component-based software engineering*. Carnegie Mellon University, Software Engineering Institute, 2000. 1

[14] Kung-Kiu Lau and Zheng Wang. Software component models. *Software Engineering, IEEE Transactions on*, 33(10):709–724, 2007. 2

[15] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software-Practice and Experience*, 36(11):1257–1284, 2006. 2, 27

[16] Eddy Caron, Frédéric Desprez, David Loureiro, and Adrian Muresan. Cloud Computing Resource Management through a Grid Middleware: A Case Study with DIET and Eucalyptus. In IEEE, editor, *CLOUD 2009: IEEE International Conference on Cloud Computing*, Bangalore, India, September 2009. Published In the Work-in-Progress Track from the CLOUD-II 2009 Research Track. 2, 11, 12

[17] Antonio Carzaniga, Alfonso Fuggetta, Richard S. Hall, Dennis Heimbigner, Andre Van Der, and Er L. Wolf. A Characterization Framework for Software Deployment Technologies. Technical report, Department of Computer Science, University of Colorado, April 10 1998. 2, 25, 26, 33

[18] Object Management Group, Inc. *Deployment and Configuration of Component-based Distributed Applications Specification, Version 4.0*, 2006. An Adopted Specification of the Object Management Group, Inc. http://www.omg.org/spec/DEPL/, 2015. 3, 26, 29, 33

[19] Alan Dearle. Software deployment, past, present and future. In *2007 Future of Software Engineering*, pages 269–284. IEEE Computer Society, 2007. 2, 25, 26

[20] Raja Boujbel. *Déploiement de systèmes répartis multi-échelles : processus, langage et outils intergiciels*. Thèse de doctorat, Université de Toulouse, Toulouse, France, janvier 2015. 2, 3, 25, 29

[21] Francesco Cesarini and Simon. Thompson. *Erlang Programming*. O'Reilly Media, Inc., 2009. 2, 58

[22] E. Caron, P.K. Chouhan, and H. Dail. GoDIET: A Deployment Tool for Distributed Middleware on Grid'5000. In IEEE, editor, *EXPGRID workshop. Experimental Grid Testbeds for the Assessment of Large-Scale Distributed Apllications and Tools. In conjunction with HPDC-15*, pages 1–8, Paris, France, June 19th 2006. 3, 26, 27, 34

[23] Zhengxiong Hou, Jing Tie, Xingshe Zhou, Ian Foster, and Mike Wilde. ADEM: Automating Deployment and Management of Application Software on the Open Science Grid. In *Grid 2009, 10th IEEE/ACM International Conference on Grid Computing*, pages 130–137, Banff, October 13-15 2009. IEEE, IEEE. 26

[24] Chef:, 2015. https://www.chef.io/chef/. 26

[25] Puppet labs: It automation software for system administrators, 2015. https://puppetlabs.com/. 3, 26

[26] Paul Marshall, Henry Tufo, and Kate Keahey. Provisioning policies for elastic computing environments. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1085–1094. IEEE, 2012. 4

[27] Alessio Gambi, Waldemar Hummer, Hong-Linh Truong, and Schahram Dustdar. Testing elastic computing systems. *Internet Computing, IEEE*, 17(6):76–82, 2013. 4

[28] Alan G Ganek and Thomas A Corbi. The dawning of the autonomic computing era. *IBM systems Journal*, 42(1):5–18, 2003. 5

[29] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003. 5, 27, 32, 44

[30] J. Andersson, R. De Lemos, S. Malek, and D. Weyns. Modeling dimensions of self-adaptive software systems. *Software Engineering for Self-Adaptive Systems*, pages 27–47, 2009. 5

[31] Rogério De Lemos, Holger Giese, Hausi A Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M Villegas, Thomas Vogel, et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013. 5, 44

[32] Jeffrey O. Kephart and Rajarshi Das. Achieving self-management via utility functions. *IEEE Internet Computing*, 11(1):40–48, 2007. 5

[33] Harald Psaier and Schahram Dustdar. A survey on self-healing systems: approaches and systems. *Computing*, 91(1):43–73, 2011. 5

[34] Harald Psaier, Florian Skopik, Daniel Schall, and Schahram Dustdar. Behavior Monitoring in Self-Healing Service-Oriented Systems. In *Proceedings of the 34th Annual IEEE International Computer Software and Applications Conference, COMPSAC 2010, Seoul, Korea, 19-23 July 2010*, pages 357–366. IEEE Computer Society, 2010.

[35] Eugen Feller, Louis Rilling, and Christine Morin. Snooze: A scalable and autonomic virtual machine management framework for private clouds. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 482–489. IEEE Computer Society, 2012. 5

[36] Edsger W. Dijkstra. Self-Stabilizing Systems in Spite of Distributed Control. *Communications of the ACM*, 17(11):643–644, 1974. 6, 17, 23, 30

[37] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems*. Prentice-Hall, 2007. 9, 15

[38] Kayhan Erciyes. *Distributed graph algorithms for computer networks*. Springer Science & Business Media, 2013. 9, 20

[39] Marco Conti and Stefano Giordano. Mobile ad hoc networking: milestones, challenges, and new research directions. *Communications Magazine, IEEE*, 52(1):85–96, 2014. 10

[40] Ian F Akyildiz and Mehmet Can Vuran. *Wireless sensor networks*, volume 4. John Wiley & Sons, 2010. 10

[41] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. 11

[42] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International journal of high performance computing applications*, 15(3):200–222, 2001. 11

[43] Fran Berman, Geoffrey Fox, and Anthony JG Hey. *Grid computing: making the global infrastructure a reality*, volume 2. John Wiley and sons, 2003. 11

[44] Ian Foster. Globus toolkit version 4: Software for service-oriented systems. In *Network and parallel computing*, pages 2–13. Springer, 2005. 11

[45] Dietmar W Erwin and David F Snelling. Unicore: A grid computing environment. In *Euro-Par 2001 Parallel Processing*, pages 825–834. Springer, 2001. 11

[46] Heba Kurdi, Maozhen Li, and Hamed Al-Raweshidy. A classification of emerging and traditional grid systems. *Distributed Systems Online, IEEE*, 9(3):1–1, 2008. 11

[47] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, pages 99–106, Washington, DC, USA, 2005. IEEE Computer Society. 11

[48] Mário David, Gonçalo Borges, Jorge Gomes, João Pina, Isabel Campos Plasencia, Enol Fernández-del Castillo, Iván Díaz, Carlos Fernandez, Esteban Freire, Álvaro Simón, et al. Validation of grid middleware for the european grid infrastructure. *Journal of Grid Computing*, 12(3):543–558, 2014. 11

[49] L.M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Linder. A Break in the Clouds: Towards a Cloud Definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2009. 11

[50] P. Sempolinski and D. Thain. A Comparison and Critique of Eucalyptus, OpenNebula and Nimbus. In *CloudCom*, pages 417–426. IEEE, 2010. 12

[51] Sonali Yadav. Comparative study on open source software for cloud computing platform: Eucalyptus, openstack and opennebula. *International Journal Of Engineering And Science*, 3(10):51–54, 2013. 12

[52] H. Nakada, S. Matsuoka, K. Seymour, J.J. Dongarra, C. Lee, and H. Casanova. A GridRPC Model and API for End-User Applications. In *GFD-R.052, GridRPC Working Group*, jun 2007. 12

[53] Keith Seymour, Hidemoto Nakada, S. Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In Manish Parashar, editor, *Grid Computing - GRID 2002, Third International Workshop*, volume 2536 of *LNCS*, pages 274–278, Baltimore, MD, USA,, November 2002. Springer. 12

[54] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984. 12

[55] Object Management Group. *The Common Object Request Broker (CORBA): Architecture and Specification*. Object Management Group, 1995. 12, 28

[56] Gérard Le Lann. Distributed systems-towards a formal approach. In *IFIP Congress*, volume 7, pages 155–160. Toronto, 1977. 13

[57] Osvaldo SF Carvalho and Gérard Roucairol. On mutual exclusion in computer-networks. *Communications of the ACM*, 26(2):146–147, 1983. 14

[58] Yuh-Jzer Joung. Asynchronous group mutual exclusion. *Distributed computing*, 13(4):189–206, 2000. 14

[59] Ousmane Thiare, Mohamed Naimi, and Mourad Gueroui. A group mutual exclusion algorithm for mobile ad hoc networks. In *Innovations and Advanced Techniques in Computer and Information Sciences and Engineering*, pages 373–377. Springer, 2007. 14

[60] SP Rana. A distributed solution of the distributed termination problem. *Information Processing Letters*, 17(1):43–46, 1983. 14

[61] Edsger W Dijkstra and Carel S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980. 14, 62

[62] Nissim Francez. Distributed termination. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):42–55, 1980. 14, 62

[63] Friedemann Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987. distributed termination detection, Mattern, DTD, terminaison, etat global stable. 14

[64] Subbarayan Venkatesan. Reliable protocols for distributed termination detection. *IEEE Transactions on Reliability*, 38(1):103–110, 1989.

[65] Jeff Matocha and Tracy Camp. A taxonomy of distributed termination detection algorithms. *The Journal of Systems and Software*, 43:207–221, 1998. distributed termination detection, terminaison.

[66] D.M. Dhamdhere, Sridhar R. Iyer, and E. Kishore Kumar Reddy. Distributed termination detection for dynamic systems. *ParallelCompufing*, 22:2025–2045, 1997. 67, 68

[67] JongBeom Lim, Kwang-Sik Chung, and Heon-Chang Yu. A termination detection technique using gossip in cloud computing environments. *J.J. Park et al. (Eds.): NPC 2012, LNCS 7513, IFIP International Federation for Information Processing 2012*, pages 429–436, 2012. DTD, distributed termination, terminaison, gossip algorithm, cloud,. 62

[68] Huiling Wu. Termination detection for synchronous algorithms in p systems. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, volume 1, 2014. 14, 62

[69] Ernest J. H. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, (4):391–401, 1982. 14

[70] Fred B Schneider and Leslie Lamport. Paradigms for distributed programs. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course, April 3-12, 1984 and April 16-25, 1985 Munich*, pages 431–480. Springer-Verlag, 1985.

[71] Jean-Michel Hélary, Aomar Maddi, Noël Plouzeau, and Michel Raynal. Parcours et apprentissage dans un réseau de processus communicants. 1986.

[72] Gerard Tel. *Introduction to distributed algorithms*. Cambridge university press, 2000. 20

[73] Sukumar Ghosh. *Distributed systems: an algorithmic approach*. CRC press, 2007. 14, 15, 17, 20

[74] P. Jesus, C. Baquero, and P.S. Almeida. A survey of distributed data aggregation algorithms. *Communications Surveys Tutorials, IEEE*, 17(1):381–404, 2015. 14

[75] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Dependability and its threats: a taxonomy. In *Building the Information Society*, pages 91–120. Springer, 2004. 15

[76] Sumit Kumar Bose, Scott Brock, Ronald Skeoch, and Shrisha Rao. CloudSpider: Combining Replication with Scheduling for Optimizing Live Migration of Virtual Machines across Wide Area Networks. In *CCGRID*, pages 13–22. IEEE, 2011. 17

[77] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *Software Engineering, IEEE Transactions on*, (1):23–31, 1987. 17

[78] Bogdan Nicolae and Franck Cappello. BlobCR: Efficient Checkpoint-Restart for HPC Applications on IaaS Clouds using Virtual Disk Image Snapshots. May 26 2011.

[79] Bogdan Nicolae, John Bresnahan, Kate Keahey, and Gabriel Antoniu. Going back and forth: efficient multideployment and multisnapshotting on clouds. In Arthur B. Maccabe and Douglas Thain, editors, *HPDC*, pages 147–158. ACM, 2011. 17

[80] Michael Barborak, Anton Dahbura, and Miroslaw Malek. The consensus problem in fault-tolerant computing. *aCM Computing Surveys (CSur)*, 25(2):171–220, 1993. 17

[81] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001. 17, 18

[82] Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993. 17

[83] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. 18

[84] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985. 18

[85] Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. Of Choices, Failures and Asynchrony: The Many Faces of Set Agreement. *Algorithmica*, 62(1-2):595–629, 2012. 18

[86] Faith Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed computing*, 16(2-3):121–163, 2003.

[87] Attiya Hagit and Ellen Faith. *Impossibility Results for Distributed Computing*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2014. 18

[88] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982. 18

[89] Nancy A Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996. 18

[90] Dinesh P. Mehta and Sartaj Sahni. *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.)*. Chapman & Hall/CRC, 2004. 19

[91] O.R. Aguilar, A.K. Datta, and S. Ghosh. Simulating shared memory in message passing model. In *Computers and Communications, 1991. Conference Proceedings., Tenth Annual International Phoenix Conference on*, pages 232–238, Mar 1991. 20

[92] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Resource bounds for self-stabilizing message-driven protocols. *SIAM Journal on Computing*, 26(1):273–290, 1997. 20

[93] Swan Dubois and Sébastien Tixeuil. A taxonomy of daemons in self-stabilization. *arXiv preprint arXiv:1110.0334*, 2011. 22, 23

[94] Lélia Blin. *Self-stabilizing algorithms for spanning tree construction and for the management of mobile entities*. Habilitation à diriger des recherches, Université Pierre et Marie Curie - Paris VI, December 2011. 22, 30

[95] Ankh Arora and Mohamed Gouda. Closure and convergence: A foundation of fault-tolerant computing. *Software Engineering, IEEE Transactions on*, 19(11):1015–1027, 1993. 23

[96] OSGi Alliance. Osgi service platform core specification, release 6, version 2.0, june 2014. http://www.osgi.org. 26

[97] Sam Malek, Nenad Medvidovic, and Marija Mikic-Rakic. An extensible framework for improving a distributed software system's deployment architecture. *Software Engineering, IEEE Transactions on*, 38(1):73–100, 2012. 26, 29

[98] Benoit Claudel, Guillaume Huard, and Olivier Richard. TakTuk, adaptive deployment of remote executions. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, pages 91–100, New York, NY, USA, 2009. ACM. 26

[99] A. Flissi, J. Dubus, N. Dolet, and P. Merle. Deploying on the Grid with DeployWare. In *CCGRID'08: Proceedings of the 8th IEEE/ACM International Symposium on Cluster Computing and the Grid*, Lyon, France, May 2008. 26, 29

[100] Ansible:, 2015. http://www.ansible.com/. 26

[101] Abbas Heydarnoori. Deploying component-based applications: Tools and techniques. In *Software Engineering Research, Management and Applications*, pages 29–42. Springer, 2008. 26

[102] Jean-Paul Arcangeli, Raja Boujbel, and Sébastien Leriche. Automatic deployment of distributed software systems: Definitions and state of the art. *Journal of Systems and Software*, 103:198–218, 2015. 27

[103] Laurent Broto, Daniel Hagimont, Patricia Stolf, Noel Depalma, and Suzy Temate. Autonomic Management Policy Specification in tune. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 1658–1663, New York, NY, USA, 2008. 27, 29, 34

[104] Sara Bouchenak, Fabienne Boyer, Benoit Claudel, Noel De Palma, Olivier Gruber, and Sylvain Sicard. From Autonomic to Self-Self Behaviors: The JADE Experience. *TAAS*, 6(4):28, 2011. 27

[105] D Garlan, SW Cheng, AC Huang, B Schmerl, and P Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004. 27

[106] Arun Mukhija and Martin Glinz. Runtime Adaptation of Applications Through Dynamic Recomposition of Components. In *Systems Aspects in Organic and Pervasive Computing - ARCS 2005*, volume 3432 of *Lecture Notes in Computer Science*, pages 124–138. Springer Berlin / Heidelberg, 2005. 27

[107] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society, 2007. 27

[108] Uwe Aßmann, Nelly Bencomo, Betty HC Cheng, and Robert B France. Models@run.time (dagstuhl seminar 11481). *Dagstuhl Reports*, 1(11):91–123, 2011. 27

[109] Franck Chauvel, Nicolas Ferry, Brice Morin, Alessandro Rossini, and Arnor Solberg. Models@ runtime to support the iterative and continuous design of autonomic reasoners. In *MoDELS@ Run. time*, pages 26–38, 2013.

[110] Sebastian Götz, Nelly Bencomo, and Robert France. Devising the future of the models@run.time workshop. *SIGSOFT Softw. Eng. Notes*, 40(1):26–29, February 2015. 27

[111] Betty HC Cheng, Kerstin I Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi A Müller, Patrizio Pelliccione, Anna Perini, Nauman A Qureshi, Bernhard Rumpe, et al. Using models at runtime to address assurance for self-adaptive systems. In *Models@ run. time*, pages 101–136. Springer, 2014. 28

[112] Filip Krikava, Philippe Collet, and Robert France. ACTRESS: Domain-Specific Modeling of Self-Adaptive Software Architectures. In *Symposium On Applied Computing*, Gyeongju, South Korea, March 2014. 28

[113] Thomas Vogel and Holger Giese. Model-driven engineering of self-adaptive software with eurema. *ACM Trans. Auton. Adapt. Syst.*, 8(4):18:1–18:33, January 2014. 28

[114] Abbas Heydarnoori and Walter Binder. A graph-based approach for deploying component-based applications into channel-based distributed environments. *Journal of Software*, 6(8):1381–1394, 2011. 28

[115] S. Lacour, C. Pérez, and T. Priol. Generic Application Description Model: Toward Automatic Deployment of Applications on Computational Grids. In *6th IEEE/ACM International Workshop on Grid Computing (Grid2005)*, Seattle, WA, USA, November 2005. Springer-Verlag. 29, 30, 34

[116] Loïc Cudennec, Gabriel Antoniu, and Luc Bougé. CoRDAGe: Towards Transparent Management of Interactions Between Applications and Ressources. In *STHEC/ICS 2008*, Island of Kos, Aegean Sea, Greece, June 2008. 29

[117] W.R. Otte, D.C. Schmidt, and A Gokhale. Towards an Adaptive Deployment and Configuration Framework for Component-based Distributed Systems. In *Proceedings of the 9th Workshop on Adaptive and Reflective Middleware (ARM'10)*, 2010. 29

[118] Distributed Management Task Force, Inc. (DMTF). *Open Virtualization Format (OVF).*, 2015. https://www.dmtf.org/sites/default/files/standards/documents/DSP0243_2.1.1.pdf. 29

[119] J. Mirkovic, T. Faber, P. Hsieh, G. Malaiyandisamy, and R. Malaviya. DADL: Distributed Application Description Language. *USC/ISI Technical Report# ISI-TR-664*, 2010. 29

[120] Alan Dearle, Graham Kirby, and Andrew McCarthy. A middleware framework for constraint-based deployment and autonomic management of distributed applications. *arXiv preprint arXiv:1006.4733*, 2010. 29

[121] Mohamed El Amine Matougui and Sébastien Leriche. A middleware architecture for autonomic software deployment. In *ICSNC'12: The Seventh International Conference on Systems and Networks Communications*, pages 13–20. XPS, 2012. 29

[122] Brice Goglin. Managing the topology of heterogeneous cluster nodes with hardware locality (hwloc). In *High Performance Computing & Simulation (HPCS), 2014 International Conference on*, pages 74–81. IEEE, 2014. 29

[123] Eddy Caron, Florent Chuffart, and Cédric Tedeschi. When self-stabilization meets real platforms: An experimental study of a peer-to-peer service discovery system. *Future Generation Computer Systems*, 29(6):1533–1543, 2013. 30

[124] Z. Xu, S. T. Hedetniemi, W. Goddard, and P. K. Srimani. A synchronous self-stabilizing minimal domination protocol in an arbitrary network graph. *IWDC 2003: Distributed Computing*, pages 832–832, 2003. 30

[125] Dominik Gall, Riko Jacob, Andrea Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. A note on the parallel runtime of self-stabilizing graph linearization. *Theor. Comp. Sys.*, 55(1):110–135, 2014. 30

[126] Riko Jacob, Stephan Ritscher, Christian Scheideler, and Stefan Schmid. A Self-stabilizing and Local Delaunay Graph Construction. In Yingfei Dong, Ding-Zhu Du, and Oscar H. Ibarra, editors, *ISAAC*, volume 5878 of *Lecture Notes in Computer Science*, pages 771–780. Springer, 2009.

[127] Riko Jacob, Andrea Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. SKIP+: A Self-Stabilizing Skip Graph. *J. ACM*, 61(6):36:1–36:26, December 2014.

[128] A. K. Datta, L. L. Larmore, and P. Vemula. A Self-Stabilizing O(k)-Time K-Clustering Algorithm. *Computer Journal*, 2008. 30

[129] Mandicou Ba, Olivier Flauzac, Bachar Salim Haggar, Florent Nolot, and Ibrahima Niang. Self-stabilizing k-hops clustering algorithm for wireless ad hoc networks. In *Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication*, page 38. ACM, 2013.

[130] C. Johnen and L. H. Nguyen. Robust Self-Stabilizing Weight-Based Clustering Algorithm. *Theoretical Computer Science*, 410(6-7):581–594, 2009. 30

[131] Yihua Ding, James Wang, and Pradip Srimani. Self-stabilizing master-slave token circulation algorithm in undirected rings and unicyclic graphs of arbitrary size and their orientations. *International Journal of Networking and Computing*, 4(1):42–52, 2014. 30

[132] Jalel Ben-Othman, Karim Bessaoud, Alain Bui, and Laurence Pilard. Self-stabilizing algorithm for efficient topology control in wireless sensor networks. *Journal of Computational Science*, 4(4):199–208, 2013. 30

[133] N. Mitton, E. Fleury, I. Guerin L., and S. Tixeuil. Self-Stabilization in Self-Organized Multihop Wireless Networks. In *ICDCSW'05: Proceedings of the Second International Workshop on Wireless Ad Hoc Networking (WWAN)*, pages 909–915, Washington, DC, USA, 2005. IEEE Computer Society. 30

[134] Shmuel Katz and Kenneth J Perry. Self-stabilizing extensions for meassage-passing systems. *Distributed Computing*, 7(1):17–26, 1993. 30

[135] Felix C Gärtner. A survey of self-stabilizing spanning-tree construction algorithms. Technical report, 2003. 30

[136] Stephane Rovedakis. *Algorithmes auto-stabilisants de constructions d'arbres couvrants*. PhD thesis, PhD thesis, Université d'Evry-Val-d-Essone, 2009. 165, 2009. 30

[137] Alain Bui, Ajoy K Datta, Franck Petit, and Vincent Villain. State-optimal snap-stabilizing pif in tree networks. In *Self-Stabilizing Systems, 1999. Proceedings. 19th IEEE International Conference on Distributed Computing Systems Workshop on*, pages 78–85. IEEE, 1999. 30

[138] Luc Onana Alima, Joroy Beauquier, Ajoy K Datta, and Sébastien Tixeuil. Self-stabilization with global rooted synchronizers. In *Distributed Computing Systems, 1998. Proceedings. 18th International Conference on*, pages 102–109. IEEE, 1998. 30

[139] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Uniform Dynamic Self-Stabilizing Leader Election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424–440, 1997. 30

[140] Ankh Arora and Mohamed Gouda. Distributed reset. *Computers, IEEE Transactions on*, 43(9):1026–1038, 1994. 30

[141] Alain Cournier, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Snap-Stabilizing PIF Algorithm in Arbitrary Networks. In *ICDCS*, pages 199–206, 2002. 30

[142] Pushpinder Kaur Chouhan. *Automatic Deployment for Application Service Provider Environments*. PhD thesis, Ecole Normale Supérieure de Lyon, 2006. 34

[143] Arnaud Legrand, Olivier Beaumont, Loris Marchal, and Yves Robert. Optimizing the steady-state throughput of Broadcasts on heterogeneous platforms. Technical Report RR-4871, INRIA, July 2003. 34

[144] Benjamin Depardon. *Contribution to the Deployment of a Distributed and Hierarchical Middleware Applied to Cosmological Simulations*. Thesis, École Normale Supérieure de Lyon, october 2010. 34

[145] Erlang/OTP. http://www.erlang.org, 2015. 58

[146] Enrique Vidal, Frank Thollard, Colin De La Higuera, Francisco Casacuberta, and Rafael C Carrasco. Probabilistic finite-state machines-part ii. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 27(7):1026–1039, 2005. 62

[147] Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. Modeling How Students Learn to Program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 153–160, New York, NY, USA, 2012. ACM. 63

[148] Gowtham Bellala, Manish Marwah, Amip Shah, Martin Arlitt, and Cullen Bash. A Finite State Machine-based Characterization of Building Entities for Monitoring and Control. In *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, BuildSys '12, pages 153–160, New York, NY, USA, 2012. ACM. 63

[149] Giordano Pola, Maria D. Di Benedetto, and Elena De Santis. Arenas of Finite State Machines, December 2011. 63

[150] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM (JACM)*, 30(2):323–342, 1983. 63