Un algorithme auto-stabilisant pour le déploiement auto-adaptatif d'un intergiciel hiérarchique : spécification, preuve, simulations

Maurice-Djibril Faye, Eddy Caron, Ousmane Thiare

Un algorithme auto-stabilisant pour le déploiement auto-adaptatif d'un intergiciel hiérarchique

spécification, preuve, simulations

Maurice Djibril Faye(*),(**), Eddy Caron(**), Ousmane Thiaré(*)

(*) UFR SAT, section Informatique
Université Gaston Berger
St-Louis
Sénégal
Ousmane.Thiare@ugb.edu.sn

(**) LIP, ENS de Lyon - INRIA - UCB Lyon 5668
Université de Lyon
Lyon
France
{Maurice.Faye, Eddy.Caron}@ens-lyon.fr

RÉSUMÉ. Dans cette article, nous nous intéressons aux moyens de rendre le déploiement d'un intergiciel auto-adaptatif. Le type d'intergiciel que nous avons considéré ici est hiérarchique (structure de graphe) et distribué. Les infrastructures de grilles/cloud étant dynamiques (perte et ajout de nœuds), un déploiement statique n'est pas la solution idéale car en cas de panne on risque de tout reprendre à zéro, ce qui est coûteux. Nous avons donc proposé un algorithme auto-stabilisant pour que l'intergiciel puisse retrouver un état stable sans intervention extérieure, au bout d'un temps fini, lorsqu'il est confronté à des pannes transitoires. Pour évaluer ces algorithmes, nous avons conçu un simulateur. Les résultats des simulations montrent qu'un déploiement, sujet à des pannes transitoires, s'auto-adapte.

ABSTRACT. *A définir par la commande* `\abstract{...}`

MOTS-CLÉS : Systèmes distribués, auto-stabilisation, intergiciel, DIET, simulateur, Machine à états finis, déploiement.

KEYWORDS : *A définir par la commande* `\keywords{...}`

1. Introduction

Un système distribué est une collection d'entités de calcul, autonomes, interconnectées [2]. De tels systèmes permettent d'échanger des données, de partager des ressources, d'améliorer nos capacités de calcul [20, 24], 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. Ils sont caractérisés par l'hétérogénéité des éléments qui les composent.

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) [3], 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 [8, 25, 29]. Pour bénéficier de leurs services, un intergiciel doit d'abord être déployé.

Le déploiement de logiciel [11, 5] 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" [5]. Il désigne l'ensemble des tâches à exécuter pour rendre un système logiciel fonctionnel. C'est une tâche complexe, surtout sur une infrastructure distribuée.

L'environnement sur lequel les intergiciels sont déployés évolue, de même que les intergiciels eux mêmes en tant que système dynamique. 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 ou de rendre un système auto-adaptatif [22, 14]. 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.

1.1. 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.

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, 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.

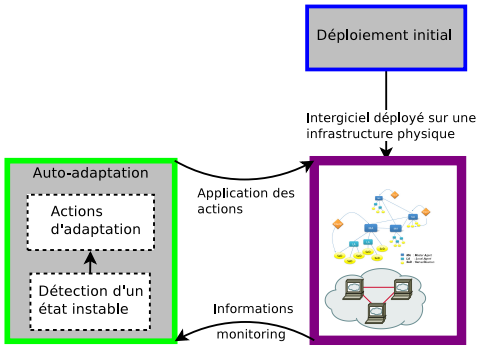

Figure 1. *Vue générale d'un déploiement auto-adaptatif*

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 (parfois avec une qualité dégradée) pour lequel il est déployé.

A chaque fois que le déploiement est 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.

1.2. L'intergiciel DIET

L'intergiciel DIET [9] nous sert de cas d'utilisation, et les travaux décrits dans ce manuscrit lui sont appliqués.

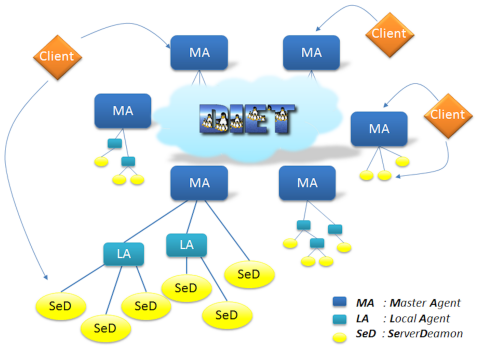

Figure 2. *Hierarchie multi-MA de* DIET.

DIET est un intergiciel GridRPC [26]. Un des objectifs de l'API GridRPC [28] est de définir clairement une syntaxe et une sémantique pour les GridRPC qui sont une extension des Remote Procedure Call (RPC) [4] appliquée au domaine des grilles de calcul. L'architecture par composant de DIET est structurée de manière hiérarchique pour améliorer le passage à l'échelle comme illustrée à la Fig. 2. DIET est implémenté en CORBA [21] et 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 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 plate-forme multi-MA.

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 [22]. Nous commentons les différents modules qui sont numérotés, comme indiqué sur la Figure 3.

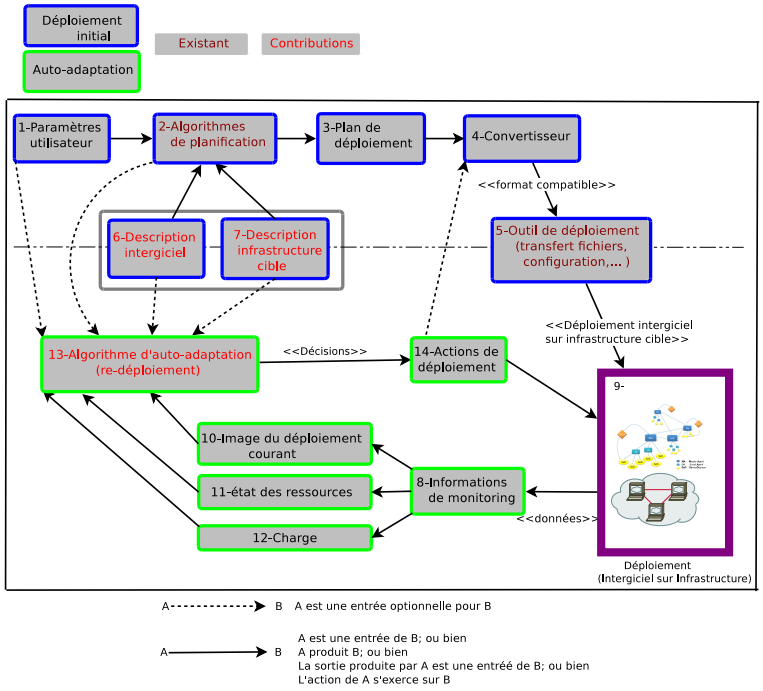

Figure 3. *Architecture pour le déploiement auto-adaptatif d'intergiciel*

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 [11, 27] 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.

Plusieurs travaux liés à l'intergiciel DIET ont été réalisés. Ceux décrits dans [13, 15] proposent des moyens d'obtenir un plan de déploiement initial pour l'intergiciel DIET. Des algorithmes de planning y sont proposés pour obtenir de manière automatique, un plan de déploiement sur des plates-formes homogènes ou hétérogènes. GoDIET [7] est un outil de déploiement spécifique à DIET. Cependant, d'autres outils de déploiement non spécifiques à DIET comme ADAGE [23] et TUNe [6] peuvent être utilisés [15] pour le déployer.

3. Déploiement initial

Une fois l'intergiciel 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. Nous avons donc deux problématiques :

– réaliser un déploiement initial;

– gérer l'adaptation du déploiement obtenu.

D'une manière générale, les besoins pour un déploiement initial de DIET sont résumés par la Figure 4.

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 four-

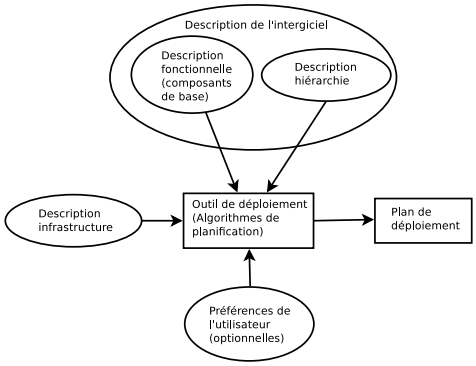

Figure 4. *Déploiement initial de* DIET

nissent 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 (Figure 3). Ces descriptions sont détaillées dans [10].

4. Auto-adaptation

Dans cette section, nous décrivons un algorithme auto-adaptatif 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. Une preuve du caractère auto-stabilisant de l'algorithme est aussi proposée.

4.1. 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.

Le code de l'algorithme auto-adaptatif est constitué d'un ensemble fini de règles de la forme :

si $(Garde_i)$ **alors** $Action_i$

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.

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

4.2. 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. section 1.2) qui nous sert de cas d'utilisation.

Définition 1 (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 2 (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. 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.

4.3.1. 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

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

if $Client \ \wedge \ (MA\_lost == Vrai) \ \wedge \ (\#MA == 0)$ **then**
  créer un fils MA ;
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

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

4.3.2. 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

if $MA \ \wedge \ (\#MA\_children == 0) \ \wedge \ (\#MA \ > \ 1)$ **then**
  $\#MA = \#MA - 1$;
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.

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 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 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

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

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

MA règle 8: R8

1 **if** $MA \wedge (MA\_charge \geq MA\_seuil\_charge)$ **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 agent comme père de tous les éléments pour chaque sous-ensemble ;
5 | les racines (2 agents) des sous arbres nouvellement créés deviennent les fils du MA ;
6 **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.

4.3.3. 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 | $\#LA = \#LA - 1$;
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 | 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 11: R11

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

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 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**
$LA \wedge (\#pre == 0) \wedge (\#\{sous{-}arbre \ : \ TypeRacine = (LA \ \vee \ MA)\} > 1)$
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** $LA \ \wedge \ (\#pre == 0) \ \wedge \ (\#\{sous - arbre \ : \ TypeRacine = (LA \ \vee \ MA)\} == 1)$ **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.

4.3.4. 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.

LA règle 14: R14

if $LA \wedge (LA\_seuil \geq LA\_seuil\_charge)$ **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 LA comme père de tous les éléments pour chaque sous-ensemble ;
    les racines (2 LA) des sous arbres nouvellement créés deviennent les fils du LA ;
end

SED règle 15: R15

if SED $\wedge$ $(\#pre == 0) \wedge (excute\ \ tche\ == Faux) \wedge (\#\{MA,\ \ LA\} == 0)$ **then**
    créer un MA comme père ;
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

if SED $\wedge$ $(\#pre == 0) \wedge (excute\ \ tche\ == Faux) \wedge (\#\{MA,\ \ LA\} > 0)$ **then**
    sélectionner un des agents (MA ou LA) comme père ;
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.

SED règle 17: R17

if SED $\wedge$ $(\#pre == 0) \wedge (excute\ \ tche\ == Vrai)$ **then**
    continuer l'exécution pour un temps maximum de T unités de temps ;
    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 */`
end

Les effets de chacune des règles sont résumés dans le Tableau 1.

Tableau 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. Preuve d'auto-stabilisation de l'algorithme

Le concept d'auto-stabilisation dans les systèmes distribués a été introduit en 1974 par E. W. Dijkstra [16].

Définition 3 (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 [17].

Pour valider le caractère auto-stabilisant d'un algorithme, il faut montrer que les propriétés de **convergence** et de **clôture** [1] sont vérifiées.

Définition 4 (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.*

Définition 5 (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.*

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), rupture de liens entre voisins.

Considérant l'algorithme distribué, spécifié sous la formes des règles définies à la Section 4.3, nous allons fournir une esquisse de preuve (sketch of proof), montrant que l'algorithme est auto-stabilisant (Définition 3); 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 4) et de **clôture** (Définition 5) de l'algorithme.

4.4.1. 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 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. $CQFD$.

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. $CQFD$.

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. $CQFD$.

preuve de $\mathcal{P}_4$:

– la règle SED R17 montre que tout calcul par un SED se termine au bout d'un temps fini. $CQFD$.

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.

4.4.2. 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. $CQFD$

5. Simulateur

Nous avons conçu un simulateur Ad hoc pour faire une évaluation de certaines propriétés de l'algorithme décrit dans la section précédente (comme le temps de stabilisation).

Le simulateur a été programmé avec le langage Erlang [12, 18]. 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.

Pour résumer, le simulateur peut réaliser plusieurs actions. Les plus importantes, et celles que nous avons utilisées le plus sont les suivantes : créer un déploiement (une hiérarchie de processus, chaque processus représentant une instance d'un élément de DIET), créer un événement de simulation (capable de rendre instable un déploiement stable), afficher l'état global du déploiement (stable ou instable), de manière périodique, avec le nombre d'instances stables, le nombre d'instances instables et le nombre total d'instances du déploiement.

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.

Un serveur de détection de la stabilité du déploiement : Il sert à détecter l'état global d'un déploiement (stable ou instable).

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.

Comme décrit dans la Section 1.2, 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) [30] 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. Chaque AEF est à la fois un serveur et un client.

La Figure 5 illustre le comportement générique d'un AEF et les types de transitions qu'il peut effectuer entre l'état initial (pris une seule fois durant le cycle de vie), l'état stable (un seul état stable) et les différents états instables (le nombre dépend du type de l'élément de DIET simulé).

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.

Le simulateur est décrit de manière détaillée dans [19](Chapitre 5).

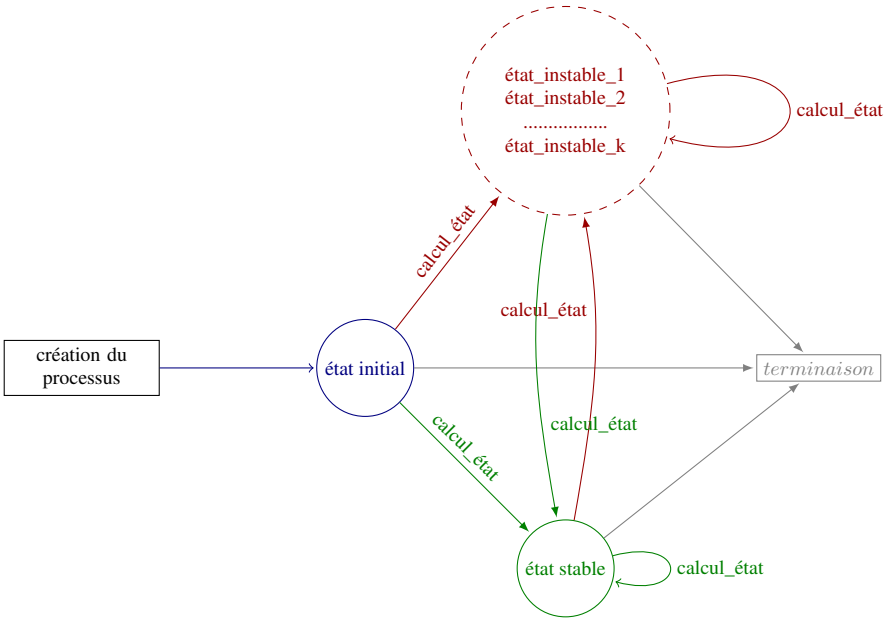

Figure 5. *Transitions possibles entre les états d'un AEF*

6. Simulations et résultats

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).

Pour toutes les simulations effectuées, nous avons utilisé cinq machines virtuelles Erlang (que nous appelons aussi nœuds erlang) déployées sur la même machine physique. 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.

Par manque d'espace, nous allons décrire une seule simulation.

6.1. 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.

Le courbe sur la Figure 7 montrent les variations des valeurs enregistrées au cours de la simulation.

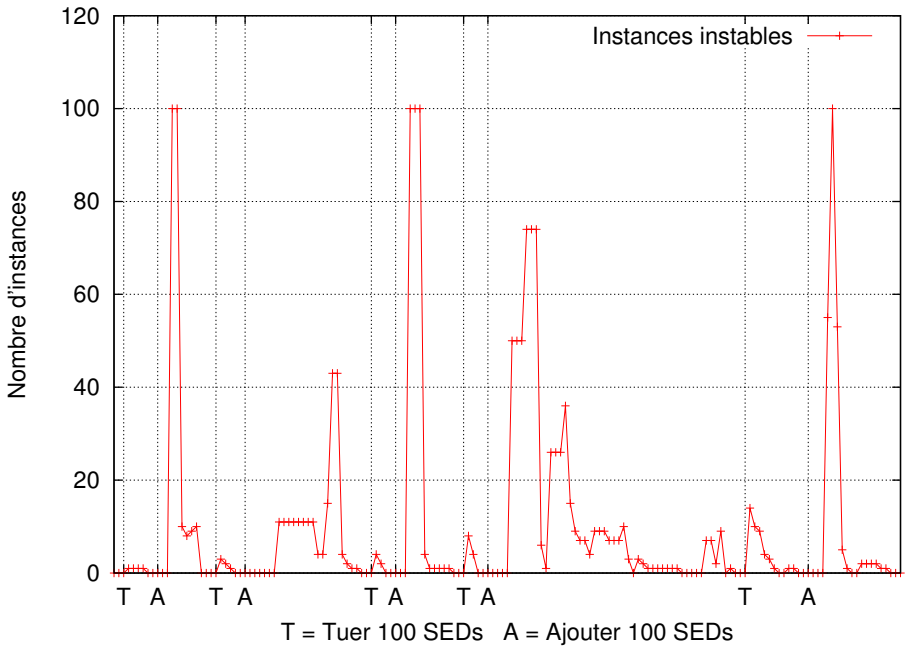

Figure 6. *Alternance des événements "tuer 100* SEDs *(T)" et "ajouter 100* SEDs *(A)". variation du nombre d'instances instables.*

On peut faire quelques observations sur la courbe de la Figure 7. 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 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.

7. Conclusion

Dans cet article, nous avons présenté un travail visant à ajouter des capacités d'auto-adaptation à un intergiciel de grille/cloud existant. A cette fin, nous avons proposé un algorithme distribué dont l'objectif est de ramener tout déploiement instable à un état stable au bout d'un nombre fini d'étapes. Nous avons proposé une preuve du caractère auto-stabilisant de l'algorithme. Nous avons aussi conçu un simulateur pour la validation de l'algorithme. Les simulations effectuées montrent que le système s'auto-adapte et recouvre un état stable après une perturbation.

8. Bibliographie

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

H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. Wiley-Interscience, 2004.

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

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

Raja Boujbel. *Déploiement de systèmes répartis multi-échelles : processus, langage et outils intergiciels*. Thèse de doctorat, Université de Toulouse, Toulouse, France, janvier 2015.

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.

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.

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.

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.

Eddy Caron, Maurice Djibril Faye, Jonathan Rouzaud-Cornabas, and Ousmane Thiare. Grid middleware modelling for self-adaptive deployment. In *Autonomous Decentralized Systems (ISADS), 2013 IEEE Eleventh International Symposium on*, pages 1–6, 2013.

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.

Francesco Cesarini and Simon. Thompson. *Erlang Programming.* O'Reilly Media, Inc., 2009.

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

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.

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

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

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

Erlang/OTP. http://www.erlang.org, 2015.

Maurice Djibril Faye. *Déploiement auto-adaptatif d'intergiciel sur plate-forme élastique*. PhD thesis, École Normale Supérieure de Lyon, 2015.

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

Object Management Group. *The Common Object Request Broker (CORBA): Architecture and Specification*. Object Management Group, 1995.

J.O. Kephart and D.M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

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.

Peter Mell and Tim Grance. The nist definition of cloud computing. 2011.

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

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.

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.

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.

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.

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.