Coordination de la Gestion autonome de la Réparation et du Dimensionnement d'un Système multi-niveaux par Contrôle Discret

Soguy Mak-Karé Gueye, Noël de Palma, Eric Rutten, Alain Tchana, Nicolas Berthier

Coordination de la Gestion Autonome de la Réparation et du Dimensionnement d'un Système multi-niveaux par Contrôle Discret

Soguy Mak-Karé Gueye[1,3], Noël De Palma[1], Éric Rutten[3], Alain Tchana[2], Nicolas Berthier[4]

soguy-makkare.gueye@imag.fr, noel.depalma@imag.fr, eric.rutten@inria.fr, Alain.Tchana@enseeiht.fr, nicolas.berthier@inria.fr

[1] *ERODS/LIG - Bât. C, 220 rue de la Chimie, 38400 St Martin d'Hères FRANCE*
[2] *INP Toulouse - Institut National Polytechnique de Toulouse FRANCE*
[3] *CTRL-A/INRIA, Grenoble Rhône-Alpes, Inovallée, 655 av. de l'Europe, Montbonnot, 38334 St Ismier Cedex FRANCE*
[4] *SUMO/INRIA/IRISA Rennes, Campus Universitaire de Beaulieu, 35042 Rennes Cedex FRANCE*

RÉSUMÉ. *Les systèmes informatiques sont de plus en plus distribués et hétérogènes, ce qui rend leur administration manuelle difficile et source d'erreurs. L'administration autonome a été proposée comme solution à ce problème. Elle consiste à automatiser l'administration des systèmes à l'aide de boucles de contrôle appelées gestionnaires autonomes. De nombreux travaux de recherche se sont intéressés à l'automatisation des fonctions d'administration de systèmes informatiques et aujourd'hui, beaucoup de gestionnaires autonomes sont disponibles. Toutefois, les gestionnaires autonomes existants sont, la plupart, spécialisés dans la gestion de quelques aspects d'administration. Cela rend nécessaire leur coexistence pour une gestion globale. La coexistence de plusieurs gestionnaires permet la gestion de plusieurs aspects, mais nécessite des mécanismes de coordination afin d'éviter des décisions incohérentes. Nous étudions l'utilisation de techniques de contrôle pour la conception de contrôleurs de coordination, nous utilisons la programmation synchrone qui fournit des méthodes formelles, et la synthèse de contrôleur discret pour automatiser la construction de contrôleur. Nous évaluons et démontrons les avantages de notre approche par la coordination de gestionnaires autonomes dédiés à la gestion de la disponibilité, et à la gestion de la performance et l'optimisation de ressources pour un système multiniveau.*

ABSTRACT. *Computing systems have become more and more distributed and heterogeneous, making their manual administration difficult and error-prone. The Autonomic Computing approach has been proposed to overcome this issue, by automating the administration of computing systems with the help of control loops called autonomic managers. Many research works have investigated the automation of the administration functions of computing systems and today many autonomic managers are available. However the existing autonomic managers are mostly specialized in the management of few administration concerns. This makes necessary the coexistence of multiple autonomic managers for a complete system management. The coexistence of several such managers is required to address multiple concerns, yet requires coordination mechanisms to avoid incoherent administration decisions. We investigate the use of control techniques for the design of coordination controllers, for which we exercise synchronous programming that provides formal semantics, and discrete controller synthesis to automate the construction of the controller. The paper details an application of the latter approach for the design of a coordination controller to orchestrate the execution of four self-repair and two self-sizing managers that address the availability and performance of a multi-tiers replication-based system. We evaluate and demonstrate the benefits of our approach by coordinating autonomic managers that address the availability, the performance and the resource optimization within a multi-tiers system.*

MOTS-CLÉS : *Systèmes informatiques, administration autonome, boucles de contrôle, systèmes à événements discrets, synthèse de contrôleur discret, programmation synchrone*

KEY WORDS : *Computer systems, autonomic computing, control loops, discrete event systems, discrete controller synthesis, synchronous programming*

1. Introduction

L'administration des systèmes informatiques est devenue une tâche de plus en plus difficile à assurer efficacement de manière manuelle. Cette difficulté résulte de la complexité des systèmes qui sont de plus en plus basés sur une architecture distribuée et à large échelle. Pour faciliter l'administration des systèmes, une approche a été proposée : *l'administration autonome [KEP 03]*. Cette approche, introduite en 2001 par IBM, consiste à automatiser les fonctions d'administration afin de minimiser l'intervention humaine et d'améliorer la réactivité quant à la détection de changements et l'application d'opérations de correction. Les fonctions d'administration sont implantées par des éléments logiciels appelés gestionnaires autonomes. Ces derniers assurent les tâches d'administration à l'exécution. De nombreux travaux de recherche se sont intéressés à la conception de gestionnaires autonomes. Aujourd'hui, plusieurs gestionnaires sont disponibles et assurent de manière cohérente les fonctions d'administration qu'ils implémentent. Cependant aucun gestionnaire n'assure intégralement l'ensemble des fonctions, ce qui rend utile l'utilisation de

plusieurs gestionnaires en parallèle pour assurer une administration complète. Toutefois, il peut être nécessaire de coordonner l'exécution des gestionnaires présents dans un même système. En effet, ils sont généralement conçus indépendamment avec une connaissance partielle du système administré. De ce fait, les actions exécutées par un gestionnaire pour atteindre ses objectifs peuvent causer la violation des objectifs des autres gestionnaires qui vont réagir à leur tour pour atteindre leurs objectifs. Cela peut mener à des actions de reconfiguration en chaîne qui ne garantissent pas nécessairement que les objectifs de tous les gestionnaires soient atteints ultimement, ce qui peut conduire à une instabilité de l'état du système administré.

La coordination de l'exécution de gestionnaires autonomes peut être considérée comme un problème de synchronisation et de contrôle de leurs actions ; ces aspects ont été largement étudiés en théorie du contrôle discret. De ce fait, nous proposons une approche basée sur des techniques issues du contrôle discret, la programmation synchrone et la synthèse de contrôleur discret, pour la conception et la validation de la coordination de gestionnaires. L'intérêt de cette approche est la génération automatique de contrôleurs de coordination corrects par construction et qui restreignent le moins possible les gestionnaires. Ce papier présente une application de notre approche de coordination pour la gestion d'un système multiniveau. Nous coordonnons l'exécution des gestionnaires autonomes pour gérer de manière cohérente la performance, l'optimisation de ressources et la disponibilité du système. La performance peut être considérée comme étant la capacité à répondre à plusieurs requêtes simultanément en un délai raisonnable et la disponibilité comme étant la capacité à résister aux pannes.

Le reste du papier est organisé comme suit. La section 2. présente l'état de l'art en coordination de gestionnaires autonomes ainsi que les techniques et les outils sur lesquels repose notre approche pour la conception d'un contrôleur de coordination. La section 3. présente un exemple de système multiniveau autonome ; elle décrit également des problèmes d'administration qui peuvent exister lorsque les gestionnaires autonomes du système ne sont pas coordonnés. La section 4. présente la conception d'un contrôleur de coordination avec notre approche pour le contrôle de l'exécution des gestionnaires du système présenté à la section 3.. La section 5. présente l'évaluation du contrôleur obtenu. La section 6. conclut le document. Elle rappelle le contexte, l'approche de coordination et donne des perspectives que nous envisageons.

2. État de l'art

2.1. *Coordination de gestionnaires autonomes*

La coordination de gestionnaires autonomes a été étudiée dans plusieurs travaux de recherche. Des approches ont été proposées pour implémenter et mettre en oeuvre la coordination. Parmi les solutions proposées pour l'implémentation

de la politique de coordination figurent des fonctions d'utilité, des fonctions d'optimisation, des protocoles de consensus et des règles (conditions-action/priorité). Les approches proposées par [KUM 09, DAS 08] sont basées sur des fonctions d'utilité. [KUM 09] s'intéresse à la gestion unifiée de la plate-forme d'exécution physique et de l'environnement virtualisé dans un data-centre pour empêcher l'exécu-tion d'actions redondantes et inutiles. Une fonction de distribution cumulative est utilisée pour calculer la probabilité que les serveurs physiques continuent à fournir suffisamment de ressources aux machines virtuelles qu'ils hébergent dans le futur durant une période de temps déterminée. [DAS 08] s'intéresse à la gestion de la performance et de la consommation énergétique dans un data-centre basé sur la répartition de charge. La politique de coordination des agents est basée sur un modèle du système et une fonction d'utilité multi-critère sur laquelle est basée le contrôle de l'agent de coordination.

Parmi les approches basées sur des règles figurent [NAT 07] et [AND 04]. [NAT 07] s'intéresse à la gestion de la consommation énergétique des ressources physiques d'un centre de données en considérant les politiques de gestion de ressources intégrées dans les machines virtuelles. Elle permet également la gestion globale de la puissance de calcul pour supporter l'environnement virtualisé en interprétant l'état des machines virtuelles pour la prise de décision. Le framework `Accord` proposé par [AND 04] est basé sur les modèles à composants pour la construction d'applications autonomes. Les applications autonomes sont formées par la composition de composants autonomes qui intègrent des agents. Les agents implémentent des règles d'administration. Les décisions conflictuelles entre les agents sont résolues grâce à des priorités.

Les approches proposées dans [TCH 09] et [KAS 10] sont basées sur des fonctions d'optimisation. [TCH 09] propose une extension de l'architecture `GANA` pour assurer la stabilité des boucles de contrôle dans un réseau autonome. Il introduit un module de synchronisation des actions (`ASM`) à intégrer dans certains des éléments de décision. Ce module est responsable de la coordination. Des indicateurs de performance sont définis, chacun avec un poids qui indique son importance. Un `ASM` choisit, parmi un ensemble des actions à synchroniser, les actions qui optimisent l'ensemble des indicateurs de performance. [KAS 10] s'intéresse à la stabilité dans un réseau autonome équipé de plusieurs boucles de contrôle. Elle propose la théorie du jeu pour l'étude de la stabilité des comportements autonomes. La conception repose sur l'architecture `GANA` qui permet une structuration hiérarchique des boucles de contrôle. `GANA` permet également la résolution de conflits via un module, une fonction de synchronisation des actions [TCH 09].

Parmi les approches basées sur le consensus, nous pouvons citer [ALD 11]. Elle propose un framework, `LIBERO`, qui permet l'implémentation de modèle comportemental de type *`Pipeline`* et *`Farm`* avec plusieurs gestionnaires autonomes coordonnés pour la gestion de plusieurs aspects non fonctionnels. La coordination des gestionnaires repose sur un consensus. Un gestionnaire qui planifie l'exécution d'une action demande la validation de l'action par les autres gestionnaires.

Nous constatons, à travers toutes ces approches ci-dessus, que la coordination requiert une synchronisation partielle de l'exécution des gestionnaires ainsi qu'un contrôle de leurs actions ; et ces aspects ont été largement étudiés en théorie du contrôle. La théorie du contrôle et les outils qui en résultent ont récemment commencé à être utilisés pour les systèmes informatiques. La plupart des cas d'utilisation reposent sur des modèles continus ; généralement pour traiter des aspects quantitatifs [HEL 04, RAG 08, HEO 11]. Des utilisations plus récentes reposent sur des modèles de la famille des systèmes à événements Discrets [CAS 06] sur lesquels des propriétés logiques sont étudiées. un contrôle. Ils utilisent les notions de contrôle supervisé [RAM 87], généralement pour garantir des propriétés logiques ou à des fins de synchronisation [WAN 07]. Le contrôle discret est basé sur des modèles de la forme de systèmes de transitions, comme les réseaux de Pétri ou automates. Il fournit des langages de haut niveau pour la spécification formelle de système ; et des outils de vérification et de synthèse de contrôleur. Nous nous intéressons à l'application des techniques issues de la théorie du contrôle pour la coordination de gestionnaires autonomes. Nous utilisons la synthèse de contrôleur discret [MAR 00] pour la construction automatique de contrôleur de coordination. La construction du contrôleur est basée sur le modèle de la coexistence des gestionnaires et une spécification de contrôle qui peuvent être réalisés par programmation synchrone [BEN 03].

2.2. *Contrôle discret*

Cette section présente les méthodes et outils issus du contrôle discret sur lesquels est basée notre approche de coordination. Nous utilisons la programmation synchrone et la synthèse de contrôleur discret. La programmation synchrone est un formalisme de modélisation et dispose de langages de haut niveau qui permettent la conception et la validation formelle de systèmes réactifs. Ces langages permettent une représentation formelle du comportement d'un système à partir de laquelle des propriétés peuvent être analysées par vérification, ou assurées par une fonction de contrôle obtenue par construction en utilisant une technique de synthèse de contrôleur.

2.2.1. *Langages synchrone*

Les langages synchrones [BEN 03] sont des langages de haut niveau introduits au début des années 1980 pour la conception de systèmes réactifs. Ils permettent une spécification formelle du comportement d'un système et disposent d'outils d'analyse offrant des garanties à la compilation sur le comportement du système à l'exécution. Ces langages reposent sur une hypothèse appelée l'*hypothèse synchrone*. Cette hypothèse fournit un niveau d'abstraction où les réactions – calculs et/ou communications – du système ont une durée nulle. Un système peut être décrit en faisant l'hypothèse qu'il réagit instantanément. Cela permet la spécification du fonctionnement d'un système sans considérer

les contraintes liées à l'architecture sur laquelle il est exécuté. L' évolution est basée sur la notion d'instant. Les événements internes et les événements en sortie sont datés précisément en respect avec le flux des événements en entrée. Cela permet au moment de la spécification de raisonner en temps logique sans tenir compte des temps réels des calculs et des communications. Cela permet également le raisonnement par rapport aux aspects de déterminisme et de concurrence sur le comportement du système souvent décrit comme la composition parallèle de sous-systèmes.

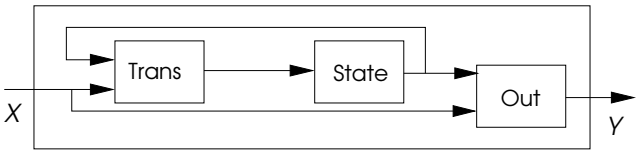

Figure 1: Système de transitions

Les langages synchrones sont basés sur différents modèles de programmation : impératif (**ESTEREL** [BER 92]), flot de données (**LUSTRE** [HAL 91]). Toutefois, les programmes implémentent le même comportement de base illustré par la figure 1. Il s'agit d'un système de transitions qui prend des entrées X et produit des sorties Y. À chaque pas, les entrées X ainsi que l'état courant de la mémoire du système sont utilisés par la fonction de transition (fonction `Trans`) pour calculer l'état suivant du système. Ils sont également utilisés par la fonction de sortie (fonction `Out`) pour calculer les sorties Y du système. La mise à jour de la mémoire du système est assurée par la fonction (fonction `State`).

2.2.2. *Synthèse de contrôleur discret (SCD)*

Parmi les méthodes de conception et de validation, la synthèse de contrôleur est l'une des plus séduisantes. Elle permet de raffiner une spécification incomplète du comportement d'un système de sorte qu'il satisfasse une (des) propriété(s) non satisfaite(s) par la spécification initiale. La synthèse de contrôleur, issue de la théorie du contrôle, permet d'obtenir une logique de contrôle par construction [MAR 00]. Elle est basée sur des méthodes formelles et permet d'obtenir un contrôleur qui restreint le système à contrôler pour assurer le respect de propriétés. Elle requiert un modèle du comportement du système à contrôler et une spécification des propriétés exprimées en terme d'objectifs de contrôle. Le modèle du système décrit de manière formelle tous les comportements possibles, les comportements corrects et incorrects vis-à-vis des propriétés désirées. Il doit également fournir des entrées de contrôle, des points de choix sur son comportement. Le système à contrôler de même que les objectifs de contrôle sont généralement modélisés au moyen de systèmes de transitions étiquetés ou automates [RAM 87], et les langages synchrones sont bien adaptés.

La synthèse de contrôleur construit une logique de contrôle, une contrainte sur les valeurs des entrées contrôlables du système à contrôler, en fonction de son état courant et des valeurs des entrées incontrôlables, de sorte que tous les comportements autorisés satisfassent les propriétés définies comme objectifs de contrôle. La logique de contrôle construite restreint le moins possible le fonctionnement du système à contrôler.

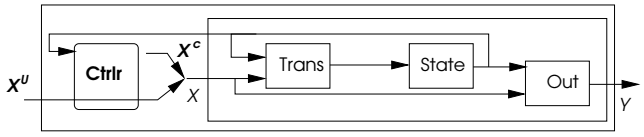

Figure 2: Système de transitions contrôlé

La figure 2 présente un exemple où le système décrit à la figure 1 est le système à contrôler. Le système de transition prend en entrée $X = \{Xu \cup Xc\}$ à chaque réaction. Les entrées Xu sont incontrôlables alors que les entrées Xc sont contrôlables. Le contrôleur (logique de contrôle) `Ctrlr`, obtenu par synthèse de contrôleur, produit les valeurs à affecter aux variables contrôlables Xc en se basant sur les valeurs des entrées incontrôlables Xu et l'état courant du système afin d'assurer les objectifs de contrôle.

Le code exécutable correspondant au modèle contrôlé décrit à la figure 2 constitue un contrôleur réel. Il permet à l'exécution de contrôler le système modélisé lorsqu'ils sont couplés.

2.2.3. *Heptagon/BZR*

`Heptagon/BZR` [1] [DEL 10a, DEL 13] est un langage de programmation appartenant à la famille des langages synchrones. Il permet de décrire un système à base d'équations de flot de données et d'automates [MAR 03]. Il facilite la description d'un système constitué de sous-systèmes par la composition parallèle et hiérarchique [COL 05] des modèles des différentes sous-systèmes. Les modèles évoluent en parallèle de manière synchrone : une réaction globale implique une réaction locale de chacun des sous-modèles.

La figure 3 présente un exemple simple de modélisation avec le langage `Heptagon/ BZR`. Le programme `tache-comp` modélise une tâche (e.g., un processus). L'activation de cette tâche peut être différée selon une condition représentée par l'entrée `c`. Initialement la tâche n'est pas active. Cet état est représenté dans le modèle par l'état `Inactive`. Dans cet état, lorsque l'activation de la tâche est demandée (`req` à vrai), si l'activation est autorisée (`c` à vrai) la tâche est activée et passe dans l'état `Active` ; sinon l'activation est retardée en attente de l'autorisation. L'attente de l'autorisation est représentée par l'état `Attente`. La sortie `dem` est la commande d'activation de la tâche dans le modèle. La sortie `active` indique l'état de la tâche, elle est à vrai lorsque la tâche

[1] `http://bzr.inria.fr`

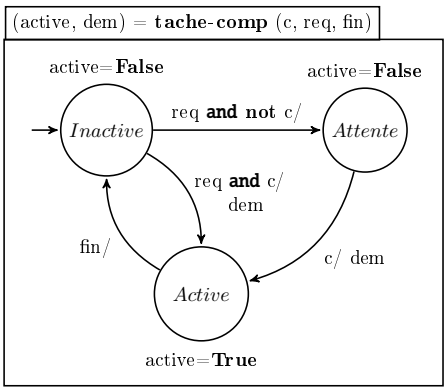

Figure 3: Modélisation avec Heptagon/BZR : Tâche différable

```
node tache-comp(req, c, fin : bool) returns (active, dem : bool)
let automaton
    state Inactive do
              active = false ; dem = req and c
          until dem then Active
              | req and not c then Attente
    state Attente do
             active = false ; dem = c
           until dem then Active
    state Active do
             active = true ; dem = false
          until fin then Inactive
   end
tel
```

Figure 4: Programme Heptagon/BZR

est en cours d'exécution. La fin de l'exécution de la tâche est représentée par l'entrée **fin** à vrai. La figure 4 présente la structure du code du programme écrit en Heptagon/BZR. Heptagon/BZR considère le premier état défini comme étant l'état initial, ici il s'agit de l'état Inactive.

La spécificité du Heptagon/BZR est qu'il intègre également l'outil de synthèse de contrôleur discret présenté dans [MAR 00] dans sa compilation. Heptagon/BZR facilite l'utilisation de cet outil de synthèse de contrôleur en introduisant la notion de contrat dans la modélisation de système. Le contrat est décrit de manière déclarative [DEL 10b] en trois parties : **assume**, **enforce** et **with**. Le contrat contient les propriétés que le fonctionnement du système doit respecter. Ces propriétés sont déclarées comme objectifs de contrôle dans la partie **enforce**. Lorsque le modèle qui décrit le fonctionnement du système à

contrôler ne garantit pas le respect des propriétés, Heptagon/BZR génère une logique de contrôle qui permet d'assurer le respect des propriétés lorsque des entrées contrôlables sont définies dans le modèle.Les entrées contrôlables dans le modèle du système sont déclarées comme variables contrôlables dans la partie **with** du contrat. La logique de contrôle qui assure le respect des propriétés détermine les valeurs à assigner à ces variables contrôlables afin de restreindre le fonctionnement aux comportements qui satisfont les propriétés. Les propriétés pertinentes concernant l'environnement d'exécution sont déclarées dans la partie **assume** du contrat. Cette information est prise en compte lors de la synthèse de la logique de contrôle.

$(\texttt{active}_1, \texttt{dem}_1, \texttt{active}_2, \texttt{dem}_2) = \texttt{deuxtaches}(\texttt{req}_1, \texttt{fin}_1, \texttt{req}_2, \texttt{fin}_2)$

assume not $(\texttt{req}_1$ **and** $\texttt{req}_2)$

enforce not $(\texttt{active}_1$ **and** $\texttt{active}_2)$

with c_1, c_2

$(\texttt{active}_1, \texttt{dem}_1) = \texttt{tache} - \texttt{comp}(\texttt{c}_1, \texttt{req}_1, \texttt{fin}_1)$;

$(\texttt{active}_2, \texttt{dem}_2) = \texttt{tache} - \texttt{comp}(\texttt{c}_2, \texttt{req}_2, \texttt{fin}_2)$

Figure 5: Heptagon/BZR contrat : exclusion mutuelle

La figure 5 illustre l'utilisation de la SCD avec Heptagon/BZR. Ce programme modélise le contrôle de l'activation de deux tâches modélisées par deux instances du programme à la figure 3. L'objectif de contrôle est d'empêcher que les deux tâches soient actives en même temps.

Cela est exprimée par la propriété : « **not** $(\texttt{active}_1$ **and** $\texttt{active}_2)$», avec $active_1$ à vrai (**true**) lorsque la tâche n°1 est active et $active_2$ à vrai (**true**) lorsque la tâche n°2 est active. Cette propriété étant l'objectif de contrôle à garantir est spécifiée dans la partie **enforce** du contrat. Les variables c_1 et c_2, étant les entrées de contrôle fournies par les modèles des tâches, sont déclarées dans la partie **with**, et vont être utilisées par la logique de contrôle construite pour empêcher l'activation de l'exécution d'une des tâches lorsque l'autre est en cours d'exécution.

A la compilation, Heptagon/BZR applique la synthèse de contrôleur. Une fois la logique de contrôle générée, il l'intègre dans le modèle et produit un programme exécutable. Ce dernier constitue un contrôleur qui permet de contrôler le système réel pour garantir le respect des propriétés à l'exécution. Heptagon/BZR permet de produire une seule solution de manière déterministe pour un problème de synthèse de contrôleur discret. Le compilateur de Heptagon/BZR favorise pour les variables contrôlables la valeur vrai (**true**) à faux (**false**) et en prenant en compte l'ordre de déclaration des variables.

Certains travaux comme [BLO 12, BLO 10] se sont intéressés aux notions de synthèse de programme. Il consiste à générer un modèle à partir de propriétés sur les entrées et sorties d'un système, exprimées en logiques temporelles. Le modèle généré est souvent représenté en termes de systèmes de transitions. Une

différence par rapport à ces travaux est que nous synthétisons une contrainte sur les variables contrôlables d'une machine à états (donnée comme modèle de l'objet à contrôler) à partir d'objectifs de contrôle. `Heptagon/BZR` est un langage impératif permettant l'écriture des automates pour les composants non encore contrôlés et déclaratif permettant de spécifier les propriétés à assurer par le contrôle. Une autre différence significative est que nous distinguons les entrées contrôlables et les entrées incontrôlables.

3. Coordination des gestionnaires d'un système multiniveau

Cette section présente l'exemple de système multiniveau considéré et deux gestionnaires autonomes pour la gestion de la disponibilité, de la performance et de l'optimisation des ressources. Les problèmes de coexistence entre les deux types de gestionnaires sont également décrits dans cette section pour montrer l'importance de la coordination de leur exécution lorsqu'ils sont présents dans le même système. des gestionnaires.

3.1. *Système multiniveau JEE*

L'exemple de système multiniveau considéré est le système JEE présenté à la figure 6. Il est constitué d'un serveur web *Apache*[2], de serveurs d'application *Tomcat*[3] dupliqués m fois, d'un serveur *Mysql-proxy*[4] et de serveurs de bases de données *Mysql*[5] également dupliqués n fois. Les requêtes entrantes sont reçues par le serveur *Apache*. Ce dernier les distribue aux serveurs *Tomcat* pour leur traitement. Les serveurs *Tomcat* accèdent aux bases de données à travers le serveur *Mysql-proxy* qui est un répartiteur de charge pour les serveurs *Mysql*. Le serveur *Mysql-proxy* distribue équitablement les requêtes de lecture aux serveurs *Mysql*.

L'une des difficultés lors du déploiement d'un système multiniveau basé sur la réplication est le dimensionnement des tiers dupliqués. En effet, la variation de la charge à traiter fait qu'il peut être difficile d'estimer le degré de réplication nécessaire au démarrage du système. La plupart du temps, une configuration statique du nombre de serveurs peut conduire à une estimation abusive du nombre de serveurs. Cela peut, peut-être, permettre d'avoir de bonnes performances mais à un coût très élevé, e.g., consommation énergétique très élevée. Ajuster dynamiquement le degré de réplication à l'exécution permet d'allouer le nombre nécessaire de serveurs en fonction du nombre de requêtes à satisfaire. De plus l'état des serveurs doit être surveillé en permanence pour détecter les

[2] `http://httpd.apache.org/`
[3] `http://tomcat.apache.org/`
[4] `http://dev.mysql.com/doc/refman/5.1/en/mysql-proxy.html`
[5] `http://www.mysql.com/`

pannes. Il est nécessaire de réparer les pannes afin d'éviter de perdre tous les serveurs.

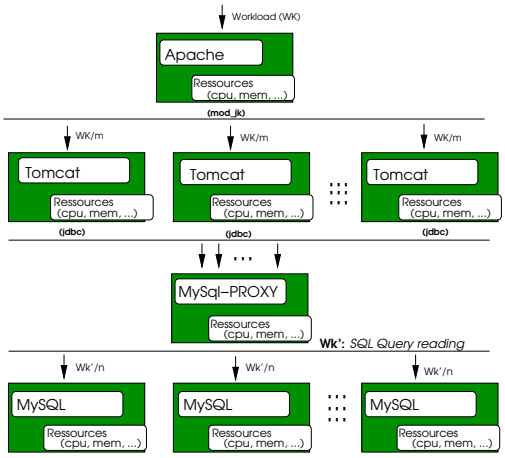

Figure 6: Application multiniveau JEE

La gestion de la performance et de la disponibilité d'un système multiniveau peut reposer sur la coexistence de gestionnaires autonomes pour le dimensionnement dynamique [TAT 06] et pour la réparation des serveurs [SIC 08, BOY 09].

3.2. *Gestionnaire d'auto-dimensionnement : Self-sizing*

Ce gestionnaire autonome est dédié au dimensionnement dynamique d'un système en fonction de la charge de travail de ce dernier. La structure du système est basée sur le canevas d'équilibrage de charge. Dans ce canevas le modèle de communication est synchrone (Client/Serveur), les serveurs sont clonés statiquement lors du démarrage du système et un aiguilleur est placé en frontal des serveurs. Le rôle de cet aiguilleur est de faire de l'équilibrage de charge entre tous les serveurs. Une requête peut donc être traitée indifféremment par n'importe lequel des serveurs. Lorsqu'un serveur reçoit une requête, il l'exécute, il met en cohérence son état avec les autres serveurs si besoin, puis il retourne le résultat de la requête au client. On considère que ce canevas s'exécute sur une grappe de machines. L'aiguilleur ainsi que chaque serveur s'exécute sur une machine différente.

Comme le montre la figure 7, le rôle de ce gestionnaire est de dimensionner dynamiquement le degré de duplication des serveurs qui constituent le système. Le dimensionnement est effectué en fonction de la charge de travail soumise au système. Il permet l'approvisionnement de ressources en cas de surcharge mais également l'optimisation de ressources en cas de sous-charge du système. Le

gestionnaire va donc ajouter ou retirer dynamiquement des clones de serveur en fonction de la charge pour traiter des surcharges ou des sous-charges du système. Il connaît la structure du système, les machines en cours d'utilisation sur lesquelles sont exécutés les serveurs. Il connaît également les machines disponibles.

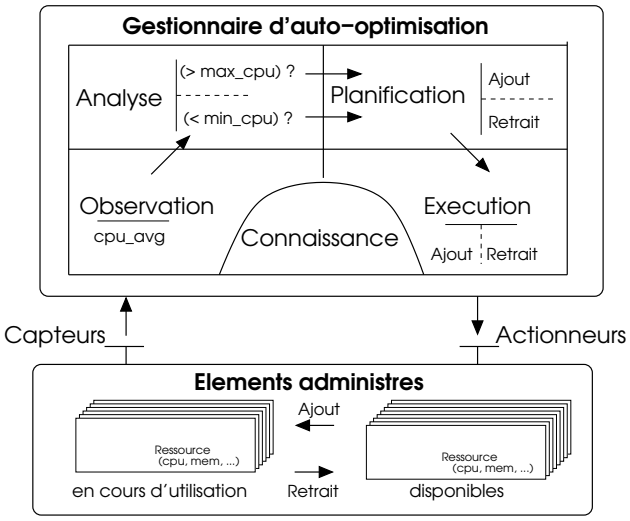

Figure 7: Gestionnaire d'auto-optimisation

Des sondes récupèrent périodiquement la charge CPU de chaque machine qui exécute un serveur dupliqué. Le gestionnaire calcule une moyenne glissante EWMA (exponentiellement pondérée) des charges CPU. Cette moyenne est utilisée pour évaluer le niveau d'utilisation des machines. Le niveau acceptable auquel il ne nécessite pas le redimensionnement du système est borné par deux seuils minimal et maximal. Le gestionnaire considère que les machines sont saturées lorsque la moyenne est supérieure au seuil maximal. Il considère les machines sous utilisées lorsque la moyenne est inférieure au seuil minimal.

Lorsqu'une surcharge du système est détectée, il démarre un nouveau serveur dupliqué sur une machine disponible et met à jour l'état de ce dernier en fonction des autres serveurs. Puis il intègre ce nouveau serveur dans la liste des serveurs dupliqués au niveau de l'aiguilleur de charge. Dans le cas d'une sous-charge, il choisit un serveur à arrêter. Il déconnecte le serveur de l'aiguilleur de charge. Il arrête le serveur et le désinstalle de la machine. Puis il remet la machine dans la liste des machines disponibles. Les systèmes administrés fournissent les actionneurs permettant d'appliquer les actions d'administration.

3.3. *Gestionnaire d'auto-réparation : Self-repair*

Ce gestionnaire autonome est dédié à la restauration d'un système ou les éléments constituant le système à l'occurrence de pannes. Le gestionnaire traite les pannes franches de machines. Il a une connaissance de la structure du système administré. Il connaît l'ensemble des machines sur lesquelles s'exécutent les éléments logiciels constituant le système. Il connaît également l'ensemble des ressources matérielles non utilisées et disponibles pour permettre la reconfiguration du système, comme le montre la figure 8.

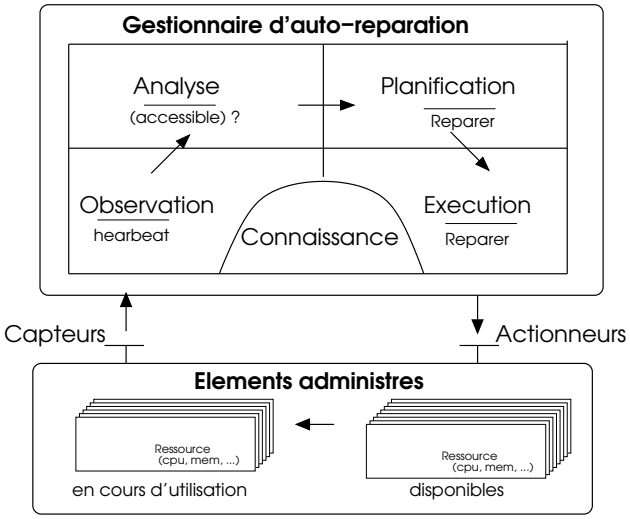

Figure 8: Gestionnaire d'auto-réparation

Des sondes contactent périodiquement les machines en cours d'utilisation afin de vérifier leur accessibilité. Les sondes utilisées dans cet exemple sont de type `Ping`. Si aucune réponse n'est reçue après l'écoulement du temps de latence alors la machine qui ne répond pas est considérée en panne. Lorsqu'une machine est considérée comme en panne, le gestionnaire détermine les éléments logiciels et matériels qui sont affectés par cette panne. Le gestionnaire détermine les logiciels que la machine défaillante exécutait et ceux qui sont liés à ces logiciels. Cette analyse est effectuée sur la base de la connaissance que le gestionnaire a de l'état courant et de la structure de système. Sans cette connaissance, la restauration du système ne peut être réalisée car l'information nécessaire pour la reconstruction serait perdue avec la défaillance. Une fois que les éléments impactés par la panne sont identifiés, le gestionnaire planifie la reconstruction du système. Cette reconstruction consiste à redémarrer sur d'autres machines disponibles les logiciels qui s'exécutaient sur la machine défaillante et rétablir les liaisons entre les éléments. L'exécution des opérations de reconfiguration est effectuée via les actionneurs fournis par le système admin-

istré qui permettent l'allocation de machines, le déploiement et la configuration des éléments logiciels et matériels du système.

Ce gestionnaire permet la disponibilité du système administré en restaurant le service après une défaillance d'une machine. Dans le cas d'un système basé sur la réplication de serveurs, il permet la restauration du degré de redondance des serveurs. Avec m serveurs répliqués, cela permet de tolérer jusqu'à m-1 pannes de serveurs durant le temps moyen de réparation (MTTR).

3.4. *Problèmes d'administration*

Dans un système basé sur une architecture multiniveau, une panne d'un serveur d'un des tiers peut affecter le tiers et les tiers qui suivent dans la chaîne de traitement des requêtes. L'occurrence d'une panne au niveau du tiers peut entraîner une baisse de charge au niveau des serveurs des autres tiers qui suivent. Ces derniers risquent de ne plus recevoir autant de requêtes à traiter qu'avant la panne. Cela peut causer une sous-charge au niveau de ces tiers. Dans le cas d'un tiers basé sur la répartition de charge, la panne d'un des serveurs peut entraîner une surcharge des autres serveurs. Lorsqu'un des serveurs dupliqués tombe en panne la charge qu'il doit traiter est répartie entre les autres serveurs restants. Cela peut entraîner une hausse de charge au niveau de ces derniers et peut causer la saturation des machines qui les exécutent.

L'utilisation de gestionnaires autonomes tels que self-sizing et self-repair peuvent faciliter la gestion des différents tiers d'un système de ce type. Toutefois, lorsqu'ils ne communiquent pas les gestionaires peuvent exécuter des opérations d'administration inutiles. En effet, les pannes de serveurs peuvent occasionner des surcharges et/ou sous-charges. La réparation d'un serveur n'étant pas instantanée, durant cette phase les sous-charges détectées au niveau des tiers précédés par le tiers où s'est produite la panne peuvent conduire à des opérations de retrait de serveurs. De plus, au niveau du tiers où s'est produite la panne les surcharges détectées durant la réparation peuvent également conduire à des ajouts de serveurs.

4. Coordination des gestionnaires self-sizing et self-repair

Cette section présente la conception d'un contrôleur de coordination des instances de self-repair et self-sizing pour une administration globale et cohérente du système multiniveau présenté à la figure 6. Pour coordonner les gestionnaires, nous identifions l'impact des actions de chaque gestionnaire sur les objectifs des autres gestionnaires. Nous identifions également les relations entre les différents événements auxquels ils réagissent. Ensuite nous définissons des objectifs globaux sur le comportement des gestionnaires. Enfin nous mettons en place un contrôle de l'exécution des gestionnaires pour assurer ces objectifs globaux. Le contrôle de l'exécution des gestionnaires est assuré par un

contrôleur de coordination obtenu par programmation synchrone et synthèse de contrôleur discret. Nous modélisons le contrôleur de coordination comme la composition des comportements des gestionnaires autonomes à laquelle est associée une politique de coordination pour éviter les comportements incohérents.

4.1. *Modélisation du contrôle des gestionnaires*

Cette section décrit les modèles des gestionnaires autonomes. Chaque gestionnaire est modélisé par un ou plusieurs automates qui décrivent son comportement et le contrôle des actions d'administration qu'il peut exécuter. Au niveau des figures 9 et 10, nous avons distingué les états relatifs aux actions d'administration que les gestionnaires peuvent entreprendre des états relatifs au contrôle qu'on peut appliquer sur le comportement des gestionnaires. Cependant, nous aurions pu représenter tous les états d'un gestionnaire par un seul automate plus simple ou plus complexe selon le niveau de détail considéré important pour la coordination.

4.1.1. *Modélisation du contrôle de self-sizing*

La figure 9 décrit le modèle du comportement contrôlable du gestionnaire self-sizing. Le modèle est constitué de trois automates. L'automate au centre représente le comportement de self-sizing et les deux autres modélisent le contrôle des actions d'ajout et de retrait de serveurs.

L'automate à droite représente le contrôle des actions de retrait de serveurs. Il est constitué de deux états : `Enable` et `Disable`. L'état `Enable`, état initial, indique que les actions de retrait sont autorisées et l'état `Disable` indique que les actions de retrait sont inhibées. Le passage de l'état `Enable` à l'état `Disable` et réciproquement est contrôlé via l'entrée `cDown`. Lorsqu'elle est à **`true`** l'automate se met dans l'état `Disable` et lorsqu'elle est à **`false`** dans l'état `Enable`. L'état courant du contrôle des actions de retrait de serveur est indiqué par la sortie `disabledDown` qui est une variable d'état. Elle est à **`true`** lorsque les actions de retrait sont inhibées. L'automate à gauche représente le contrôle des actions d'ajout de serveurs. Cet automate est similaire à celui du contrôle des retraits. Le passage de l'état `Enable` à l'état `Disable` et réciproquement est contrôlé via l'entrée `cUp`. L'état courant du contrôle des actions d'ajout est indiqué par la sortie `disabledUp`.

L'automate au centre représente le comportement du gestionnaire self-sizing. Il est constitué de quatre états. Le gestionnaire est initialement dans l'état `UpDown` dans lequel il peut exécuter aussi bien des opérations d'ajout que des opérations de retrait. Il est dans l'état `Down` quand le nombre maximum de serveurs actifs autorisé est atteint. Dans cet état il ne peut exécuter que des opérations de retrait. Le gestionnaire est dans l'état `Up` lorsque le nombre minimum de serveurs actifs autorisé est atteint. Dans cet état il ne peut exécuter que des opérations d'ajout. L'exécution d'une opération d'ajout est représentée par l'état `Adding`. Contrairement aux ajouts, l'exécution des opérations

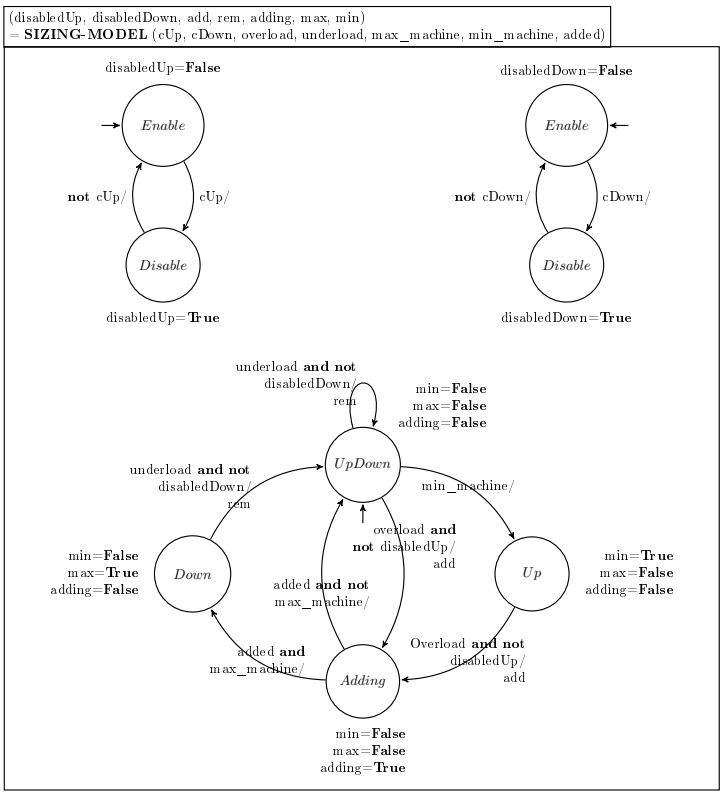

Figure 9: Modèle de contrôle de self-sizing

de retrait est considérée instantanée dans le modèle. L'occurrence d'une sous-charge (`underload` à **`true`**), lorsque le gestionnaire est dans l'état `UpDown` ou dans l'état `Down`, entraîne l'exécution d'une opération de retrait (`rem` à **`true`**) si les opérations de retrait sont autorisées (`disabledDown` à **`false`**). Dans l'état `UpDown`, le gestionnaire passe dans l'état `Up` lorsque le nombre minimum de serveur est atteint (`min_machine` à **`true`**). L'occurrence d'une surcharge (`overload` à **`true`**), lorsque le gestionnaire est dans l'état `UpDown` ou dans l'état `Up`, entraîne l'exécution d'une opération d'ajout (`add` à **`true`**) si les opérations d'ajout sont autorisées (`disabledUp` à **`false`**). A l'exécution d'un ajout le gestionnaire passe dans l'état `Adding` où aucune autre opération d'ajout ou de retrait de serveur ne peut être entamée. À la fin de l'opération d'ajout, il passe dans l'état `Down` si le nombre maximum de serveurs est atteint (`max_machine` à **`true`**) sinon il retourne dans l'état `UpDown`.

4.1.2. *Modélisation du contrôle de self-repair*

Le modèle de self-repair est constitué de deux automates. L'automate à gauche modélise le contrôle des actions de réparation. Il a deux états : `Enable` et `Disable`. L'état `Enable`, état initial, indique que les actions de réparation sont autorisées et l'état `Disable` indique que les actions de réparation sont interdites. Le changement d'états est contrôlé par l'entrée `c`. Lorsqu'elle est à **`true`** les actions sont interdites. L'état du contrôle des actions de réparation est indiqué en sortie par la variable d'état`disabled` qui est à **`true`** lorsque les actions sont interdites.

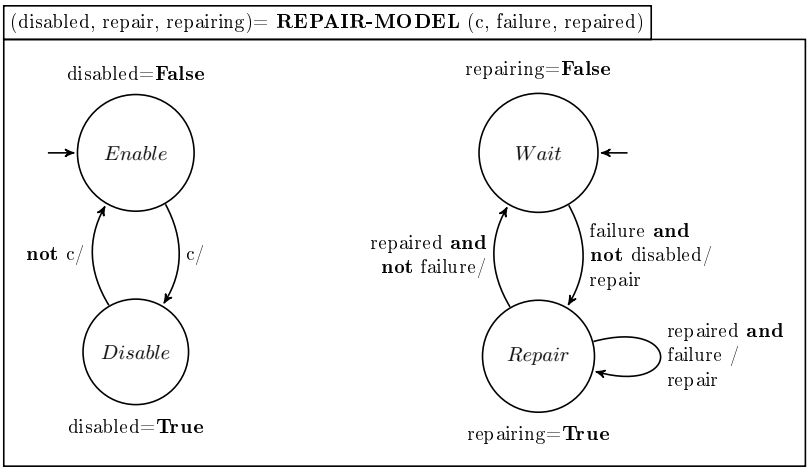

Figure 10: Modéle de contrôle de self-repair

L'automate à droite modélise le comportement du gestionnaire. Il est constitué de deux états : `Wait` et `Repair`. L'état `Wait`, état initial, représente l'état dans lequel self-repair attend la détection d'une panne. L'état `Repair` représente l'état dans lequel le gestionnaire est en train d'effectuer la réparation de la panne. L'occurrence d'une panne est représentée par l'entrée `failure` à **`true`**. Dans l'état `Wait`, à l'occurrence d'une panne, self-repair réagit, si autorisée, en produisant l'action `repair` et se met dans l'état `Repair`. La fin de la réparation est représentée par l'entrée `repaired` à **`true`**. Si aucune panne n'est détectée, self-repair retourne dans l'état d'attente `Wait`. En cas de panne, il réagit en réparant la panne. La sortie `repairing` indique l'état courant de l'automate. Elle est à **`true`** lorsque l'automate est dans l'état `Repair`.

4.2. *Spécification de la coordination*

La coexistence des gestionnaires self-sizing et self-repair est représentée par la composition d'instances des automates qui décrivent leur comportement,

illustrée à la figure 11. Nous avons quatre instances du modèle du gestionnaire self-repair et deux instances du modèle du gestionnaire self-sizing.

```
(A_disabled, A_repair_server, A_repairing,
 T_disabled, T_repair_server, T_repairing,
 T_disabledUp, T_disabledDown, T_add_server, T_remove_server, T_adding, T_max, T_min,
 ...) = MULTITIER (A_c, T_c, P_c, M_c, T_cUp, T_cDown, M_cUp, M_cDown,
                   A_failure, A_repaired,
                   T_failure, T_repaired,
                   T_Underload, T_Overload, T_max_machine, T_min_machine...)
```

```
(A_disabled, A_repair_server, A_repairing) = REPAIR-MODEL (A_c, A_failure, A_repaired);

(T_disabled, T_repair_server, T_repairing) = REPAIR-MODEL (T_c, T_failure, T_repaired);
(T_disabledUp, T_disabledDown, T_add_server, T_remove_server, T_adding, T_max, T_min) =
    SIZING-MODEL (T_cUp, T_cDown, T_Underload, T_Overload, T_max_machine, T_min_machine...);

(P_disabled, P_repair_server, P_repairing) = REPAIR-MODEL (P_c, P_failure, P_repaired);

(M_disabled, M_repair_server, M_repairing) = REPAIR-MODEL (M_c, M_failure, M_repaired);
(M_disabledUp, M_disabledDown, M_add_server, M_remove_server, M_adding, M_max, M_min) =
    SIZING-MODEL (M_cUp, M_cDown, M_Underload, M_Overload, M_max_machine, M_min_machine...);
```

Figure 11: Composition des modèles de self-sizing et self-repair

Les entrées et sorties des instances d'automates sont renommées, en ajouter un préfixe, pour distinguer les gestionnaires qu'ils représentent : "`A_`" pour le gestionnaire dédié au tiers `Apache`, "`T_`" pour le tiers `Tomcat`, "`P_`" pour le tiers `Mysql-proxy`, et " `M_`" pour le tiers `Mysql`. Par exemple, l'entrée `A_failure` représente une panne du serveur `Apache`. Les entrées (sorties resp.) de la composition **`multitier`** correspondent à l'union de toutes les entrées (sorties resp.) des automates contenus.

4.2.1. *Stratégie de coordination*

La stratégie de coordination consiste à empêcher les gestionnaires self-sizing d'exé-cuter des actions d'ajout ou de retrait de serveurs en cas de panne. En effet puisqu'une panne d'un serveur d'un tiers peut entraîner une surcharge au niveau du tiers et une sous-charge au niveau des tiers qui suivent, traiter d'abord la panne avant de tenir compte des événements de surcharge ou de sous-charge peut être plus pertinent. Cela peut empêcher des réactions inutiles qui conduisent à des ajouts et/ou des retraits de serveurs. Le contrôle des actions des gestionnaires self-sizing est basé sur les activités des gestionnaires self-repair. La stratégie est décrite de manière textuelle ci-dessous :

Condition 1 : **ignorer la surcharge détectée au niveau du tiers *Tomcat*** — En cas de panne dans le tiers *Tomcat*, toute surcharge dans ce tiers est ignorée tant que la réparation n'est pas terminée.

Condition 2 : **ignorer la sous-charge détectée au niveau du tiers *Tomcat*** — En cas de panne dans le tiers *Apache*, toute sous-charge dans le tiers `Tomcat` est ignorée tant que la réparation de la panne n'est pas terminée.

Condition 3 : **ignorer la surcharge détectée au niveau du tiers *Mysql*** — En cas de panne dans le tiers *Mysql*, toute surcharge dans ce tiers est ignorée tant que la réparation n'est pas terminée.

Condition 4 : **ignorer la sous-charge détectée au niveau du tiers *Mysql*** — En cas de panne dans les tiers *Apache*, *Tomcat* et *Mysql-proxy*, toute sous-charge dans le tiers *Mysql* est ignorée tant que la réparation de la panne n'est pas terminée.

4.2.2. *Spécification du contrat*

Nous définissons la stratégie sous forme de propriétés d'invariance. Ces propriétés sont exprimées via les sorties des automates à la figure 11. Certaines propriétés sont spécifiées dans le contrat et assurées par le contrôleur généré. D'autres sont programmées manuellement. Ci-dessous nous décrivons les spécifications formelles de la stratégie de coordination.

Condition 1 : cette condition consiste à ignorer les surcharges détectées dans le tiers *Tomcat* en cas de panne d'un serveur *Tomcat* en cours de réparation. Lors de la réparation d'un *Tomcat*, le gestionnaire self-sizing chargé du dimensionnement dynamique de ce tiers ne doit pas réagir aux surcharges. Cela est exprimé par :

1. $invariant_{1.1}$ = `T_repairing` **XOR**[6] **not** `T_disabledUp`

   `T_repairing` étant à **`true`** exprime le fait que self-repair est en train de faire une opération de réparation de serveur *Tomcat*. (**not** `T_disabledUp`) étant à **`true`** exprime le fait que les opérations d'ajout de self-sizing sont autorisées. Empêcher (**not** `T_disabledUp`) d'être à **`true`** quand `T_repairing` est à **`true`** assure qu'aucune opération d'ajout de serveur *Tomcat* ne sera exécutée durant une réparation d'un serveur *Tomcat*. Cette propriété permet, au niveau du tiers *Tomcat*, de «désactiver» les opérations d'ajout de serveurs lorsqu'une opération de réparation est en cours d'exécution.

2. $invariant_{1.2}$ = **not** (`T_repair_server` **and** `T_add_server`)

   `T_repair_server` étant à **`true`** exprime le fait que self-repair entame une opération de réparation. `T_add_server` étant à **`true`** exprime le fait que self-sizing entame une opération d'ajout. Cette propriété permet d'éviter une exécution simultanée d'une opération d'ajout et d'une opération de réparation au niveau du *Tomcat*.

[6] Nous avons utilisé l'opérateur `XOR` pour éviter que les deux opérandes puissent être à **`false`** en même temps. Cela permet d'éviter que les opérations d'ajout soient désactivées lorsqu'il n'y a pas de réparation en cours.

Une seule expression aurait pu suffir pour ces deux propriétés. Toutefois nous avons adopté cette approche pour présenter les différentes possibilités offertes par `Heptagon/BZR`.

Condition 2 : cette condition consiste à ignorer les sous-charges détectées dans le tiers *Tomcat* en cas de panne du serveur *Apache* et en cours de réparation. Cela implique l'inhibition des opérations de retrait de serveurs au niveau du tiers *Tomcat* lorsque le serveur *Apache* est en train d'être réparé :

3. $invariant_{2.1}$ = `A_repairing` **XOR not** `T_disabledDown`

   Cette propriété concerne les opérations de réparation du tiers `Apache` et l'inhibition des opérations de retrait au niveau du tiers *Tomcat*.

4. $invariant_{2.2}$ = **not** (`A_repair_server` **and** `T_remove_server`)

   Cette propriété évite qu'au même instant une opération de réparation du serveur *Apache* et une opération de retrait de serveur au niveau du *Tomcat* soient entamées.

Condition 3 : cette condition consiste à ignorer les surcharges détectées dans le tiers *Mysql* en cas de panne d'un serveur *Mysql* en cours de réparation. Lors de la réparation d'un *Mysql*, le gestionnaire self-sizing chargé du dimensionnement dynamique de ce tiers ne doit pas réagir aux surcharges. Cela est exprimé par :

5. $invariant_{3.1}$ = `M_repairing` **XOR not** `M_disabledUp`

   Cette propriété est similaire à $invariant_{1.1}$.

6. $invariant_{3.2}$ = **not** (`M_repair_server` **and** `M_add_server`)

   Cette propriété est similaire à $invariant_{1.2}$.

Condition 4 : cette condition consiste à ignorer les sous-charges détectées dans le tiers *Mysql* en cas de panne du serveur *Apache*, du serveur *Mysql-Proxy* ou d'un serveur *Tomcat* et en cours de réparation. Cela implique l'inhibition des opérations de retrait de serveurs au niveau du tiers *Mysql* lorsqu'un serveur au niveau des tiers qui précédent est en train d'être réparé :

Soit `APT_repairing` défini par (`A_repairing` **or** `P_repairing` **or** `T_repairing`), et `APT_repair_server` défini par (`A_repair_server` **or** `P_repair_server` **or** `T_repair_server`).

7. $invariant_{4.1}$ = `APT_repairing` **XOR not** `M_disabledDown`

La propriété $invariant_{4.1}$ concerne l'inhibition des opérations de retrait au niveau du tiers *Mysql* lorsqu'une (des) opération(s) de réparation est en train d'être réalisée au niveau des tiers précédents.

8. $invariant_{4.2}$ = **not** (`APT_repair_server` **and** `M_remove_server`)

   La propriété $invariant_{4.2}$ évite qu'au même instant une opération de réparation du serveur au niveau des tiers précédents et une opération de retrait de serveur au niveau du *Mysql* soient entamées.

4.2.3. *Programme final*

La figure 12 décrit le modèle de la coordination des gestionnaires. Ce modèle est constitué du modèle de la coexistence des gestionnaires auquel est associé un contrat exprimant la stratégie de coordination. Les propriétés d'invariance décrites à la section 4.2.2. peuvent être assurées soit par synthèse de contrôleur et/ou par programmation. Certaines propriétés sont assurées par programmation alors que d'autres sont assurées par le contrôleur qui est généré à la compilation. Il s'agit de montrer que les propriétés peuvent être programmées. Toutefois, la synthèse de contrôleur facilite le respect des propriétés surtout lorsque la programmation devient complexe.

```
(A_repair_server, T_repair_server, T_add_server, T_remove_server...) =
    COORDINATED_MULTITIER (A_failure, A_repaired,
                           T_failure, T_repaired,
                           T_Underload, T_Overload, T_max_machine, T_min_machine...)
enforce (invariant1.1 and invariant2.1 and invariant3.1 and invariant4.1 and
         invariant1.2 and invariant2.2 and invariant3.2 and invariant4.2)
with A_c, T_c, P_c, M_c, T_cUp, T_cDown, M_cUp, M_cDown

 APT_failure = A_failure or P_failure or T_failure;
     ...
 T_Overload' = not T_failure and T_Overload;
T_Underload' = not A_failure and T_Underload;
 M_Overload' = not M_failure and M_Overload;
M_Underload' = not APT_failure and M_Underload;
     ...
(A_disabled, A_repair_server, A_repairing,
 T_disabled, T_repair_server, T_repairing,
 T_disabledUp, T_disabledDown, T_add_server, T_remove_server, T_adding, T_max, T_min,
 ...) = MULTITIER (A_c, T_c, P_c, M_c, T_cUp, T_cDown, M_cUp, M_cDown,
                   A_failure, A_repaired,
                   T_failure, T_repaired,
                   T_Underload', T_Overload', T_max_machine, T_min_machine...);
```

Figure 12: Coordination des instances de self-sizing et self-repair

Les propriétés $invariant_{1.2}$, $invariant_{2.2}$, $invariant_{3.2}$ et $invariant_{4.2}$ sont assurées par programmation du code Heptagon/BZR qui les réalise et non par SCD. Elles sont vérifiées à la compilation. La compilation réussit si ces propriétés sont satisfaites. Pour la propriété $invariant_{1.2}$ (**not** (`T_repair_server` **and** `T_add_server`)), on définit une variable `T_Overload'`. La valeur de `T_Overload'` est :
(**not** `T_failure` **and** `T_Overload`)

T_Overload' est à **true** lorsque T_Overload est à **true** et T_failure est à **false**.

Pour assurer la propriété $invariant_{1.2}$, on considère T_Overload' comme entrée de notification de surcharge pour le modèle du self-sizing du tiers Tomcat. Cela évite les situations dans lesquelles T_repair_server et T_add_server sont à **true** au même instant. Le même principe est utilisé pour les autres propriétés programmées manuellement. A la compilation, BZR génère la logique de contrôle qui restreint la composition **multitier** aux comportements qui respectent les propriétés $invariant_{1.1}$, $invariant_{2.1}$, $invariant_{3.1}$ et $invariant_{4.1}$. Cette logique de contrôle est automatiquement intégrée dans le modèle global.

5. Expérimentations

L'objectif des expérimentations est d'évaluer le comportement du contrôleur de coordination construit pour coordonner les gestionnaires self-sizing et self-repair afin d'éviter des décisions d'administration incohérentes. La plate-forme expérimentale est constituée de machines avec des processeurs Dual-Core de 1.66Ghz et 1.9Go de mémoire, de machines avec des processeurs Dual-Core de 2,53 Ghz et 3.4Go de mémoire, et de machines avec des processeurs Dual-Core de 1.20Ghz et 1.5Go de mémoire. Les machines sont inter-connectées via un réseau Ethernet (1 Gbit/s). Pour l'ensemble des expérimentations, une seule instance de serveur *Apache* et une instance de serveur *Mysql-Proxy* sont utilisés. Les tiers *Tomcat* et *Mysql* sont basés sur la réplication. Chaque machine héberge un seul serveur.

Quatres instances du gestionnaire self-repair sont utilisées, chaque tiers est géré par une instance pour la gestion des pannes. Les tiers basés sur la réplication sont les tiers *Tomcat* et *Mysql*. Deux instances du gestionnaire self-sizing sont utilisées, chacun de ces deux tiers est géré par une instance pour le dimensionnement dynamique du nombre de serveurs dupliqués actifs. Le contrôleur de coordination généré est responsable de la coordination des actions des quatre instances de self-repair et des deux instances de self-sizing.

5.1. *Configuration*

Les expérimentations ont été réalisées avec l'application multiniveau JEE RUBiS [CEC 03]. Elle implante un site de vente aux enchères [SAR 08] et définit plusieurs types d'interactions Web (e.g., l'enregistrement de nouveaux utilisateurs, la navigation, l'achat ou la vente d'objets). Le déploiement de RUBiS est basé sur une architecture distribuée constituée d'un front-end et d'un back-end. Le front-end est un cluster constitué des serveurs d'application *Tomcat* dupliqués et d'un serveur web *Apache* comme équilibreur de charge. Et le back-end est un cluster constitué des serveurs de bases de données *Mysql* et du serveur *Mysql-proxy* comme équilibreur de charge pour les serveurs *Mysql*.

L'évaluation représente un réel problème qui peut être rencontré dans un environnement de *cloud computing* avec des capacités de recouvrement et des capacités d'élasticité.

Initialement lors de chaque exécution, le système est démarré avec un serveur au niveau de chaque tiers : un serveur *Apache*, un serveur *Tomcat*, un serveur de *Mysql-proxy* et un serveur *Mysql*. Nous injectons une charge croissante (correspondant à la période de la création des threads qui simulent les actions des clients) puis une charge constante (correspondant à l'achèvement de la création de l'ensemble des threads). Nous attendons qu'il y ait deux serveurs *Tomcat* actifs et deux serveurs *Mysql* actifs pour déclencher des pannes.

5.2. *évaluation*

Nous avons effectué des expérimentations durant lesquelles les gestionnaires ne sont pas coordonnés et d'autres durant lesquelles les gestionnaires sont coordonnés. La même configuration et le même profil de charge de travail sont utilisés lors des différentes exécutions aussi bien non coordonnées que coordonnées. Les expérimentations non coordonnées permettent de voir le comportement des gestionnaires à l'occurrence d'une panne. Les expérimentations coordonnées permettent de voir si le contrôleur de coordination contrôle les gestionnaires afin d'assurer le respect de la politique de coordination.

Nous déclenchons des pannes lorsque la charge est constante pour voir leurs impacts au niveau des tiers et aussi pour voir comment les différentes instances des gestionnaires réagissent.

Les courbes qui indiquent le nombre de serveurs actifs ne contiennent que des valeurs entières même si les évolutions sur certaines courbes peuvent laisser penser le contraire. Périodiquement une sonde interroge les serveurs pour détecter s'ils sont accessibles et enregistre le nombre de serveurs accessibles. Entre deux enregistrements il peut y avoir un délai causé par le délai de communication mais également par la reconfiguration de la sonde pour prendre en compte les modifications effectuées.

5.2.1. *Comportement non coordonné*

Comme dit plus haut, une panne au niveau d'un tiers peut avoir un impact sur le tiers mais également sur les tiers qui suivent.

La figure 13 présente une exécution durant laquelle le serveur *Apache* tombe en panne. L'occurrence de la panne du serveur *Apache*, 17 minutes après le début de l'exécution, provoque une baisse de charge au niveau des tiers *Tomcat* et *Mysql* (18 min). Une sous-charge est détectée au niveau de ces tiers et a conduit au retrait de serveurs dupliqués. Cependant après la réparation du serveur *Apache* (19 min), la charge est redevenue normale et les serveurs précédemment retirés ont été démarrés à nouveau à cause d'une sur-charge (*Tomcat* : 20 min et *Mysql* : un peu après 25 min).

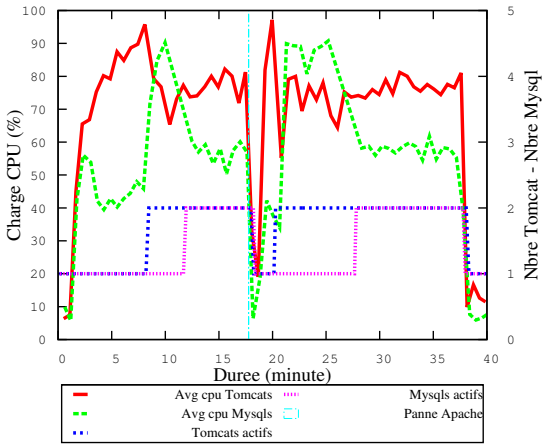

Figure 13: Exécution non coordonnée : panne du serveur Apache

La figure 14 présente une exécution durant laquelle un serveur *Tomcat* tombe en panne. Cette panne, survenue 26 minutes après le début de l'exécution, a provoqué une hausse de charge au niveau du tiers (27 min) et aussi une baisse de charge au niveau du tiers *Mysql* (27 min). La hausse de charge a conduit à une surcharge du tiers *Tomcat* conduisant à l'ajout d'un nouveau serveur *Tomcat*. La baisse de charge au niveau du tiers *Mysql* a conduit au retrait d'un serveur à cause d'une sous-charge des serveurs. Mais après la restauration (28 min) du serveur *Tomcat* tombé en panne, le serveur *Tomcat* précédemment ajouté est retiré et le serveur *Mysql* qui était retiré est rajouté à nouveau à cause d'une surcharge (31 min).

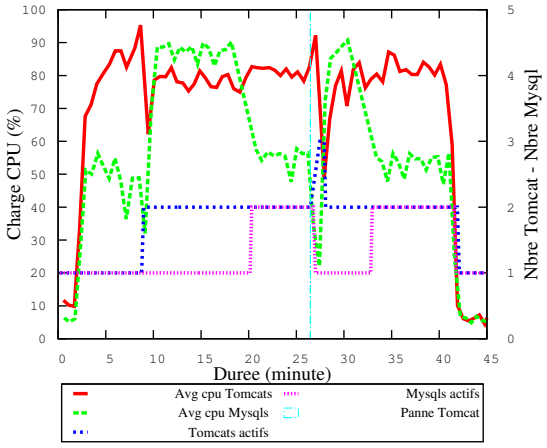

Figure 14: Exécution non coordonnée : panne d'un serveur Tomcat

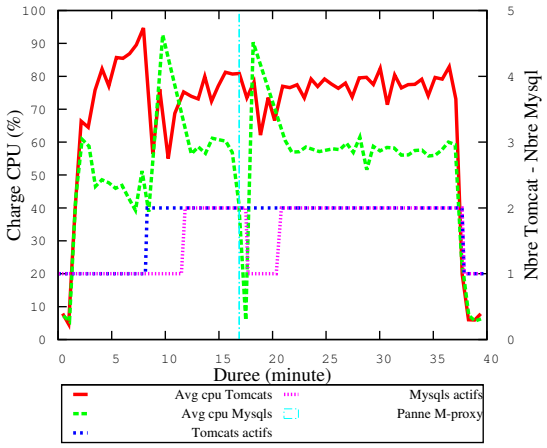

Figure 15: Exécution non coordonnée : Panne du serveur Mysql-proxy

La figure 15 présente une exécution durant laquelle le serveur *Mysql-proxy* tombe en panne. L'occurrence de la panne du serveur *Mysql-proxy*, survenue 17 minutes après le début de l'exécution, provoque une baisse de charge au niveau du tiers *Mysql* (18 min). Une sous-charge est détectée et a conduit au retrait d'un serveur *Mysql*. Cependant après la réparation du serveur *Mysql-proxy* (19 min), la charge est redevenue normale et le serveur précédemment retiré a été démarré à nouveau à cause d'une surcharge (20 min).

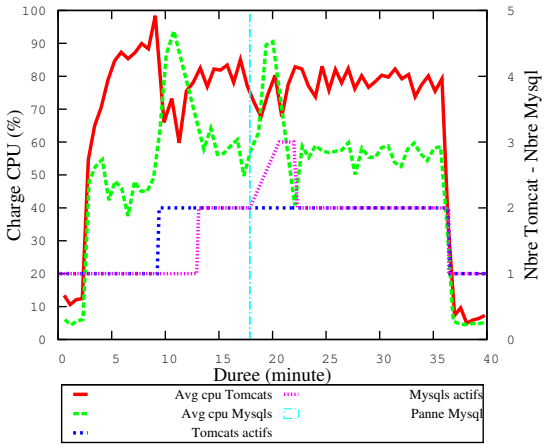

Figure 16: Exécution non coordonnée : panne d'un serveur Mysql

La figure 16 présente une exécution durant laquelle un serveur *Mysql* tombe en panne. Cette panne, survenue 18 minutes après le début de l'exécution, a provoqué une surcharge du serveur *Mysql* restant (20 min). Cela a conduit à

un ajout d'un serveur *Mysql* alors que celui tombé en panne est en cours de restauration. Après la restauration, le serveur ajouté est retiré à cause d'une baisse de charge (22 min).

Les pannes peuvent entraîner des réactions des instances du gestionnaire self-sizing à cause de leur impact sur la charge à traiter. Cependant, à moins que la charge en entrée ait changé, les actions de ces instances de self-sizing ne sont pas nécessaires car une fois les pannes réparées, le degré de réplication, avant les pannes, est restauré.

5.2.2. *Comportement coordonné*

Les figures suivantes présentent les exécutions durant lesquelles les instances de self-sizing et de self-repair sont coordonnées. La coordination est assurée par le contrôleur modélisé à la section 4..

La figure 17 présente une exécution durant laquelle le serveur *Apache* tombe en panne. L'occurrence de la panne (min. 19) provoque une diminution de la charge à la fois au niveau du tiers *Tomcat* et au niveau du tiers *Mysql*. Une sous-charge (10 % de charge cpu) au niveau de ces tiers est observée cependant aucune opération de retrait de serveur n'est exécutée ni au niveau du tiers *Tomcat* ni au niveau du *tiers Mysql*. A la fin de la réparation du serveur *Apache*, la charge est redevenue normale au niveau des tiers *Tomcat* et *Mysql*.

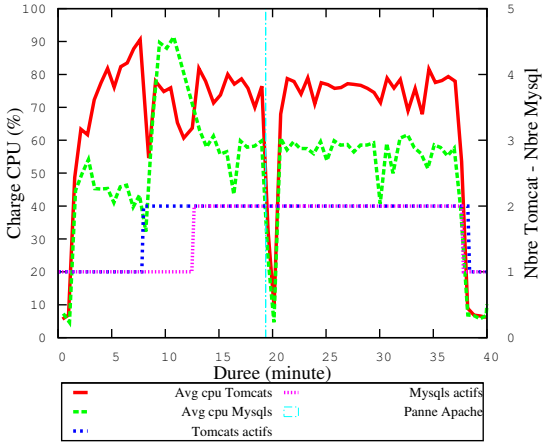

Figure 17: Exécution coordonnée : panne du serveur Apache

La figure 18 présente une exécution durant laquelle un serveur *Tomcat* est en panne. La panne survient 17 minutes après le début de l'expérimentation et provoque une hausse de la charge du serveur *Tomcat* restant. Cette hausse de charge a conduit à une surcharge du serveur *Tomcat* restant (19 min). Cependant aucune opération d'ajout de serveur n'est exécutée par l'instance de self-sizing qui gère le tiers *Tomcat*. Une baisse de la charge est également observée au niveau du tiers *Mysql* jusqu'en dessous du seuil minimal toléré. Mais sur ce

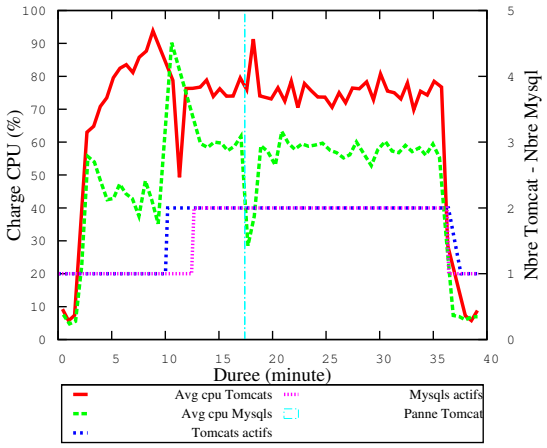

Figure 18: Exécution coordonnée : panne d'un serveur Tomcat

tiers aussi aucune opération de retrait de serveur n'est exécutée par l'instance de self-sizing qui le gère. La charge au niveau des tiers est redevenue normale après la réparation du serveur *Tomcat*.

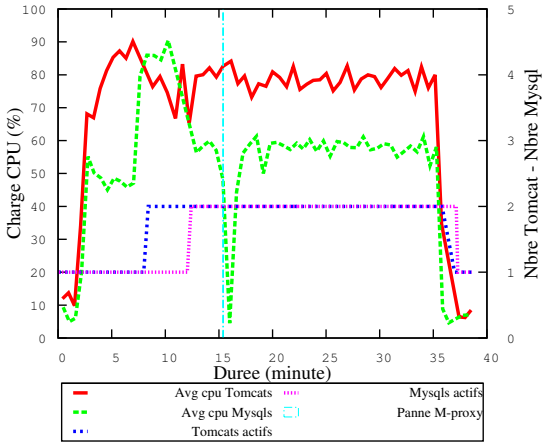

Figure 19: Exécution coordonnée : panne du serveur Mysql-proxy

La figure 19 présente une exécution durant laquelle on observe une panne du serveur *Mysql-proxy*. Cette panne survenue 16 minutes après le début de l'expérimenta-tion a occasionné une baisse de charge au niveau du tiers *Mysql*. Cependant aucune opération de retrait n'est effectuée sur le tiers *Mysql* et la charge est redevenue normale après la réparation.

L'occurrence d'une panne de serveur *Mysql*, observée 17 minutes après le

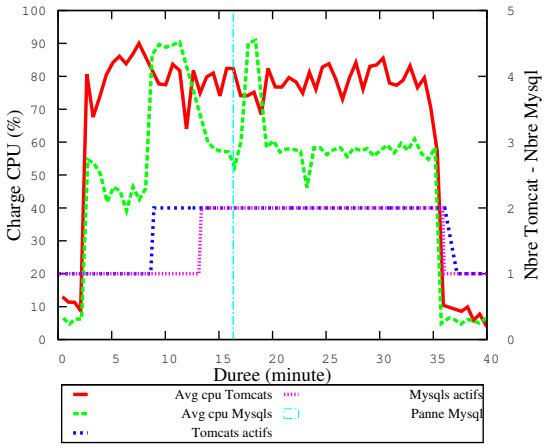

Figure 20: Exécution coordonnée : panne d'un serveur Mysql

début de l'expérimentation sur la figure 20, occasionne une hausse de charge au niveau du *Mysql* restant. Cependant là également aucune opération d'ajout n'est observée. Le degré de réplication au niveau du tiers *Mysql* n'a pas varié durant la réparation de la panne. La charge est redevenue normale après la réparation de la panne.

5.3. *Synthèse*

Le contrôleur de coordination obtenu est en mesure de coordonner les gestionnaires afin d'assurer le respect de la politique de la coordination. Il empêche les gestionnaires self-sizing d'ajouter un nouveau serveur dans un tiers où une panne est en train d'être réparée. Il empêche également la suppression de serveurs au niveau des autres tiers. Il évite des opérations d'acquisition et de libération répétitive de machines ce qui est particulièrement pertinent dans la gestion de centre de données, où les ressources de calcul sont partagées entre plusieurs applications clientes isolées grâce à la virtualisation. Dans un tel environnement un quota de ressources de calcul est attribué à chacune des applications clientes. Le quota affecté à une application peut être ajusté dynamiquement en fonction de sa charge de travail. Une acquisition inutile de ressources peut empêcher une autre application d'atteindre son objectif de performance. Et la libération inutile de ressources nécessaires pour une application peut avoir un impact sur ses performances si les ressources sont affectées à d'autres applications.

6. Conclusions

L'automatisation des fonctions d'administration de systèmes informatiques a été étudiée dans plusieurs travaux de recherche. Ces travaux ont démontré la faisabilité de cette approche en proposant différents gestionnaires autonomes qui assurent de manière cohérente les fonctions d'administration. Aujourd'hui de nombreux gestionnaires autonomes sont disponibles, mais aucun n'implante intégralement l'ensemble des fonctions d'administration nécessaires pour la gestion complète d'un système informatique. La complexité de concevoir un gestionnaire complet rend nécessaire la coexistence de plusieurs gestionnaires pour une administration complète. Toutefois leur coordination est utile pour assurer une cohérence des actions d'administration exécutées par les gestionnaires autonomes qui peuvent avoir des politiques contradictoires.

Nous proposons une approche de coordination basée sur la synchronisation partielle de l'exécution des gestionnaires et le contrôle logique de leurs actions pour éviter des incohérences. Nous proposons une approche basée sur les langages synchrones et les techniques de synthèse de contrôleur discret pour la conception de contrôleur de coordination. Les langages synchrones sont des langages de haut niveau permettant une spécification formelle de système, et associés à des outils de vérification. La synthèse de contrôleur permet de raffiner une spécification incomplète de manière à atteindre un certain objectif comme la satisfaction d'une propriété non encore vérifiée sur le système initial.

Dans ce papier, nous évaluons notre approche en coordonnant des gestionnaires autonomes pour la gestion de la performance, de la disponibilité et de l'optimisation des ressources pour un système multiniveau. Les expérimentations, présentées dans ce papier, montrent la faisabilité de notre approche pour la conception et la validation d'un contrôleur de coordination qui assure de manière cohérente les politiques de coordination définies.

Nous envisageons d'appliquer cette approche pour une coordination à large échelle avec des politiques de coordination plus complexes et plus élaborées. Une perspective pour la SCD est d'explorer par exemple la distribution du contrôle ([Bel 11]), le contrôle sur des chemins ([DUM 10]) et des cas d'étude industriels sur des centres de données dans le contexte du projet ANR Ctrl-Green.

7. Remerciement

Ce travail a été effectué dans le cadre du projet Ctrl-Green (ANR-11-INFR 012 11). Ce projet de recherche est financé par l'ANR (Agence Nationale de la Recherche) avec le soutien de MINALOGIC. `www.ctrlgreen.org`

8. References

[ALD 11] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Vamis Xhagjika. LIBERO: A Framework for Autonomic Management of Multiple Non-functional Concerns. MarioR. Guarracino, Frédéric Vivien, JesperLarsson Träff, Mario Cannatoro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, Michael Alexander, , *Euro-Par 2010 Parallel Processing Workshops*, 6586 *Lecture Notes in Computer Science*, 237–245. Springer Berlin Heidelberg, 2011.

[AND 04] Hua L. And, Hua Liu, Manish Parashar, Salim Hariri. « A Component Based Programming Framework for Autonomic Applications ». *Proc. of 1st International Conference on Autonomic Computing*, 10–17, 2004.

[Bel 11] M. A. Belhaj Seboui, N. Ben Hadj Alouane, G. Delaval, E. Rutten, M. Yeddes. « An approach for the Synthesis of Decentralized Supervisors for Distributed Adaptive Systems ». *International Journal of Critical Computer-Based Systems*, 2(3/4), 2011.

[BEN 03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, Robert de Simone. « The Synchronous Languages Twelve Years Later ». *Proc. of the IEEE*, 91(1), January 2003.

[BER 92] Gerard Berry, Georges Gonthier, Ard Berry Georges Gonthier, Place Sophie Laltte. « The Esterel Synchronous Programming Language: Design, Semantics, Implementation », 1992.

[BLO 10] Roderick Paul Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, Richard Seeber. « RATSY - A new Requirements Analysis Tool with Synthesis ». Springer, , *Computer Aided Verification*, 6174 *Lecture Notes in Computer Science*, 425 – 429, 2010.

[BLO 12] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, Y. Sa'ar. « Synthesis of Reactive(1) Designs ». *Journal of Computer and System Sciences, Special Issue in Memory of Amir Pneuli*, 78(3):911–938, May 2012. Full version of VMCAI06, DATE07, and COCV07 papers.

[BOY 09] Fabienne Boyer, Noël de Palma, Olivier Gruber, Sylvain Sicard, Jean-Bernard Stefani. « *Architecting Dependable Systems, Lectures Notes in Computer Science, edited by R. De Lemos and J.C. Fabre* », Self-Repair of Distributed Applications. Springer Verlag, 2009.

[CAS 06] Christos G. Cassandras Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[CEC 03] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite, Willy Zwaenepoel. « Performance comparison of middleware architectures for generating dynamic web content ». *Proceedings of the ACM-IFIP-USENIX 2003 International Conference on Middleware*, Middleware '03, 242–261, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[COL 05] Jean-Louis Colaço, Bruno Pagano, Marc Pouzet. « A conservative extension of synchronous data-flow with state machines ». *Proceedings of the 5th ACM International Conference on Conference on Embedded Software*, EMSOFT '05, 173–182, New York, NY, USA, 2005. ACM.

[DAS 08] Rajarshi Das, Jeffrey O. Kephart, Charles Lefurgy, Gerald Tesauro, David W. Levine, Hoi Chan. « Autonomic multi-agent management of power and performance in data centers ». *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent systems: industrial track*, AAMAS '08, 107–114, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[DEL 10a] Gwenaël Delaval, Hervé Marchand, Eric Rutten. « Contracts for modular discrete controller synthesis ». *Proc ACM Comf. LCTES*, 2010.

[DEL 10b] Gwenaël Delaval, Hervé Marchand, Éric Rutten. « Contracts for modular discrete controller synthesis ». *Proceedings of the ACM SIGPLAN-SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '10, 57–66, New York, NY, USA, 2010. ACM.

[DEL 13] G. Delaval, É. Rutten, H. Marchand. « Integrating Discrete Controller Synthesis into a Reactive Programming Language Compiler ». *Discrete Event Dynamic Systems*, 23(4):385–418, December 2013.

[DUM 10] E. Dumitrescu, A. Girault, H. Marchand, E. Rutten. « Multicriteria optimal discrete controller synthesis for fault-tolerant tasks ». *Proc. 10th Int Workshop on Discrete Event Systems (WODES 2010)*, 2010.

[HAL 91] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud. « The synchronous dataflow programming language LUSTRE ». *Proceedings of the IEEE*, 1305–1320, 1991.

[HEL 04] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, Dawn M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[HEO 11] Jin Heo, Praveen Jayachandran, Insik Shin, Dong Wang, Tarek Abdelzaher, Xue Liu. « OptiTuner: On Performance Composition and Server Farm Energy Minimization Application ». *IEEE Trans. Parallel Distrib. Syst.*, 22(11):1871–1878, November 2011.

[KAS 10] Timotheos Kastrinogiannis, Nikolay Tcholtchev, Arun Prakash, Ranganai Chaparadza, Vassilios Kaldanis, Hakan Coskun, Symeon Papavassiliou. « Addressing Stability in Future Autonomic Networking. ». Kostas Pentikousis, Ramon Aguero Calvo, Marta GarcÃa-Arranz, Symeon Papavassiliou, , *MONAMI*, 68 *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, 50–61. Springer, 2010.

[KEP 03] Jeffrey O. Kephart David M. Chess. « The Vision of Autonomic Computing ». *Computer*, 36:41–50, January 2003.

[KUM 09] Sanjay KUMAR, Vanish TALWAR, Vibhore KUMAR, Parthasarathy RANGANATHAN, Karsten SCHWAN. « vManage: loosely coupled platform and virtualization management in data centers ». *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, 127–136, New York, NY, USA, 2009. ACM.

[MAR 00] Hervé MARCHAND, Patricia BOURNAI, Michel LE BORGNE, Paul LE GUERNIC. « Synthesis of Discrete-Event Controllers Based on the Signal Environment ». *Discrete Event Dynamic Systems*, 10:325–346, October 2000.

[MAR 03] Florence MARANINCHI Yann RÉMOND. « Mode-automata: a new domain-specific construct for the development of safe critical systems ». *Sci. Comput. Program.*, 46:219–254, March 2003.

[NAT 07] Ripal NATHUJI Karsten SCHWAN. « VirtualPower: coordinated power management in virtualized enterprise systems ». *Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, 265–278, New York, NY, USA, 2007. ACM.

[RAG 08] Ramya RAGHAVENDRA, Parthasarathy RANGANATHAN, Vanish TALWAR, Zhikui WANG, Xiaoyun ZHU. « No "power" struggles: coordinated multi-level power management for the data center ». *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, 48–59, New York, NY, USA, 2008. ACM.

[RAM 87] P.J. RAMADGE W.M. WONHAM. « Supervisory Control of a Class of Discrete Event Processes ». *SIAM J. on Control and Optimization*, 25(1):206–230, January 1987.

[SAR 08] Ebada SARHAN, Atif GHALWASH, Mohamed KHAFAGY. « Specification and implementation of dynamic web site benchmark in telecommunication area ». *Proceedings of the 12th WSEAS International Conference on Computers*, ICCOMP'08, 863–867, Stevens Point, Wisconsin, USA, 2008. World Scientific and Engineering Academy and Society (WSEAS).

[SIC 08] Sylvain SICARD, Fabienne BOYER, Noel DE PALMA. « Using Components for Architecture-Based Management: The Self-Repair Case ». *30th International Conference on Software Engineering (ICSE)*. ACM, May 2008.

[TAT 06] Christophe TATON, Sara BOUCHENAK, Noël DE PALMA, Daniel HAGIMONT, Sylvain SICARD. « Self-Sizing of Clustered Databases ». *Proceedings of the 2006 International Symposium on World of Wireless, Mobile and Multimedia Networks*, WOWMOM '06, 506–512, Washington, DC, USA, 2006. IEEE Computer Society.

[TCH 09] Nikolay TCHOLTCHEV, Ranganai CHAPARADZA, Arun PRAKASH. « Addressing Stability of Control-Loops in the Context of the GANA Architecture: Synchronization of Actions and Policies ». *IWSOS*, 262–268, 2009.

[WAN 07] Yin WANG, Terence KELLY, Stéphane LAFORTUNE. « Discrete control for safe execution of IT automation workflows ». *Proceedings of the 2nd ACM SIGOPS-EuroSys European Conference on Computer Systems 2007*, EuroSys '07, 305–314, New York, NY, USA, 2007. ACM.