

Une Gestion de Ressources Sensible au Contexte & Opportuniste pour les Systèmes d'Information Pervasifs

David Beserra, Manuele Kirsch Pinheiro, Carine Souveyet

▶ **To cite this version:**

David Beserra, Manuele Kirsch Pinheiro, Carine Souveyet. Une Gestion de Ressources Sensible au Contexte & Opportuniste pour les Systèmes d'Information Pervasifs. INFORSID 2021, Jun 2021, Dijon, France. pp.123-138. hal-03877533

HAL Id: hal-03877533
https://paris1.hal.science/hal-03877533

Submitted on 29 Nov 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Une Gestion de Ressources Sensible au Contexte & Opportuniste pour les Systèmes d'Information Pervasifs

David Beserra[1], Manuele Kirsch-Pinheiro[1], Carine Souveyet[1]

1. Centre de Recherche en Informatique, Université Paris 1 Panthéon-Sorbonne
90 rue de Tolbiac, 75013 Paris, France
David-Willians.Beserra@univ-paris1.fr, Manuele.Kirsch-Pinheiro@univparis1.fr, Carine.Souveyet@univ-paris1.fr

RESUME. *Les Systèmes d'Information (SI) sont en train d'évoluer, en intégrant de plus en plus des ressources hétérogènes et mobiles, en complément aux ressources traditionnellement disponibles dans les centres de données et plateformes cloud computing, de façon à créer un nouveau genre de système d'Information nommée Système d'Information Pervasif (SIP). Les SIP combinent les SI traditionnels avec les environnements pervasifs pour : (i) améliorer les processus-métier, et (ii) capturer des données issus des environnements physiques et des infrastructures informatiques pour adapter leur comportement selon la situation en mains. Parmi les tendances actuelles, nous avons observé l'introduction du fog computing, qui favorise une utilisation opportuniste des ressources aux bords du réseau. L'objectif du fog computing est de permettre le traitement de données proche à ses sources de production. Ce nouveau paradigme permet de réduire certains problèmes communs aux plateformes cloud computing, tels quels ceux liés à la latence et à la confidentialité. Néanmoins, il introduit aussi des défis liés à la gestion des ressources. L'hétérogénéité des environnements fog computing complexifie l'ordonnancement de tâches ainsi que d'autres activités liées à la gestion de ressources pour les SIP. Dans ce travail nous présentons une architecture de gestionnaire de ressources qui favorise une gestion opportuniste des ressources d'un SIP, ainsi qu'un algorithme d'ordonnancement de tâches conçu pour fonctionner dans cette architecture. Le pierre angulaire de notre solution est l'utilisation des informations de contexte pour: (i) guider l'ordonnancement de taches dans des environnements hétérogènes et dynamiques ; (ii) exprimer plusieurs politiques dans les ressources et services pour détailler leurs possibilités d'utilisation (ressources) et d'exécution (services).*

ABSTRACT. *Information Systems (IS) are evolving, integrating more and more mobile & heterogenous resources, in addition to resources traditionally available on data centers and cloud platforms in order to build a new kind of IS called Pervasive Information Systems (PIS). They combine traditional IS with pervasive environments for (i) improving business processes and (ii) capturing data from the physical environments & IT infrastructures to adapt their functioning according to the situation in hand. Among current trends, we observe the introduction of fog computing, which promotes an opportunistic use of devices on the edge of the network. The goal is to bring computation nearest the user and the data production. This*

new paradigm allows to reduce common problems of cloud platforms, such as latency & privacy, but introduces also challenges related to the resource management. Heterogeneity of fog environments complexify scheduling and other resource management tasks, making of this a key aspect. In this paper, we tackle it by proposing a conceptual architecture supporting the opportunistic resource management of fog resources for PIS. The cornerstone of the solution is (i) the use of context information for guiding the tasks scheduling on heterogeneous & dynamic environment and (ii) to express various policies on the resources & services for detailing their use. We present the proposed architecture and the policies to express for supporting an opportunistic resource management on PIS.

Mots-clés : Gestion de Ressources Sensible au Contexte, Ordonnancement Sensible au Contexte, Systèmes d'Information Pervasifs.

KEYWORDS: Computing, Resource management, Context-aware scheduling, Pervasive information system.

1. Introduction

Les Systèmes d'Information évoluent de plus en plus vite ces derniers années, en intégrant de plus en plus des dispositifs mobiles et hétérogènes, tels quels les *Smartphones*, les *Tablets* (Pernici 2006), et les divers genres de capteurs et dispositifs embarqués de l'IdO (Internet des Objets) (Xu et al. 2014). Tout cela en addition aux ressources traditionnelles des centrés de données ou des plateformes *cloud computing*. Nous appelons cette nouvelle catégorie de SI de Système d'Information Pervasif (SIP), étant donnée que ces environnements pervasifs contribuent à améliorer les processus-métier ou/et les interactions des utilisateurs, mais aussi pour capturer des données de l'environnement physique et des infrastructures informatiques de façon à adapter leur comportement selon la situation en mains.

Cette intégration apporte aux SIP une hétérogénéité de ressources sans précédents, en combinant des dispositifs avec des caractéristiques variables, des fonctionnalités différentes et des degrés distincts de disponibilité. Elle permet également d'offrir des nouveaux services ou d'améliorer des applications existantes. Néanmoins, ces nouvelles technologies produisent aussi un impact non-négligeable dans la gestion des ressources des SIP, que devienne de plus en plus complexe parce que ces ressources sont devenues de plus en plus hétérogènes et dynamiques.

Plusieurs tendances peuvent être associées à ce phénomène : l'IdO, le *Big Data*, mais aussi le *Fog Computing*. Le terme *fog computing* est souvent utilisé pour exprimer l'idée de services proches aux utilisateurs et aux sources de production de données (Bonomi et al. 2012), résultant dans une expérience d'utilisation superior et dans une latence réduite (Huang et Wu, 2018). Le paradigme *fog computing* considere l'utilisation de ressources aussi proches que possible des sources de données et des utilisateurs finaux, en habilitant le traitement des données aux bords du réseaux. Il offre une alternative intéressante aux solutions complètement basées sur le *cloud computing*, notablement pour les applications de l'IdO (Ghobaei-Arani et al. 2019) (Hong and Varghese. 2019). Cela permet : (i) la détection de

l'emplacement (*location awareness*) et le support a la mobilité de l'utilisateur, (ii) une faible latence dans l'offre de services, et (iii) une haute évolutivité des services que ne peut pas nécessairement être prise en charge par les plateformes *cloud computing* (Ghobaei-Arani et al. 2019).

Le *fog computing* propose l'utilisation de dispositifs bas-de-gamme en tant que serveurs de proximité, qui peuvent performer des services basiques et prétraiter les données. Néanmoins, l'hétérogénéité et la dynamicité de tels dispositifs requirent un placement de données et de tâches adéquat (Breitbach et al. 2019). Les dispositifs de *fog* sont hétérogènes, et il n'y a pas de garantie que ces dispositifs auront des capacités similaires. Ils peuvent éventuellement avoir des capacités très différentes en termes de quantité de mémoire RAM, performance de CPU, espace de stockage, et bande passante de réseaux. Aussi, la connectivité réseau aux dispositifs *fog* ne peut pas être garantie, car ils ne sont pas si fiables que les serveurs des *clouds* (Hao et al. 2017). Outre, les ressources sont éparpillés sur des plateformes personnelles et véhiculaires hétérogènes et intermittemment connectées (Huang et Wu, 2018). Cet environnement hautement variable et imprédictible demande de considérer la gestion de ressources comme l'un des problèmes les plus difficiles dans l'univers *fog* (Ghobaei-Arani et al. 2019).

En effet, l'hétérogénéité et la dinamicité des environnements *fog* rendent complexes les activités liées à la gestion de ressources. Particulièrement quand comparés aux environnements *cloud* (Hong and Varghese. 2019). Pour être efficaces, les plateformes *fog* doivent assurer que les services soient performés au plus proche possible des dispositifs d'extrémité et que les ressources informatiques soient attribuées selon leurs capacités en temps d'exécution. Puisque les applications *fog* sont typiquement distribuées à travers plusieurs nœuds (les termes nœud et ressource sont utilisés de façon concomitante dans cet article) hétérogènes, décider dans quelle ressource exécuter une tâche est plus difficile que dans les environnements *cloud* (Hao et al. 2017). Plus que jamais l'ordonnancement dans ces environnements doit pendre en compte les capacités des ressources disponibles ainsi que leur état actuel, ce que mène à la notion de sensibilité au contexte. La sensibilité au contexte peut être vue comme la capacité d'un système d'observer sont environnement et d'adapter sont comportement en fonction de cela (Baldauf et al. 2007). L'information de contexte peut être utilisée pour caractériser l'état actuel de la ressource dans cet environnement dynamique, en offrent des indications importantes sur la capacité que une ressource a pour exécuter (ou non) une tâche donnée. Utiliser ces informations en tant que base pour l'ordonnancement de tâches ouvre des nouvelles perspectives pour un ordonnancement « intelligent » et opportuniste.

Nous considérons comme ordonnancement opportuniste l'ordonnancement que essaye de profiter des ressources en mains, quand ils sont disponibles, selon leurs capacités disponibles et restrictions, d'une façon indéterministe. Au lieu de considérer l'optimisation d'un pool de ressources dédies, comme dans l'ordonnancement traditionnel qu'on trouve dans l'informatique d'haute performance, l'objectif principal ici est de permettre l'exécution de tâches dans des

ressources de proximité toujours que possible et sans surcharger telles ressources, dont le propos primaire n'est pas nécessairement exécuter ce genre de tache.

Néanmoins, pour proposer tel ordonnancement sensible au contexte dans les Systèmes d'Information Pervasifs (SIP), une architecture conceptuelle doit être considérée. Dans un côté, l'information de contexte demande un support adéquat pour pouvoir gérer leur incertitude et nature dynamique de son acquisition jusqu'à sa modélisation (Kirsch-Pinheiro et Souveyet, 2018). Dans un autre côté, les SIP peuvent être vus comme un organisme vivant que doit évoluer avec leur organisation. Cela fait que l'extensibilité soit un aspect-clé pour les futures applications et infrastructures.

Dans cet article nous proposons une architecture conceptuelle pour la gestion de ressources particulièrement conçue pour supporter l'hétérogénéité et la dynamicité des environnements *fog*, ainsi que les besoins d'extensibilité des SIP. Nous discutons aussi les politiques de placement et d'ordonnancement de tâches requises dans un ordonnancement sensible au contexte pour les environnements pervasifs.

Cet article est organisé de la façon suivante : la section 2 discute les travaux connexes en *fog computing* ; la section 3 introduit notre architecture conceptuelle, pendant que la section 4 présente les politiques de placement et d'ordonnancement que peuvent être exprimées dans des ressources et dans des services d'un SIP, avec l'algorithme d'ordonnancement, avant de conclure dans la section 5.

2. Travaux Connexes

Fog computing est un paradigme émergent qui gagne de plus en plus l'attention de la communauté de Systèmes d'Information parce qu'il offre une alternative intéressante aux solutions « *tout-dans-le-cloud* ». En fait, malgré ses avantages, le *cloud computing* a aussi des nombreux inconvénients (Chen et al. 2017) (Hofmann and Woods, 2010), souvent liées à la latence et à la confidentialité. Par exemple, les applications qui s'appuient seulement dans des serveurs éloignés peuvent présenter des défaillances ou des délais non négligeables si la latence est trop élevée. Placer telles applications dans des dispositifs dans la périphérie du réseau peut réduire la quantité de transferts de données en explorant au même temps les capacités de traitement de données et de stockage proposés par ces ressources. Le principe de base derrière le *fog computing* est alors d'explorer le pouvoir de traitement de ces ressources qui se trouvent dans la périphérie du réseau pour accomplir des tâches usuellement déléguées à des facilités éloignées, comme les plateformes *cloud.*

Diverses initiatives partagent ce principe avec le *fog computing* (Huang et Wu, 2018): les *mobile clouds* (Huang et Wu, 2018), l'*edge computing* (Dey et al. 2013) (Olaniyan et al. 2018), l'informatique osmotique (Villari et al. 2016), les *cloudlets* (Satyanarayanan et al. 2006), juste pour nommer certaines. Toutes partageant le même principe : distribuer des tâches informatiques dans des ressources de proximité de façon à mieux répondre les besoins des applications, tels quels la réduction de la latence de communication ou la réduction des transferts de données à travers le réseau.

Par ailleurs, le terme *fog computing* est souvent associé aux applications de l'IdO (Ghobaei-Arani et al. 2019) (Hong and Varghese. 2019) comme une façon de traiter facilement des données collectés par capteurs attachés à des dispositifs embarqués. Toutefois, d'autres applications peuvent aussi se bénéficier de ce principe dans un Système d'Information. Par exemple, les applications d'analyse de données (surtout celles de *Big Data*), mais aussi des applications interactives ou des applications qui demandent le traitement de donnés en temps réel.

Similairement au *cloud computing*, le *fog computing* permet de fournir des ressources de traitement, de stockage, et de réseau. Néanmoins, plusieurs défis émergent de leur utilisation (Ghobaei-Arani et al. 2019) (Hong and Varghese. 2019) (Yi et al. 2015), notablement sur comment gérer les ressources et déployer sur eux des services adéquats, en considérant que le *fog* potentiellement s'étend du réseau local jusqu'à le *cloud*. En raison de la dynamicité, de la hétérogénéité, et et incertitude de ces environnements, un mécanisme de gestion de ressources est inévitable pour faire le *fog computing* devenir une réalité (Ghobaei-Arani et al. 2019). En fait, contrarièrent aux ressources de *cloud*, les ressources dans le périphérie du réseau sont : (i) limités, c'est-à-dire : ils ont une capacité de traitement de données réduite, car ils ont des processeurs plus petits à faible consommation énergétique ; (ii) hétérogènes, en incluant des processeurs de plusieurs architectures différentes ; et (iii) dynamiques, étant donnée que leur charge de travail change, et les applications compètent pour les ressources limités. Par conséquent, la gestion des ressources est un des défis-clé en *fog computing* (Hong and Varghese. 2019).

Les plateformes *fog* doivent faire face à un *pool* de ressources dynamique et hétérogènes dont la composition et l'état peuvent varier pendant l'exécution. Telles ressources ne sont pas dédiés à ces plateformes, car leur propos principal n'est pas d'exécuter des tâches de la plateforme *fog*. Tous ces aspects affectent la gestion de ressources, en commençant par son organisation générale. Généralement, les solutions adoptées par les plateformes *cloud* s'appuient sur un *hardware* spécifique ou sur des serveurs centralisés, et cela limite l'évolutivité et la compatibilité des solutions avec des dispositifs génériques. Donc, ces solutions ne sont pas adéquates aux environnements *fog* dynamiques et hétérogènes. C'est aussi le cas des solutions basées sur des topologies hybrides (Breitbach et al. 2019) (Edinger et al. 2017) (Garcia-Lopez et al. 2015), dans lesquelles un serveur (*broker* ou *proxy*) est en charge d'un ensemble de nœuds dans un certain périmètre, ce que le rend vulnérable. Aussi, des qu'un serveur centralisé est responsable pour traiter les données et pendre des décisions dans les approches centralisées, cela limite l'évolutivité à cause de ses limitations en termes de capacité de bande passante et de traitement. En opposition, les approches décentralisées sont hautement évolutives, mais elles introduisent une surcharge non-négligeable dans la communication.

Une autre caractéristique importante des environnements de *fog* est son incertitude. Non seulement les ressources peuvent apparaître et disparaître dans n'importe quel moment, comme aussi des nouvelles tâches peuvent être proposées dans n'importe quel moment. Ce n'est pas toujours possible d'avoir toutes les connaissances nécessaires avant l'ordonnancement dans tels environnements hétérogènes. Aussi, les approches statiques ne peuvent pas fournir un ordonnancement optimal pour les ressources *fog*, pointant vers l'ordonnancement

dynamique que fournit des tâches dont l'heure d'arrivée est inconnue, et ces tâches sont ordonnancées au moment où elles sont soumises (Ghobaei-Arani et al. 2019).

En autre, les environnements *fog* peuvent être caractérisés comme étant des environnements multi-tenant. Ce terme fait référence à quand (ou non) les ressources sont partagées entre plusieurs entités avec le but d'optimiser l'utilisation des ressources et leur efficacité énergétique (Hong and Varghese. 2019). Typiquement, les ressources placés aux bords du réseau peuvent héberger plusieurs applications appartenant à plusieurs utilisateurs (selon la classification de (Hong and Varghese. 2019)), et pas nécessairement toutes ces applications seront exécutées comme une partie (ou au nom) de la plateforme *fog*. Le gestionnaire de ressources n'est pas nécessairement en charge de toutes les tâches en exécution sur chaque nœud, ce que rendre essentielle l'observation de l'état du nœud.

Donc, le placement des données et des applications doit considérer les conditions en temps d'exécution sur chaque nœud, de façon à consommer leurs ressources de façon efficace. Aussi, il doit prioriser l'utilisation des nœuds qu'ont plus de ses capacités disponibles en détriment de ceux qui sont déjà surchargés par d'autres tâches, étant donnée que ces nœuds ne sont pas de nœuds dédiées, comme ceux qu'on trouve dans les infrastructures de cluster. Ainsi, le contexte d'exécution des nœuds doit être considéré par des fins d'ordonnancement. Le contexte peut être défini comme toute l'information que peu être utilisée pour caractériser la situation d'une entité (une personne, un endroit, un objet, etc.) considérées comme étant pertinentes à l'interaction entre l'utilisateur et le système (Dey 2001).

Différentes informations peuvent être considérées comme étant des informations de contexte, selon l'objectif su système : mémoire disponible, charge de CPU, stockage disponible, connexion réseau, etc. Des travaux comme (Cassales et al. 2016) (Kumar et al. 2012) démontrant l'intérêt de considérer des informations de contexte dans l'ordonnancement de tâches dans des environnements hétérogènes, pendant que (Breitbach et al. 2019) (Edinger et al. 2017) ont adressé cette question dans les environnements *fog*. Edinger et al. (Edinger et al. 2017) propose un ordonnanceur tolérant aux défaillances que dans lequel un *broker* collecte des informations de contexte des *fog* nœuds et estime leur stabilité et fiabilité de façon à placer des tâches dans les nœuds les plus appropriées. En (Breitbach et al. 2019), les auteurs se concentrent sur le placement de données, en essayent d'estimer la quantité de répliques nécessaires pour une certaine donnée ainsi que le dégrée de stabilité du nœud. Les tâches sont alors distribuées selon la présence de répliques et un indicateur de performance. Une fois encore, nous observons la présence d'un *broker* centralisé que décide sur le placement de données et de tâches sur les nœuds.

Donc, pour fournir une gestion de ressources opportuniste pour les environnements *fog*, il est nécessaire de fournir un ordonnancement de tâches sensible au contexte et considérer une organisation décentralisée où les ressources peuvent collaborer sans un nœud central de façon à exécuter des tâches sur cet environnement sans connaissance préalable de ces tâches. Dans la prochaine section nous proposons une architecture qui adresse ces exigences en identifiant les « blocs de construction » que permettent des solutions évolutives et extensibles. Avoir une telle architecture évolutive est une étape obligatoire pour introduire les plateformes

de *fog* dans les SIP, étant donnée que ces systèmes devraient évoluer au même temps que les besoins et les restrictions de leurs organisations.

3. Une architecture de Gestion de Ressources Sensible au Contexte

Dans notre architecture, les entités gérées par le gestionnaire de ressources sont les Ressources et les Services. Les *Ressources* (c'est-à-dire : les nœuds) sont définis par leur habilité d'exécuter des tâches ou des services. Les *Services* sont des modules d'application autonomes que doivent être ordonnancés et exécutés (c'est-à-dire : les tâches). Avant de détailler l'architecture de gestionnaire de ressources que nous proposons pour les SIP, il faut expliquer quelles sont les caractéristiques des SIP que doivent être considérées.

3.1. Caractéristiques de la situation de la gestion de ressources

Comme mentionné avant, les ressources que seront gérés sont hétérogènes et dynamiques et le pool de ressources peut être partagé par plusieurs organisations et propriétaires. Cette gestion vise à fournir l'ordonnancement et l'exécution de services sur demande, ce qui signifie que les exécutions de tâches ne peuvent pas être anticipées. La caractéristique spécifique de cette gestion n'est pas de chercher d'optimiser l'utilisation de ressources (par exemple : répartition de charge entre les ressources, ou minimiser la quantité de ressources utilisées), mais plutôt de proposer une usage opportuniste de ceux-là pour exécuter des services sur demande. Cela signifie que les critères ou métriques observées peuvent varier selon les ressources ou services qu'on observe/considère.

En ce que concerne aux SIP, il est actuellement possible d'envisager un grand nombre de ressources à gérer et de services à exécuter. Étant donné que l'évolutivité est une des caractéristiques majeures des SIP, nous inférons qu'une architecture de gestionnaire de ressources distribuée est plus adaptée à ceux-ci qu'une architecture centralisée. Il est aussi nécessaire d'organiser les ressources des SIP en plusieurs communautés, pour pouvoir contrôler la gestion de l'évolutivité. Les SIP doivent être capables d'évoluer en réponse à son environnement, mais aussi pour améliorer son processus-métier en réponse aux décisions venues du niveau métier.

Finalement, l'utilisation de stratégies opportunistes et distribuées dans la gestion de ressources implique qu'une ressource doit être capable de décider par soi-même quand elle exécute un service ou le transmet vers la ressource plus proche d'elle. En d'autres termes, un comportement opportuniste implique en considérer une stratégie de « meilleur effort » pour l'exécution de services au lieu d'une stratégie déterministe.

3.2. Architecture Conceptuelle du Gestionnaire de Ressources

Dans notre architecture, la gestion de ressources est distribuée et le gestionnaire de ressources est idéalement présent sur chaque ressource. En ce que concerne son organisation physique, les ressources sont organisées typologiquement dans un réseau P2P. L'architecture du gestionnaire est composée par un ensemble de modules dont chacun est responsable pour accomplir une fonction spécifiquement, et chaque ressource exécute (idéalement) une instance de chaque module. L'ensemble des modules qui composent l'architecture est illustré dans la Figure 1.

Dans notre architecture, le composant ***« Gestionnaire de Communications »*** est le responsable pour assurer la communication entre les gestionnaires de ressources installés dans des nœuds voisins. Il doit garder et actualiser une liste de ressources voisines de temps en temps. Les communications possibles entre les ressources sont : (i) envoyer/recevoir un service pour être exécuté après une décision prise par un ordonnanceur ; et (ii) envoyer/recevoir des informations de contexte (ou des propriétés) depuis la ressource pour supporter l'ordonnanceur dans sa décision.

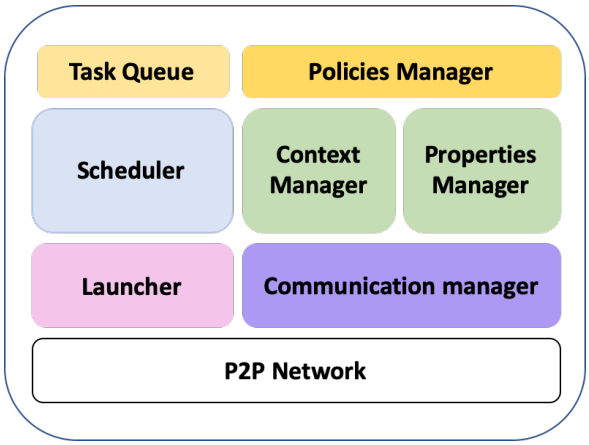


Figure 1. Architecture conceptuelle du gestionnaire de ressources.

À son tour, la ***« File d'Attente »*** est le composant responsable pour recevoir les services que seront prises en compte par l'ordonnanceur local, et les ajouter dans une file d'attente. Nous assumons que certaines connaissances préalables sur les services sont disponibles avant leur soumission. Le composant File d'Attente notifie aussi l'ordonnanceur quand des nouveaux services sont arrivés.

Le service d'ordonnancement demande des informations sur le nœud et sur les services à exécuter. Nous proposons de considérer ces informations comme faisant partie de deux catégories différentes : un ensemble de *propriétés*, nommé *descripteur*, et un ensemble *d'éléments de contexte* nommé *contexte*. Le contexte et les propriétés sont définis par chaque nœud **ainsi que** pour chaque service à exécuter. Les propriétés ne sont pas censées de changer en fonction du temps

pendant que les informations de contexte sont des informations qu'on doit observer pendant le temps. Les questions relatives à la capture et a la gestion de contexte ne sont pas adressées dans cet article et seront adressées dans une autre étude.

Pour être en mesure de gérer les propriétés et les informations de contexte de façon à rendre ces informations utiles pour la gestion des ressources, nous avons ajouté deux autres composants à notre architecture : le ***« Gestionnaire de Contexte »***, et ***« Gestionnaire de Propriétés »***. Le premier est responsable pour observer le contexte de la ressource locale, et pour demander le contexte des ressources voisines quand nécessaire. Il est aussi responsable pour évaluer les exigences des services par rapport aux informations de contexte. Le deuxième est responsable pour accéder et manipuler les propriétés d'une ressource locale ou d'un service considéré par l'ordonnanceur (local ou d'un des ressources voisines).

Dans notre architecture, des *politiques* doivent être définies extérieurement à l'algorithme de décision et au gestionnaire, à la fois au niveau des ressources comme au niveau des services. Dans un point de vue d'un nœud, les politiques sont utilisées pour exprimer des restrictions qui permettent de contrôler (et de limiter) son utilisation opportuniste (c'est-à-dire, on peut réduire « l'espace d'opportunités » d'utilisation d'une ressource). Le composant de notre architecture responsable pour accéder et manipuler les politiques définies pour la ressource locale ainsi que les politiques liées aux services qui seront considérés par l'ordonnanceur est le ***« Gestionnaire de Propriétés »***.

L'élément central de l'architecture que nous proposons est le ***« Ordonnanceur »***. Il est chargé de décider quand un service sera (ou poudra) être exécuté dans la ressource locale. Pour pendre sa décision il peut utiliser les gestionnaires de politiques, de contexte, et de propriétés afin d'évaluer (selon un critère dévaluation donnée et variable en fonction de l'implémentation du module) si les politiques de la ressource locale et du service en considération sont satisfaites ou non. En fonction de cette évaluation, le module d'ordonnancement peut pendre les décisions suivantes: (i) exécuter le service localement, (ii) l'envoyer vers une autre ressource voisine, ou, (iii) le remettre dans la file d'attente locale.

Finalement, le composant **« Lanceur »** est responsable de mettre la tâche en exécution dans la ressource locale, après une décision favorable de l'ordonnanceur, et appelé par celui-là. Le défi plus important, ici, reste l'hétérogénéité des technologies qui peuvent être utilisées pour implémenter les tâches (services). L'utilisation de micro-services représente une approche intéressante pour surmonter ce défi. Dans ce cas, le lanceur est responsable pour résoudre les dépendances locales nécessaires pour permettre l'exécution des tâches.

Comme nous avons vu dans cette section, l'architecture proposée a comme plus value les notions de contexte, de propriétés, et les politiques. Le propos de la prochaine section est détailler ces trois notions et comment elles peuvent être utilisées pour gérer des ressources avec notre architecture.

4. Contexte, Propriétés, et Politiques pour la Gestion de Ressources

Les entités principales adressées par le gestionnaire de ressources sont les Ressources et les Services. Pour ces deux types d'entités, le gestionnaire peut exploiter deux types d'information : (i) les *propriétés* que les caractérisent ; et, (ii) les *informations de contexte* qui sont pertinentes d'être observées. L'utilisation opportuniste de ressources requière la définition de *politiques* à la fois dans les ressources et dans les services qui seront considérées pendant le processus d'ordonnancement. Dans cette section nous présentons et discutons un méta-modèle qui englobe toutes ces entités, ainsi que la formalisation de politiques.

4.1. Méta-modèle des entités manipulées par le gestionnaire de ressources

Notre méta-modèle décrit un ensemble d'entités pertinentes pour la gestion de ressources. Il est illustré dans la Figure 2. Dans notre méta-modèle, nous considérons que les entités qui sont gérés par le gestionnaire de ressources sont les entités des types Ressource et Service. Les ressources ont comme caractéristique majeure la capacité d'exécuter des services. À sont tour, nous considérons que les services sont des modules d'application autonomes définis au niveau de l'entreprise (niveau métier). Ces services sont caractérises par: (i) l'intention qui les utilisateurs peuvent atteindre par son exécution (c'est-à-dire, un objectif : (Najar et al. 2011), (Najar et al. 2015)) ; et (ii) les types de données qu'il consomme (*input data*) et qu'il produit (*output data*) lors de son exécution.

Ces deux entités sont caractérisées par un ensemble de *types-propriétés* : le *modèle descripteur.* Un type-propriété (désormais : *propriété*) est défini par un *nom*, un *identificateur*, et une *spécification sémantique*. Pour une propriété avec une seule valeur, une *spécification de valeur* doit être ajoutée, pendant qu'une propriété de valeur composite doit être désignée par la composition de plusieurs sous-propriétés. Les propriétés sont utilisées pour exprimer l'information qui ne change pas en fonction du temps. Dans notre architecture, le gestionnaire de propriétés aide l'ordonnanceur à accéder aux modèles descripteurs de la ressource locale et du service en train de se faire analyser par l'ordonnanceur. Il peut aussi : (i) restreindre les deux descripteurs en fonction des informations utilisées dans les politiques ; et (ii) identifier les propriétés manquantes.

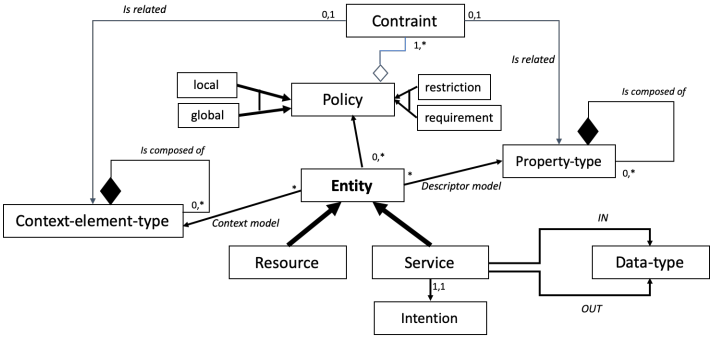


Figure 2. Méta-modèle d'entités utilisées par le gestionnaire de ressources

Par exemple, un service peut avoir les propriétés suivantes: *processus métier de référence*, *unité commerciale* (qui offre le service), *niveau de sécurité requis*, *utilisateur ciblé*, etc. Tous ces exemples peuvent être considérés comme étant des propriétés avec une seule valeur. Néanmoins, la propriété *mémoire installée* d'une ressource donnée peut être définie comme une composition des propriétés *quantité installée* et *fréquence d'opération*. En plus des propriétés techniques de base normalement considérées par les ordonnanceurs (*mémoire installée, processeurs, etc*.), les ressources peuvent aussi avoir des propriétés issues du niveau métier, telles quelles : *unité commerciale* (qui dispose primairement de la ressource), *administrateur*, ou *niveau de sécurité assuré*. Toutes ces informations sont considérées comme étant des propriétés parce que elles ne changent pas en fonction du temps. Les avantages issues de l'utilisation d'un descripteur configurable pour les services et pour les ressources sont : (i) customiser leur caractérisation pour la situation en mains, (ii) ajouter des informations qui viennent à la fois du niveau technique, mais aussi du niveau métier, et (iii) exploiter les deux pour la gestion de ressources.

Dans notre méta-modèle, le *modèle de contexte* se concentre sur les informations relatives aux ressources et aux services qui évoluent en fonction du temps et qui requièrent un mécanisme pour les observer. Un modèle de contexte est exprimé comme un ensemble de *types-élément-de-contexte* (désormais : *élément de contexte*). Similairement aux propriétés, un élément de contexte doit être défini par un *nom*, un *identificateur*, et une *identification sémantique* correspondante à un modèle de contexte. En autre, un élément de contexte basique doit être caractérisé par une *spécification de valeur* et aussi une référence au *type de capteur* qui l'observe. Un élément de contexte composé est défini par le lien de composition avec ses sous-éléments.

Au niveau technique, nous pouvons observer plusieurs informations de contexte dans une ressource donnée, telles quelles : *nombre de cœurs de processeur disponibles*, *occupation de mémoire*, *nombre de services invités en exécution* (des services appartenant à d'autres organisations qui ne sont pas les propriétaires de la ressource), sa *localisation* (pour les ressources mobiles), etc. Toutes ces informations sont considérées comme contexte parce que elles peuvent changer en fonction du temps, mais aussi parce que ces informations sont liées à un capteur spécifique afin d'être observées.

Les avantages d'avoir un modèle de contexte associé aux ressources et aux services sont : (i) pouvoir customiser pour chaque ressource/service les éléments de contexte qui doivent être observés, (ii) ajouter des informations provenant à la fois des niveaux métier et technique, et (iii) les exploiter pour la gestion des ressources.

Le gestionnaire de contexte ajoute, à chaque donnée observée pour un élément de contexte, un descripteur de qualité aidant à déterminer sa qualité. Le gestionnaire de contexte aide l'ordonnanceur à accéder au modèle de contexte de la ressource locale et également au modèle de contexte du service considéré. Il peut également (i)

restreindre les deux contextes en fonction des informations utilisées dans les politiques; et (ii) identifier les éléments de contexte manquants.

Il est à noter qu'une manière flexible et extensible de mettre en œuvre ces deux modèles est nécessaire en raison de leur amélioration ou évolution dans le temps tant au niveau métier qu'au niveau technique. Le descripteur et les modèles de contexte sur les ressources et services peuvent être utilisés pour exprimer des politiques. Ils seront détaillés dans la section suivante.

4.2. Définition de Politiques

Une politique est définie comme une conjonction de contraintes simples, chacune d'entre elles basée sur un élément de contexte ou sur une propriété. Le but d'une politique est d'exprimer l'ensemble d'états souhaités sur lesquels un service peut être exécuté sur une ressource. Ces états souhaités sont appelés *exigences* alors que les politiques exprimant des états interdits sont appelés *restrictions*. Chaque politique a un attribut de pondération (c'est-à-dire : un poids-propre), pour exprimer son importance pour l'ordonnanceur.

Les *politiques attachées à une ressource* expriment comment la ressource peut être utilisée de façon opportuniste. Ce genre de politique peut établir une contrainte sur les services autorisés à être exécutés sur une ressource, ou sur le sous-ensemble de ses capacités qui pourra être utilisée pour cela.

Pour illustrer l'utilisation de politiques dans la gestion de ressources, considérons un scénario où deux entreprises installées dans un centre commercial (Entreprise-A et Enterprise-B) partagent une infrastructure informatique afin d'exécuter des services sur demande. Ces services peuvent être exécutés sur des ressources empruntées pour plusieurs raisons, par exemple: (i) parce que la ressource où le service est originalement placé est déjà surchargée au moment où il essaye de se faire exécuter, (ii) parce que on souhaite réduire la consommation énergétique des ressources mobiles afin d'augmenter la longévité de leurs batteries, ou alors (iii) parce que les ressources de l'une ou l'autre organisation ont des caractéristiques uniques et qui sont nécessaires à l'exécution du service, etc.. Les ressources partagées par l'entreprise A sont: 1 serveur (RA1#), 3 notebooks (RA2#, .., RA5#) et 2 écrans tactiles positionnés à des emplacements stratégiques du centre commercial (RA6#, RA7#). À son tour, l'entreprise B partage 5 écrans tactiles (RB#1, .., RB#5).

Lors de la modélisation de ce scénario pour le gestionnaire de ressources, l'entité Ressources a été spécialisée dans les entités Serveur, Notebook, Smartphone et Écran-Tactile. Le modèle-descripteur de ce scénario a toutes les propriétés mentionnées dans les sections précédentes. En outre, il inclut également la propriété composée *batterie* (*capacité*, *tension*, *courant*) pour les Notebooks et pour les Smartphones. De même, le modèle de contexte utilisé dans ce scénario contient tous les éléments de contexte mentionnés ci-dessus, ainsi que quelques autres qui peuvent être observés dans les Notebooks et dans les Smartphones: *source-alimentation-en-use*, *niveau-de-charge-batterie*, *temps-de-charge*, et *temps-de-décharge*, etc.

Afin d'assurer ses propres intérêts et d'exécuter plus de ses propres services que de services d'autres organisations, l'Enterprise-A énonce les politiques suivantes (politiques de ressources, PR) pour toutes ses ressources:

- **PR1:** une ressource peut utiliser jusqu'à la moitié de ses processeurs pour effectuer des services invités (**exigence**); et,
- **PR2:** une ressource ne doit pas exécuter de services complémentaires si le nombre de cœurs disponibles est inférieur à 2 et si sa température est supérieure à 50 ° C (**restriction**).
- **PR3:** une ressource ne doit pas exécuter de services complémentaires si le niveau de charge de sa batterie est inferieur à 40% (**restriction**).

Ces politiques sont résumées comme suit (le premier paramètre est le poids de la politique, tandis que le deuxième paramètre est l'entité qui possède la politique, le troisième paramètre sont les contraintes de politique):

- PR1: (1, Resource, DIF(*service.Owner, "Enterprise-A"*) & GTEQUAL(*local_node.context.NumberOfCoresAvailable*, *local_node.property.NumberOfCores/2*));
- PR2: (2, Resource, LT(*local_node.context.NumberOfCoresAvailable*, 2:0) & GT (*local.node.context.Temperature*, 50:5));
- PR3: (2, Resource, LT(*local_node.context.BatteryChargeLevel*, 40:10));

Les politiques attachées à un service doivent expliquer ce qui est requis ou interdit sur une ressource pour exécuter le service. Une politique attachée à un service peut faire référence à des informations de contexte ou à des propriétés relatives à une ressource, car l'exécution d'un service sur une ressource donnée est attachée aux capacités disponibles de celle-là ou à leurs caractéristiques (techniques et administratives).

Suivant notre scénario, l'Enterprise-B dispose d'un service (ServiceB1) qui peut être exécuté dans des ressources appartenant à d'autres organisations. Afin d'augmenter les chances de que ce service s'exécute sur des ressources capables de l'exécuter dans des conditions satisfaisantes, l'Enterprise-B a défini certaines politiques liées à ce service (politiques de service, PS):

- **PS1:** Le ServiceB1 nécessite 5 personnes devant une ressource afin d'être exécuté (**exigence**);
- **PS2:** Le ServiceB1 ne doit pas être exécuté dans des ressources avec moins de 512 Mo de mémoire disponible (**restriction**);
- **PS3 :** Le ServiceB1 ne doit pas être exécuté dans des ressources dont le niveau de sécurité assuré est « faible » (**exigence**);

Ces politiques pour le ServiceB1 peuvent être résumées comme suit (le premier paramètre est le poids de la politique, tandis que le deuxième paramètre est l'entité qui possède la politique, le troisième paramètre sont les contraintes de la politique):

- PS1:(3, ServiceB1, GT(*local_node.context.Presence, 5*));
- PS2:(1, ServiceB1, LT(*local_node.context.MemoryAvailable*, 512));
- PS3:(1, ServiceB1, EQUAL(*local_node.descripteur.SecurityLevel, weak*));

Une politique peut être *locale* à un service ou à une ressource, mais elle peut également être *globale* sur un ensemble de services ou de ressources appartenant à la même organisation. Toujours dans notre exemple, les politiques précédemment définies pour le ServiceB1 sont locales à ce service, pendant que les politiques définies pour l'ensemble de ressources de l'Enterprise-A sont des politiques globales.

Dans une ressource, l'ordonnanceur local utilise les politiques afin de pendre sa décision sur l'exécution (ou non) d'un service localement. L'ordonnanceur fait appel au gestionnaire de politiques pour extraire l'ensemble des politiques (locales et globales) liées à la ressource locale et l'ensemble des politiques (locales et globales) liées aux services. Afin d'aider l'ordonnanceur à évaluer les politiques, le gestionnaire de contexte et le gestionnaire de propriétés extraient respectivement le contexte et le descripteur liés à la ressource *n* et au service *s* et les restreindront aux éléments utilisés dans les politiques.

Une API d'opérateurs utilisés pour exprimer des contraintes est proposée afin d'opérationnaliser l'évaluation des politiques, en gérant aussi la qualité du contexte observé et les éléments de contexte manquants (c'est-à-dire : les éléments qui n'ont pas pu être observés). Le résultat renvoyé par ces operateurs n'est pas un booléen traditionnel mais une valeur réelle entre 0 et 1. Cette valeur indique la proximité entre la valeur observée pour un certain élément de contexte et la valeur attendue pour ce même élément, tout en considérant la qualité de l'information de contexte observée. Des implémentations personnalisées de certains opérateurs fondamentaux sont fournies, telles quels : l'égalité (EQUAL), la non-égalité (DIF), inférieure à (LT), supérieure à (GT), etc. Pour illustrer l'utilisation générale des opérateurs, considérons l'opération EQUAL(*local_node.context.Température, 50:5*). Cette opération attend à ce que la valeur observée pour l'élément de contexte *Température* soit égale à 50, et accepte comme observations valables pour cet élément de contexte toutes les observations qui ne varient pas plus que 5 points (vers le haut ou vers le bas). Cette fonction renvoie une valeur comprise entre 0 et 1 en fonction de la valeur observée pour l'élément de contexte *Temperature*, de l'intervalle de précision donné, et du descripteur de qualité associé à l'observation.

Les critères utilisés pour les décisions d'ordonnancement sont spécifiques à chaque ressource et à chaque service à prendre en compte. Cette personnalisation est soutenue dans notre architecture par la notion de politique et par la gestion du contexte dans la ressource. Dans la section suivante, nous présentons un algorithme

d'ordonnancement de tâches sensible au contexte et qui utilise les différents composants de l'architecture de gestionnaire de ressources que nous proposons, ainsi que politiques que nous venons de décrire et que s'expriment avec les éléments mentionnés dans le méta-modèle.

4.3. Un Algorithme d'Ordonnancement pour le composant Ordonnanceur

Dans notre algorithme (Algorithme 1), l'ordonnanceur décide sur l'exécution (ou non) d'un service (appelé *s* dans l'algorithme) dans la ressource locale (appelée *local_node* dans l'algorithme). L'algorithme d'ordonnancement commence à exécuter sur une ressource lorsque l'exécution d'un service est terminée ou quand le composant file d'attente informe à l'ordonnanceur de l'arrivée d'un service. Comme nous avons dit précédemment, les gestionnaires de politiques, de contexte, et de propriétés permettent d'accéder et d'extraire les informations provenant du contexte de la ressource et du descripteur du service en train d'être analysé par l'ordonnanceur, quand celui-là les demande.

L'algorithme de décision que nous proposons rassemble dans une même collection les politiques locales et globales, liées soit à la ressource *local_node*, soit au service *s*. Cependant, le traitement est différent si la politique est une exigence ou une restriction; c'est pourquoi nous avons deux collections appelées *PolReq* et *PolRest* (respectivement : collection d'exigences et collection de restrictions attachées à la ressource *local_node* ou au service *s*). L'évaluation de ces politiques nécessite le contexte de *local_node* (appelé *ExecutionContext*), et cette collection est générée par le gestionnaire de contexte, étant limitée aux éléments de contexte requis dans les politiques de *PolReq* et *PolRest*. Un traitement similaire est effectué par le gestionnaire de propriétés afin de produire la collection appelée *Properties*. Le gestionnaire de contexte peut également fournir les éléments de contexte manquants en fonction des politiques à évaluer. Les éléments manquants seront utilisés lors de l'évaluation d'une politique pour calculer une pénalité (voir la deuxième ligne marque en rouge sur l'Algorithme 2).

L'algorithme prend sa décision basée sur le résultat de la soustraction de la moyenne pondérée de l'évaluation de toutes les restrictions et de la moyenne pondérée de l'évaluation de toutes les exigences. Un bonus est appliqué sur ce résultat pour tenir compte de la priorité attribuée au service *s* (ligne rouge sur l'Algorithme 1). Si le résultat est supérieur ou égal au *seuil d'exécution local*; le lanceur est appelé pour exécuter le service localement (troisième ligne bleue sur l'Algorithme 1). Si le service ne peut pas être exécuté localement, alors sa priorité est augmentée, et l'incrément de priorité est plus important si la ressource et le service ont le même propriétaire. Un seuil spécifique est utilisé pour déterminer si le service doit être transféré à un voisin. Si le résultat est inférieur à ce *deuxième seuil*, le service est remis dans la file d'attente locale (quatrième ligne bleue sur l'Algorithme 1).

Lorsque le service doit être transféré à un voisin; le problème devient alors « obtenir la liste des voisins du gestionnaire de communication et de sélectionner le voisin ». Pour sélectionner le voisin, l'exigence et la restriction les plus pondérées

sont sélectionnées dans l'ensemble des politiques attachées au service *s*. Le premier voisin satisfaisant ces deux politiques est sélectionné. Pour effectuer cette évaluation, le gestionnaire de communication est utilisé pour obtenir le contexte et les propriétés du voisin. Cela implique que les gestionnaires de contexte et de propriétés du voisin fournissent les informations requises. Le gestionnaire de communication prend en charge le transfert d'informations entre les deux ressources. La cinquième ligne bleue sur l'Algorithme 1 exprime la manière dont le service est transféré au voisin sélectionné en utilisant le gestionnaire de communication. À la fin, le service est ajouté dans la file d'attente du voisin sélectionné.

Algorithme 1. Algorithme d'ordonnancement sensible au contexte

```
/*
The resource manager is installed on a resource called local_node
S= Service.
PolReq : Collection of requirements policies attached to local_node and service s
PolRest : Collection of restrictions policies attached to local_node and service s
ExecutionContext : Collection of Context elements of the local node and service s
used in the policies belonging PolReq or PolRest
MissingContext : Collection of Missing context elements of local_node and service s
used in the policies belonging in PolReq or PolRest
Properties : Collection of Properties attached to local_node and service s
used in the policies belonging to  PolReq or PolRest
*/
Scheduler.decide (s, PolReq, PolRest, ExecutionContext, MissingContext, Properties)
SUM_Req=SUM_Treq=SUM_Rest= SUM_Trest=SUM=0;

For each req  IN  PolReq do // requirements handling
    SUM_Req = SUM_Req + (req.weight *
                evaluationPolicy (req, ExecutionContext, MissingContextElements, Properties)) ;
    SUM_Treq = SUM_Treq + req.weight ;

For each rest de PolRest do // restrictions handling
     SUM_Rest = SUM_Rest + (rest.weight *
                 evaluationPolicy (rest, ExecutionContext, MissingContextElements, Properties) ) ;
     SUM_Trest = SUM_Trest + rest.weight ;
SUM = ((1 + s.getPriority()) *((SUM_Req)/(SUM_Treq))) – ((SUM_Req)/(SUM_Trest)));
// decide if the service will be executed on the local node
if (SUM >= Scheduler.getThresholdRun())
  then Launcher.execute (s);
 else
      // Increment the priority of the service s
       if (s.getOwner()== local_node.getOwner())
      then  s.setPriority(s.getPriority () + Scheduler.getPriorityIncrementOwner() );
      else s.setPriority(s.getPriority () + Scheduler.getPriorityIncrementNotOwner());
     // Decide to put back the service to the local task queue or to transfer it to a neighbor
     if (SUM  < Scheduler.getThresholdNeighbor())
       then TaskQueue.addAgain(s);
     else
         // Get the list of neighbors from the communication manager
         L_neighbors = CommunicationManager.getNeighbors();
         PolRests-Maxweight=PoliciesManager.getMaxWeight-Restriction (PolS);
          PolReq-Maxweight= PoliciesManager.getMaxWeight-Re (PolS);
          // select the appropriate neighbor.
          target = SelectNeighbor(L_neighbors, PolReq, PolRest, Scheduler.getThresholdTransfer());
          if (target != null)
             then communicationManager.sendService(target, s); // transfer to the selected neighbor
             else TaskQueue.addAgain(s);
// end of decide
```

Comme nous l'avons vu dans la section précédente, une politique est une conjonction de contraintes, et l'Algorithme 2 explique comment une politique est évaluée. Dans cet algorithme, la fonction d'évaluation de politique *evaluationPolicy* renvoie un *taux de satisfaction* de la politique (une valeur réelle entre 0 et 1). L'évaluation de chaque contrainte de la politique nécessite le contexte d'exécution et les propriétés ainsi que les éléments de contexte manquants. La fonction d'évaluation de contraintes est spécifique selon le type d'opérateur utilisé dans l'expression de contrainte. Il gère: (i) la situation où la contrainte est basée sur une propriété; (ii) la situation où l'élément de contexte utilisé dans la contrainte est manquant; et (iii) la situation où la qualité de l'élément de contexte à considérer doit être utilisée pour estimer sa satisfaction. L'évaluation de la politique est la moyenne de l'évaluation de ses contraintes. Une pénalité basée sur l'incomplétude du contexte d'exécution peut être appliqué à la valeur renvoyée (la troisième ligne rouge sur la figure 4).

Algorithme 2. Algorithme d'evaluation de politiques

```
/* policy p
ExecutionContext : Context elements required for all the policies
attached to the local_node and the service s
MissingElements : Missing context elements for all the policies
attached to the local_node and the service s
POR : Properties attached to the local_node and the service s
 */
evaluationPolicy( p, ExecutionContext, MissingElements, POR )
Constraints  = p.getContraints() // Constraints composing the policy p
NbConst=constraints.size() ;
NbContextElements= Constraints.getNbContextElements () ;
RequiredContextElements=Constraints.getRequiredContextElements () ;
NbMissingElement=0;

// Determine the number of missing context elements for the policy p
for each ec in RequiredContextElements do

   if MissingElements.contains(ec)
    then NbMissingElement= NbMissingElement++ ;
SUM=0 ;
for each cons IN Constraints do
   // Evaluation of each constraint
   SUM = SUM + cons.evaluate (ExecutionContext, MissingElements, POR);
//Value must be between 0 and 1
SUM = SUM/NbConst ;
//penalty applied is proportional to the number of missing context elements
penalty = NbMissingElement / NbElementsContextePol ;

//Result is a value where 0 is completely not satisfied and 1 is completely satisfied
satisfactionRate = SUM * (1 – penalty);

RETURN satisfactionRate;
//end of evaluationPolicy
```

Pour illustrer le fonctionnement de notre ordonnanceur, supposons qu'à un moment donné une instance du ServiceB1 est arrivée à la file d'attente de la ressource RA1# de l'Enterprise-A. L'ordonnanceur récupère le ServiceB1 de la file d'attente et va pendre une décision sur son exécution. Pour le faire, il: (i) extrait les exigences et les restrictions du ServiceB1 et de la ressource RA1# avec l'aide du gestionnaire de politiques, (ii) observe le contexte et les propriétés de RA1# avec

l'aide des gestionnaires de contexte et de propriétés, (iii) évalue la conformité entre les politiques et les informations (contexte/propriétés) observées, et (iv) prend sa décision.

Supposons aussi que, lors des étapes (ii) et (iii), on vérifie que la politique PS3 du ServiceB1 est satisfaite par la ressource RA1# car le niveau de sécurité assuré par cette ressource (*SecurityLevel*) est élevé. Par contre, les politiques PS1 et PS2 ne sont pas complètement satisfaites par la ressource RA1#, car il n'y a personne devant (*Presence*) et il n'y a que 256 Mo de mémoire disponible (*AvailableMemory*). En plus, les politiques de la ressource PR1 et PR2 ne sont pas satisfaites non plus car il n'y a pas assez de cœurs de processeur disponibles. Par conséquent, l'ordonnanceur de la ressource RA1# décide de transférer le service à un voisin. Le premier voisin qu'il trouve est la ressource RA4# (un Notebook), mais la manque de l'information de contexte *Presence* ne peut pas satisfaire les politiques du ServiceB1, et l'algorithme cherche le prochain voisin. Finalement, on trouve que le voisin RA6# (un EcranTactile) peut satisfaire les politiques du ServiceB1, car elle a une présence de dix personnes, sa mémoire disponible est de 625 Mo, et son niveau de sécurité est moyen. Par conséquent, l'ordonnanceur de RA1# décide transférer le ServiceB1 vers la file d'attente de RA6#.

Notre exemple aide à illustrer comment l'algorithme d'ordonnancement que nous proposons permet de faire l'utilisation opportuniste d'une ressource par un service tout en adoptant un style «patate chaude». L'idée originale de cet algorithme est de fonder sa décision sur un ensemble de politiques définies: (i) au niveau du service pour exprimer les besoins métiers et techniques d'un service en termes d'environnement d'exécution; et (ii) au niveau de la ressource pour expliquer la limitation de son utilisation de manière opportuniste. De plus, il exploite des informations issues d'un modèle de contexte ou/et d'un modèle de descripteur sur les ressources et les services. Enfin, sa mise en œuvre est extensible car le modèle de contexte, le modèle de descripteur et les politiques peuvent être facilement modifiés.

5. Conclusions

L'un des principaux défis liées au *fog computing* est de fournir des mécanises d'ordonnancement des tâches qui puissent faire face à l'hétérogénéité et à la dynamicité ce type d'environnement. Dans cet article, nous abordons ce problème en considérant une architecture conceptuelle dans laquelle plusieurs composants collaborent pour proposer un ordonnancement sensible au contexte. Notre objectif avec une telle architecture est de promouvoir une utilisation opportuniste des ressources disponibles en fonction des capacités de chaque ressource et de son contexte d'exécution. Par conséquent, notre objectif n'est pas nécessairement de terminer les tâches plus tôt ou d'optimiser l'utilisation d'un pool de ressources, mais de faire de notre mieux pour exécuter les tâches chaque fois que cela est possible sans surcharger excessivement les ressources disponibles.

Nous supposons que pour rendre possible ce type de gestion de ressources dans les SIP, des politiques provenant du niveau métier ainsi que des niveaux techniques doivent être abordées et le contexte d'exécution des ressources et des services doit

être géré de manière appropriée. Nos travaux futurs consistent, dans un premier temps, à simuler l'algorithme d'ordonnancement proposé afin de valider sa pertinence pour l'objectif «faire une usage opportuniste des ressources» dans les SIP. Deuxièmement, nous réfléchissons à la manière d'améliorer la description des exigences des tâches et des ressources elles-mêmes afin de faire face aux contraintes des Systèmes d'Information, telles que la sécurité.

Notre objectif final reste de proposer un gestion de ressources qui exploite pleinement l'idée de contexte dynamique et variable traitant non seulement des ressources informatiques et des descriptions de tâches, mais étendue aux éléments métiers pertinents afin d'améliorer son impact sur les SIP.

Bibliographie

Baldauf, M., Dustdar, S., Rosenberg, F. (2007): A survey on context-aware systems. International Journal of Ad Hoc and Ubiquitous Computing 2(4), 263-277.

Bonomi, F., Milito, R., Zhu, J., Addepalli, S. (2012): Fog Computing and Its Role in the Internet of Things. In: Proceedings of the 1st MCC Workshop on Mobile Cloud Computing, pp. 13-16. ACM.

Breitbach, M., Schäfer, D., Edinger, J., Becker, C. (2019): Context-Aware Data and Task Placement in Edge Computing Environments. In: 2019 IEEE International Conference on Pervasive Computing and Communications (PerCom), pp. 1-10. IEEE.

Cassales, G.W., Schwertner Charão, A., Kirsch-Pinheiro, M., Souveyet, C., Steffenel, L.A. (2016): Improving the performance of Apache Hadoop on pervasive environments through context-aware scheduling. Journal of Ambient Intelligence and Humanized Computing 7(3), 333-345.

Chen, S., Zhang, T., Shi, W. (2017) : Fog Computing. IEEE Internet Computing 21(2), 4-6.

Dey, A. (2001): Understanding and using context. Personal and Ubiquitous Computing 5(1), 4-7.

Dey, S., Mukherjee, A., Paul, H. S., Pal, A. (2013): Challenges of Using Edge Devices in IoT Computation Grids. In: Int. Conf. on Parallel and Distributed Systems, pp. 564-569.

Edinger, J., Schäfer, D., Krupitzer, C., Raychoudhury, V., Becker, C. (2017): Fault-avoidance strategies for context-aware schedulers in pervasive computing systems. In: 2017 IEEE Int. Conference on Pervasive Computing and Communications (PerCom), pp. 79-88.

Garcia Lopez, P., Montresor, A., Epema, D., Datta, A., Higashino, T., Iamnitchi, A. et al. (2015): Edge-centric Computing: Vision and Challenges. SIGCOMM Comput. Commun. Rev. 45(5), 37-42, ACM.

Ghobaei-Arani, M., Souri, A., Rahmanian, A. A. (2019): Resource Management Approaches in Fog Computing: a Comprehensive Review. Journal of Grid Computing, DOI 10.1007/s10723-019-09491-1.

Hao, Z., Novak, E., Yi, S., Li, Q. (2017): Challenges and Software Architecture for Fog Computing. IEEE Internet Computing, 21(2), 44-53.

Hofmann, P., Woods, D.: Cloud Computing (2010): The Limits of Public Clouds for Business Applications. Internet Computing 14(6), 90-93.

Hong, C.-H., Varghese, B.: Resource Management in Fog/Edge Computing (2019): A Survey on Architectures, Infrastructure, and Algorithms. ACM Comput. Survey 52(5), Article n° 97.

Huang, D., Wu, H. (2018): Mobile Cloud Computing. Morgan Kaufmann.

Kirsch-Pinheiro, M., Souveyet, C.: Supporting context on software applications (2018): a survey on context engineering (Le support applicatif à la notion de contexte : revue de la littérature en ingénierie de contexte). Modélisation et utilisation du contexte, 2(1), ISTE OpenScience. https://www.openscience.fr/Le-support-applicatif-a-la-notion-de-contexte-revue-de-la-litterature-en/.

Kumar, K. A., Konishetty, V. K., Voruganti, K., Rao, G. V. P.: CASH (2012): context aware scheduler for Hadoop. In: Proceedings of the International Conference on Advances in Computing, Communications and Informatics, pp. 52-61. ACM.

Olaniyan, R., Fadahunsi, O., Maheswaran, M., Zhani, M. F. (2018): Opportunistic Edge Computing: Concepts, opportunities and research challenges. Future Generation Computer Systems 89, 633-645.

Pernici, B. (2006): Mobile information systems. Springer.

Satyanarayanan, M., Bahl, P., Caceres, R., Davies, N. (2009): The case for vm-based cloudlets in mobile computing. IEEE Pervasive Computing 8(4), 14-23.

Steffenel, L.A., Kirsch-Pinheiro, M. (2015): When the cloud goes pervasive: approaches for IoT PaaS on a mobiquitous world. In: Mandler B. et al. (eds), EAI Int. Conf. on Cloud, Networking for IoT systems (CN4IoT 2015), Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering (LNICST),.

Villari, M., Fazio, M., Dustdar, S., Rana, O., Ranjan, R. (2016): Osmotic Computing: A New Paradigm for Edge/Cloud Integration. IEEE Cloud Computing 3(6), 76-83.

Xu, B., Da Xu, L. Cai, H., Xie, C., Hu, J., Bu, F. (2014): Ubiquitous data accessing method in iot-based information system for emergency medical services. IEEE Trans. Industrial Informatics, 10(2), 1578-1586.

Yi, S., Hao, Z., Qin, Z., Li, Q. (2015): Fog Computing: Platform and Applications. In: Third IEEE Workshop on Hot Topics in Web Systems and Technologies, pp. 73-78. IEEE.

Najar, S., Kirsch Pinheiro, M., Souveyet, C. (2011): "The Influence of Context on Intentional Service". 5th Int. IEEE Workshop on Requirements Engineerings for Services (REFS'11), IEEE Conference on Computers, Software, and Applications (COMPSAC'11), Munich, Germany, pp. 470-475.

Najar, S., Kirsch-Pinheiro, M (2015): "Service discovery and prediction on Pervasive Information System", Journal of Ambient Intelligence and Humanized Computing, vol. 6, issue 4, June 2015, Springer Berlin Heidelberg, pp. 407-423. ISSN: 1868-5137. DOI: http://dx.doi.org/10.1007/s12652-015-0288-5