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 sont en train d'évoluer, en intégrant de ressources plus hétérogènes et mobiles, en complément aux ressources disponibles dans les centres de calcul et les plateformes cloud. Parmi les tendances conduisant à cette nouvelle génération, nommée Système d'Information Pervasif, se trouve le fog computing. Ce paradigme favorise une utilisation opportuniste des ressources en périphérie du réseau en déplaçant le traitement de données à proximité des utilisateurs et des sources de production. Néanmoins, l'hétérogénéité des environnements fog complexifie la gestion des ressources, et notamment l'ordonnancement des services nécessaires aux SIP. Nous proposons ici une architecture conceptuelle pour un gestionnaire de ressources d'un SIP qui favorise une gestion opportuniste de celles-ci, ainsi que son algorithme d'ordonnancement. Cette solution se caractérise par l'utilisation des informations de contexte pour : (i) guider l'ordonnancement de tâches dans ces environnements hétérogènes et dynamiques ; et (ii) exprimer les politiques relatives à l'usage des ressources et les exigences des services pour leur exécution).*

ABSTRACT. *Information Systems are evolving, integrating more and more mobile & heterogenous resources, in addition to those traditionally available on data centers and cloud platforms. Among the current trends leading to this new generation, called Pervasive Information Systems, we may cite the fog computing. This paradigm promotes an opportunistic use of devices on the edge of the network, by bringing computation nearest the user and the data production. Nevertheless, heterogeneity of fog environments complexify the resource management, and notably the task scheduling. In this paper, we tackle this issue by proposing a conceptual architecture supporting an opportunistic resource management of fog resources on PIS. The cornerstone of the solution is the use of context information for (i) guiding the tasks scheduling on heterogeneous & dynamic environment; and (ii) expressing the resources & services policies conditioning their use.*

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

KEYWORDS: *Opportunistic use of resources, Context-aware Resource management, Pervasive Information System.*

1. Introduction

Les Systèmes d'Information évoluent rapidement ces dernières années, en intégrant de plus en plus de dispositifs mobiles et hétérogènes, ainsi que des capteurs et des dispositifs embarqués pour l'IdO (Internet des Objets) (Xu *et al.* 2014), en complément aux ressources issues des plateformes de *cloud computing*. Cette nouvelle catégorie de SI est appelée Système d'Information Pervasif (SIP). L'intégration d'environnements pervasifs au SI doit permettre d'améliorer les processus métier, les interactions avec les utilisateurs, mais également de capturer les données issues de l'environnement et des infrastructures informatiques de façon à adapter leur comportement selon la situation.

Plusieurs tendances technologiques 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 des utilisateurs et des sources de production des données (Bonomi *et al.* 2012), permettant ainsi d'obtenir une expérience utilisateur de meilleure qualité et une latence réduite. Par exemple, les entreprises situées dans un centre commercial pourraient utiliser les ressources placées directement sur l'infrastructure informatique du centre pour fournir des services à la demande à leurs clients, sans avoir à déployer ces services sur des ressources distants. En ramenant le traitement des données à la frontière du réseau, le paradigme *fog computing* offre une alternative intéressante aux solutions complètement basées sur le *cloud computing*, notamment pour les applications de l'IdO (Hong et Varghese. 2019).

Le *fog computing* propose l'utilisation de dispositifs bas-de-gamme en tant que serveurs de proximité, lesquels peuvent exécuter des services basiques et prétraiter des données. Néanmoins, l'hétérogénéité et le caractère dynamique de tels dispositifs requièrent un placement pertinent des services (Breitbach *et al.* 2019). En effet, ces dispositifs peuvent avoir des capacités très différentes en termes de quantité de mémoire RAM, de performance de CPU, d'espace de stockage, et de bande passante réseau. De plus, la connectivité réseau de ces dispositifs ne peut pas non plus être garantie, car ils ne sont pas aussi fiables que les serveurs *cloud* (Hao *et al.* 2017). L'usage de ressources *fog* ouvre également la possibilité d'un partage de ressources entre organisations, comme dans l'exemple d'un centre commercial, où plusieurs entreprises pourraient se partager des ressources sur la même infrastructure. La gestion des ressources dans ce type d'environnement devient un problème majeur dans l'univers *fog* (Ghobaei-Arani *et al.* 2019).

Puisque les applications *fog* sont typiquement réparties sur plusieurs ressources hétérogènes, décider à quelle ressource est affectée l'exécution d'une tâche est plus difficile que dans les environnements *cloud* (Hao *et al.* 2017). Plus que jamais l'ordonnancement dans ces environnements doit prendre en compte les capacités des ressources disponibles ainsi que leur état actuel, ceci conduit à considérer la notion de contexte et sa gestion. La sensibilité au contexte peut être vue comme la capacité d'un système à observer son environnement et à adapter son comportement en fonction des changements observés (Baldauf *et al.* 2007). L'information de contexte peut être utilisée pour caractériser l'état actuel de la ressource dans cet environnement

dynamique, offrant des indications importantes sur la capacité qu'une ressource a pour exécuter (ou non) une tâche donnée. Utiliser ces informations pour l'ordonnancement de services ouvre de nouvelles perspectives pour un ordonnancement « intelligent » et opportuniste. Par exemple, dans l'exemple du centre commercial, on pourrait imaginer l'exécution d'un service appartenant à une entreprise A sur une ressource appartenant à une entreprise B, lors que les ressources appartenant à A sont surchargées et que celles de B seraient inoccupées (ou peu sollicitées).

Un ordonnancement opportuniste essaie de profiter des ressources disponibles, en fonction de leurs capacités et de leurs restrictions d'usage, d'une façon indéterministe. Au lieu de considérer l'optimisation d'un pool de ressources dédiées, comme dans l'ordonnancement traditionnel utilisé dans le calcul haute performance, l'objectif ici, est de permettre l'exécution de services sur des ressources de proximité lorsque cela est possible et sans les surcharger, alors que leur rôle initial n'est pas d'exécuter ce genre de tâche.

Néanmoins, pour qu'un tel ordonnancement sensible au contexte soit possible dans les SIP, une architecture conceptuelle supportant la capture, la gestion, et l'exploitation du contexte pour l'ordonnancement de services doit être proposée. Les caractéristiques de l'information de contexte comme l'incertitude et sa nature dynamique (Kirsch-Pinheiro et Souveyet, 2018) doivent être prises en compte dans l'algorithme de décision. De plus, le SIP doit pouvoir évoluer en fonction de l'organisation et de son environnement physique. L'extensibilité est un facteur clé pour les futures applications et infrastructures, car il faut pouvoir : (i) disposer d'une gestion modulaire permettant de modifier chaque élément de l'architecture ; (ii) changer les critères de décision de l'algorithme d'ordonnancement, si nécessaire, sans qu'il soit nécessaire de modifier l'algorithme d'ordonnancement. Donc, le gestionnaire de ressources doit pouvoir être aussi extensible.

Par ailleurs, les problèmes causés par une gestion centralisée des ressources sensible au contexte peuvent être accentués. En effet, plus la latence pour obtenir et traiter les informations de contexte et prendre une décision est importante, plus grand est le risque que ces informations deviennent obsolètes (puisque que le contexte évolue) et perdent leur utilité.

Dans cet article, nous proposons une architecture conceptuelle distribuée pour la gestion de ressources conçue plus particulièrement pour supporter l'hétérogénéité et la dynamicité des environnements *fog*, ainsi que les besoins d'extensibilité des SIP. Nous discutons aussi des politiques qui s'expriment au niveau des ressources et des services pour la mise en œuvre d'un ordonnancement sensible au contexte pour les environnements pervasifs.

Cet article est organisé de la façon suivante : la section 2 est dédiée à l'état de l'art spécifique au *fog computing* ; la section 3 introduit notre architecture conceptuelle, alors que la section 4 présente les politiques qui peuvent être exprimées au niveau des ressources et des services, et l'algorithme d'ordonnancement utilisant ces politiques, avant de conclure par la section 5.

2. Etat de l'art

Plusieurs aspects affectent la gestion de ressources, nous commençons par l'organisation générale du gestionnaire. Généralement, les solutions adoptées par les plateformes *fog* s'appuient sur des serveurs centralisés, ce qui peut limiter l'évolutivité de ces solutions, car toutes les informations nécessaires à la prise de décision doivent être envoyées à une ressource centrale, ce qui peut surcharger le réseau si de nombreuses ressources doivent être gérées simultanément. De plus, la ressource utilisée pour gérer l'environnement peut devenir surchargée par l'excès de données à traiter, et cela peut entraîner une latence dans la prise de décision, ce qui peut augmenter la latence dans l'exécution des services. C'est aussi le cas de nombreuses solutions basées sur des topologies hybrides (Breitbach *et al.* 2019) (Edinger *et al.* 2017) (Garcia-Lopez *et al.* 2015), dans lesquelles un serveur est en charge d'un ensemble de ressources dans un certain périmètre.

En outre, les environnements *fog* peuvent être partagées entre plusieurs entités avec le but d'optimiser leur utilisation et leur efficacité énergétique. Les ressources peuvent héberger des applications appartenant à plusieurs utilisateurs (Hong et Varghese 2019), et pas nécessairement toutes ces applications seront exécutées par la plateforme *fog*. Le gestionnaire de ressources n'est pas nécessairement en charge de tous les services en exécution sur chaque ressource, ce qui rend essentiel l'observation de l'état de la ressource. Ainsi, le contexte d'exécution de ces ressources doit être considéré à des fins d'ordonnancement.

La notion de contexte peut être définie comme toute l'information pouvant être utilisée pour caractériser la situation d'une entité (une personne, un endroit, un objet, etc.) considérée comme pertinente à l'interaction entre un utilisateur et un système (Dey 2001). Différentes informations peuvent être considérées comme étant des informations de contexte, selon l'objectif du système : la mémoire disponible, la charge de la CPU, le stockage disponible, la connexion réseau, etc. Des travaux comme (Cassales *et al.* 2016) (Kumar *et al.* 2012) démontrent l'intérêt de considérer des informations de contexte dans l'ordonnancement de services pour des environnements hétérogènes. Différents auteurs, comme (Breitbach *et al.* 2019) (Edinger *et al.* 2017), ont traité cette question dans les environnements *fog*. Edinger *et al.* (2017) propose un ordonnanceur tolérant aux défaillances dans lequel un *broker* collecte des informations de contexte issues des nœuds *fog* et estime leur fiabilité afin de placer les services dans les nœuds les plus appropriés. Breitbach *et al.* (2019) se concentrent sur le placement des données, en essayant d'estimer la quantité de répliques nécessaires pour obtenir une certaine donnée ainsi que le degré de stabilité de chaque nœud. Là aussi, nous observons la présence d'un *broker* centralisé.

Nous pensons que, pour fournir une gestion de ressources opportuniste pour les environnements *fog*, il est nécessaire de fournir un ordonnancement de services sensible au contexte et de considérer une organisation décentralisée où les ressources peuvent collaborer sans un élément central de manière à exécuter des services sur cet environnement sans nécessairement avoir de connaissances préalables sur ces services. Nous proposons ainsi une architecture capable de répondre à ces exigences. Cette architecture identifie un ensemble d'éléments permettant la mise en place de solutions évolutives et extensibles. Une telle architecture évolutive est une étape

obligatoire pour l'introduction de plateformes de *fog computing* dans les SIP, étant donné que ces systèmes doivent évoluer en 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 essentiellement les Ressources et les Services proposés par un SIP à ces utilisateurs. Les *Ressources* (c'est-à-dire les nœuds) sont définies par leur habilité à exécuter des services. Les *Services* sont des modules d'application autonomes qui sont proposés au gestionnaire de ressources à travers des mécanismes de sélection de services (hors du cadre de ce travail) et que doivent être ordonnancés et exécutés.

Dans l'architecture proposée, la gestion de ressources est répartie et le gestionnaire de ressources est idéalement présent sur chaque ressource. L'organisation physique des ressources est organisée dans un réseau pair à pair (P2P). L'architecture du gestionnaire est composée d'un ensemble de modules dont chacun est chargé d'accomplir une fonction spécifique, et chaque ressource exécute une instance de chaque module. Figure 1 illustre l'ensemble des modules composant l'architecture.

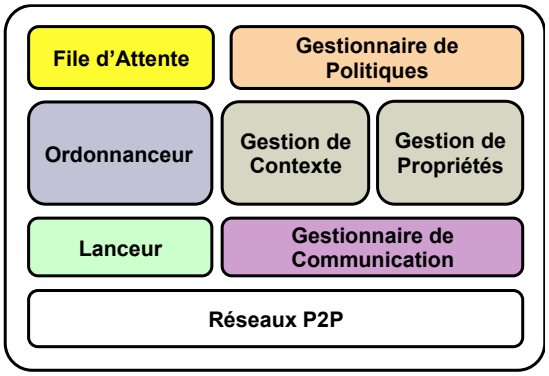

Figure 1. Architecture conceptuelle du gestionnaire de ressources.

Le composant ***« Gestionnaire de Communication »*** assure la communication entre les gestionnaires de ressources installés dans les nœuds voisins. Il maintient la liste de ressources voisines connues. Les communications possibles entre les ressources sont : (i) envoyer/recevoir une tâche à exécuter après une décision prise par un ordonnanceur ; et (ii) envoyer/recevoir des informations de contexte (ou des propriétés) depuis la ressource pour aider l'ordonnanceur dans sa décision.

À son tour, la ***« File d'Attente »*** est le composant responsable pour recevoir les services qui seront proposés à l'ordonnanceur local. Nous faisons l'hypothèse que certaines connaissances préalables sur les services sont disponibles avant leur soumission, comme son niveau de priorité et son propriétaire (l'organisation qui génère la tâche). Le composant File d'Attente notifie aussi l'ordonnanceur lorsque des nouveaux services arrivent.

Le service d'ordonnancement requiert, pour sa prise de décision, des informations sur la ressource et sur les services à exécuter. Nous considérons ces informations comme faisant partie de deux catégories distinctes : un ensemble de *propriétés*, et un ensemble *d'éléments de contexte,* nommé *contexte*. Le contexte et les propriétés sont définis pour chaque ressource ainsi que pour chaque tâche à exécuter. Les propriétés sont considérées comme des informations stables, ne changeant pas (ou peu) en fonction du temps, alors que les informations de contexte sont des informations plutôt dynamiques, que l'on observe en temps d'exécution.

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 gérer le contexte de la ressource locale, accéder au contexte du service considéré 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 second est responsable pour accéder et manipuler les propriétés d'une ressource locale ou d'une tâche considérée par l'ordonnanceur (local ou issu d'une ressource voisine).

Par ailleurs, dans notre architecture, les *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. Du point de vue d'une ressource, les politiques sont utilisées pour exprimer des conditions qui permettent de contrôler (et de limiter) son utilisation de manière opportuniste (c'est-à-dire, on peut réduire « l'espace d'opportunités » d'utilisation d'une ressource). Du point de vue d'un service, les politiques expriment les conditions dans lesquelles le service peut être exécuté ou les conditions dans lesquelles il serait préférable de ne pas l'exécuter. Le composant de notre architecture en charge de l'accès et de la manipulation des politiques définies pour la ressource locale ainsi que les politiques liées aux services qui seront considérées par l'ordonnanceur est le ***« Gestionnaire des Politiques »***.

L'élément central de l'architecture est l'***« Ordonnanceur »***. Il est chargé de décider si et quand un service pourra être exécuté dans la ressource locale. Pour prendre sa décision, il utilise le gestionnaires des politiques, de contexte, et de propriétés afin d'évaluer 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 décider de : (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* »** permet de mettre un service en exécution sur la ressource locale, après une décision favorable de l'ordonnanceur. Le défi ici reste l'hétérogénéité des technologies qui peuvent être utilisées pour implémenter les services. L'utilisation de micro-services représente une approche intéressante pour surmonter ce défi. Les services peuvent avoir des dépendances locales pour leur bonne exécution (p. ex : besoin d'une bibliothèque spécifique). Pour l'ordonnanceur ces dépendances sont considérées comme étant des propriétés liées aux ressources et aux services et qui peuvent être exprimées à travers les politiques. Ces concepts (propriétés, politiques) sont définis dans la section suivante. Le lanceur n'a à résoudre les dépendances qu'en cas d'exécution du service.

Comme nous l'avons vu dans cette section, l'architecture proposée a comme plus-value l'usage des notions de contexte, de propriétés, et des politiques. La prochaine section détaille ces trois notions et comment elles peuvent être utilisées pour gérer des ressources dans notre architecture.

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

L'utilisation opportuniste de ressources requiert la définition de *politiques* à la fois pour l'usage les ressources et pour l'exécution des services. Ces politiques seront considérées lors du processus d'ordonnancement. Dans cette section, nous présentons et discutons un méta-modèle qui englobe toutes les entités gérées par le gestionnaire de ressources, ainsi que la formalisation de politiques de gestion.

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

Le méta-modèle présenté dans la Figure 2 décrit un ensemble d'entités pertinentes pour la gestion de ressources. Dans celui-ci, nous considérons les entités gérées par le gestionnaire de ressources, à savoir les entités de type *Ressource* et *Service*. Les ressources sont des entités capables d'exécuter des services. À leur tour, les services représentent des modules d'application autonomes définis au niveau de l'entreprise (niveau métier). Ces services sont caractérisés par : (i) l'intention utilisateur satisfaite par leur exécution (c'est-à-dire, un objectif) (Najar *et al.* 2011) ; et (ii) les types de données qu'ils consomment (*input data*) et qu'ils produisent (*output data*) lors de leur exécution.

Ces deux entités sont caractérisées par un ensemble de *propriétés-types*, que nous appelons *propriétés*. Une propriété-type (désormais *propriété*) est définie par un *nom*, un *identificateur*, et une *spécification sémantique*. Pour une propriété avec une seule valeur, une *spécification de valeur* doit être ajoutée, permettant d'indiquer la nature des valeurs qu'elle supporte, alors qu'une propriété de valeur composite doit être désignée par la composition de plusieurs sous-propriétés. Les propriétés sont utilisées pour exprimer une information qui ne change pas (ou peu) en fonction du temps. Dans notre architecture, le gestionnaire de propriétés aide l'ordonnanceur à accéder aux propriétés de la ressource locale et des services analysés par l'ordonnanceur.

Par exemple, un service peut avoir les propriétés suivantes : *processus métier de référence*, *unité organisationnelle* (qui offre le service), *niveau de sécurité requis*, etc. Tous ces exemples peuvent être considérés comme étant des propriétés avec une seule valeur. Néanmoins, la propriété *mémoire installée* d'une ressource 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 que : *unité organisationnelle* (qui dispose primairement de la ressource), et *administrateur*.

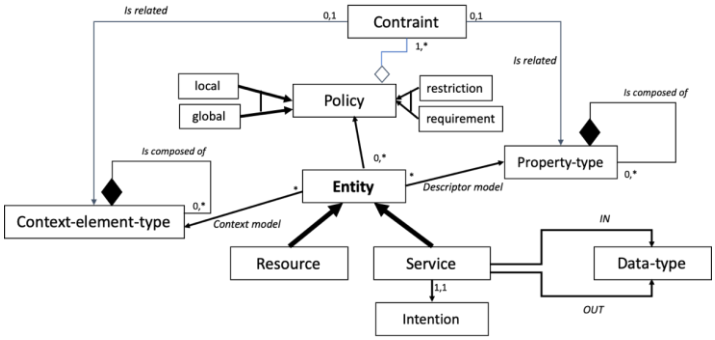

Figure 2. Méta-modèle d'entités utilisées par le gestionnaire de ressources

Les propriétés peuvent être propres à chaque ressource ou service, et leur usage offre plusieurs avantages : (i) permettre la caractérisation et la configuration de ces ressources et services ; (ii) ajouter des informations qui viennent aussi bien du niveau technique que du niveau métier ; et (iii) permettre d'exploiter les informations issues de ces deux niveaux pour la gestion de ressources.

Dans notre méta-modèle, le *modèle de contexte* concentre les informations de contexte relatives aux ressources et aux services. Ces informations évoluent fortement en fonction du temps et requièrent un mécanisme approprié pour les observer. Un modèle de contexte est exprimé comme un ensemble d'*élément-de-contexte-type* (désormais *élément de contexte*). Similairement aux propriétés, chaque élément de contexte doit être défini par un *nom*, un *identificateur*, et une *identification sémantique* correspondant 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 sur une ressource : *nombre de cœurs de processeur disponibles*, *occupation de mémoire*, *nombre de services invités en exécution* (des services appartenant à d'autres organisations, pas celle propriétaire de la ressource), sa *localisation* (pour les ressources mobiles), etc. Toutes ces informations sont considérées comme contexte parce qu'elles peuvent changer en fonction du temps, mais aussi parce que ces informations sont liées à un capteur spécifique afin d'être observées.

Le gestionnaire de contexte gère, pour les éléments de contexte observés, un descripteur de qualité aidant à déterminer la qualité de l'observation. Le gestionnaire de contexte permet ainsi à l'ordonnanceur d'accéder au modèle de contexte associé à la ressource locale et également à celui associé au service considéré. Les modèles de contexte permettent l'usage des informations de contexte sur les ressources et services afin d'exprimer des politiques qui seront utilisées par l'ordonnanceur pour la prise de décision. Ces politiques sont détaillées dans la section suivante.

4.2. Définition de Politiques

Une politique est définie comme une conjonction de contraintes simples, chacune d'entre elles basées sur un élément de contexte ou sur une propriété. Le but d'une politique est d'exprimer l'ensemble d'états souhaités dans lesquels un service peut être exécuté sur une ressource. Ces états sont appelés *exigences*, alors que les politiques exprimant des états interdits sont appelées *restrictions*. Chaque politique a un attribut de pondération (un poids) qui exprime 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 pourront être utilisées pour cela.

Pour illustrer l'utilisation des politiques dans la gestion de ressources, considérons le scénario présenté dans l'introduction où deux entreprises situées dans un centre commercial partagent une infrastructure informatique pour fournir des services à la demande. Les ressources partagées par l'entreprise A sont : un serveur (RA1#), trois notebooks (RA2# à RA5#) et deux écrans tactiles (RA6#, RA7#) positionnés à des emplacements stratégiques du centre commercial. L'entreprise B, quant à elle, partage cinq écrans tactiles (RB#1 à RB#5).

Lors de la modélisation de ce scénario, l'entité *Ressources* a été spécialisée dans les entités *Serveur*, *Notebook*, *Smartphone* et *Écran-Tactile*. L'ensemble de propriétés de ce scénario a toutes les propriétés mentionnées dans les sections précédentes. 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 donner la priorité à ses propres services (par rapport à ceux issus 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**) ;
- **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 inférieur à 40% (**restriction**).

Ces politiques sont résumées comme suit (le premier paramètre est le poids de la politique, tandis que le deuxième correspond à l'entité qui possède la politique, et le troisième correspond aux contraintes associées à la 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 de favoriser l'exécution de ses services de manière satisfaisante sur les différentes ressources, l'Enterprise B a défini certaines politiques liées à ce service (politiques de service, PS) :

- **PS1 :** Le ServiceB1 nécessite 5 personnes devant la ressource afin d'être exécuté (**exigence**) ;
- **PS2 :** Le ServiceB1 ne doit pas être exécuté sur des ressources avec moins de 512 Mo de mémoire disponible (**restriction**) ;
- **PS3 :** Le ServiceB1 ne doit pas être exécuté sur 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 correspond à l'entité à qui appartient cette politique, et le troisième correspond à ses contraintes) :

- 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 une même organisation. Toujours dans l'exemple du centre commercial, les politiques définies pour le ServiceB1 sont des politiques locales à ce service, alors que les politiques définies pour les ressources de l'Enterprise A sont des politiques globales.

Dans une ressource, l'ordonnanceur local utilise les politiques afin de prendre 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 et l'ensemble des politiques (locales et globales) liées au service. Afin d'aider l'ordonnanceur à évaluer les politiques, le gestionnaire de contexte et le gestionnaire de propriétés extraient respectivement les informations de contexte et les propriétés liées à la ressource *n* et au service *s* et les restreindront aux éléments utilisés dans les politiques.

Une API d'opérateurs est proposée afin d'opérationnaliser l'évaluation des politiques, en gérant 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 opérateurs 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, tels que : l'égalité (EQUAL), la non-égalité (DIF), inférieure à (LT), supérieure à (GT), etc. Pour illustrer l'utilisation générale des opérateurs, considérons l'opération indiquée dans la politique PR2 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 degrés, et accepte comme observations valables pour cet élément de contexte toutes les observations qui ne varient pas plus que 5 degrés (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 services 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.

4.3. Un Algorithme d'Ordonnancement pour le composant Ordonnanceur

Au cœur du composant Ordonnanceur du gestionnaire de ressources il y a un algorithme de prise de décision (Figure 3). Cet algorithme décide de l'exécution (ou non) d'un service (appelé *s* dans l'algorithme) sur la ressource (appelée *local_node* dans l'algorithme). Il se déclenche lorsque le composant file d'attente informe l'ordonnanceur de l'arrivée d'une demande de service. Pour prendre sa décision, l'algorithme cherche à vérifier l'équilibre entre les exigences (issues de la ressource et du service lui-même) qui peuvent être satisfaites et les restrictions qui s'appliquent, afin d'identifier si l'exécution du service sur la ressource est possible et opportune.

L'algorithme de la Figure 3 prend sa décision sur la base du 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. Cette somme représente la proportion des exigences qui ont été satisfaites par rapport aux restrictions satisfaites. Un bonus est appliqué sur ce résultat pour tenir compte de la priorité attribuée au service *s* (ligne rouge de la Figure 3). Si le résultat est supérieur ou égal au *seuil d'exécution local* qui représente la proportion des exigences à respecter par rapport aux restrictions satisfaites ; alors le lanceur est appelé pour exécuter le service localement (troisième ligne bleue de la Figure 3). 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 la Figure 3). Les seuils sont définis localement par le responsable de chaque ressource. On a deux seuils pour augmenter les possibilités de choix, mais rien n'empêche d'utiliser la même valeur pour le deux.

```
/*
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
```

Figure 3. Algorithme d'ordonnancement sensible au contexte

Lorsque le service doit être transféré à un voisin, il faut obtenir la liste des voisins à partir du gestionnaire de communication, puis sélectionner le voisin qui sera sollicité. Cette liste de voisins est tenue par le gestionnaire de communication, se basant sur les informations issues du réseau pair-à-pair lui-même. 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 des voisins concernés. Cela implique que les gestionnaires de contexte et de propriétés de chaque voisin puissent fournir les informations requises. Le gestionnaire de communication

prend en charge le transfert de ces informations entre les deux ressources. La cinquième ligne bleue de la Figure 3 exprime la manière dont le service est transféré au voisin sélectionné. À la fin, le service est ajouté dans la file d'attente du voisin sélectionné.

Comme nous l'avons vu dans la section précédente, une politique est une conjonction de contraintes, et la Figure 4 montre l'algorithme d'évaluation d'une politique (*evaluationPolicy*) appelé par l'algorithme de prise de décision. Pour évaluer une politique, l'algorithme nécessite les informations issues de ces politiques (exigences et restrictions). A l'aide du gestionnaire de politiques, le composant Ordonnanceur rassemble dans une même collection les politiques locales et globales, liées soit à la ressource *local_node*, soit au service *s*, et les transmet en tant que paramètre à l'algorithme de prise de décision. Dans cet algorithme, 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*). Cette collection est générée par le gestionnaire de contexte et est limitée aux éléments de contexte requis par les politiques de *PolReq* et *PolRest*. Un traitement similaire est effectué grâce au 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 sont utilisés lors de l'évaluation d'une politique pour calculer une pénalité (voir la deuxième ligne rouge de la Figure 4). Cette pénalité est appliquée pour augmenter l'impact des politiques avec plus d'informations disponibles au détriment des politiques où les informations sont moins disponibles lors de la prise de décision par l'ordonnanceur.

Cet algorithme renvoie un *taux de satisfaction* de la politique (une valeur réelle entre 0 et 1). L'évaluation de chaque contrainte d'une politique nécessite le contexte d'exécution et les propriétés ainsi que les éléments de contexte manquants. La fonction d'évaluation de contraintes invoque l'implémentation correspondante de l'opérateur selon le type des opérandes utilisés 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 de la Figure 4).

Pour illustrer le fonctionnement de l'algoritme de prise de décision par l'ordonnanceur, supposons qu'à un certain moment 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) appelle l'algorithme de prise de décision pour prendre sa décision.

```
/* 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
```

Figure 4. Algorithme d'évaluation de politiques

Lors de l'étape (iii), l'algorithme de prise de décision appelle l'algorithme d'évaluation d'une politique décrit par la Figure 4 pour évaluer la conformité entre les politiques et les informations (contexte/propriétés) observées. Supposons qu'à ce moment 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 la ressource (*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. Supposons donc que la somme de décision soit inférieure au seuil d'exécution local, mais qu'elle soit encore suffisamment importante pour éviter d'être renvoyée dans la file d'attente local. Dans ce cas, 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 l'information de contexte *Presence* est manquante et ne permet pas de satisfaire les politiques du ServiceB1, et l'algorithme cherche donc un autre 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 de transférer le ServiceB1 vers la file d'attente de RA6#.

Notre exemple illustre comment l'algorithme d'ordonnancement que nous proposons permet une utilisation opportuniste des ressources. L'idée originale de cet algorithme est d'exploiter des informations issues d'un modèle de contexte ou/et d'un ensemble de propriétes sur les ressources et les services afin d'utiliser une ressource lorsqu'il est possible. Enfin, sa mise en œuvre est extensible car le modèle de contexte, l'ensemble de propriétes, et les politiques peuvent être modifiés.

5. Conclusions

L'un des principaux défis liés au *fog computing* est de fournir des mécanismes d'ordonnancement de services 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 l'exécution des services plus tôt ou d'optimiser l'utilisation d'un pool de ressources, mais de faire de notre mieux pour exécuter les services 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 du niveau technique doivent être exprimées et le contexte d'exécution des ressources et des services doit être géré de manière appropriée. Une des limitations de notre algorithme d'ordonnancement est qu'il est incapable de résoudre avec précision les incohérences entre les politiques. Il se limite à utiliser les poids des politiques pour favoriser une politique en relation à une autre. Il s'agit d'une conséquence du caractère opportuniste de la proposition : l'algorithme essaie de trouver un équilibre entre les exigences et les restrictions, sans vouloir résoudre les possibles incohérences entre elles, puisque ce qui est évalué est le degré de satisfaction global d'un ensemble de politiques. Néanmoins, l'existence d'un module de gestion des politiques dans notre architecture rend possible la mise en œuvre ultérieure d'un mécanisme de gestion d'incohérences entre politiques avant qu'elles ne soient évaluées par l'ordonnanceur.

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

Bibliographie

Baldauf, M., Dustdar, S., Rosenberg, F. (2007): A survey on context-aware systems. International Journal of Ad Hoc and Ubiquitous Computing, vol. 2, n° 4, p. 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, p. 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), p. 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 vol. 7, n° 3, p. 333-345.

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

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), p. 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., vol. 45, n° 5, p. 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, vol. 21, n° 2, p. 44-53.

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

Kirsch-Pinheiro, M., Souveyet, C. (2018): Supporting context on software applications: 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, vol. 2, n° 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. (2012): CASH: context aware scheduler for Hadoop. In: Proceedings of the International Conference on Advances in Computing, Communications and Informatics, p. 52-61. ACM.

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, p. 470-475.

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, vol. 10, n° 2, p. 1578-1586.