Modèles de conception pour des applications collaboratives dans le cloud

Nadir Guetmi

THESE

Pour l'obtention du Grade de

DOCTEUR DE L'ÉCOLE NATIONALE SUPÉRIEURE DE MÉCANIQUE ET D'AÉROTECHNIQUE

(Diplôme National — Arrêté du 25 Mai 2016)

Ecole Doctorale : Science et Ingénierie pour l'Information, Mathématiques
Secteur de Recherche : INFORMATIQUE ET APPLICATIONS

Présentée par :

Nadir GUETMI

****

Modèles de Conception pour des Applications Collaboratives Mobiles dans le Cloud

****

Directeurs de Thèse : **Ladjel BELLATRECHE et Abdessamad IMINE**

Soutenue le 12/12/2016
devant la Commission d'Examen

JURY

Rapporteurs :	Sophie CHABRIDON	MCF (HDR), Télécom SudParis
	Daniela GRIGORI	Professeur, Université Paris Dauphine
Examinateurs :	Mirian HALFELD FERRARI	Professeur, Université Orléans
	Yacine GHAMRI-DOUDANE	Professeur, Laboratoire L3i, La Rochelle
	Ladjel BELLATRECHE	Professeur, ISAE-ENSMA, Poitiers
	Abdessamad IMINE	MCF (HDR), Université de Lorraine & LORIA, Nancy

Remerciements

Mes sincères remerciements à :

Ladjel BELLATRECHE, pour sa générosité, pour la confiance qu'il m'a témoigné, pour ses précieux conseils et orientations et pour ces idées innovantes et créatives. Je mesure cette chance d'avoir un directeur de thèse aussi compétent, vif et toujours en bonne humeur.

Abdessamade IMINE, pour son encadrement minutieux, ses orientations significatives et ses qualités scientifiques et pédagogiques. Je tiens à le remercier de m'avoir appris beaucoup de choses.

Un grand merci à Sophie CHABRIDON et Daniela GRIGORI de m'avoir fait l'honneur d'être rapporteurs de ma thèse, ainsi pour Alves FERRARI et Yacine GHAMRI-DOUDANE d'avoir accepté d'être examinateurs. Je suis très honoré de l'intérêt qu'ils ont porté à mes travaux.

Mes remerciements sont destinés également :

Au directeur du laboratoire, Emmanuel GROLLEAU pour sa sympathie ainsi que son bien traité.

À tout le personnel du LIAS.

À mes amis et collègues.

Enfin, un spécial remerciement à mes parents qu'ils ont tout sacrifier afin que je puisse arriver à ce stade et à mes filles Maram et Nadine, mes sources de force et de bonheur.

À mon Algérie bien-aimée.

Mes parents,

Mes filles Maram et Nadine.

Table des matières

Chapitre 1

Introduction générale

Sommaire

1 Contexte

Des statistiques récentes montrent que le monde évolue vers l'utilisation de dispositifs mobiles de plus en plus connectés. En effet, à la fin de l'année 2014, le nombre d'abonnements mobiles à large bande avait atteint 2,3 milliards à l'échelle mondiale [96]. Cet engouement s'explique par la manipulation d'applications puissantes qui tirent parti de la disponibilité croissante des réseaux de communication et la performance des dispositifs mobiles pour l'échange des données. Ainsi, les flux de données en temps réel, ainsi que les services web proposés via internet (tels que le commerce mobile, les réseaux sociaux et la collaboration ad-hoc) sont parfaitement intégrés dans les applications mobiles.

Cependant, les dispositifs mobiles demeurent toujours pauvres en ressources, moins sécurisés, instables en terme de connectivité et contraints par la courte durée de vie des batteries. Cette limitation de ressources est donc considérée comme un enjeu majeur pour de nombreuses applications en dépit d'une évolution continuelle des technologies mobiles tendant progressivement à améliorer les configurations matérielles et logicielles des dispositifs mobiles [97]. À titre exemple, l'édition collaborative d'un document partagé en temps réel via un réseau mobile ad-hoc et pair-à-pair est souvent très coûteuse, car elle nécessite une énorme consommation d'énergie pour (i) gérer l'évolutivité du groupe de collaborateurs (c-à-d, la gestion des événements pour créer, joindre et quitter un groupe) et, surtout pour, (ii) synchroniser plusieurs copies du document partagé afin de préserver la propriété de cohérence. Les calculs intensifs peuvent même conduire à une congestion ou une panne matérielle et/ou logicielle. De plus, il est impossible d'assurer une collaboration continue en raison des fréquentes déconnexions.

Pour faire face à la limitation des ressources mobiles, une solution simple consiste à migrer le maximum des traitements mobiles vers le cloud. Ainsi, profiter des avantages de la virtualisation du cloud permet d'étendre de telles ressources en substituant chaque dispositif mobile par un clone (ou machine virtuelle) qui s'exécutera sur le cloud et prendra en charge les tâches coûteuses du mobile. Dans notre contexte, le cloud computing permet aux utilisateurs de créer des réseaux virtuels pair-à-pair où un dispositif mobile peut rester connecté en permanence (par l'intermédiaire de son clone) à d'autres mobiles afin de réaliser des tâches communes. L'adoption de la technologie de virtualisation pour implémenter de tels réseaux et machines virtuelles prouve son intérêt dans l'isolement des traitements et communications mobiles de leur environnement cloud (c-à-d, les serveurs hébergeurs). Cet isolement contribue à l'indépendance de toute plateforme construite par rapport aux technologies matérielles et logicielles adoptées par n'importe quel environnement ou fournisseur cloud. Par exemple, l'utilisation des logiciels de virtualisation existants (tel que VirtualBox [8]) permet (i) d'exécuter une machine virtuelle Android [12] sur des serveurs opérant avec des systèmes d'exploitation hétérogènes et (ii) configurer ses modes d'accès à distance. En utilisant une bonne configuration, les traitements effectués ainsi que les données stockées sur cette machine virtuelle seront inaperçus aux serveurs hébergeurs (en supposant que leurs administrateurs ne sont pas curieux) et aux utilisateurs n'ayant pas droit d'accès.

2 Motivation

Dans le cadre de cette thèse, nous présentons un travail de recherche dans le domaine du *Mobile Cloud Computing* (MCC) qui consiste à combiner les domaines du mobile et du cloud avec comme objectif la conception et le développement des applications collaboratives et mobiles dans le cloud. Cet objectif vise à fournir une plateforme virtuelle de collaboration mobile, permettant un accès rapide et simultané pour la manipulation des ressources partagées (telles que, les documents, les images, les vidéos, etc.) par des utilisateurs mobiles et dispersés. Afin d'assurer une disponibilité maximale ou permanente des données, chaque utilisateur dispose d'une copie locale des ressources partagées. En général, la collaboration s'effectue comme suit : les mises-à-jour de chaque utilisateur sont exécutées localement d'une manière non bloquante, puis elles sont diffusées vers d'autres utilisateurs pour les exécuter sur d'autres copies. Dans ce qui suit, nous présentons les problèmes que nous avons traités dans cette thèse.

Problème 1 : Gestion d'un déploiement efficace et résilient sur le cloud. Le déploiement mobile consiste à préparer une plateforme virtuelle composée de clones des dispositifs mobiles (structurés en groupes) afin de leur déléguer le maximum des tâches de collaboration et de communication. La gestion d'une virtualisation efficace assurant la continuité de la collaboration pour des réseaux pair-à-pair est une tâche très difficile. En effet, l'aspect dynamique des groupes où les utilisateurs peuvent joindre, quitter ou changer leurs groupes peut mener à un partitionnement intempestif des réseaux virtuels dans le cloud. Le déploiement doit donc être capable de rendre (en temps réel) visible et opérationnelle toute nouvelle modification dans l'organisation des utilisateurs ainsi que leurs groupes sur la plateforme virtuelle. En outre, chaque pair (le mobile et/ou son clone) dans les réseaux virtuels est vulnérable aux pannes matérielles et/ou logicielles. Aussi, il est nécessaire de concevoir un protocole de déploiement efficace et résilient, qui doit gérer tout un cycle de vie pour les clones de manière à assurer la pérennité de la collaboration ainsi que la disponibilité et la cohérence des ressources partagées. Ce cycle doit inclure, entre autres, l'installation, la configuration, la gestion d'appartenance aux réseaux virtuels et la restauration après panne.

Cependant, les systèmes de cloud actuels fournissent une gestion incomplète et approximative d'infrastructures ayant comme seul service offert le provisionnement de ressources. D'autres approches [36, 33, 73] proposent des mécanismes de déploiement basés sur la virtualisation mais sans préciser comment gérer le cycle de vie des machines virtuelles distantes. Par ailleurs, les solutions actuelles de tolérance aux pannes sont basées sur une réplication instantanée des ressources dans le cloud. Avec un système collaboratif à grande échelle, cette solution est énormément coûteuse. Dans notre contexte, toute méthode de reprise après panne doit être fine, optimale et non contraignante pour le cloud.

Problème 2 : Gestion de la cohérence des données partagées. Mettre en ligne un service d'édition collaborative et le déployer pour des dispositifs mobiles sur une plateforme virtuelle dans le cloud n'est pas sans problèmes. Comme le facteur humain est prépondérant dans l'édi-

tion collaborative, ce service doit posséder les caractéristiques suivantes : (i) Une réactivité locale maximale : le système doit assurer une réactivité identique à celle offerte par les éditeurs mono-utilisateur [43, 107, 109] ; (ii) Un degré élevé de concurrence : les utilisateurs doivent être en mesure de modifier n'importe quelle partie de la ressource partagée d'une manière simultanée et sans aucune restriction [43, 107] ; (iii) Une cohérence à terme des données : les utilisateurs doivent être en mesure de voir une vue identique de toutes les copies distribuées [43, 107] ; (iv) Une coordination décentralisée : toutes les mises-à-jour simultanées doivent être synchronisées de façon décentralisée afin d'éviter un point de défaillance unique ; (v) Un passage à l'échelle sans contraintes : un groupe doit être dynamique dans le sens où les utilisateurs peuvent le joindre ou le quitter à tout moment ; (vi) Une reprise après panne rapide : les utilisateurs doivent être capables de récupérer toutes les ressources partagées lorsqu'un problème technique (par exemple une panne matérielle, un vol ou une perte du dispositif mobile) se produit, et continuer la collaboration d'une manière transparente. Il est clair que le recours au cloud se prête bien à la satisfaction de ces caractéristiques, ne serait-ce que pour alléger la consommation des ressources (capacité de calcul, espace de stockage et énergie des batteries) sur les mobiles.

Mais, qu'en est-il de la coordination décentralisée pour préserver la cohérence des données partagées dans le cloud ? D'autant plus que chaque utilisateur possède deux copies des données partagées : une copie stockée au niveau du dispositif mobile et une autre sur son clone. L'utilisateur doit donc être en mesure de modifier concurremment n'importe quelle partie de sa copie locale et à tout moment. Les mises-à-jour effectuées localement sur le mobile doivent être envoyées au clone correspondant dans le cloud et diffusées ensuite vers les autres clones. Les accès simultanés à ces copies vont inéluctablement produire des vues incohérentes au niveau des différents clones. De plus, un décalage d'exécution des opérations (de mise-à-jour) locales entre le clone et son dispositif mobile est inévitable ; ce décalage constitue un autre problème pour maintenir la cohérence des ressources partagées.

Dans la littérature, CloneDoc [72] a été proposé comme un système d'édition collaborative en temps-réel, s'exécutant entre les dispositifs mobiles et le cloud. Ce système est lui même inspiré d'un autre protocole de collaboration client/serveur, appelé Sporc [45]. Ce protocole est basé sur l'approche des transformées opérationnelles (OT) [43, 106] et un serveur central de synchronisation. Cependant, CloneDoc et son prédécesseur Sporc peuvent être sujets à des goulots d'étranglement, et par conséquent, ils sont plus vulnérables aux pannes [84]. Par exemple, si tous les sites collaborateurs en cours d'exécution dans le cloud continuent à envoyer des mises-à-jour fréquentes au serveur de synchronisation, les performances du serveur seront sérieusement dégradées au détriment de la réactivité. D'autre part, si le serveur de synchronisation échoue (en raison d'une défaillance matérielle ou d'un déni de service [104]), certains sites collaborateurs vont perdre leurs mises-à-jour. En outre, ce serveur peut être une source malveillante ou être lui même vulnérable à des attaques malveillantes [23]. De tels comportements peuvent affecter le processus de synchronisation et engendrer beaucoup plus de procédures de maintenance du côté du cloud.

Contrairement à la synchronisation centralisée, une coordination distribuée n'est pas basée sur un serveur central et chaque site collaborateur doit communiquer et se synchroniser avec tous les autres sites. Ainsi, cette synchronisation permet de meilleures performances via l'élimination des goulots d'étranglement, et un haut degré de tolérance aux pannes. En effet, dans le cas où un clone échoue, le reste du système peut continuer à fonctionner.

Problème 3 : Absence de patrons de conception pour les applications collaboratives en MCC. D'une manière générale, la réutilisabilité implique un processus de création des systèmes logiciels à partir de logiciels existants plutôt que de les construire à partir de zéro [75, 94, 112]. Elle présente un impact positif sur la qualité des logiciels, la minimisation des efforts et coûts ainsi que sur la productivité [94]. La réutilisation est possible grâce à des architectures génériques, spécifiquement conçues pour des domaines d'application particuliers (par exemple, les environnements du MCC). Une architecture générique (de haut niveau) composée d'un ensemble de patrons de conception peut être définie pour répondre aux différents besoins de développement de chaque domaine d'application. Les patrons de conception permettent de conduire le processus de réutilisation des architectures génériques et de gérer le cycle de vie des logiciels.

Malgré l'importance de la réutilisabilité pour la conception et le développement des applications collaboratives mobiles dans des environnements MCC, cette notion a été négligée par la totalité des travaux de recherche. Dans la littérature, seul le travail présenté dans [88] a fourni une architecture réutilisable destinée au développement des applications collaboratives dans des environnements mobiles. Mais cette architecture reste incomplète quant à un déploiement sur le cloud.

D'un point de vue architectural, la définition de modèles de conception est plus difficile pour le développement de nouvelles applications offrant des services de collaboration mobile dans le cloud, car la combinaison des environnements mobiles et cloud soulève de nombreuses questions de conception, comme l'hétérogénéité de ces environnements. En effet, le déploiement des tâches collaboratives mobiles vers le cloud implique une interaction de deux domaines différents, à savoir cloud et mobile, qui sont eux-mêmes basés sur des technologies différentes. Les constructeurs de mobiles proposent une variété de dispositifs qui sont équipés de systèmes d'exploitation différents (par exemple, Android [5] et Apple [6]). D'autre part, les plateformes de cloud sont aussi fondées sur des technologies de virtualisation hétérogènes (par exemple Xen [9] et VirtualBox [8]). Tout modèle proposé doit être générique dans le sens où il peut être facilement réutilisé afin de l'adapter à ces environnements hétérogènes.

3 Contributions

Afin de répondre aux différents problèmes soulevés, nos travaux nous ont permis d'apporter les contributions suivantes :

1. **Des patrons de cloud pour la gestion du partage des données mobiles.** Ces patrons

sont considérés comme des briques de conception spécifiques à la modélisation des tâches de collaboration mobiles dans le cloud. Nous proposons une architecture réutilisable et extensible pour l'implémentation des mécanismes de synchronisation décentralisée, permettant de préserver la cohérence des données partagées, tout en respectant les contraintes et exigences des environnements mobiles (par exemple, limitation de la durée de vie de la batterie et instabilité des réseaux mobiles) par la migration des tâches intensives vers le cloud. Les patrons de conception proposés sont répartis sur deux niveaux. Le premier niveau sert à décrire le middleware de déploiement qui agit (de manière autonome) sur le cloud afin de cloner des dispositifs mobiles, gérer des groupes d'utilisateurs et assurer le bon fonctionnement de cette plateforme virtuelle. Le deuxième niveau présente des mécanismes de collaboration pour la synchronisation décentralisée des données partagées en temps réel.

Nos patrons de conception encapsulent un ensemble de classes abstraites et d'interfaces pour l'implémentation des tâches requises au clonage des dispositifs mobiles, à la gestion des réseaux privés virtuels (c-à-d, VPN) pair-à-pair et au partage des données mobiles dans le cloud. Ils confèrent aux développeurs un cadre générique et abstrait pour développer des applications collaboratives mobiles dans le cloud. Une réutilisation de nos modèles implique donc une redéfinition concrète des différentes méthodes d'interfaces pour les adapter à des environnements de virtualisation spécifiques et réaliser des travaux de collaboration flexibles.

2. **MidBox, un middleware de déploiement.** Conformément à notre objectif global visant à alléger les dispositifs mobiles, MidBox est un middleware de déploiement conçu pour déléguer les tâches importantes (par exemple, synchronisation et trafic réseau) vers les clones des dispositifs mobiles dans un cloud privé basé sur l'hyperviseur de virtualisation VirtualBox [8]. Ce middleware est présenté via des patrons de clonage spécifiques et qui sont dérivés de la réutilisation de notre architecture générique. D'une manière générale, l'objectif de MidBox est de concevoir, développer et mettre en œuvre un middleware de déploiement, agissant sur l'hyperviseur de virtualisation VirtualBox pour la construction et le maintien en bon état d'une plateforme collaborative virtuelle composée principalement des clones pour les dispositifs mobiles et des réseaux privés virtuels.

3. **MobiRdf, un protocole d'édition collaborative des données liées mobiles.** En réutilisant les patrons de collaboration de l'architecture générique, nous concevons un protocole, appelé MobiRdf, qui prend en charge l'édition collaborative en mode pair-à-pair (sans la nécessité d'une coordination centrale) des graphes RDF partagés. En se basant sur un schéma de réplication optimiste, notre protocole offre un accès simultané à des copies RDF et utilise la propriété de commutativité pour préserver leur cohérence. En raison de la limitation de l'espace de stockage sur les dispositifs mobiles, nous utilisons une réplication partielle basée sur la sélection des graphes RDF utiles aux besoins des utilisateurs mobiles depuis le cloud. Cette sélection s'appuie sur les informations contextuelles des utilisateurs.

Les résultats expérimentaux de notre prototype démontrent que MobiRDF apporte un énorme gain en termes de consommation d'énergie, temps de réponse et trafic réseau pour les dispositifs mobiles.

Contributions vs. approches existantes. En comparaison avec des approches existantes en terme de gestion optimale des ressources mobiles (capacités de calcul, énergie, stockage et instabilité réseau), notre middleware assure un déploiement à-la-demande des données, des tâches et du réseautage mobiles, sans recourir aux mécanismes de surveillance des ressources et partitionnement des applications qui s'avèrent coûteuses en consommation pour les mobiles. La combinaison des mécanismes de clonage, réseaux privés virtuels, sauvegarde/restauration, gestion des fichiers, auto-tolérance aux pannes ainsi que l'utilisation des services web offrent un degré très élevé de réutilisabilité, passage à l'échelle, interchangeabilité et disponibilité.

Quant à la conception des applications collaboratives, elle est basée sur un protocole de synchronisation totalement décentralisée (sans aucun rôle assigné à un serveur central). Ce protocole assure un degré très élevé de déploiement des tâches de synchronisation et réseautage dans le cloud, et par conséquent, un important gain est obtenu en matière de consommation des ressources mobiles. Le bon fonctionnement de ces applications collaboratives est assuré par notre middleware de déploiement. En cas d'échec, il procède à une restauration transparente des applications sans affecter le travail collaboratif.

4 Plan de la thèse

Ce mémoire de thèse est principalement structuré en deux parties. La première partie est dédiée à la description du contexte de recherche et de l'état de l'art. Ainsi, le chapitre 2 présente un aperçu des différents concepts liés au cloud computing ainsi qu'aux applications collaboratives. Le chapitre 3 est consacré à la présentation des différents travaux liés à nos deux principaux axes de recherche, à savoir le déploiement des tâches mobiles dans le cloud et les applications collaboratives mobiles.

La deuxième partie de ce mémoire présente les contributions pour la conception et le développement des applications collaboratives mobiles dans le cloud. Dans le chapitre 4, nous décrivons notre architecture générique basée sur les patrons de conception qui correspondent à deux niveaux, à savoir le clonage et la collaboration. Les patrons correspondant à ces deux niveaux sont raffinés afin de permettre la conception et le développement des systèmes : (i) MidBox, un middleware de déploiement agissant sur l'hyperviseur de virtualisation VirtualBox [8], décrit au chapitre 5 et (ii) MobiRdf, un éditeur collaboratif des graphes RDF dans le cloud, présenté dans le chapitre 6. Nous concluons par un bref survol sur le travail effectué dans cette thèse et quelques perspectives de recherche. La figure 1.1 résume les liens entre les différentes contributions, ainsi que leur répartition sur les différents chapitres.

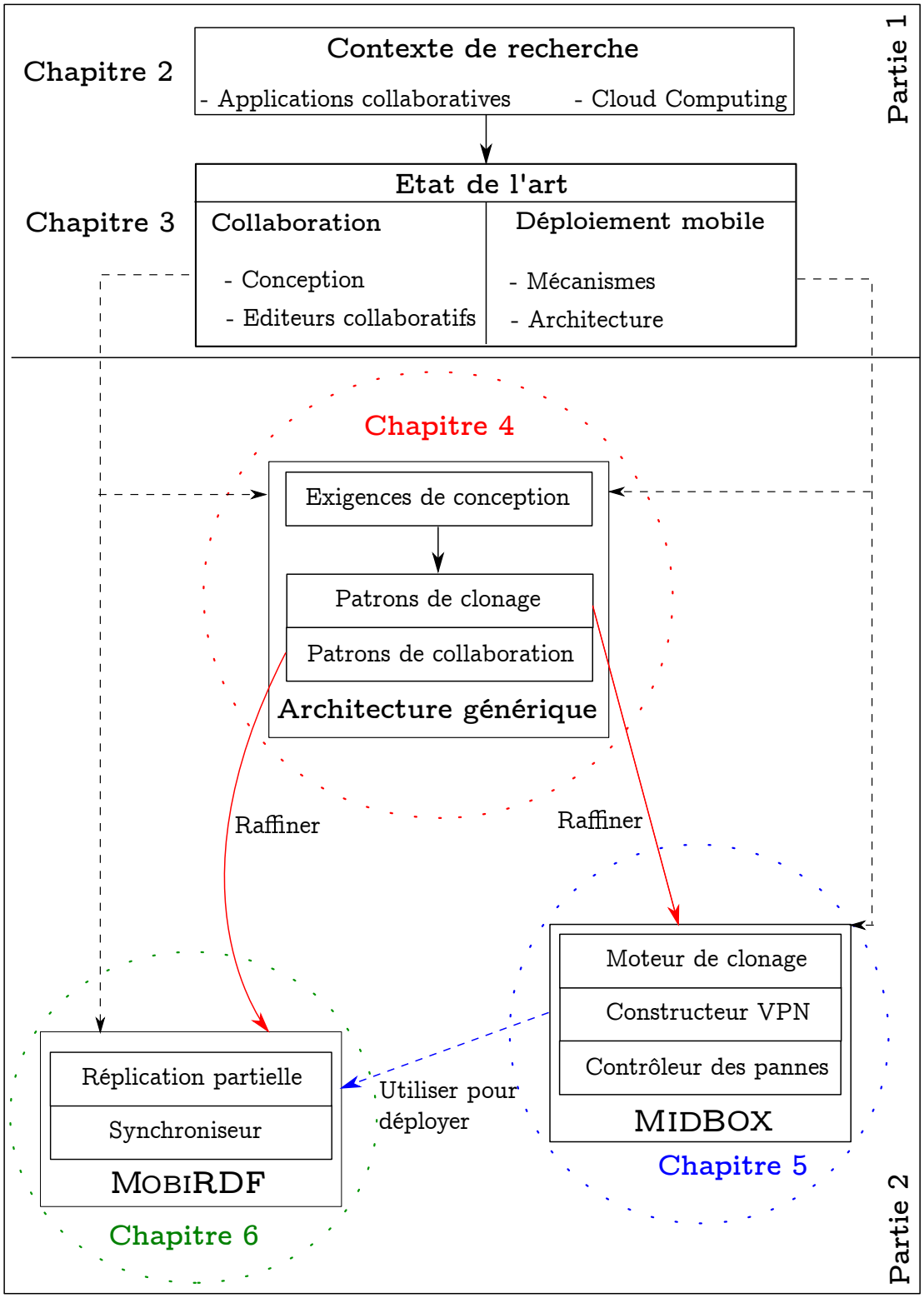


FIGURE 1.1 – Plan de la thèse.

Publications

Journal :

1. Mechaoui, M. D., **Guetmi, N**., Imine, A. (2016). MiCa: Lightweight and mobile collaboration across a collaborative editing service in the cloud. In Journal Peer-to-Peer Networking and Applications, 1-28, Springer.

Chapitre :

1. **Guetmi, N**., Imine, A. (2016). Designing Mobile Collaborative Applications for Cloud Environments. In A. Rosado da Cruz, S. Paiva (Eds.) Modern Software Engineering Methodologies for Mobile and Cloud Environments (pp. 34-60). Hershey, PA: Information Science Reference.

Conférences, Workshops et Magazines :

1.**Guetmi, N**., Mechaoui, M. D., Imine, A., Bellatreche, L. (2015, April). Mobile collaboration: a collaborative editing service in the cloud. In Proceedings of the 30th Annual ACM Symposium on Applied Computing (pp. 509-512). ACM.

2.**Guetmi, N**., Abdessamad, I. (2015). A Cloud-Based Reusable Design for Mobile Data Sharing. In Model and Data Engineering (pp. 62-73). Springer International Publishing.

3. Mechaoui, M. D., **Guetmi, N**., Imine, A. (2015). Mobile Co-Authoring of Linked Data in the Cloud. In New Trends in Databases and Information Systems (pp. 371-381). Springer International Publishing.

4. Mechaoui, M. D., **Guetmi, N**., Imine, A. (2015). Towards Real-Time Co-authoring of Linked-Data on the Web. In Computer Science and Its Applications (pp. 538-548). Springer International Publishing.

5. **Guetmi, N**., Mechaoui, M. D., Imine, A.(2015). Resilient Collaboration for Mobile Cloud Computing. ERCIM News 2015(102).

Première partie

Contexte et État de l’art

Chapitre 2

Contexte de recherche

Sommaire

1 Introduction

Dans la littérature, le domaine du MCC a été soutenu par une large gamme d'applications mobiles [46]. Ces applications se répartissent sur plusieurs domaines tels que le partage du GPS [114], la perception des données de capteurs [81], le traitement d'images et le traitement du langage naturel [31], l'enregistrement vidéo et la recherche multimédia [81] et le partage au sein des réseaux sociaux. La majorité de ces applications sont caractérisées par un aspect collaboratif à travers des calculs distribués.

Ce chapitre est dédié à la présentation des différents concepts liés à nos principaux axes de recherches, à savoir le MCC et les applications collaboratives et qui réapparaissent tout au long de cette thèse. Ainsi, un aperçu sur ces concepts sera réparti en deux parties. La première partie est consacrée à la définition du cloud ainsi que la description de ses modèles, services et technologies de base. Quant à la deuxième partie, elle vise à décrire les applications collaboratives.

2 Cloud computing

Plusieurs définitions du cloud computing peuvent être trouvées dans la littérature. L'institut national des normes et de la technologie NIST (en anglais, National Institute of Standards and Technology) définit le cloud computing comme suit [85] :

"*Le cloud computing est un modèle qui permet un accès réseau à-la-demande à un pool partagé de ressources informatiques configurables (Par exemple, réseaux, serveurs, stockage, applications et services) qui peuvent être rapidement provisionnées et publiées avec un effort de gestion minimal via un service d'interaction de fournisseur*".

En se basant sur plusieurs définitions de la littérature, les auteurs de [116] ont conclu une définition globale décrivant les caractéristiques minimales du cloud computing.

"*Les clouds sont un large pool de ressources virtuelles, facilement utilisables et accessibles (matériels, plateformes et/ou des services de développement). Ces ressources peuvent être dynamiquement reconfigurées afin d'être ajustées aux déploiements variables (passage à l'échelle). Ceci permet également une utilisation optimale des ressources. Ce pool de ressources est généralement exploité par un modèle payer-par-utilisation dans lequel des garanties sont offertes par le fournisseur de l'infrastructure par le biais de contrats SLA (Service Level Agreement) personnalisés*".

2.1 Taxonomie du cloud computing

Le concept de cloud computing a été considéré à partir de différents points de vue. D'une manière générale, les clouds peuvent être répartis selon les types d'accès (via des modèles standards) ou selon la prestation des services proposés.

2.1.1 Modèles

Il y a trois modèles connus du cloud :

Clouds publics. Les fournisseurs des clouds publics offrent leurs ressources telles que les applications et les moyens de stockage comme des services au grand public sur Internet [98]. Le principal avantage pour un client est son exonération des frais d'investissement en infrastructures [122]. Vu la prise en charge des coûts des matériels, applications et réseaux par les fournisseurs cloud, les clients peuvent créer et installer leurs environnements d'applications d'une manière rapide et moins coûteuse. Les clients ont la possibilité d'accéder aux ressources du cloud au moment et avec la manière qu'ils le souhaitent. Cependant, le plus gros problème confronté avec les services du cloud public est le manque d'un contrôle précis sur les données, le réseau et les paramètres de sécurité.

Clouds privés. Les clouds privés sont conçus pour une utilisation restreinte par une seule organisation [74]. Ils sont basés sur les mêmes principes des clouds publics où les ressources informatiques sont mises en commun derrière un pare-feu (les ressources ne sont pas disponibles aux utilisateurs externes à l'organisation) [118]. Les clouds privés peuvent être construits et gérés par l'organisation elle-même ou par des fournisseurs externes. Les grands fournisseurs de cloud comme Google et Amazon sont initialement fondés sur le principe de clouds privés. Ils ont profité des avancements technologiques en virtualisation et calculs distribués afin de devenir parmi les leaders des fournisseurs de cloud. D'une manière générale, les clouds privés possèdent les mêmes caractéristiques de fondement et de fonctionnement des clouds publics, mais leur champ opérationnel est limité à une échelle réduite (c-à-d, au niveau de l'entreprise).

En matière de contrôle, performance, fiabilité et sécurité, cette approche assure un degré élevé. Cependant, en raison de leurs coûts élevés, les clouds privés ne sont adoptés comme solution que par les grandes entreprises où le contrôle et la sécurité/confidentialité des données sont primordiales et nécessaires.

Clouds hybrides. Un cloud hybride est un environnement issu de la combinaison des modèles de clouds publics et privés. Ce modèle vise à remédier aux limitations de chaque approche. Son utilisation est généralement recommandée en cas de nécessité pour étendre les capacités de ressources épuisées du cloud privé.

2.1.2 Services

Les services du cloud sont généralement présentés ou étiquetés sous la forme "X AS A SERVICE", où X peut prendre des valeurs telles que : Infrastructure, Hardware, Software, Application, Database, voire même Datacenter.

La classification la plus adoptée des services cloud est donnée par le NIST [85] qui distingue trois principaux types : l'infrastructure comme un service, la plateforme comme un service et le logiciel comme un service.

Le diagramme de la figure 2.1 illustre une classification par paquets des services fournis par des fournisseurs de cloud computing. Le paquet le plus à gauche représente une infrastructure privée détenue par l'organisation/client. Il assure un niveau élevé de la confidentialité et de contrôle des données. Cependant, ce paquet est caractérisé par sa plus grande complexité. Sur le paquet le plus à droite, moins de flexibilité offerte pour le contrôle et la sécurité des données mais aussi moins de complexité à gérer.

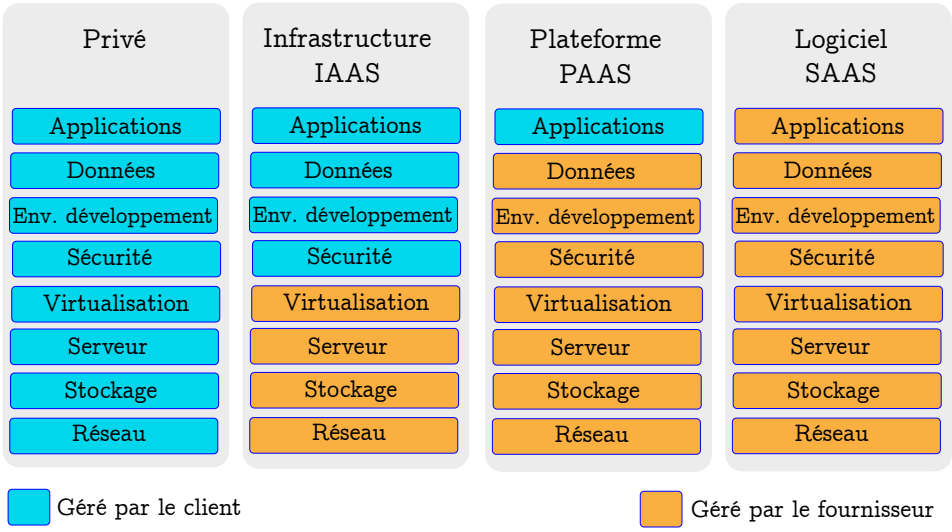


FIGURE 2.1 – Les services du cloud computing.

Infrastructure comme Service (IaaS). L'IaaS propose des ressources d'infrastructure à la demande. Cet approvisionnement permet de louer des ressources matérielles, tels que des serveurs, équipements réseau et espaces de stockage mais il ne fournit pas d'applications. L'infrastructure peut être étendue ou réduite d'une manière dynamique en fonction des besoins en ressources. Amazon Elastic Compute Cloud (*EC2*) est un exemple idéal pour les services cloud IaaS.

Plateforme comme Service (PaaS). La plateforme fournie par cette catégorie de services est considérée comme un environnement destiné pour le déploiement et l'exécution des applications. Elle est généralement proposée aux développeurs d'applications informatiques. Cette plateforme peut inclure des systèmes d'exploitation, des environnements de développement, des systèmes de gestion des bases de données et des outils de configuration et accès réseau.

D'une manière similaire aux services IaaS, les ressources logicielles louées via le PaaS sont quantifiables et mesurables. Cependant, les utilisateurs bénéficiant de la plateforme comme service sont épargnés des complexités d'interaction avec les ressources matérielles de l'infrastructure. Les environnements logiciels proposés par la plateforme tels que les systèmes d'exploitation et les SGBDs encapsulent toute complexité d'interaction avec les ressources matérielles aux utilisateurs. Un exemple bien connu des services PaaS est Google App Engine [54].

Logiciel comme Service (SaaS). Le SaaS est un modèle permettant d'appeler et exploiter un ensemble d'applications distantes installées sur des serveurs cloud. On peut y trouver plusieurs

catégories d'applications allant de la gestion de la relation client (en anglais, "CRM" : Customer Relationship Management) à la gestion des comptabilités, outils collaboratifs et messagerie. L'application Web de bureautique Salesforce [17] est un exemple des logiciels fournis comme un service. Cet éditeur offre un service en ligne pour la distribution des logiciels tout en diminuant les charges clients pour l'achat des licences ainsi que l'installation et la maintenance des logiciels.

2.2 Technologies utilisées

Cette section répertorie les technologies de base qui sont nécessaires pour construire un cloud et de comprendre son principe de fonctionnement. Toute application doit avoir besoin d'un modèle de stockage, un modèle de calcul, et un modèle de communication [21]. La conception de ces modèles doit favoriser le passage à l'échelle à partir des serveurs simples à des milliers de nœuds de calcul. En général, les clouds sont basés sur les datacenters pour l'implémentation de leurs infrastructures et services. En l'occurrence, la mise en œuvre de tels centres de données a bien contribué à plusieurs avancées technologiques. La technologie de virtualisation est particulièrement la plus importante.

2.2.1 Datacenter

Un datacenter (ou centre de données) est un site physique servant à rassembler et héberger une grande collection d'équipements informatiques, tels que les serveurs, supports de stockage et connecteurs réseau. La co-localisation de ces équipements est due aux exigences environnementales communes, besoins de sécurité physique et facilité d'entretien [25]. D'un point de vue structurel, un datacenter est construit à base de baies (en anglais, racks) rassemblant l'ensemble d'équipements nécessaires (c-à-d, serveurs, unités de stockage, commutateurs réseau, etc.). La taille d'un datacenter peut varier d'une pièce à un bâtiment tout entier.

Il est à noter que les datacenters sont généralement fixes, mais il existe d'autres solutions sous forme de datacenters mobiles. En effet, plusieurs fournisseurs proposent des solutions de cloud privé à travers des datacenters inclus dans des conteneurs mobiles. Par exemple, la société "Huawei" propose le datacenter "IDS1000-A All-in-One Container" [3]. Cette solution est bien adaptée à des scénarios mobiles (par exemple, la gestion des catastrophes et la prospection pétrolière).

2.2.2 Virtualisation

La virtualisation est un outil technologique clé dans le cloud computing. Elle permet d'exécuter plusieurs machines virtuelles avec des systèmes d'exploitation différents sur un même serveur physique. Comme le montre la figure 2.2, le logiciel de virtualisation joue le rôle d'in-

termédiaire entre la couche hardware du serveur hébergeur et les machines virtuelles. En l'occurrence, il permet de gérer le partage et l'allocation dynamique des ressources matérielles (c-à-d, processeurs, mémoires, cartes réseau, etc.).

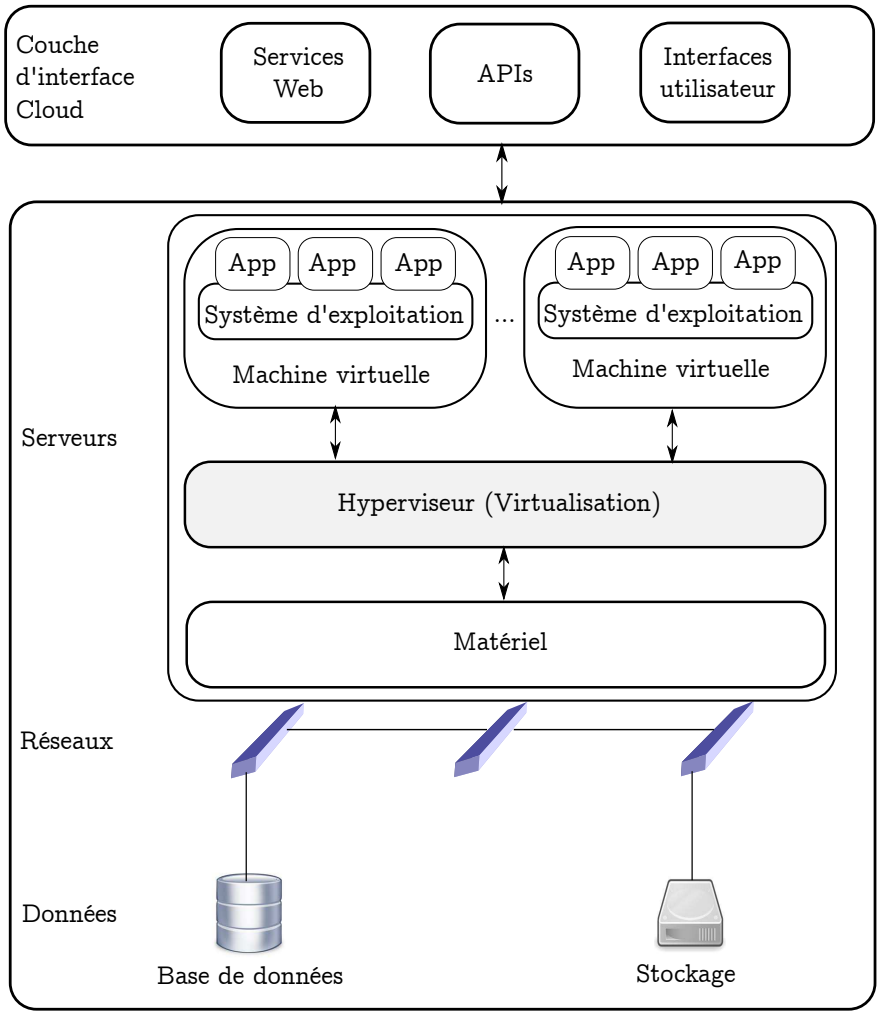


FIGURE 2.2 – Situation de l'hyperviseur dans une architecture cloud simplifiée.

Un autre terme souvent utilisé en décrivant la virtualisation est l'hyperviseur. Il peut être soit de type 1 ou de type 2 [18]. Le type 1 signifie que l'hyperviseur est installé directement sur le matériel. Xen[9] est un exemple d'hyperviseurs classés comme type 1. D'autre part, les logiciels de virtualisation de type 2 ne sont pas directement installés sur le matériel, et sont installés au dessus d'un système d'exploitation déjà en cours d'exécution. Un exemple de ce type est VirtualBox[8]. Mais d'une manière générale, comme le montre la figure 2.2 , le principe reste le même. Un hyperviseur sert d'interface entre machines virtuelles et matériels. Il est à noter que dans plusieurs livres et articles portant sur ce sujet, le terme "Virtual Machine Monitor" ou "VMM" est fréquemment utilisé [68, 49]. Ceci est un autre mot pour désigner l'hyperviseur et les deux termes sont interchangeables.

L'hyperviseur VirtualBox. Comme le montre la figure 2.3 (figure reprise du guide de programmation de VirtualBox [8]), VirtualBox est un logiciel de virtualisation architecturé en plusieurs couches.

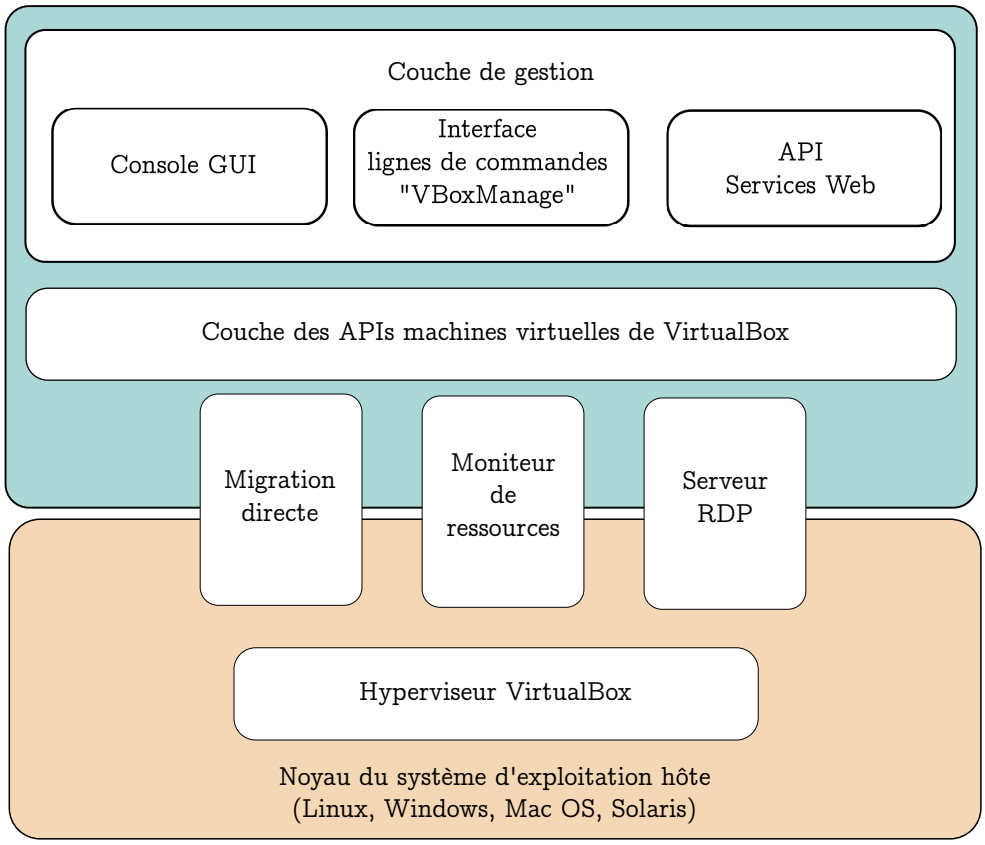


FIGURE 2.3 – Architecture de VirtualBox.

La couche en couleur orange représente les tâches exécutées au niveau noyau du système d'exploitation du serveur hôte. Au-dessus de ce noyau réside l'hyperviseur VirtualBox qui contrôle l'exécution des machines virtuelles tout en assurant qu'elles ne soient pas en situations inter-conflictuelles les unes avec les autres. Au sommet de l'hyperviseur, des modules internes additionnels offrent des fonctionnalités supplémentaires. Le serveur RDP (en anglais, Remote Desktop Protocol) fournit des sorties graphiques liés aux performances des machines virtuelles à un client RDP distant. Le module de migration directe et le moniteur de ressources sont des composants additionnels à rajouter à VirtualBox. La couche bleue représente l'espace utilisateur et elle est essentiellement composée des APIs permettant la création et configuration des machines et réseaux virtuels. Un utilisateur peut bénéficier de ces APIs via plusieurs interfaces, à savoir : (i) la console d'interface graphique utilisateur, (ii) la ligne de commandes "VboxManage" et (iii) les APIs de services web. Le chapitre 5 présente MIDBox, un middleware de déploiement pour le clonage des dispositifs mobiles et la création des réseaux privés virtuels sur VirtualBox par implémentation des commandes "VboxManage".

Clonage. En général, le clonage est l'opération qui consiste à copier une instance d'un objet

donné. La nouvelle instance créée comportera les mêmes informations que celles possédées par l'objet d'origine. Le chapitre 5 fournit un auto-mécanisme basé sur les techniques de virtualisation, pour créer des copies virtuelles des dispositifs mobiles dans un environnement de cloud computing. Les clones (machines virtuelles) sont capables de contenir les mêmes données et applications, mais effectueront les tâches mobiles intensives pour décharger leurs dispositifs réels.

2.2.3 Réseautage

L'architecture du mécanisme de réseautage au niveau d'un datacenter est organisée en plusieurs niveaux [25]. Les nœuds de calcul contenus dans une seule baie sont reliés par un commutateur de réseau Gigabit Ethernet. Le regroupement de ces nœuds de calcul constitue un cluster. Les baies sont reliées par un autre niveau de réseau ou commutateurs avec un nombre de ports plus important. Dans de tels réseaux, les programmeurs doivent être en mesure d'agir sur la couche réseau comportant des équipements d'interconnexion des nœuds de calcul au niveau cluster, voire même aller plus loin en manipulant les connecteurs réseau au niveau baies (voir la figure 2.4).

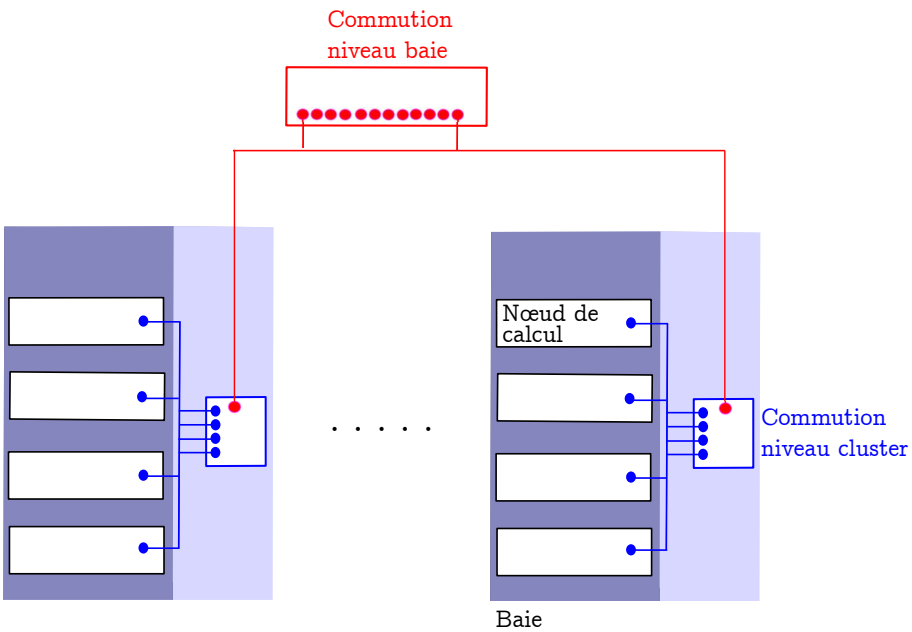


FIGURE 2.4 – Structure d'un datacenter [25].

2.2.4 Stockage

Actuellement, les données traitées et partagées via Internet sont issues de plusieurs sources (par exemple, documents des individus ou organisations, données de capteurs, coordonnées GPS, etc.) et sont de plus en plus qualifiées de massives (en anglais, BigData) en raison de leurs tailles qui peuvent atteindre plusieurs téraoctets. L'évolution croissante des masses de données fut la cause d'apparition de nouveaux modèles de stockage et de calcul. Les systèmes de gestion des fichiers classiques sont inadaptés avec le stockage des données volumineuses. À cet effet, les systèmes de fichiers distribués (DFS) sont présentés comme des solutions typiques répondant aux besoins d'un stockage bien adapté aux données massives.

Les systèmes de fichiers distribués les plus connus sont Google File System (GFS) [53] et Hadoop Distributed File System (HDFS) [48]. HDFS est une version open-source inspiré du GFS original. D'un point de vue technique, le principe de fonctionnement des systèmes de fichiers distribués est basé sur les mécanismes de fragmentation et réplication. Chaque fichier peut être divisé en fragments (64 ou 128 mégaoctets). La réplication des différents fragments sur plusieurs nœuds de calcul qui sont à leur tour distribués sur des baies différentes doit assurer un niveau de disponibilité très élevé. La conduite de ce processus de distribution des fichiers suit une architecture client/serveur où un nœud maître est le seul responsable de l'indexation des différents fichiers au niveau fragments, ainsi que d'autres opérations comme ouvrir, fermer et renommer des fichiers et des répertoires.

2.2.5 Calcul

Auparavant, les calculs intensifs destinés au traitement des données massives n'étaient possibles qu'avec des ordinateurs à usage spécialisé munis de plusieurs processeurs. Vu leurs tailles, ainsi que leurs coûts très élevés, cette catégorie d'ordinateurs n'était exploitable que par les grandes entreprises et organisations (par exemple, la NASA). Cependant, avec l'apparition des technologies de calculs distribués et parallèles, les calculs intensifs associés aux données massives sont devenus possibles en les distribuant sur plusieurs nœuds de calcul. Ces nœuds ne doivent pas posséder de capacités particulières. Par conséquent, l'implémentation des mécanismes de calculs distribués a rendu le traitement des données massives moins coûteux et à la portée de tout le monde.

Le modèle de calcul le plus répandu, qui est associé aux systèmes de fichiers distribués est MapReduce [37]. C'est un environnement logiciel introduit par Google en 2004 afin de supporter des calculs distribués appliqués sur des bases de données volumineuses en utilisant des clusters d'ordinateurs (c-à-d, des collections de nœuds de calcul). Les programmes MapReduce sont conçus de manière permettant d'exécuter des calculs parallèles sur des données massives tout en gérant les pannes matérielles au cours des calculs.

La simplification du traitement des données massives est considérée comme un atout principal du système MapReduce. Le programmeur n'a besoin d'écrire que deux fonctions, appelées

Map et *Reduce*. Le serveur médiateur de MapReduce se chargera du reste. Il doit gérer la planification et réplication des données, l'exécution parallèle et la synchronisation des tâches de calcul sur un cluster d'ordinateurs. Il gère aussi les erreurs et les pannes d'exécution des différentes tâches.

2.3 Mobile Cloud Computing

Le terme "Mobile Cloud Computing" (MCC) a été introduit peu après que le concept de "Cloud Computing" a fait son apparition.

Le Forum du Mobile Cloud Computing défini le MCC comme suit [4] :

"*Le Mobile Cloud Computing se réfère à une infrastructure où à la fois le stockage et le traitement des données se produisent à l'extérieur du dispositif mobile. Les applications cloud mobile déplacent la puissance des calculs et de stockage des données loin des dispositifs mobiles vers le cloud*".

D'une manière plus précise, le MCC implique un ensemble d'utilisateurs mobiles avec des traitements et des services de stockage des données se produisant dans le cloud. En suivant ce concept, les dispositifs mobiles n'auront pas besoin d'une configuration puissante (par exemple, vitesse du processeur et capacité de stockage) puisque tous les modules de calculs complexes et intensifs peuvent être traités dans le cloud [102].

D'une manière générale, comme le montre la figure 2.5, le MCC peut être brièvement défini comme une combinaison du Cloud Computing et Mobile Computing [32, 79]. Ainsi, les dispositifs mobiles (samartphones, tablettes et ordinateurs portables) peuvent établir des connexions sans fil via 3G, 4G et WIFI avec le cloud computing afin de bénéficier de la puissance des calculs et stockages offertes par le cloud. Les utilisateurs mobiles peuvent exprimer leurs requêtes et besoins à travers un ensemble d'applications et interfaces web offrant des services cloud à la demande.

3 Applications collaboratives

La collaboration est un mécanisme qui permet à plusieurs utilisateurs/systèmes de traiter et manipuler des données partagées en vue d'atteindre un objectif commun. Les applications collaboratives offrent des outils pour créer des espaces de travail communs et d'ajouter/proposer des tâches collaboratives.

En particulier, les applications collaboratives mobiles sont spécifiquement destinées aux environnements mobiles où des utilisateurs distants utilisent leurs dispositifs mobiles (par exemple, smartphones et tablettes) afin d'achever des tâches collaboratives. La conception de telles applications particulières doit prendre en compte des exigences spécifiques liées à la limitation

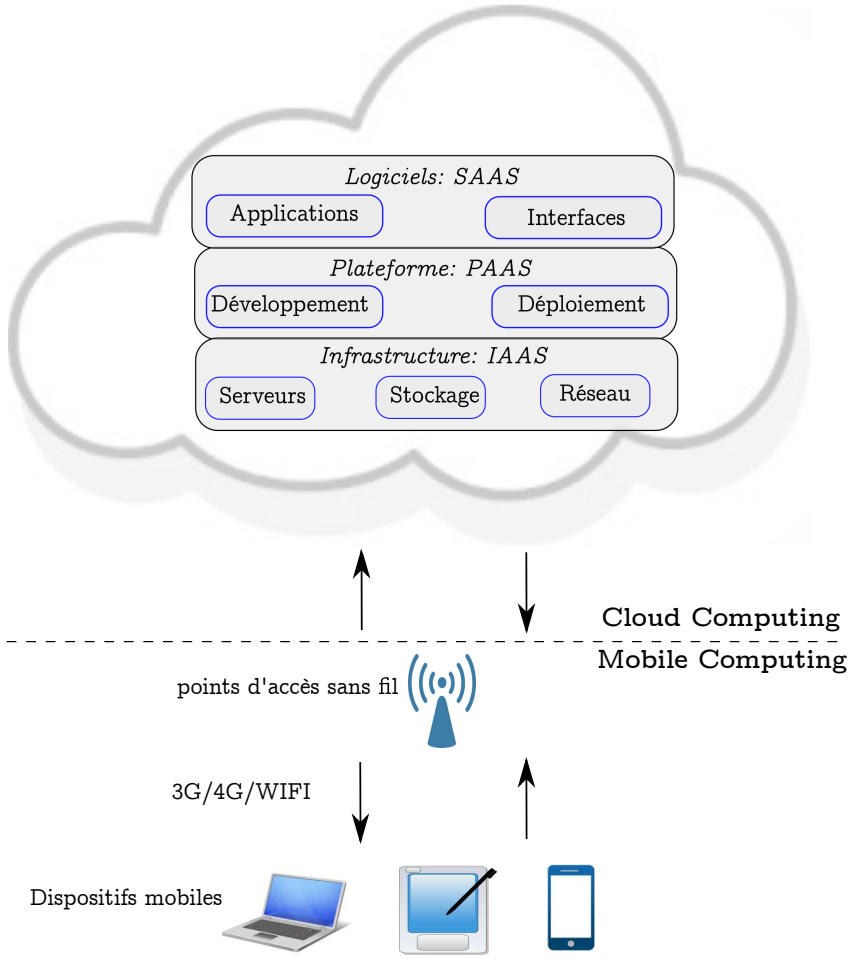


FIGURE 2.5 – Architecture simplifiée du Mobile cloud Computing.

des ressources mobiles ainsi qu'aux connexions réseau instables.

3.1 Formes de collaboration

D'un point de vu opérationnel, chaque application collaborative doit être basée sur un protocole de synchronisation. Ceci implique un mécanisme qui vise à maintenir la cohérence des données partagées comme vues par les différents sites dans des environnements collaboratifs et concurrents. Ainsi, les utilisateurs distants peuvent simultanément accéder/traiter des ressources partagées et la synchronisation sera assurée (d'une manière transparente) par les applications collaboratives. En l'occurrence, la collaboration peut avoir deux formes de synchronisation :

D'une part, en utilisant une forme de collaboration synchrone (aussi appelée collaboration en temps réel), les partenaires manipulent simultanément des ressources partagées et communiquent en travaillant. Les mises-à-jour effectuées par chaque utilisateur sont immédiatement propagées vers les autres afin d'être visibles en temps réel. Les jeux multi-joueurs en ligne et

les éditeurs collaboratifs [43, 91, 105, 110, 64] sont des exemples de cette forme de collaboration. Il est important de noter que l'instabilité des connexions réseau peut être considérée comme un obstacle majeur pour des applications mobiles basées sur une collaboration synchrone. Notre solution basée sur un protocole de synchronisation en temps réel déployée dans le cloud remédie à ce problème.

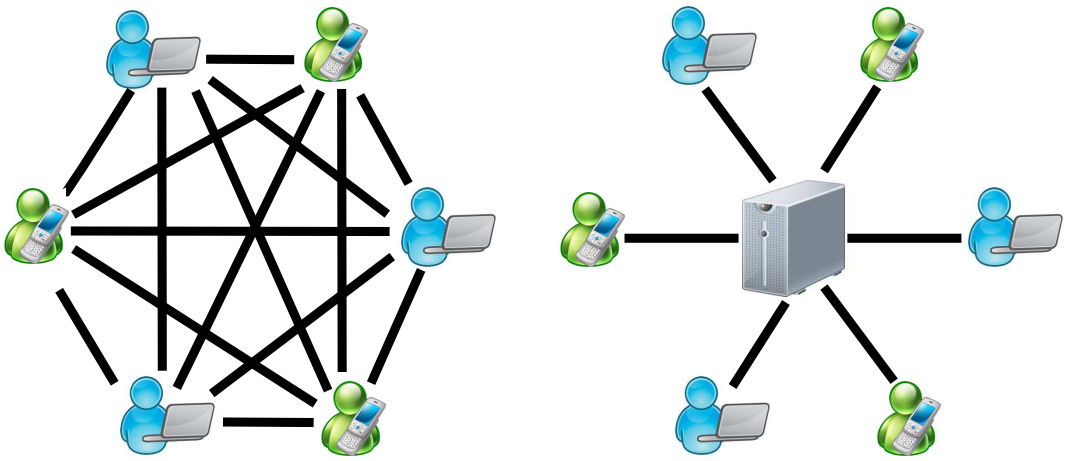
Quant à une collaboration asynchrone, les collaborateurs ne doivent pas nécessairement manipuler les ressources partagées d'une manière simultanée et par conséquent, une communication en temps réel n'est pas requise. Les forums peuvent être considérés comme un exemple typique de cette forme de collaboration.

3.2 Modèles de collaboration

Nous pouvons distinguer deux modèles de collaboration : centralisée et décentralisée.

Le modèle de collaboration centralisée est basé sur une architecture réseau client/serveur. Les différents calculs de synchronisation sont réalisés par un serveur central qui coordonne les différentes tâches de collaboration des clients (voir la figure 2.6 (b)).

D'autre part, un modèle de collaboration décentralisée est basé sur une architecture réseau pair-à-pair. Les ressources partagées sont répliquées et les calculs sont distribués sur les différents sites collaborateurs (voir la figure 2.6 (a)). Le modèle de collaboration décentralisée est plus avantageux par rapport à son opposé en matière de sécurité et de tolérance aux pannes. Il évite ainsi d'avoir un seul point de contrôle.



(a) Modèle de collaboration décentralisée. (b) Modèle de collaboration centralisée.

FIGURE 2.6 – Modèles de collaboration.

3.3 Réplication

La réplication des données est une technologie clé dans les systèmes de partage de données distribués qui permet d'assurer un degré élevé de disponibilité des données partagées et de performance [93]. Elle consiste à maintenir des copies multiples d'objets de données, appelés répliques, sur plusieurs sites [82]. Les données répliquées doivent être synchronisées afin de préserver leur cohérence.

Il ne faut pas confondre la réplication avec la sauvegarde : les données sauvegardées ne changent pas au fil du temps, tandis que les données répliquées ne cessent d'évoluer. Par ailleurs, il existe deux modes de réplication :

(i) La réplication pessimiste essaye de donner aux utilisateurs une illusion d'avoir un seul exemplaire hautement disponible de la ressource partagée [28]. Cet objectif peut être atteint de plusieurs façons, mais le concept de base reste le même : la mise-à-jour des répliques est toujours effectuée par un seul utilisateur et par conséquent l'accès à une réplique sera bloqué jusqu'à ce que sa mise-à-jour soit achevée. La réplication pessimiste se prête bien aux réseaux locaux où les latences sont faibles et les pannes sont rares.

(ii) La réplication optimiste [93] offre un ensemble de techniques pour un partage des données efficace et beaucoup plus adapté à des environnements étendus et/ou mobiles. La principale caractéristique qui distingue les algorithmes de réplication optimistes de leurs homologues pessimistes est leur approche de contrôle de concurrence. Les algorithmes pessimistes coordonnent les répliques d'une manière synchrone lors des accès et bloquent les autres utilisateurs lors d'une mise-à-jour. En revanche, les algorithmes optimistes favorisent les lectures et écritures des données sans imposition de synchronisation a priori. Les mises-à-jour sont propagées en arrière plan et les conflits occasionnels sont résolus après qu'ils se produisent.

En résumé, une réplication optimiste permet de mettre à jour plusieurs répliques d'une manière simultanée et concurrente. Les conflits entre les mises-à-jour au niveau des différentes répliques sont détectés et résolus après qu'ils (c-à-d, les conflits) se sont produits. Par conséquent, ce mécanisme permet aux utilisateurs d'accéder à toute réplique à tout moment, ce qui implique une plus grande disponibilité d'écriture sur les différents sites. Alors qu'avec une réplication pessimiste, l'accès en écriture est limité à une seule réplique.

3.4 Éditeurs collaboratifs

Un éditeur collaboratif consiste à faire collaborer plusieurs utilisateurs en appliquant des opérations d'édition (insertion, suppression, copiage, collage, fusion, etc.) sur un ensemble de ressource partagées (textes, images, etc.). Ces manipulations sont effectuées d'une manière simultanée, concurrente et depuis de différents sites (physiquement dispersés).

Dans le cadre de notre mémoire de thèse, nous nous intéressons particulièrement aux éditeurs collaboratifs mobiles dans des réseaux pair-à-pair. De tels éditeurs doivent assurer [108,

64] : (i) une réactivité locale maximale, (ii) un degré élevé de concurrence, (iii) une cohérence à terme des données, (iv) une coordination décentralisée, (v) un passage à l'échelle efficace et (vi) une reprise après panne rapide. D'un point de vue technique, chaque site (ou utilisateur) collaborateur possède et manipule librement sa copie (réplique) locale de la ressource partagée. Chaque opération d'édition localement générée sera immédiatement diffusée vers les autres sites afin d'y être intégrée. Les opérations générées et appliquées sur une ressource partagée peuvent être liées par les relations suivantes :

Relation de dépendance causale. Une opération o est causalement dépendante d'une autre opération o', si l'effet d'exécution de o sera appliqué à la donnée précédemment générée par l'application de l'opération o'. Par exemple, une opération $del(x, p)$ qui supprime un élément x à la position p est causalement dépendante d'une opération d'insertion ($ins(x, p)$) qui a généré le même élément à la position p.

Dépendance sémantique. Une opération o est sémantiquement dépendante d'une opération o', si l'exécution de l'opération o' suivie par l'exécution de l'opération o possède un sens logique (le cas contraire peut être faux, ou possède un autre sens logique). Par exemple, si on considère les deux opération *o1 : mettre_le_sucre* et *o2 : mélanger_le_café* pour la préparation d'un café, alors $o2$ est sémantiquement dépendante de $o1$.

Les accès concurrents aux ressources partagées peuvent engendrer des situations conflictuelles et des vues (sur les différents ressources partagées) incohérentes. À cet effet, un modèle de contrôle de la concurrence doit être implémenté et mis en œuvre afin de maintenir la cohérence et corriger les divergences tout en respectant les relations de dépendance entre les opérations d'édition. Ce module constitue le noyau de chaque éditeur de collaboration et il est généralement basé sur les deux approches suivantes :

Approche des Transformées Opérationnelles. Dans la littérature des éditeurs de collaboration, l'approche des Transformées Opérationnelles [43] (OT) est considérée comme une méthode sûre et efficace pour le maintien de la cohérence. En effet, elle vise à assurer la convergence des copies indépendamment de l'ordre d'exécution des mises-à-jour appliquées par les utilisateurs sur les différentes copies. Dans cette approche, les opérations de mise-à-jour sont appliquées localement et puis diffusées vers les autres sites distants. Chaque site recevant une opération distante doit la transformer par rapport à toutes les opérations concurrentes déjà appliquées. Ceci vise à préserver et inclure les effets des opérations concurrentes. Cette approche a été plus appliquée aux structures de données linéaires moyennant deux opérations : (i) *ins(e,p)* : pour insérer un élément e à la position p et (ii) *del(p)* : pour supprimer un élément à la position p. Les développeurs utilisant cette approche doivent définir comment transformer une opération distante reçue selon plusieurs cas de conflit.

Afin de bien comprendre cette approche, considérons l'exemple suivant : étant donné deux sites, Site1 et Site2, partant d'un état commun d'un document partagé "XYZ" (voir la figure 2.7). Au niveau du Site1, un utilisateur mobile exécute *Op1* = Ins (2,A) pour insérer le caractère 'A' à la position 2 et se retrouve avec "XAYZ". Parallèlement, un autre utilisateur au niveau

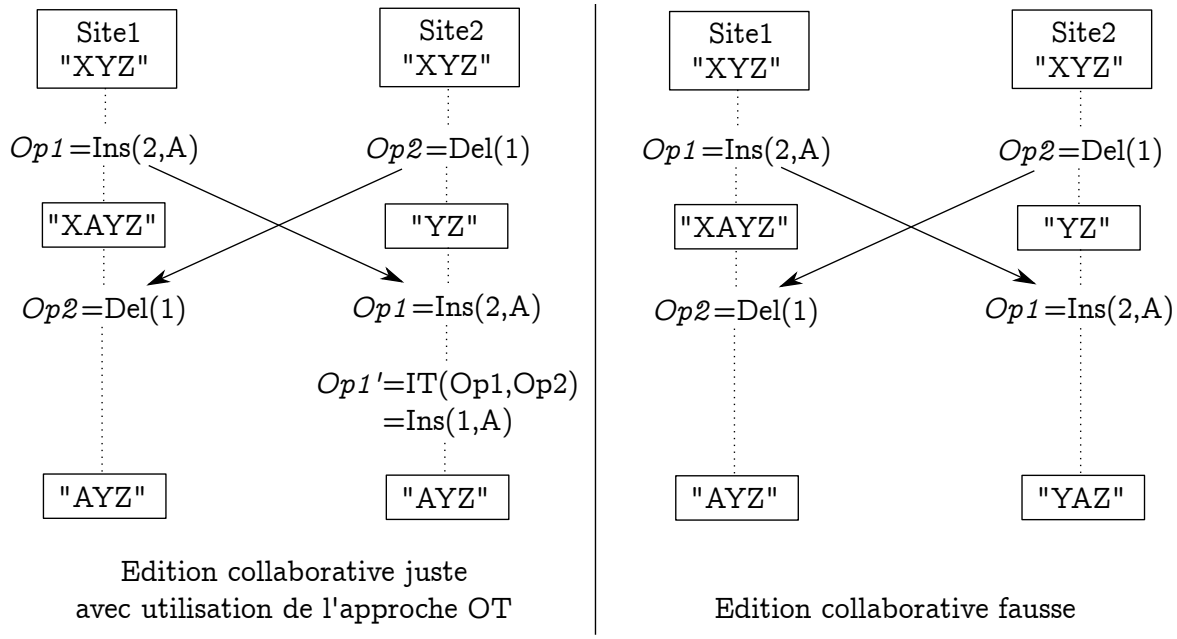


FIGURE 2.7 – Exemple d'utilisation de l'approche OT lors d'une édition collaborative.

du Site2 exécute l'opération *Op2*= *Del*(1) pour supprimer le caractère 'X' à la position 1 et obtient l'état "YZ". Après que les opérations ont été échangées entre Site1 et Site2, si elles sont naïvement appliquées, les deux sites obtiennent des états incohérents : Site1 avec "AYZ" et Site2 avec "YAZ". La transformation opérationnelle est considérée comme une méthode sûre et efficace pour le maintien de la cohérence. En général, elle applique des algorithmes de transformation appelés IT (en anglais, "Inclusive Transformation"), de telle sorte que pour chaque paire possible d'opérations simultanées, le programmeur d'application doit préciser comment intégrer ces opérations indépendamment de l'ordre de leur réception. Ainsi, au Site2, l'opération *Op1* doit être transformée pour inclure l'effet de *Op2* : *Op1'*= IT(*Op1*, *Op2*) = $Ins(1, A)$. Quant au Site1, l'opération *Op2* reste inchangée. Par conséquent, les deux sites obtiennent le même état "AYZ".

L'approche CRDT. Cette approche consiste à définir un type de données répliqué et commutatif (en anglais : "CRDT : Commutative Replicated Data Type"). Ceci implique un type de données où toutes les opérations commutent indépendamment de leur ordre d'arrivée. Les répliques d'un CRDT convergent automatiquement sans contrôle de concurrence complexe [100, 101]. Un exemple typique d'un tel CRDT est l'ensemble des entiers muni des deux opérations commutatives d'addition et de soustraction. Cette approche se voit plus adaptée à des structures hiérarchiques telles que les arbres et les graphes. Dans le chapitre 5 nous présentons MobiRdf, un protocole d'édition collaborative des données liées mobiles où le modèle de concurrence est basé sur la commutativité des requêtes de mise-à-jour. Dans le cas du système MobiRdf, il ne s'agit pas de définir un nouveau CRDT, mais seulement utiliser un sous ensemble de requêtes du langage SPARQL-UPDATE qui sont naturellement commutatives.

4 Patrons de conception

Pour la conception des applications collaboratives déployées dans le cloud, nous utilisons les patrons de conception. Plusieurs définitions des patrons de conception peuvent être trouvées dans la littérature. La définition la plus recommandée est la suivante [67] :

"*Un patron pour une architecture logicielle décrit un problème de conception récurrente particulière qui se pose dans des contextes spécifiques de conception et présente un schéma générique éprouvé pour sa solution. Le schéma de solution est spécifié en décrivant ses éléments constitutifs, leurs responsabilités et leurs relations, et la manière dont ils collaborent.*"

Les auteurs de [50], ont donné une définition basée sur les critères suivants :

— *Les patrons de conception sont considérés comme une façon de réutiliser un ensemble de connaissances abstraites d'un problème et sa solution.*
— *Un patron est une description du problème et l'essentiel de sa solution.*
— *Un patron est une abstraction réutilisable qui peut être appliquée dans différents contextes.*
— *Un patron est une abstraction d'une forme concrète qui peut être développée d'une manière récurrente selon des contextes spécifiques.*

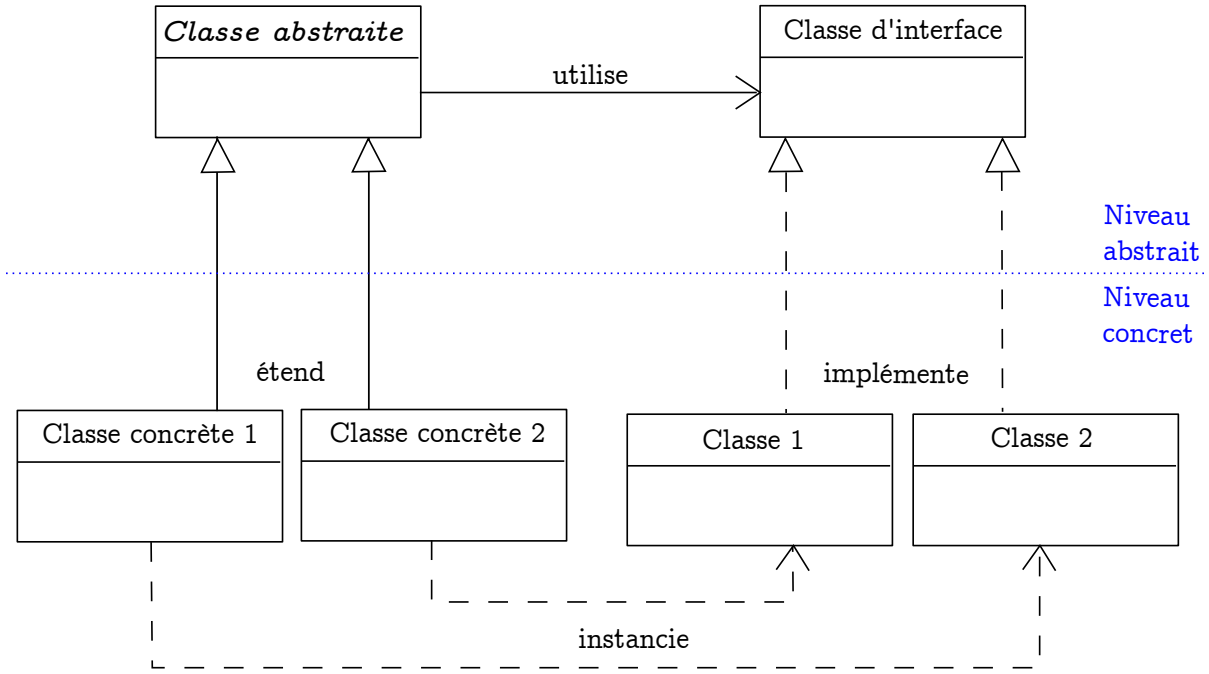


FIGURE 2.8 – Patrons de conception : fabriques abstraites.

4.1 Fabriques abstraites

Plusieurs types de patrons de conception existent. Nous nous intéressons particulièrement aux fabriques abstraites que nous utiliserons pour la conception de nos modèles génériques.

D'une manière générale une fabrique de conception sert à regrouper un ensemble de méthodes et données qui sont communes à une famille d'objets. La figure 2.8 résume le principe des fabriques de conception. Il s'agit de définir des classes d'interface regroupant des méthodes associées à une catégorie d'objets. Cette catégorie est définie par une classe abstraite. Selon un contexte particulier, des classes concrètes qui étendent la classe abstraite vont instancier d'autres classes qui implémentent les classes d'interfaces par redéfinition de leurs méthodes.

Par exemple, on peut définir une classe d'interface appelée "*soins*" qui regroupe un ensemble de méthodes, telles que "*Opérations_chirurgicales*" et "*Premiers_Soins*". Cette interface peut être utilisée par une classe abstraite appelée "*Staff_Médical*". Ceci représente un niveau abstrait de conception. Quant au niveau concret, on peut définir les classes concrètes "*Dentiste*" et "*Infirmier*" qui étendent la classe "*Staff_Médical*" et respectivement instancient les classes "*Extraction_Dentaire*" et "*Lavage_Oculaire* " qui implémentent l'interface "*soins*".

5 Conclusion

Dans ce chapitre, nous avons introduit les principaux concepts et technologies liés à nos deux axes de recherche, à savoir le cloud et la collaboration mobile. La première partie de ce chapitre donne un aperçu sur les modèles et services du cloud ainsi que ses technologies de fondement. Quant à la deuxième partie, elle a été consacrée à la présentation des différents concepts liés aux applications collaboratives, à savoir : les formes et modèles de collaboration, la réplication et la synchronisation.

Le MCC résultant de la combinaison de ces domaines implique le déploiement des tâches et données vers le cloud. Nous nous intéressons particulièrement au déploiement des applications collaboratives mobiles. Ainsi, dans le chapitre suivant, nous allons présenter un état de l'art des travaux de recherche autour du déploiement mobile et des applications collaboratives dans le cloud.

Chapitre 3

État de l'art

Sommaire

1 Introduction

Le développement des applications mobiles a connu des avancées technologiques qui ne cessent de s'accroître depuis longtemps. Un tel investissement attractif a créé une compétitivité intense entre les grandes entreprises (par exemple Google, Samsuung, Apple, etc.) afin de proposer des applications mobiles modernes dans plusieurs domaines (comme la santé, le commerce, les jeux et éditeurs collaboratifs, etc.). En outre, le web d'aujourd'hui est caractérisé à la fois par des aspects social, collaboratif et mobile. Par conséquent, une importante part de ce grand marché mondial de développement est exclusivement dédiée aux applications collaboratives mobiles. Cependant, une exploitation efficace de telles applications en utilisant des mobiles limités en ressources demeure un défi. En effet, la durée de vie de la batterie et l'instabilité des connexions réseau sont considérées comme les préoccupations majeures des utilisateurs mobiles ainsi que les développeurs. Dans la littérature, la solution la plus populaire est le déploiement des tâches mobiles vers le cloud.

Dans ce chapitre, nous explorons l'état de l'art des travaux effectués dans le cadre de nos deux axes de recherches majeurs : (i) le déploiement des tâches mobiles vers le cloud et (ii) la collaboration en temps réel. En outre, cette étude nous a permis d'examiner les différents travaux de conception proposés dans ces deux domaines.

2 Déploiement des tâches mobiles vers le cloud

Les auteurs d'une étude menée sur le déploiement [76] ont défini le déploiement des tâches mobiles vers le cloud [1] comme suit : "*Le déploiement mobile consiste à pousser les tâches mobiles intensives vers des serveurs distants et recevoir des résultats depuis ces serveurs*".

Dans cette section nous présentons un aperçu des différentes approches existantes en matière de déploiement mobile. Comme illustré à la figure 3.1, nos critères pour définir cet aperçu sont basés sur les questions clés qui ont été abordées dans le domaine du déploiement mobile [41]. Nous nous concentrons sur :

— Mécanismes de déploiement mobile : ces mécanismes représentent les différentes méthodes et techniques utilisées pour déployer la totalité ou seulement des fragments de programmes mobiles vers le cloud. Ce déploiement est généralement lié à la limitation de la durée de vie de la batterie du mobile ainsi que la nécessité d'une puissance de calcul plus élevée.
— Gestion des données mobiles : cette gestion implique les opérations de sauvegarde, transfert et restauration des données mobiles depuis et/ou vers le cloud. Ce type de déploiement est lié à l'insuffisance de l'espace de stockage des mobiles.

1. Dans le reste du document, nous utilisons le terme "déploiement mobile" pour désigner le déploiement des tâches mobiles vers le cloud.

— Tolérance aux pannes : ceci concerne le maintien du bon fonctionnement des tâches et la préservation des données mobiles déployées dans le cloud.
— Conception : ce critère est lié à la possibilité de réutilisation et implémentation des modèles de déploiement existant à travers des architectures et APIs fournis.

Pour plus de précision, nous pouvons étendre la définition précédente comme suit : "*Le déploiement mobile consiste à pousser les tâches intensives et sauvegarder les données mobiles vers des serveurs distants et recevoir des résultats depuis ces serveurs, tout en assurant le bon fonctionnement des tâches et la préservation des données mobiles déployées*".

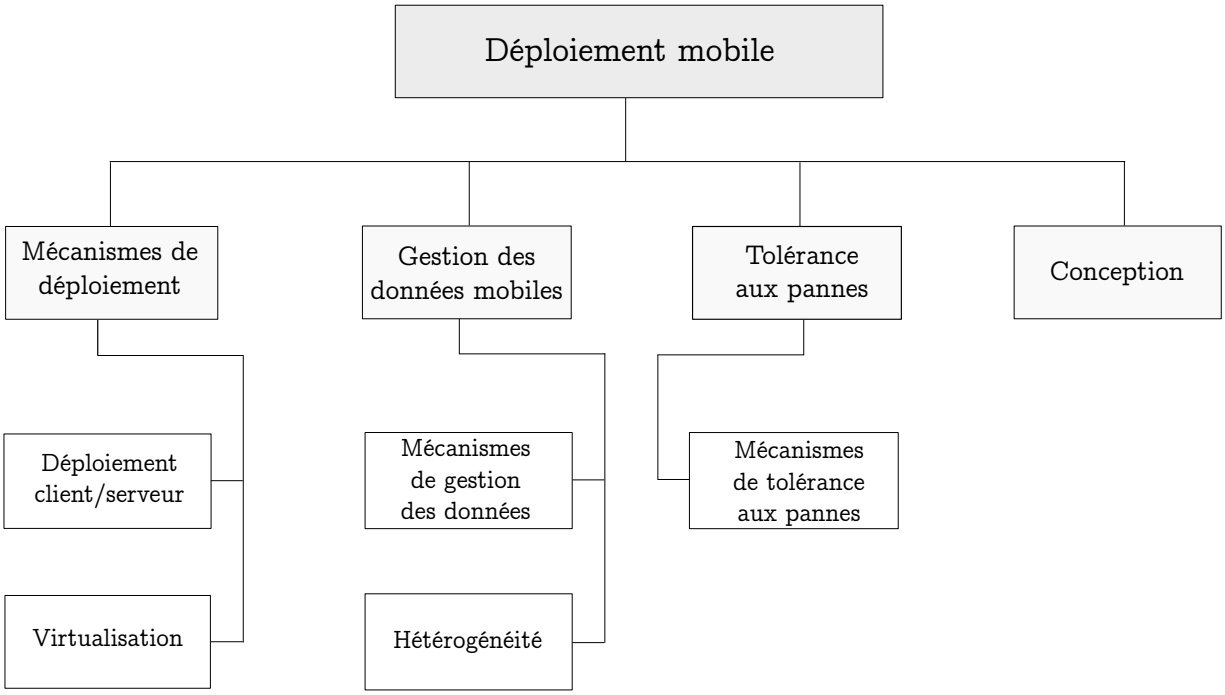


FIGURE 3.1 – Déploiement mobile.

Il est à noter qu'un tel déploiement est souvent basé sur un processus de partitionnement de programmes. D'une manière générale, ce processus consiste à diviser une application en un ensemble d'unités logiques (par exemple : modules, procédures et threads [2]) en vue de leurs exécutions sur un ou plusieurs serveurs. En outre, ce processus est généralement associé à un modèle de coût aidant à choisir les partitions nécessitant un déploiement.

2.1 Mécanismes de déploiement mobile

Les travaux de recherche actuels pour le déploiement des tâches mobiles vers le cloud sont axés sur deux directions : déploiement client/serveur et déploiement par virtualisation.

2. Pour une raison de simplicité, nous utilisons le terme thread pour désigner un processus léger (ou fil d'exécution) d'un programme.

2.1.1 Déploiement client/serveur

Le principe de ce mode de déploiement est généralement basé sur le mécanisme d'appels de procédures distantes. Les mobiles (clients) peuvent faire appel à des services installés sur des serveurs distants afin de déployer des tâches et des données mobiles.

SPECTRA. *SPECTRA* [47] est un système qui peut être utilisé pour le déploiement des tâches mobiles via un mécanisme d'exécution à distance. L'architecture de ce système implique un client exécutant des applications sur un mobile et un serveur accueillant et exécutant un ensemble d'opérations déployées par le client. Le principe de fonctionnement de ce système peut se résumer en trois points : (i) *SPECTRA* prend une application en entrée et procède à son analyse afin de déterminer l'ensemble des opérations qu'elle contient ; (ii) pour chaque opération le système spécifie un plan d'exécution et décide quelle opération nécessite une exécution distante ; (iii) en utilisant le plan d'exécution précédemment élaboré, *SPECTRA* fait des appels de procédure à distance pour exécuter les opérations sélectionnées sur un serveur distant.

Afin de mettre en œuvre ces tâches de déploiement, *SPECTRA* est basé sur des modules de surveillance des différentes ressources locales et distantes (par exemple, CPU, réseau et batterie).

Il est à noter que *SPECTRA* ne fait pas de partitionnement automatique. Les développeurs souhaitant utiliser ce système doivent préalablement partitionner leurs applications d'une manière manuelle.

CHROMA. *CHROMA* [24] est identique à *SPECTRA* dans le sens de sa mise en œuvre. C'est un système d'exécution où des clients font appel à des procédures depuis des serveurs distants. Le principe de fonctionnement de ce système est basé sur un concept nommé tactique (en anglais, tactic). Comme le montre la figure 3.2, pour chaque opération d'une application client, le système prépare un ensemble de tactiques où chaque tactique doit définir une seule combinaison d'un ensemble d'appels de procédures distantes ordonnées pour l'exécution de cette opération sur un serveur distant. *CHROMA* doit choisir la meilleure tactique selon la disponibilité des ressources requises. À cet effet, le moteur de sélection des tactiques est basé sur deux mécanismes : (i) la prévision instantanée des ressources demandées par chaque opération et (ii) la surveillance continue de la disponibilité de ces ressources du côté des clients ainsi que du côté serveur.

HYRAX. L'auteur de [81] a présenté *HYRAX* pour l'exécution des applications mobiles supportées par le système d'exploitation Android sur une plateforme mobile. Le principe de *HYRAX* est basé sur la distribution des calculs ainsi que des données en utilisant une plateforme *HADOOP* [2] déployée sur plusieurs dispositifs Android. L'idée est de fournir un cloud mobile sous forme d'un cluster de mobiles.

HADOOP est une implémentation open-source de *MapReduce* [37] et offre une interface d'accès à un cluster de nœuds de calcul. Ce système est destiné à la coordination des traitements distribués des données à grande échelle et qui sont stockées dans un système de fichiers

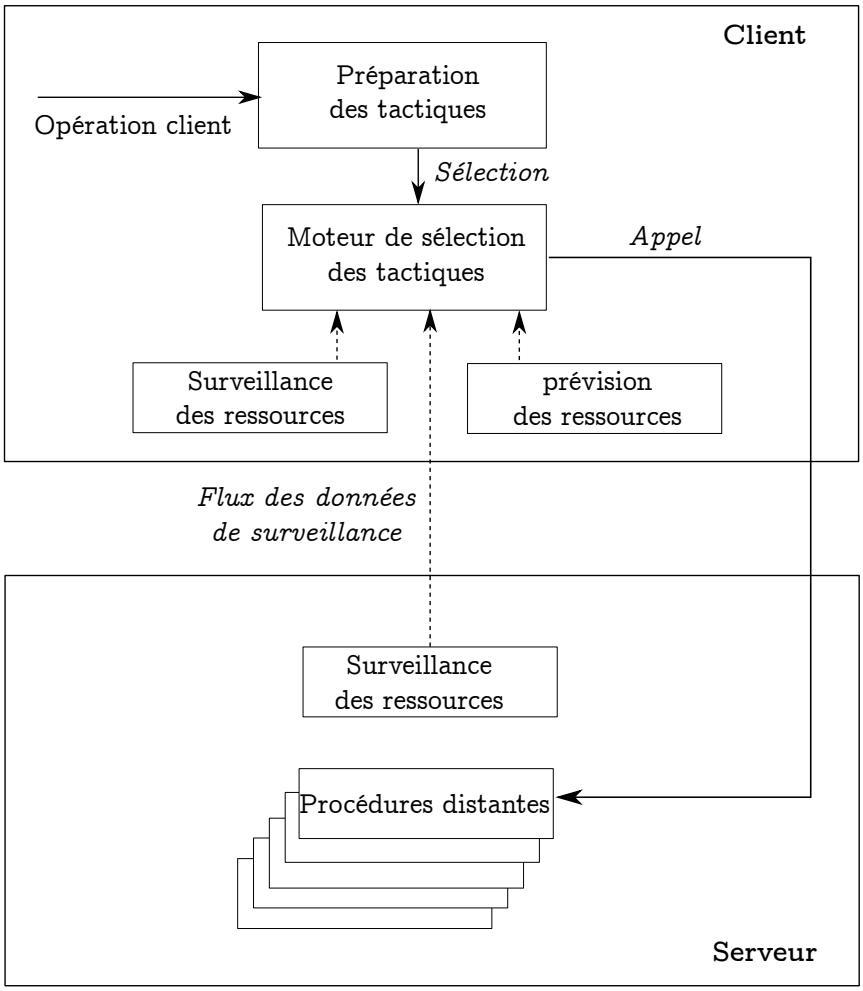


FIGURE 3.2 – Principe de fonctionnement du système CHROMA.

distribués (HDFS). D'un point de vue opérationnel, *HADOOP* peut créer quatre types de processus différents [29, 99, 117] : *NameNode*, *JobTracker*, *DataNode* et *TaskTracker*. Le *NameNode* gère un répertoire de blocs de données qui composent les fichiers HDFS. Le *JobTracker* gère les travaux et coordonne les sous-tâches entre les *TaskTracker*. Le *DataNode* sauvegarde et donne accès aux blocs de données, et le *TaskTracker* exécute les tâches qui lui sont assignées par le *JobTracker*.

Le principe de fonctionnement de *HADOOP* peut être résumé comme suit : un seul service *NameNode* et un autre *JobTracker* s'exécutent sur un serveur central, tandis que plusieurs instances des processus *DataNode* et *TaskTracker* s'exécutent sur les différents nœuds de calcul (c-à-d, un processus par nœud, voir la figure 3.3).

Le *NameNode* envoie des commandes aux *DataNode* pour créer, supprimer et répliquer des blocs de données au niveau des nœuds de calcul. Il est aussi responsable du "mapping" de ces blocs de données. D'autre part un processus *JobTracker* est responsable du partitionnement des travaux (programmes) en un ensemble de tâches qui vont être redirigées vers les différents

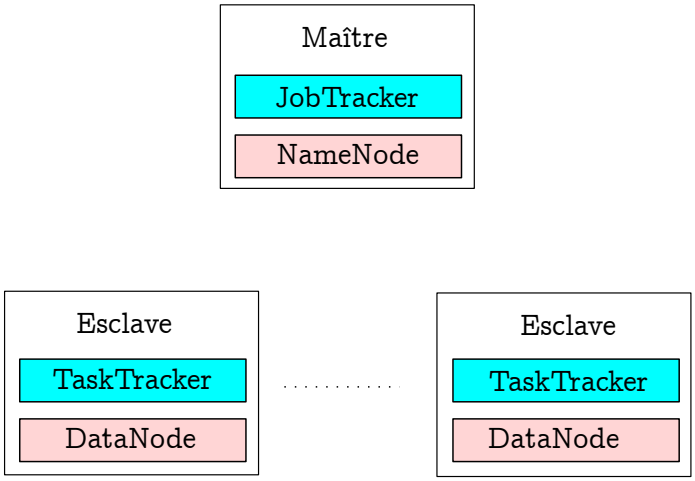


FIGURE 3.3 – Architecture typique d'un cluster *Hadoop* [81].

nœuds de calcul. C'est le *TaskTracker* qui se chargera de l'exécution de la tâche confiée.

Comme le montre la figure 3.4, d'une manière similaire à *HADOOP*, l'architecture distribuée de *HYRAX* est basée sur un serveur central qui coordonne et accède à l'ensemble des mobiles d'un cluster.

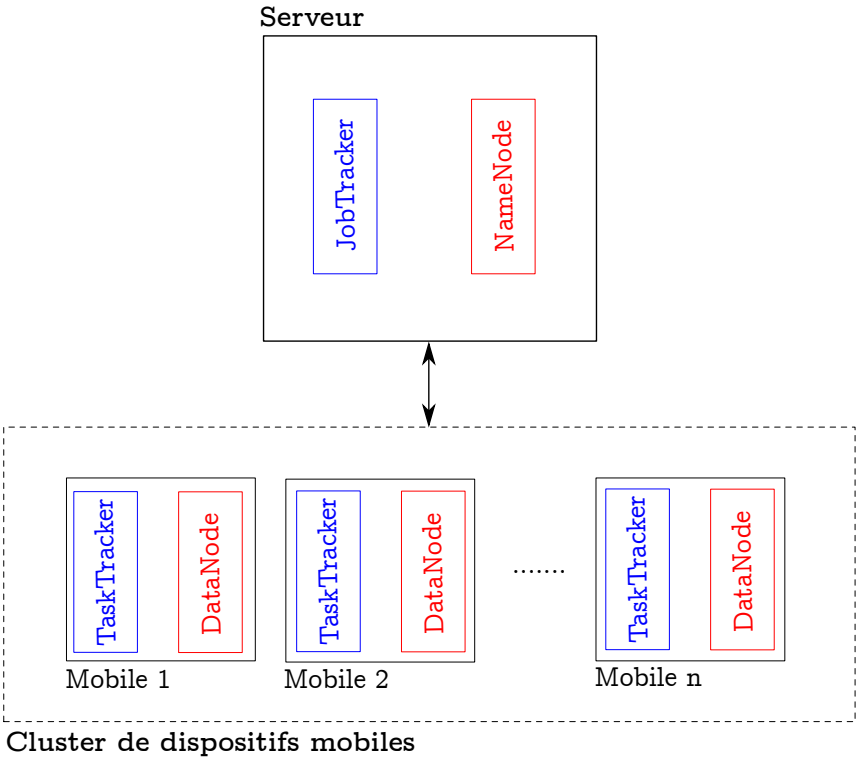


FIGURE 3.4 – Architecture simplifiée de HYRAX.

Ce système dispose également d'un *NameNode* et d'une instance *JobTracker* s'exécutant

sur le serveur central. Bien qu'il n'effectue aucun traitement, ce serveur est responsable de la coordination des données stockées et des tâches exécutées sur les mobiles. Comme prototype de ce système, *HyraxTube* [81] est présenté comme une application mobile distribuée pour la recherche et le partage des objets multimédia. Avec cette application, chaque mobile est capable d'enregistrer des fichiers multimédia dans un système de fichiers *HADOOP* distribué sur l'ensemble de ces mobiles et de lancer des tâches d'enregistrement sonore.

Cuckoo. *Cuckoo* [65] est un système pratique qui peut être utilisé pour migrer les calculs d'applications android. Les applications déployées avec *Cuckoo* doivent être réécrites en utilisant un modèle de programmation offert aux développeurs. Ce modèle prend en charge l'instabilité des connexions réseau afin de permettre l'exécution des applications en deux modes, local et distant. Le système *Cuckoo* est basé sur un gestionnaire de ressources s'exécutant sur les mobiles pour décider si une méthode doit être exécutée localement ou invoquée à distance selon la disponibilité des ressources. Ce gestionnaire est aussi utilisé afin d'identifier des ressources distantes susceptibles d'être utilisées lors de l'invocation des méthodes distantes. Afin de mettre en œuvre cette communication entre le gestionnaire de ressource du côté mobile et des ressources situées sur des serveurs distants, *Cuckoo* utilise le middleware de communication *Ibis* [115]. Un client utilise son identificateur *Ibis* pour établir une connexion avec un serveur et invoquer des services distants. Cependant, si le service demandé n'est pas disponible, le client procèdera par un envoi d'une requête d'installation comportant le fichier compressé *.jar* (en anglais : Java ARchive) du service concerné.

2.1.2 Déploiement par virtualisation

La virtualisation consiste à créer des copies de ressources matérielles et logicielles sur des serveurs distants. C'est une technologie clé dans la conception des clouds qui vise à offrir des capacités de stockage et de calcul à grande échelle. Elle permet l'extension des ressources des mobiles par migration des calculs et données vers des machines virtuelles dans le cloud.

CLOUDLET. Les auteurs de [97] considèrent que la latence des réseaux étendus (en anglais, WAN : Wide Area Network) ainsi que les délais associés aux bandes passantes sont des obstacles majeurs pour effectuer le déploiement des tâches mobiles vers des clouds publics. Il suggèrent la solution de *Cloudlet* (petit cloud) où un déploiement peut avoir lieu depuis des mobiles vers des petits datacenters qui sont situés à proximité.

Le mobile est connecté au *Cloudlet* par un réseau sans fil à haut débit caractérisé par une faible latence et une large bande passante, garantissant ainsi une réponse interactive en temps réel. Cependant, si l'utilisateur perd sa connexion avec le *Cloudlet* suite à son éloignement, il peut basculer vers un cloud public, ou même travailler en mode offline au pire des cas.

Pour une migration moins contraignante, *Cloudlet* impose un minimum de restrictions sur le logiciel tout en simplifiant la gestion. Cette solution est basée sur une "*personnalisation transitoire*" de l'infrastructure du *Cloudlet* en utilisant la technologie des machines virtuelles. L'as-

pect transitoire signifie qu'un *Cloudlet* doit pouvoir restaurer son état original après chaque utilisation. Durant cette utilisation transitoire, les logiciels clients sont exécutés par des machines virtuelles qui les encapsulent et les séparent des environnements logiciels de l'hôte *Cloudlet*.

MAUI. Le système de déploiement des tâches mobiles *MAUI* [36] est présenté comme une combinaison de deux approches : la migration des machines virtuelles et le partitionnement des applications. Tout d'abord, les développeurs doivent participer via le partitionnement de leurs programmes tout en repérant les méthodes susceptibles d'être exécutées sur des serveurs *MAUI* distants. Au cours de l'exécution d'une application mobile, le composant *MAUI* installé côté mobile détermine le coût et le bénéfice attendus de la migration de chaque méthode. En outre, *MAUI* surveille constamment les connexions réseau tout en estimant la latence et la bande passante. Toutes ces mesures sont nécessaires pour la prise des décisions de déploiement. En l'occurrence, des méthodes seront sélectionnées pour un déploiement sur des serveurs distants tandis que d'autres vont continuer à s'exécuter localement.

La figure 3.5, présente un aperçu sur une architecture globale et simplifiée du système *MAUI*. Du côté mobile, cette architecture fait abstraction de trois composantes : (i) un moniteur de ressources qui évalue les coûts d'exécution des différentes méthodes d'un programme en terme de consommation énergétique, trafic réseau et déploiement, (ii) un moteur de décision qui sélectionne les méthodes qui sont susceptibles a être exécutées à distance selon les informations fournies par les moniteurs de ressources et (iii) un client proxy qui prend en charge le déploiement des méthodes sélectionnées. D'autre part, le système *MAUI* côté serveur comprend les trois principales composantes suivantes : (i) un moniteur de ressource, (ii) un serveur proxy qui reçoit les opérations déployées à partir des clients proxy et (iii) un coordinateur qui gère l'allocation des ressources au profit de ces méthodes déployées.

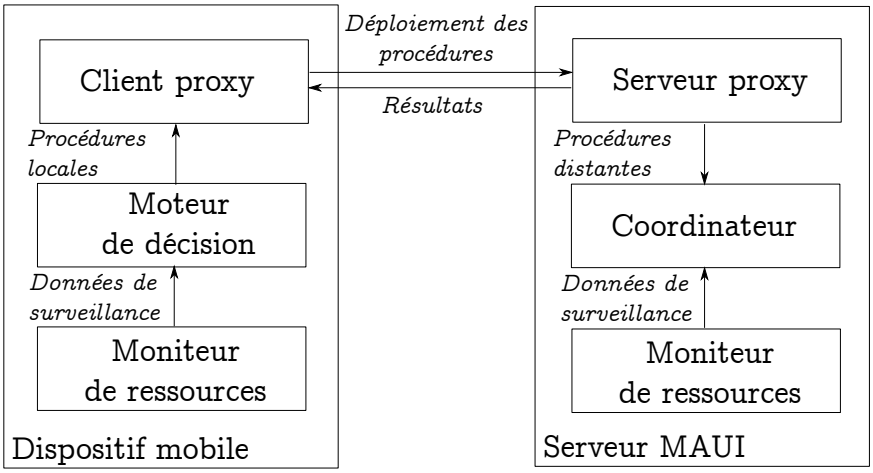


FIGURE 3.5 – Architecture simplifiée du système MAUI [36].

Il est important de noter que lorsqu'un mobile perd contact avec un serveur alors que ce serveur est en train d'exécuter une méthode distante, le système *MAUI* renvoie la commande de déploiement au proxy local. Ce dernier tentera de trouver un nouveau serveur, sinon cette

méthode sera exécutée localement.

CLONECLOUD. *CloneCloud* [33] est également un système basé sur la virtualisation visant à déployer une partie d'une application mobile en cours d'exécution vers un serveur de ressources, soit par 3G ou WiFi. Le principe de *CloneCloud* consiste à migrer des threads depuis un mobile vers un clone (c-à-d, une machine virtuelle) de ce dispositif dans la cloud. L'exécution d'une application mobile est distribuée sur le mobile et son clone. L'application côté mobile continue son exécution normale tout en suspendant exclusivement tout accès aux threads migrés. Après achèvement de son exécution sur un serveur distant, le thread déployé sera réintégré à son processus original dans le mobile réel.

La migration via *CloneCloud* est similaire à celle de *MAUI* [36]. Elle est principalement basée sur un mécanisme de partitionnement d'applications. Ce mécanisme est lui-même composé de trois processus : (i) un analyseur statique, qui prend le code d'une application en entrée et le marque avec des points de repère correspondant aux emplacements des threads à migrer vers des serveurs distants, (ii) un moniteur dynamique qui construit un modèle pour l'évaluation des coûts de migration et d'exécution distante sur le cloud, et (iii) un moteur de décision pour choisir les méthodes à déployer pour une exécution distante dans le cloud en se basant sur l'évaluation offerte par le modèle de coût. Mais contrairement au système *MAUI*, *CloneCloud* n'impose aucune restriction sur la modification des applications. En effet, il n'est pas requis aux développeurs d'effectuer un repérage préalable pour les méthodes nécessitant un déploiement.

2.2 Gestion des données mobiles

La gestion des données mobiles au niveau du cloud soulève plusieurs préoccupations pour les utilisateurs et les fournisseurs. Les données mobiles n'étaient exploitables (c-à-d, stockées, traitées ou échangées) que via des mobiles (smartphones). De nos jours, plusieurs fournisseurs présentent des offres pour le stockage et le traitement des données mobiles sur le cloud. Cependant, cette gestion des données sur des plateformes distantes peut se heurter à des obstacles liés aux modes de stockage et accès, ainsi qu'à l'hétérogénéité des environnements MCC. Comme nous allons le constater ci-dessous, la majorité des fournisseurs clouds proposent des solutions centralisées et une minorité d'approches traitent le problème d'interopérabilité [3]au sein des environnements MCC hétérogènes.

2.2.1 Stockage

Le domaine de stockage des données mobiles est principalement monopolisé par les fournisseurs géants du cloud public (par exemple, Google est Amazon). Cependant, ce stockage est centralisé dans des serveurs distants dont les utilisateurs n'ont aucun contrôle. Ceci est totale-

3. l'interopérabilité désigne la capacité d'un système à s'exécuter, communiquer et transférer des données avec d'autres systèmes ou environnements différents.

ment possédé par les prestataires de services qui peuvent affaiblir les politiques de confidentialité pour une raison ou pour une autre. Ces fournisseurs de cloud ont été confrontés à des pressions gouvernementales depuis le monde entier afin de communiquer des renseignements concernant leurs abonnés [45]. Malgré les réserves, les gens ont tendance à utiliser leurs mobiles afin de stocker leurs données personnelles dans le cloud. La table 3.1 présente un aperçu des principaux services cloud publics de stockage des données mobiles [35].

Fournisseur	Stockage gratuit	Max déploiement	Environnement	Partage
Google Drive	5 GB	10 GB	Android, iOS	Oui
Dropbox	2 GB	300 MB	Android, iOS, Blackberry	Oui
SkyDrive	7 GB	2 GB	Windows Phone, iOS	Oui
iCloud	5 GB	25 MB / 250 MB	iOS	Limité
Box	5 GB	25 MB / 1 GB	Android, iOS, Blackberry	Oui

TABLE 3.1 – Principaux services des clouds publics pour le stockage des données mobiles.

D'autre part, les approches offrant des mécanismes décentralisés pour un stockage distribué des données mobiles dans le cloud sont très limitées. *STACEE* [87] propose une solution P2P où plusieurs mobiles (smartphones, tablettes, etc.) peuvent contribuer pour le stockage des données dans le cloud. En outre, ce travail prend en charge la qualité du service (QoS) afin de minimiser la consommation énergétique et satisfaire les besoins des utilisateurs. Les données générées et provenant des différents pairs sont classées en quatre catégories selon leurs contenus : temporaire, intermittente, de sauvegarde et de travail. À chaque catégorie est associé un protocole définissant des règles de stockage, des modes d'accès et des exigences de disponibilité.

Comme le montre la figure 3.6, l'architecture fonctionnelle du système *STACEE* fait abstraction d'un empilement de trois couches :

— Une couche d'interface rassemblant des outils et protocoles pour créer, afficher, modifier et supprimer des données depuis/vers l'espace de stockage dans le cloud.
— Une couche de services assurant la coordination centrale pour une gestion optimale des données. Cette gestion est effectuée via un ensemble d'actions telles que la réplication et la migration des données.
— Une couche d'adaptation qui assure l'autonomie du système et initialise des commandes de contrôle qui peuvent être exploitées par la couche des services.

SmartBox [124] est un système de stockage et partage des données mobiles dans le cloud. Inspiré du système de gestion des fichiers *Google* (GFS) [53] et du système *HADOOP* [2], il est construit à base de nœuds de calcul distribués afin d'offrir une capacité maximale des données stockées. En outre, ce travail est basé sur la notion d'attribut visant à indexer les fichiers pour faciliter l'utilisation via les mobiles. Avec *SmartBox*, chaque mobile est associé à un espace de stockage individuel "d'ombre". Cet espace agit comme un entrepôt de données personnel

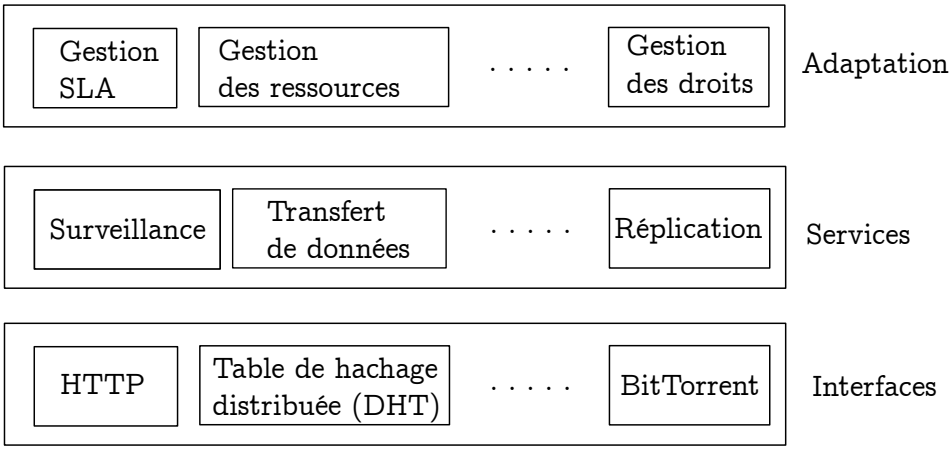


FIGURE 3.6 – Architecture simplifiée du système STACEE [87].

lorsque le mobile est connecté. L'échange des données entre les mobiles est effectué via les différents espaces de stockage d'ombre correspondants. Pour le partage des données au sein d'un groupe d'utilisateurs, *SmartBox* offre un espace de stockage public. D'un point de vue architectural, ce système est composé de trois couches : (i) la couche des ressources requises en termes de nœuds de calcul et supports de stockage (ordinateurs, serveurs, etc.), (ii) le système de gestion des fichiers et (iii) la couche des utilitaires (explorateurs, portails web, etc.).

2.2.2 Hétérogénéité des plateformes MCC

Le domaine du MCC implique une utilisation de plusieurs types de plateformes logicielles et matérielles (serveurs, smartphones, tablettes, équipements de connexion, etc.). Cependant, l'hétérogénéité de ces plateformes peut avoir un impact négatif sur l'interchangeabilité des données ainsi que la capacité d'exécution des programmes au sein de ces environnements hétérogènes. Nous classons cette hétérogénéité en trois catégories : (i) hétérogénéité mobile, liée à la diversité des mobiles et leurs systèmes d'exploitation (par exemple, Android, BlackBerry et iPhone), (ii) hétérogénéité cloud, en relation avec la différence des technologies matérielles et logicielles des différents fournisseurs cloud et (iii) hétérogénéité réseau en termes des différentes interfaces réseaux sans fil existantes.

MABOCCF (en anglais, Mobile Agent Based Open Cloud Computing Federation) [123] est un système qui combine les avantages de l'agent mobile et du cloud computing afin de fournir une interface de fédération offrant des mécanismes d'interopérabilité inter-clouds. Avec ce système, les programmes et les données sont échangés par l'intermédiaire d'agents mobiles où chaque agent est exécuté dans une machine virtuelle appelée "*MAP*" (en anglais, Mobile Agent Place). Les agents mobiles sont capables de se déplacer entre les *MAPs*, comme aussi de communiquer et négocier les uns avec les autres afin d'atteindre les objectifs d'interopérabilité entre des fournisseurs de services cloud hétérogènes. Pour atteindre cet objectif, chaque programme

utilisateur encapsulé à l'intérieur d'un agent mobile est muni d'un ensemble d'informations complémentaires, telles que l'indicateur de mobilité de l'agent et l'ensemble de ressources requises à l'exécution du programme. D'autre part, le travail présenté dans [66] est un système qui étend *MABOCCF* [123] dans le sens où il est basé sur la mobilité d'agents pour la réalisation de l'interopérabilité et la portabilité entre des fournisseurs cloud hétérogènes. Mais à la différence de son prédécesseur, ce système est muni d'une certaine intelligence dans le sens où chaque agent doit se déplacer vers un nouveau cloud dans le cas où il n'arrive pas à trouver la machine virtuelle (c-à-d, *MAP*) adaptée à ses exigences d'exécution. À cet effet, il utilise le protocole "*XMPP*" (en anglais, Extensible Messaging and Presence Protocol) pour la découverte des services offerts par les différents fournisseurs de cloud.

L'hétérogénéité en terme d'interfaces réseau sans fil est considérée comme un autre souci d'interchangeabilité dans le domaine du MCC [42]. Ceci est lié à la différence des technologies mettant en œuvre les points d'accès mobiles tels que *WCDMA*, *GPRS*, *WiMAX*, *CDMA2000*, et *WLAN*. Le travail présenté dans [70] propose un mécanisme intelligent pour l'accès aux différents réseaux mobiles. Ce système est construit sur la base d'un concept d'accès réseau radio intelligent (*IRNA*) [69] qui traite et cache les problèmes et complexités liés à la dynamique et hétérogénéité des réseaux d'accès disponibles. Pour appliquer *IRNA* dans le domaine du MCC, l'architecture de ce système est basée sur la gestion du contexte. Les informations contextuelles manipulées par ce système dépendent des exigences imposées par les environnements MCC en termes de passage à l'échelle, disponibilité et qualité des points d'accès réseaux mobiles.

2.3 Tolérance aux pannes

La tolérance aux pannes est une préoccupation majeure pour garantir la disponibilité des données mobiles ainsi que la fiabilité et la continuité de l'exécution des applications déployées dans le cloud. Cela peut être effectué en utilisant deux directions principales : des solutions logiques telle que la réplication [81] et des solutions matérielles [115] .

Concernant les solutions logiques, nous pouvons citer deux principaux travaux qui sont respectivement basés sur des mécanismes de réplication et de migration. Le travail de *Hyrax* [81] utilise des techniques de réplication afin de permettre la réexécution des tâches sur un nouveau nœud stable après échec d'un autre nœud. Dans [95], le mécanisme de tolérance aux pannes est basé sur une instanciation dynamique des services de communication au niveau des nœuds distribués dans le cloud. Ce mécanisme vise à maintenir un degré élevé de disponibilité des services de communication pour remédier aux difficultés liées aux exigences de mobilité des utilisateurs ainsi qu'aux pannes affectant le fonctionnement des supports de communication. Pour assurer la continuité de la communication entre les différents nœuds, une technique de migration de proxy est utilisée. Elle consiste à créer et démarrer une nouvelle instance du service proxy sur un nouveau nœud après la défaillance du premier nœud hôte.

Pour les solutions matérielles, l'approche *IBIS* [115] consiste à utiliser une grille de calcul

afin de bénéficier de la très grande puissance de calcul résultant des différents nœuds qui la composent. Chaque utilisateur peut déployer des calculs intensifs des données massives depuis son mobile vers la grille *IBIS* afin de surmonter le problème des ressources limitées. La tolérance aux pannes est atteinte grâce à un modèle de suivi des ressources qui implémente des APIs "*JEL*" (Rejoignez, Élire, et Quitter, en anglais : "Join, Elect and Leave"). La malléabilité (en anglais, "Malleability") est le terme utilisé pour décrire un système qui persiste dans son état de fonctionnement suite à l'échec ou la déconnexion de l'un de ces nœuds. En effet, il offre aux applications des mécanismes nécessaires pour leur adaptation à une nouvelle forme de la grille de calcul. Une nouvelle forme de cette grille aura lieu suite à l'occurrence de l'un des événements suivants : (i) arrivée d'un nouveau nœud qui rejoint la grille (reconnexion), (ii) sélection d'un nœud leader pour accomplir la tâche de coordination des calculs au sein de la grille et (iii) départ d'un nœud qui quitte la grille suite à une panne ou une déconnexion. Dans ces cas, des opérations "*JEL*" sont exécutées afin de notifier les applications pour leur adaptation à de nouvelles formes de la grille *IBIS*.

2.4 Conception des mécanismes de déploiement

Dans cette section nous présentons un aperçu sur les architectures et APIs proposées par les travaux de déploiement mobile dans le cloud. Nous nous concentrons sur l'aptitude des différents systèmes proposés d'être réutilisés dans les différents environnements clouds mobiles. En effet, et comme nous l'avons déjà mentionné dans la section 2.2.2, l'hétérogénéité des différentes plateformes impliquées dans le domaine du MCC est un facteur très important et non négligeable. Rappelons que les environnements MCC sont basés sur le fondement du cloud et du mobile computing qui sont eux-mêmes fondés sur des technologies hétérogènes (par exemple, Android, Blackbery et iPhone pour le mobile computing et VirtuelBox, VMWare et Xen pour le cloud).

Le développement d'un système de déploiement efficace doit être basé sur une architecture réutilisable/extensible afin de pouvoir l'implémenter sur ces divers environnements hétérogènes. Une architecture réutilisable/extensible est considérée comme un moule de raffinement pour la construction de différents systèmes de déploiement en vue de leur adaptation à plusieurs environnements.

Comme il est présenté dans la table 3.2, la totalité des travaux de déploiement mobile dans le cloud ne prend pas en considération cet aspect de réutilisation/extensibilité. Ils se sont concentrés sur la présentation d'architectures fonctionnelles sans illustrer comment ces systèmes peuvent être adaptés à des environnements cloud mobile différents. Le système *IBIS* [115] est considéré comme une exception dans le sens où il offre des APIs permettant l'adaptation à plusieurs technologies d'interface réseaux disponibles.

Approche	Architecture fournie	API fournie	Réutilisable
SPECTRA [47]	/	*SPECTRA API*	Non
CHROMA [24]	Architecture fonctionnelle de haut niveau présentant le mécanisme d'élection des tactiques.	Non	Non
HYRAX [81]	Architecture fonctionnelle basée sur la plateforme HADOOP.	*HYRAX API*	Non
CUCKOO [65]	Architecture fonctionnelle du modèle de déploiement.	Non	Non
CLOUDLET [97]	Architecture fonctionnelle de haut niveau présentant le mécanisme de déploiement.	Non	Non
MAUI [36]	Architecture fonctionnelle de haut niveau.	Non	Non
CLONECLOUD [33]	Architecture fonctionnelle présentant les mécanismes de migration et de partitionnement.	Non	Non
STACEE [87]	Architecture en couches du système de stockage.	Non	Non
SMARTBOX [124]	Architecture en couches de la plateforme de stockage.	Non	Non
MABOCCF [123]	Architecture fonctionnelle de haut niveau présentant le mécanisme de fédération cloud.	Non	Non
Khan et al. [66]	Architecture fonctionnelle présentant : - Le mécanisme de fédération cloud. - Diagramme de flux de données. - Un scénario d'interopérabilité.	Non	Non
Klein et al. [70]	Architecture fonctionnelle de haut niveau du gestionnaire du contexte.	Non	Non
Samimi et al. [95]	Architecture générale en couches du modèle.	Non	Non
IBIS [115]	Architecture fonctionnelle du système IBIS	*IBIS API*	*Oui*

TABLE 3.2 – Aperçu des architectures et APIs proposées pour le déploiement mobile dans le cloud.

2.5 Synthèse

Dans cette section, nous avons fait un tour d'horizon sur les principales approches liées au déploiement mobile dans le cloud. Nous avons présenté un résumé des travaux existants dans la littérature selon une classification basée sur quatre critères : (i) mécanismes de déploiement ([47], [24], [81], [62], [65], [97], [36], [33]), (ii) gestion des données mobiles ([87], [124], [123],[66], [70]), (iii) tolérance aux pannes ([81], [95], [115]) et (iv) architectures et APIs proposées par toutes les approches. Dans ce qui suit nous présentons une nouvelle étude comparative

des différentes approches afin de souligner leurs lacunes.

Comme il est illustré dans la table 3.3, cette étude synthétique est basée sur cinq classes de critères, à savoir :

Approche	Surveillance/ Partitionnement			Réseau	Gestion des données		Performance			Conception
	SR	PA	MC	DR	S/R	TF	SC	TP/AU	IN	RE
SPECTRA [47]	✓	✓								
CHROMA [24]	✓	✓								
HYRAX [81]		✓				✓	✓	✓		
Huerta-Canepa et al. [62]	✓	✓				✓	✓			
CUCKOO [65]	✓	✓								
CLOUDLET [97]							✓			
MAUI [36]	✓	✓	✓							
CLONECLOUD [33]	✓	✓	✓							
STACEE [87]						✓				
SMARTBOX [124]						✓				
MABOCCF [123]							✓		✓	
Khan et al. [66]							✓		✓	
Klein et al. [70]							✓		✓	
Samimi et al. [95]							✓	✓		
IBIS [115]							✓	✓		✓

SR : surveillance des ressources — MC : modèle de coût — S/R : sauvegarde restauration
PA : partitionnement — DR : déploiement réseau — TF : transfert de fichiers
RE : réutilisabilité — SC : passage à l'échelle — IN : interopérabilité
TP/AU : tolérance aux pannes / autonomie

TABLE 3.3 – Synthèse des différentes approches de déploiement mobile dans le cloud.

1. Surveillance et partitionnement (SR : surveillance, PA : partitionnement et MC : modèle de coût). La totalité des mécanismes de déploiement ([47], [24], [81], [62], [65], [97], [36], [33]) est basée sur des processus s'exécutant en arrière plan sur les mobiles afin de : (i) surveiller la consommation des ressources (CPU, mémoire et énergie) requises pour l'exécution des applications (ii) partitionner les programmes et (iii) évaluer les coûts attendus par le déploiement des différentes partitions. Cependant, ces processus de surveillance et partitionnement qui sont exécutés sur les mobiles s'avèrent coûteux en terme de consommation des ressources (énergie et trafic réseau) [41].

2. Déploiement des tâches de réseautage (DR). Un trafic réseau important est énormément coûteux en consommation énergétique [55, 84]. Cependant, malgré l'importance d'un tel déploiement, il a été négligé par la totalité des approches étudiées.
3. Gestion des données mobiles (S/R : sauvegarde/restauration et TF : transfert de fichiers). La majorité des approches [87, 124] et fournisseurs de services (*Google Drive*, *DropBox*, *SkyDrive*, *iCloud* et *Box*) qui proposent un gestionnaire de données mobiles dans le cloud, utilisent une architecture centralisée tout en négligeant le déploiement des calculs et tâches de réseautage.
4. Performances (SC : passage à l'échelle, TP/AU : tolérance aux pannes/autonomie et IN : interopérabilité). Seuls les systèmes *HYRAX* [81] et *IBIS* [115] offrent des mécanismes de tolérance au pannes adaptés aux modèles de calcul distribué dans le cloud. Le travail présenté dans [95] offre un mécanisme assurant la continuité des communications entre les différents nœuds de calcul dans le cloud. Malgré que ces systèmes passent à l'échelle, leurs mécanismes de tolérance à la panne sont basés sur une réplication et une réinstanciation continue des ressources et des tâches dans le cloud ; ceci peut être coûteux en terme de consommation de ressources dans le cloud. D'autre part, le système *MABOCCF* [123] et son successeur [66] offrent un mécanisme basé sur les agents mobiles pour la fédération des clouds hétérogènes. Ces systèmes nécessitent une extension afin d'être adaptés au déploiement mobile dans le cloud. En outre, le système présenté dans [70] s'est limité au traitement de l'hétérogénéité des interfaces réseau.
5. Conception (RE : réutilisabilité). À l'exception du système *IBIS* [115] qui s'est limité à offrir des APIs permettant son adaptation à plusieurs types d'interfaces réseaux, la fourniture d'architectures ou APIs réutilisables a été négligée par toutes les approches étudiées.

Nos contributions pour le déploiement mobile dans le cloud. En comparaison avec les approches étudiées du MCC, nos contributions consistent à offrir des patrons de cloud génériques pour la conception des middlewares de déploiement mobile selon les différents environnements de cloud existants. Notre protocole assure à la fois un déploiement à-la-demande des données, tâches et réseautage mobiles sans recourir aux mécanismes de surveillance des ressources et partitionnement des applications qui s'avèrent coûteux en consommation des ressources mobiles. La combinaison des mécanismes de clonage, réseaux privés virtuels, sauvegarde/restauration, gestion des fichiers, auto-tolérance aux pannes ainsi que l'utilisation des services web offrent un degré de réutilisabilité, passage à l'échelle, interopérabilité et disponibilité très élevé.

3 Applications collaboratives

Dans cette section nous allons explorer les travaux de recherches en relation avec notre deuxième axe de recherche, à savoir les applications collaboratives. Nous présentons un aperçu

sur ces travaux selon deux directions : (i) les architectures proposées pour concevoir des applications collaboratives et (ii) les systèmes d'édition collaborative dans le cloud.

3.1 Conception des applications collaboratives

Concernant la conception des applications collaboratives, nous nous intéressons exclusivement aux architectures basées sur les patrons de conception ainsi qu'à l'aspect de réutilisation. Le domaine lié au partage des données mobiles a connu plusieurs travaux de recherche, mais peu de travaux ont présenté un modèle réutilisable pour la conception des applications mobiles collaboratives. De plus, à notre connaissance il n'existe pas de travail qui a proposé un modèle réutilisable spécialement conçu pour le MCC.

Dans [83] un modèle architectural général a été introduit pour aider les développeurs à concevoir des applications et services mobiles. L'objectif de ce modèle est de faire abstraction de l'ensemble des composants de haut niveau des applications mobiles tout en offrant des patrons de conception afin de raffiner ces composants. Ainsi, pour chaque composant, le modèle proposé peut suggérer les patrons qui dépendent de lui durant le processus de raffinement. En se basant sur ces suggestions de patrons, le concepteur peut choisir le modèle raffiné le plus adapté à son application.

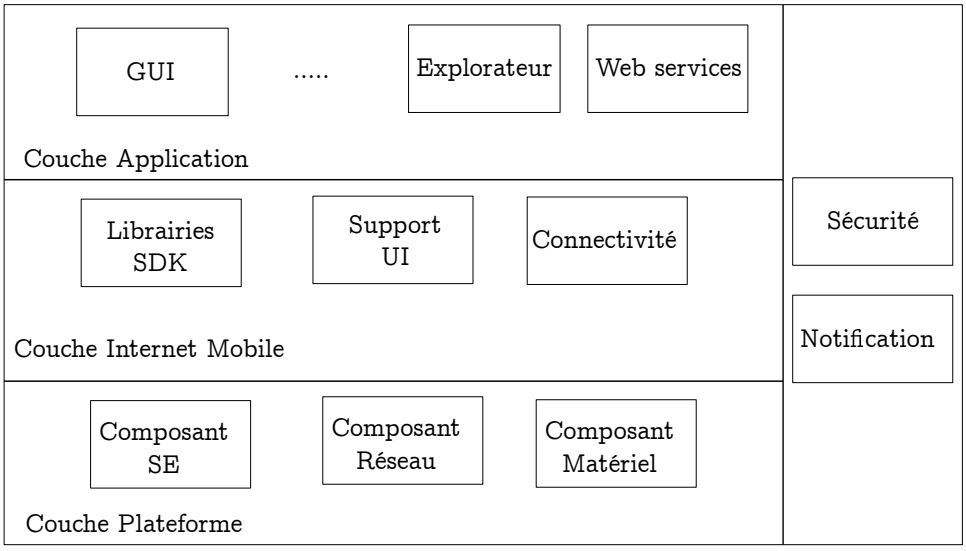


FIGURE 3.7 – Architecture d'un modèle général des applications mobiles [83].

Comme le montre la figure 3.7, le modèle général proposé suit une architecture en couches. Chaque couche est composée d'un certain nombre de composants possédant le même niveau d'abstraction. Un composant peut représenter un élément logiciel ou matériel qui a pour mission d'exécuter certaines fonctions spécifiques. En général, les composants d'une couche utilisent des fonctionnalités fournies par le niveau inférieur et d'autres fournies au sein de la même

couche. Un composant peut ainsi servir d'intermédiaire afin de réémettre des fonctionnalités offertes par un niveau inférieur à des niveaux supérieurs. Ce modèle général est composé de trois couches : (i) une couche matérielle comportant les composants du système d'exploitation, pilotes réseau et matériels, (ii) une couche Internet Mobile contenant des librairies et supports permettant d'accéder aux fonctionnalités du système d'exploitation et de gérer la connectivité réseau via les différents protocoles internet, et (iii) une couche application fournissant des interfaces d'accès pour les utilisateurs.

Le travail présenté dans [38] fournit une architecture générique qui définit un espace de conception des architectures d'applications collaboratives qui peuvent différer selon leur façon de répondre à plusieurs questions importantes, à savoir :

— Concurrence : quels sont les composants de l'application pouvant s'exécuter simultanément ?
— Distribution : lequel de ces composants nécessite une exécution sur plusieurs nœuds de calcul ?
— Gestion des versions/réplication : lequel de ces composants doit être répliqué ?
— Sensibilisation de collaboration : lequel de ces composants est sensible à la collaboration ?

Pour répondre à ces préoccupations de conception, l'espace des architectures proposées est basé sur d'autres approches de la littérature. La conception de telles architectures est essentiellement basée sur une phase préliminaire consistant à définir les différents composants d'une application collaborative donnée. En l'occurrence, deux types de décomposition de ces applications sont pris en considération : (i) par unités de calcul où l'application est divisée en un ou plusieurs modules de communication. Un module peut être composé d'un ou plusieurs niveaux et chaque niveau peut être composé d'une ou plusieurs couches répliquées et (ii) par unités de concurrence qui suppose qu'une application peut être décomposée en un ou plusieurs processus distribués, où chaque processus peut déclencher un ou plusieurs threads simultanés.

Le travail présenté dans [92] offre une structure abstraite rassemblant l'essentiel des couches et composants pour la conception des espaces de travail mobiles partagés (en anglais, *MSW* : Mobile Shared Workspace). Il montre comment une telle structure peut être instanciée pour obtenir des espaces de travail particuliers. La solution proposée requiert deux éléments de base pour supporter le travail collaboratif mobile : un mobile et un espace de travail partagé. Comme le montre la figure 3.8, un *MSW* est un système collaboratif mobile composé de trois principaux éléments : (i) un espace de travail, contenant l'ensemble des données et services qui peuvent être manipulés en mode mono-utilisateur comme en collaboration, (ii) des services de communication qui offrent des fonctionnalités de connectivité et réseautage inter-mobiles et (iii) des services de coordination ayant pour rôle de manipuler ou distribuer des ressources partagées (par exemple, transfert de fichiers et synchronisation des données).

Une architecture réutilisable qui peut être appliquée dans le développement de plusieurs systèmes de collaboration mobiles a été présentée dans [88]. En se basant sur une étude anté-

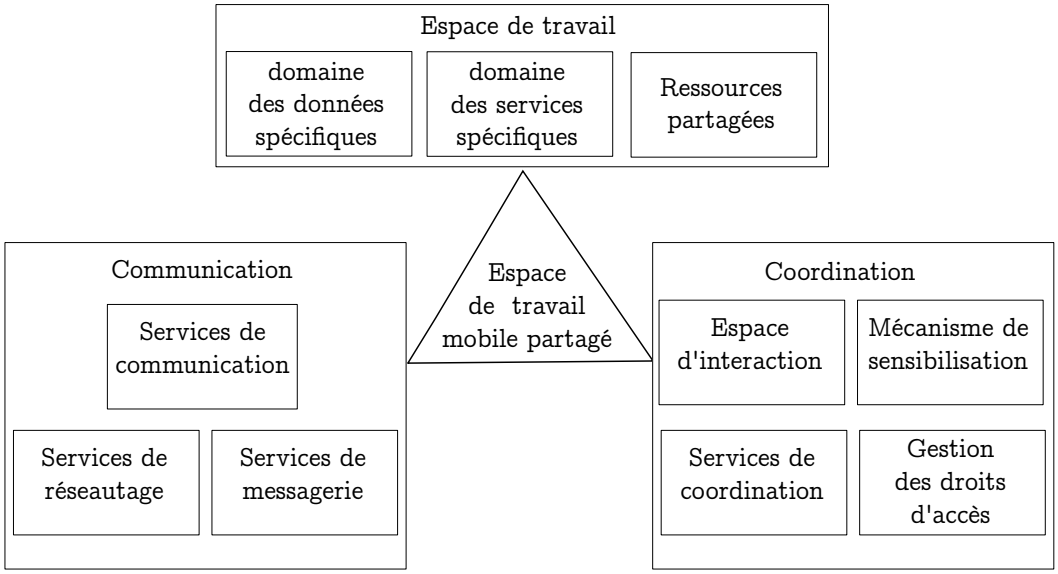


FIGURE 3.8 – Architecture abstraite d'un espace de travail mobile partagé [92].

rieure autour des problématiques et expériences liées aux "Groupwares" [44], cette architecture distribuée est composée de trois couches (voir la figure 3.9) :

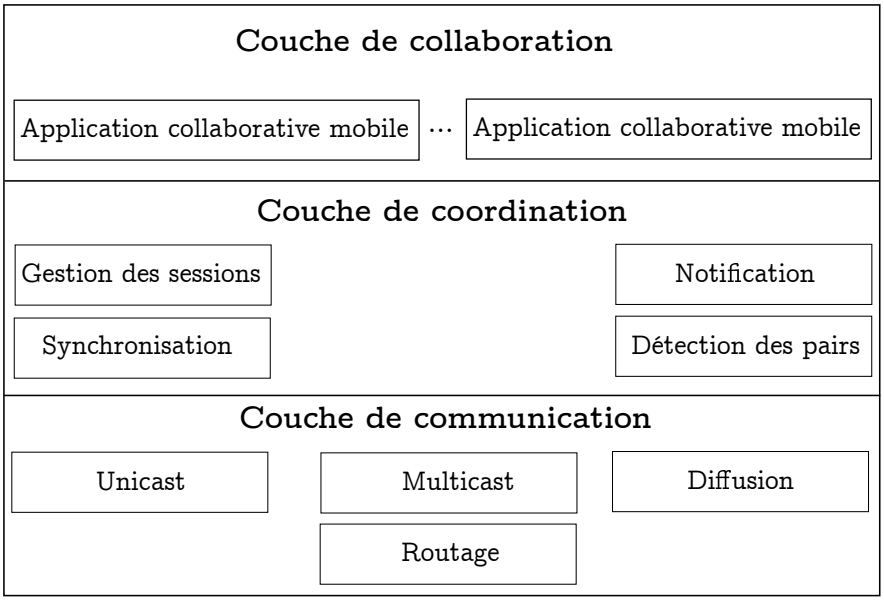


FIGURE 3.9 – Architecture en couches pour supporter la collaboration mobile [88].

— Une couche de collaboration qui contient les applications collaboratives des utilisateurs.
— Une couche de coordination qui fournit des solutions liées à la coordination des travaux de collaboration (par exemple, la gestion des sessions et la synchronisation des données partagées).
— Une couche de communication, chargée de fournir des modules servant aux échanges

des messages entre les mobiles. Ce composant permet à un utilisateur d'envoyer un message à d'autres utilisateurs de différentes manières : à ceux qui sont liés à une même session (*multicast*), pour un groupe d'utilisateurs (diffusion) ou à un utilisateur unique (*unicast*).

Cette architecture a été réutilisée afin de concevoir une plateforme nommée *SOMU* [89] par implémentation de ses services de communication et de coordination.

Une plateforme composée d'éléments contextuels qui peuvent être utilisés pour aider les développeurs au cours des différentes phases de conception d'applications collaboratives mobiles a été présentée dans [19]. Cette plateforme fait abstraction de trois niveaux d'informations contextuelles qui peuvent être respectivement liés à trois phases nommées : conception, analyse et conception architecturale (en anglais, conception, analysis and architectural design phases). En l'occurrence, l'utilisation de cette plateforme permet de cadrer une application via un ensemble d'informations contextuelles tout en identifiant les besoins non-fonctionnels et les restrictions de conception.

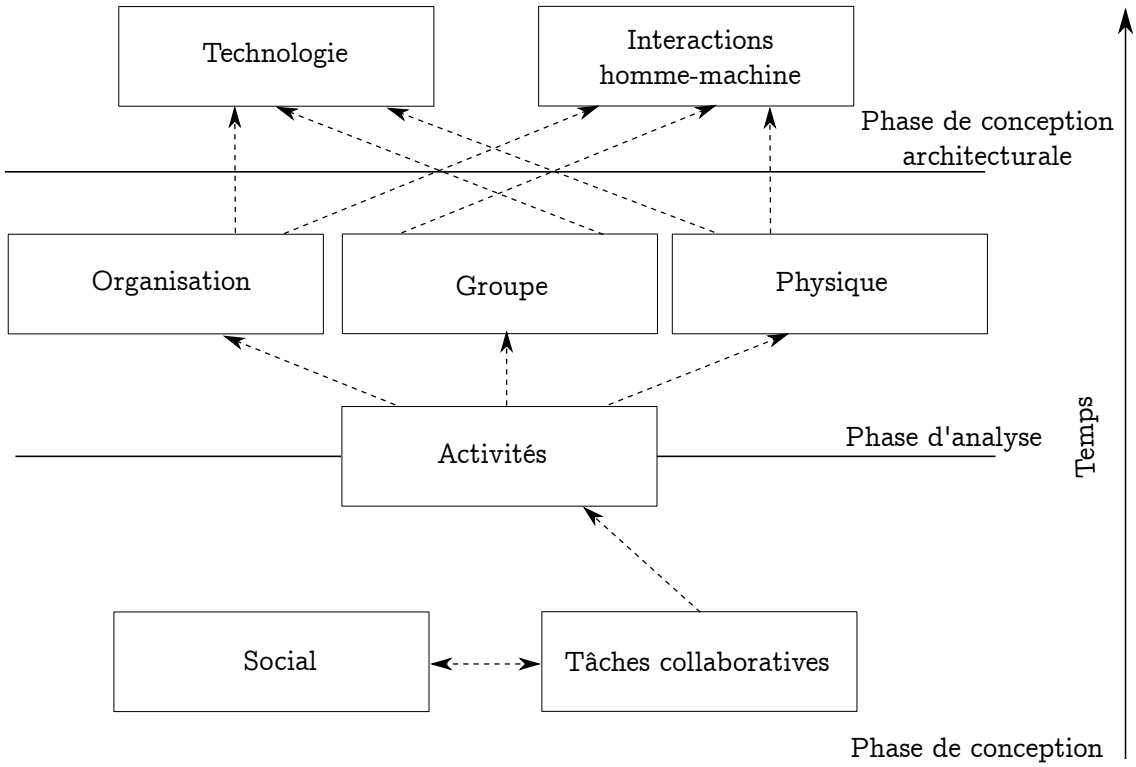


FIGURE 3.10 – Une architecture basée sur le contexte pour les applications collaboratives mobiles [19].

Comme le montre la figure 3.10, cette plateforme permet de classer les éléments du contexte à travers ses trois phases. La phase de conception contient des informations contextuelles qui sont liées aux tâches de collaboration ou possédant un aspect social (la couche la plus basse dans la figure 3.10). Les informations contextuelles de la phase de conception aident les dé-

veloppeurs à comprendre les exigences du système d'une manière générale. Dans un niveau plus élevé, l'objectif de la phase d'analyse est de poursuivre la compréhension du problème afin d'identifier les restrictions de conception et les besoins non-fonctionnelles en matière de contexte lié à l'organisation, le physique et les groupes. En remontant vers la dernière couche, l'objectif de cette phase est de créer un modèle architectural final de l'application en tenant compte des besoins non-fonctionnels et des restrictions de conception identifiés dans les phases précédentes.

3.2 Éditeurs collaboratifs pour le cloud

Dans cette section nous présentons un aperçu sur les éditeurs de collaboration en temps réel opérant dans des environnements cloud.

SPORC. *SPORC* [45] est un système collaboratif permettant à plusieurs utilisateurs de modifier des documents partagés par réplication. Afin de maintenir la cohérence de ces documents partagés, *SPORC* implémente l'approche des transformées opérationnelles (OT) et utilise un serveur central d'ordonnancement, ayant pour rôle de maintenir un ordre global aux opérations de mise-à-jour concurrentes.

Le système *SPORC* est composé de quatre états :

1. Un *état local* représentant la copie locale d'un document manipulé par un client.
2. Un *log serveur* contenant l'ensemble des opérations stockées et commandées par le serveur.
3. Un *log client* contenant l'ensemble de toutes les opérations appliquées et échangées par les clients.
4. Une *file d'attente* contenant la liste des opérations locales qui ont été déjà appliquées à l'état local d'un client, mais qui doivent être validées (étiquetées par le serveur).

La figure 3.11 illustre le principe de fonctionnement du système *SPORC* qui peut être résumé comme suit : (1) après qu'un client génère une nouvelle opération, il l'applique d'abord à son état local et l'ajoute à la fin de sa file d'attente. (2) Cette opération sera envoyée au serveur d'ordonnancement qui lui attribuera une estampille (ou un numéro) suivant un ordre global, l'insèrera à la fin de son log et (3) la diffusera à tous les clients. Un client qui reçoit une opération qu'il a initialement créée, doit (4) l'extraire à partir de la file d'attente et (5) l'insérer dans son log. Sinon si un client n'est pas à l'origine de cette opération distante reçue, il l'insèrera dans son log et (6) l'appliquera à sa copie locale du document partagé après transformation.

CloneDoc. *CloneDoc* [72] est un système de collaboration destiné aux mobiles. Une partie des tâches de collaboration est migrée vers une plateforme virtuelle composée de clones des mobiles dans le cloud. Inspiré du protocole de synchronisation *SPORC* [45], *CloneDoc* est aussi basé sur l'approche OT et un serveur central d'ordonnancement pour le maintien de la cohérence des documents partagés.

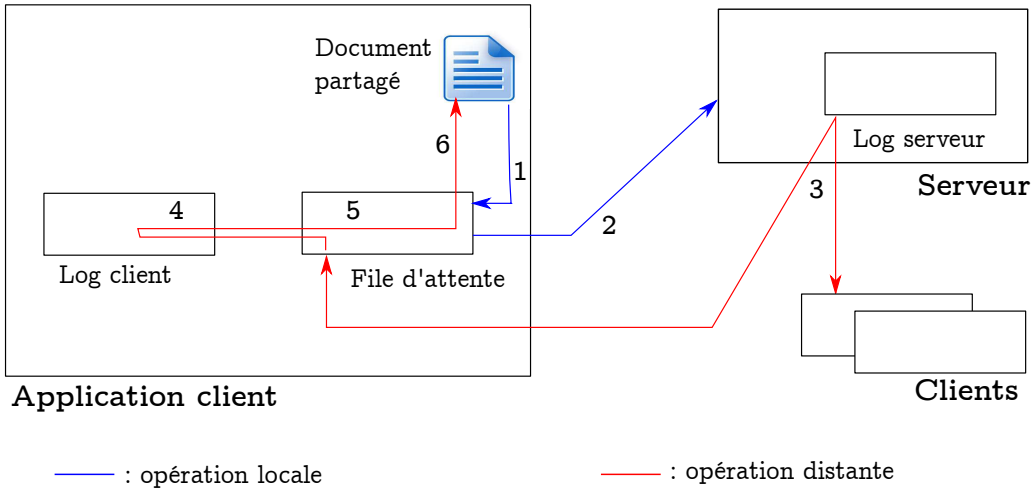


FIGURE 3.11 – Principe de fonctionnement du système SPORC [45].

La figure 3.12 illustre le principe de fonctionnement du système *CloneDoc* selon l'ordre suivant : (1) après qu'un mobile génère et applique une opération locale, (2) il l'enverra à son clone dans le cloud qui l'appliquera directement sur sa copie du document partagé ; (3) une fois cette opération appliquée, elle sera envoyée au serveur central d'ordonnancement ; (4) ce serveur attribuera une estampille à l'opération reçue et la diffusera vers les autres clones. (5) Après avoir reçu une opération distante, un clone doit la transformer contre les opérations de son log avant de l'appliquer sur sa copie du document partagé ; (6) une fois l'opération appliquée et insérée au log, elle sera envoyée au mobile qui doit la transformer encore une fois avant de l'appliquer sur son état local.

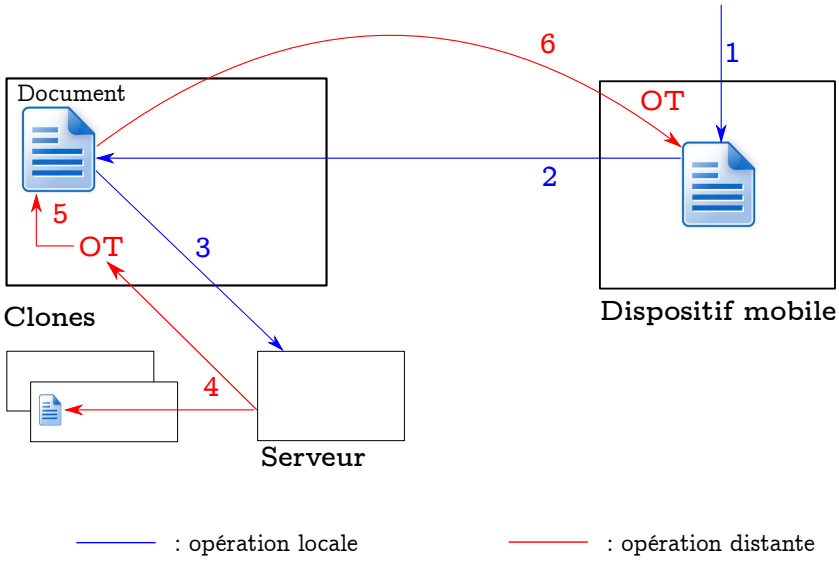


FIGURE 3.12 – Principe de fonctionnement du système CloneDoc.

Il est évident que *CloneDoc* traite le décalage d'exécution des opérations distantes entre

le mobile et son clone via des transformations opérationnelles supplémentaires. En l'occurrence, les tâches de collaboration sont distribuées, voire même redondantes, entre le mobile et son clone, puisque la même opération distante sera transformée deux fois, ce qui génère une consommation d'énergie au niveau du mobile.

Hermes. *Hermes* [119] est un système de collaboration permettant de gérer l'interopérabilité (d'une manière transparente) des services d'édition collaborative hétérogènes dans le cloud. Les utilisateurs ont la possibilité d'utiliser leurs services d'édition de cloud favoris afin de participer à une collaboration inter-clouds et *Hermes* se chargera de leur interopérabilité. Le mécanisme de synchronisation de ce système est basé sur l'approche OT pour résoudre les conflits et maintenir la cohérence des documents partagés.

Hermes est basé sur des "utilisateurs robots" pour l'automatisation de la synchronisation entre les services d'édition hétérogènes. Ainsi, un utilisateur n'a besoin de partager un document qu'avec un utilisateur robot qui prendra en charge la collaboration avec d'autres robots. Chaque utilisateur robot est composé de trois éléments de base : (i) une interface utilisateur, utilisée par un robot pour participer à une collaboration via le service d'édition de l'utilisateur, (ii) un adaptateur, qui applique des mises-à-jour reçues d'autres robots et synchronise l'état de son document local avec un serveur de synchronisation et (iii) un moteur de collaboration, qui est responsable de la détection des changements effectués sur les documents et de la synchronisation avec d'autres utilisateurs robots.

rbTree-Doc. *rbTree-Doc* [120] est un service d'édition cloud qui permet à plusieurs utilisateurs d'éditer des documents partagés en ligne tout en prenant en considération des aspects de sécurité tels que la confidentialité des données partagées. Le document est représenté comme un arbre bicolore [26]. Avec ce service, un utilisateur n'a besoin que de télécharger une partie du document concernée par une mise-à-jour. Ainsi, *rbTree-Doc* permet de réduire la taille des données devant être chiffrées en analysant seulement le contenu du sous-arbre qui a été mis à jour.

Comme le montre la figure 3.13, le système *rbTree-Doc* est principalement composé d'un serveur distant dans le cloud et un ensemble de clients. Le rôle du serveur consiste à authentifier l'identité des clients, gérer les documents (partage et stockage), contrôler les droit d'accès et diffuser les mises-à-jour. Quant aux clients, ils sont capables de créer et manipuler des vues sur les documents partagés (c-à-d, des arbres bicolores). Après avoir été chiffré, chaque arbre bicolore modifié sera envoyé au serveur qui le diffusera aux autres clients. Ces derniers procèdent au déchiffrement et reconstitution de cet arbre reçu.

3.3 Synthèse

Dans cette section nous avons donné un aperçu sur les travaux de la littérature liés aux applications collaboratives mobiles. Nous les avons présentés selon les architectures de conception proposées dans la littérature [83, 38, 92, 88, 19] ainsi que celles utilisées pour l'édition collabo-

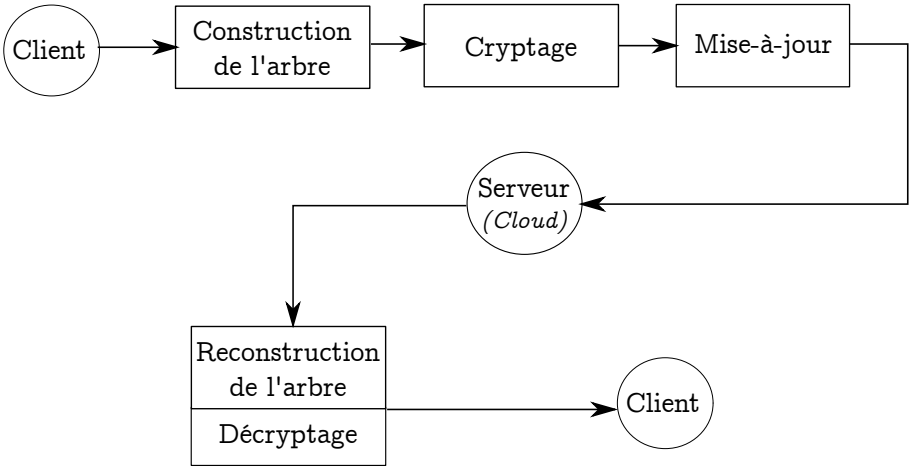


FIGURE 3.13 – Architecture du système rbTree-Doc [120].

rative moyennant le cloud [45, 72, 119, 120]. Dans ce qui suit nous présentons une autre étude comparative des différents systèmes selon de nouveaux critères afin de tirer leurs points faibles par rapport au contexte de conception et de développement des applications collaboratives mobiles. Comme le montre la table 3.4, cette étude synthétique est basée sur les trois catégories de critères suivantes :

Approches	Architecture				Collaboration					Performance	
	C/S	PP	RE	MCC	SD	SC	EM	DS	DR	PE	TP
SPORC [45]	✓					✓				✓	
CloneDoc [72]	✓			✓		✓	✓	Faible	moyen	✓	
Hermes [119]	✓					✓				✓	
rbTree-Doc [120]	✓					✓					
Mazhelis et al. [83]			✓								
Dewan et al. [38]	✓	✓	✓				✓				
Rodríguez-Covili et al. [92]		✓	✓		✓		✓				
Neyem et al. [88]		✓	✓		✓		✓			moyen	
Alarcon et al. [19]		✓	✓								

C/S : Client/Serveur — SD : Synchronisation distribuée — DR : Déploiement Réseautage
PP : Pair-à-Pair — SC : Synchronisation Centralisé — PE : Passage à l'échelle
RE : Réutilisabilité — EM : Environnement Mobile — TP : Tolérance aux Pannes
MCC : Mobile Cloud Computing — DS : Déploiement Synchronisation

TABLE 3.4 – Synthèse des différentes approches liées aux applications collaboratives mobiles.

1. Architecture (C/S : Client/Serveur, PP : Pair-à-Pair, RE : Réutilisabilité et MCC : Mobile

Cloud Computing). Bien que la totalité des travaux d'architectures proposés [83, 38, 92, 88, 19] offre l'avantage d'être réutilisés, ils ne considèrent pas la contrainte de limitation des ressources mobiles et par conséquent, ils ne sont pas adaptés aux environnements MCC.

2. Collaboration (SD : Synchronisation distribuée, SC : Synchronisation Centralisée, EM : Environnement Mobile, DT : Déploiement des Tâches de Synchronisation et DR : Déploiement Réseautage). Les mécanismes de synchronisation des éditeurs collaboratifs [45, 72, 119, 120] sont basés sur un serveur central. Ce serveur peut être responsable de la gestion d'un ensemble de tâches clés, telles que l'ordonnancement des opérations d'édition, le contrôle des droits d'accès et la gestion des utilisateurs (joindre/quitter un groupe et authentification). Cependant, une panne d'un tel point central de synchronisation peut avoir des conséquences indésirables sur la continuité du travail collaboratif. En outre, il faut aussi prévoir une stratégie certaine en matière de sécurité afin d'éviter toute éventuelle falsification des messages échangés entre les collaborateurs. Un serveur malveillant peut causer l'incohérence des différentes vues d'une ressource partagée.

   D'un autre côté, à l'exception du système *CloneDoc* [72], la majorité des éditeurs collaboratifs ne sont pas adaptés aux environnements mobiles. Cependant, malgré la prise en considération de la contrainte de limitation des ressources mobiles par *CloneDoc*, ce système reste coûteux en terme de consommation énergétique et trafic réseau [55, 84] durant l'exécution des tâches de collaboration.

3. Performance (PE : Passage à l'échelle et TP : Tolérance aux Pannes). Malgré son importance, la tolérance aux pannes a été négligée par toutes les approches étudiées.

Nos contributions pour la conception des applications collaboratives. À la différence des approches existantes, nous proposons un modèle générique de collaboration mobile dans le cloud. Ce niveau est aussi conçu à base des patrons de conception afin de concevoir des applications collaboratives mobiles sous contraintes de limitation des ressources mobiles (durée de vie des batteries, capacités de calcul et de stockage et instabilité des réseaux mobiles). Nous proposons cette architecture réutilisable pour aider les développeurs à implémenter les différents protocoles de synchronisation décentralisée existants (par exemple l'approche OT). En utilisant les services offerts par notre premier niveau de déploiement, ce protocole de collaboration assure un important gain en matière d'énergie, trafic réseau et temps de réponse pour les mobiles. La détection des différentes situations de pannes ainsi que la restauration des applications sont effectuées d'une manière transparente sans affecter le travail de collaboration. Quant au passage à l'échelle, nous utilisons une technique simple basée sur des dépendances directes entre les différentes opérations de mise-à-jour appliquées qui préserve la dynamité des utilisateurs et groupes de collaboration.

4 Conclusion

Ce chapitre a été consacré à la présentation d'un état de l'art sur les travaux de recherche réalisés autour du déploiement mobile et des applications collaboratives dans le cloud.

D'une part, nous avons exposé les travaux de déploiement existants selon une classification basée sur un ensemble de questions clés. L'analyse de ces travaux nous a conduit à déterminer les lacunes suivantes : (i) utilisation de mécanismes de surveillance et de partitionnement qui sont eux-mêmes coûteux en terme de consommation de ressources mobiles, (ii) manque de déploiement des tâches de réseautage, (iii) manque de mécanismes de tolérance aux pannes et (iv) la non prise en considération de l'hétérogénéité des environnements MCC.

D'autre part, notre étude menée sur les travaux de collaboration dans le cloud a été principalement axée sur les architectures et éditeurs collaboratifs proposés. La synthèse de ces travaux a permis de déduire les manques suivants : (i) inexistence d'une architecture réutilisable spécialement dédiée au domaine MCC et (ii) absence d'un mécanisme de tolérance aux pannes qui peuvent être provoquées par la centralisation de la synchronisation des éditeurs collaboratifs.

Dans nos contributions nous apportons des solutions à l'ensemble des problèmes détectés au niveau des approches existantes. Dans le chapitre suivant, nous allons présenter une architecture réutilisable pour la conception des applications collaboratives et mobiles dans le cloud.

Deuxième partie

Contributions

Chapitre 4

Des “patrons de cloud” pour les applications collaboratives mobiles

Sommaire

1 Introduction

La définition des modèles de conception est plus difficile dans le cas des nouvelles applications visant à mettre œuvre des services de collaboration mobiles basés sur des solutions cloud. En effet, la combinaison des environnements mobile et cloud soulève de nombreuses questions de conception telles que la gestion des travaux de collaboration en temps réel (par exemple, des tâches d'édition collaborative). Ces applications collaboratives mobiles impliquent de nombreux utilisateurs mobiles pour manipuler des données partagées. Donc, ils ont besoin des modèles de conception appropriés pour la modélisation des services de communication et de synchronisation en vue de leur exploitation par plusieurs mobiles en migrant la majorité des tâches intensives sur le cloud. En outre, d'autres processus tels que l'installation, le déploiement, la configuration, la surveillance et la gestion des modules logiciels doivent être bien modélisés afin de fournir un service collaboratif complet et cohérent au profit des utilisateurs mobiles dans le cloud.

Ce chapitre propose un modèle de conception réutilisable d'applications collaboratives mobiles dans le cloud. Il vise à fournir des mécanismes de synchronisation purement décentralisés afin de préserver la cohérence des ressources partagées en tenant compte des contraintes des applications mobiles, à savoir la courte durée de vie de la batterie et l'instabilité des connexions réseau. En conséquence, des solutions de conception seront illustrées pour modéliser des services de clonage et de collaboration dans le cloud. Les patrons de conception proposés concernent deux niveaux : le premier niveau offre un auto-mécanisme de contrôle pour créer des clones de mobiles, gérer des groupes d'utilisateurs et rétablir les clones en panne. Le deuxième niveau présente des mécanismes de collaboration en temps réel.

Le présent chapitre est structuré en quatre sections. Nous présentons à la section 2 notre modèle de collaboration. La section 3 énumère les exigences requises pour la conception des applications collaboratives mobiles. La section 4 décrit deux modèles de conception réutilisables pour gérer le clonage des mobiles et assurer la collaboration entre ces dispositifs. Enfin, la section 5 conclut ce chapitre.

2 Modèle de collaboration

Un modèle de collaboration est présenté pour permettre à plusieurs utilisateurs géographiquement dispersés de collaborer pour atteindre un but commun. L'adoption des mobiles pour une telle collaboration repose sur des facteurs essentiels tels que la capacité des traitements, la durée de vie de la batterie et la couverture du réseau. Étendre cette collaboration dans l'espace et dans le temps butera inévitablement aux limitations de ces facteurs. Pour surmonter ces limitations, la plupart des tâches de calcul des mobiles ainsi que la communication entre ces dispositifs sont déléguées au cloud. En effet, la virtualisation étend les ressources en déchargeant les mobiles de l'exécution des tâches coûteuses. Ces tâches seront prises en charge par

les clones dans le cloud. Le cloud permet aux utilisateurs de construire des réseaux virtuels P2P où un mobile peut être connecté (par l'intermédiaire de son clone) en permanence à d'autres mobiles pour réaliser des tâches communes.

Quant au partage des ressources, chaque utilisateur possède deux copies des données partagées où la première copie est stockée dans le mobile tandis que la seconde est sur son clone. L'utilisateur modifie la copie mobile, puis envoie les modifications locales au clone de son mobile afin de mettre à jour la deuxième copie et propager ces modifications à d'autres clones (c-à-d d'autres mobiles). Par ailleurs, il faut noter qu'un utilisateur peut travailler même en cas de déconnexion (en mode hors ligne) au moyen de la copie du mobile.

3 Exigences de conception

La phase de conception d'une application doit tenir compte des exigences fonctionnelles et non fonctionnelles telles que la flexibilité, la performance, l'interopérabilité et la sécurité afin d'accroître l'utilité de l'application [34]. Le contexte des applications collaboratives mobiles ne constitue pas une exception et doit se conformer à cette règle de conception dans sa forme générale. Mais compte tenu des particularités liées aux applications mobiles, d'autre exigences sont nécessaires [59] : la flexibilité des interactions des utilisateurs (ou l'évolutivité), la protection de telles interactions, la communication, l'hétérogénéité et l'interopérabilité, la sensibilisation des utilisateurs, et la cohérence et disponibilité des données. Nous avons prolongé cette liste par une exigence importante, à savoir la restauration après pannes. Ceci concerne l'aptitude de l'application collaborative mobile à supporter des mécanismes de tolérance aux défaillances afin de détecter/réparer automatiquement différentes situations de pannes.

Dans ce qui suit, nous listons les exigences liées à la conception d'applications collaboratives mobiles destinées aux environnements MCC :

Autonomie. Le système conçu doit être doté de mécanismes autonomes assurant la continuité et le bon fonctionnement des tâches collaboratives déployées sur le cloud, telles que l'intégration et la configuration des machines virtuelles, la gestion des réseaux virtuels, la synchronisation distante et la restauration après panne.

Disponibilité des données. L'application doit être réactive dans les travaux en ligne et hors ligne de telle manière que les données partagées doivent être disponibles à tout instant. Même si l'utilisateur perd son mobile, il doit pouvoir récupérer des données à partir du cloud. Pour ce faire, une réplication explicite des données entre le mobile et son clone sera utilisée.

Cohérence des données. Les utilisateurs doivent être en mesure de modifier librement et concurremment n'importe quelle partie des données partagées à n'importe quel moment (en ligne et hors ligne). Ces modifications vont potentiellement générer des vues incohérentes sur les données partagées. Plus précisément, cette incohérence est causée par les mises-à-jour concurrentes des interactions clone-clone et mobile-clone. Par conséquent les applications collaboratives mo-

biles doivent prévoir des mécanismes de synchronisation légers et décentralisés afin de préserver la cohérence des données. La synchronisation doit être effectuée principalement au niveau du cloud de telle manière que la collaboration va diminuer les dépenses d'énergie sur le mobile. Il est à mentionner aussi que la décentralisation de la synchronisation permettra aux utilisateurs d'éviter le problème du point unique de défaillance.

Communication. La communication entre les utilisateurs mobiles doit être disponible et permanente. En effet, elle constitue le support de toute forme de collaboration (par exemple l'échange de plusieurs types de données pour effectuer une tâche commune) et de synchronisation afin de maintenir la cohérence des données. Il est bien connu que la communication à travers des mobiles souffre des déconnexions fréquentes. Pour remédier à cette situation, le modèle proposé de collaboration basé sur le cloud permet à chaque utilisateur mobile de communiquer directement avec le clone qui est toujours disponible dans le cloud. Ainsi, n'importe quel utilisateur est accessible via ce clone à tout moment.

Évolutivité. Les applications mobiles doivent être adaptées à des environnements dynamiques où la taille du groupe de collaboration pourrait changer fréquemment. Typiquement, des événements tels que la création d'un nouveau groupe ou rejoindre/quitter un groupe ne doivent pas affecter la disponibilité et l'intégrité des données partagées. Ainsi, d'une part, les utilisateurs doivent être en mesure de participer à n'importe quelle session de travail collaboratif. D'autre part, la gestion des groupes d'utilisateurs doit être effectuée de manière transparente et cohérente, et surtout sans aucun coût de traitement supplémentaire pour les mobiles. Notre modèle de collaboration basé sur le cloud est bien adapté pour développer des applications collaboratives mobiles extrêmement évolutives.

Sensibilisation des utilisateurs. Chaque clone doit être capable de détecter en temps réel et de partager avec d'autres clones (membres du même groupe) des informations relatives à l'état de connexion et de disponibilité des utilisateurs mobiles. Afin de supporter cette prise de sensibilisation, nous utilisons des mécanismes identiques à ceux utilisés par [58], à savoir la joignabilité de l'utilisateur (c-à-d connecté/déconnecté) et la disponibilité de l'utilisateur (par exemple disponible/occupé). Ces moyens de sensibilisation permettent à chaque utilisateur d'être informé par l'intermédiaire du clone sur la joignabilité/disponibilité des autres utilisateurs en temps réel.

Hétérogénéité. Le type du dispositif (ou système d'exploitation) utilisé par les utilisateurs mobiles ne doit pas être un obstacle pour effectuer la collaboration à-la-demande entre eux. La conception proposée offre une architecture de référence pour la mise en œuvre d'un middleware de déploiement basé sur les services Web *WSDL* (Langage de Description des Services Web) afin de cloner des mobiles et gérer leurs réseaux ad-hoc dans le cloud. Cette interface permet de créer des clones avec un système d'exploitation unique (c-à-d, Android). Cela permettra d'éliminer le problème lié à l'hétérogénéité des mobiles pour la communication inter-clones.

D'autre part, la communication entre les clones et leurs mobiles qui sont munis de systèmes d'exploitation autres qu'Android est basée sur des échanges *SOAP* standard (Simple Object Access Protocol) à travers des services web fournis par le middleware de déploiement. Il convient

également de noter que notre middleware de déploiement ne réalise pas des clones complets pour tous les mobiles. Le système créera seulement une machine virtuelle Android équipée d'un protocole de collaboration sans possibilité de sauvegarde/restauration des données et des applications depuis/vers des mobiles non-Android.

Restauration après panne. Les utilisateurs doivent récupérer facilement toutes les données partagées quand un incident technique (par exemple, panne, vol ou perte du mobile, ou défaillance du clone) se produit et continuer la collaboration de manière transparente. Pour résoudre ce problème, un gestionnaire du cycle de vie des clones est nécessaire. Ce gestionnaire permet de détecter les clones défaillants et restaurer leurs états sans affecter les tâches collaboratives en cours.

4 Patrons de conception

Cette section présente des patrons de conception que nous proposons pour les applications collaboratives mobiles dans le cloud. La figure 4.1 illustre l'architecture globale du système qui se compose de trois niveaux :

1. La couche d'application fournit des interfaces utilisateur graphiques (GUI) pour interagir avec les deux couches restantes. L'interaction avec la couche de clonage (le middleware de déploiement) permet le traitement des requêtes utilisateurs afin de lancer des processus de clonage des mobiles et de gestion des groupes de collaboration dans le cloud, tandis que l'interaction avec la couche de collaboration permet de synchroniser le mobile et son clone.

2. La couche de déploiement fournit des services pour : (a) la création des clones et (b) la gestion des groupes d'utilisateurs à travers la création et la manipulation des réseaux privés virtuels (VPN) dans le cloud.

3. La couche de collaboration supporte des mécanismes de synchronisation pour manipuler des données partagées par plusieurs utilisateurs dans le cloud. Elle doit assurer (en temps réel) la cohérence des données qui sont partagées par une réplication explicite sur l'ensemble des clones.

Les couches de déploiement et de collaboration représentent la partie "*Back-End*" du système et comprennent des modèles (détaillés ci-dessous) afin de répondre aux problèmes soulevés au niveau des différents contextes liés à la conception des applications collaboratives dans un environnement MCC.

Afin d'illustrer les fonctionnalités des modèles de conception proposés, des diagrammes de classes de nos patrons (voir les figures 4.2 et 4.4) seront présentés dans cette section.

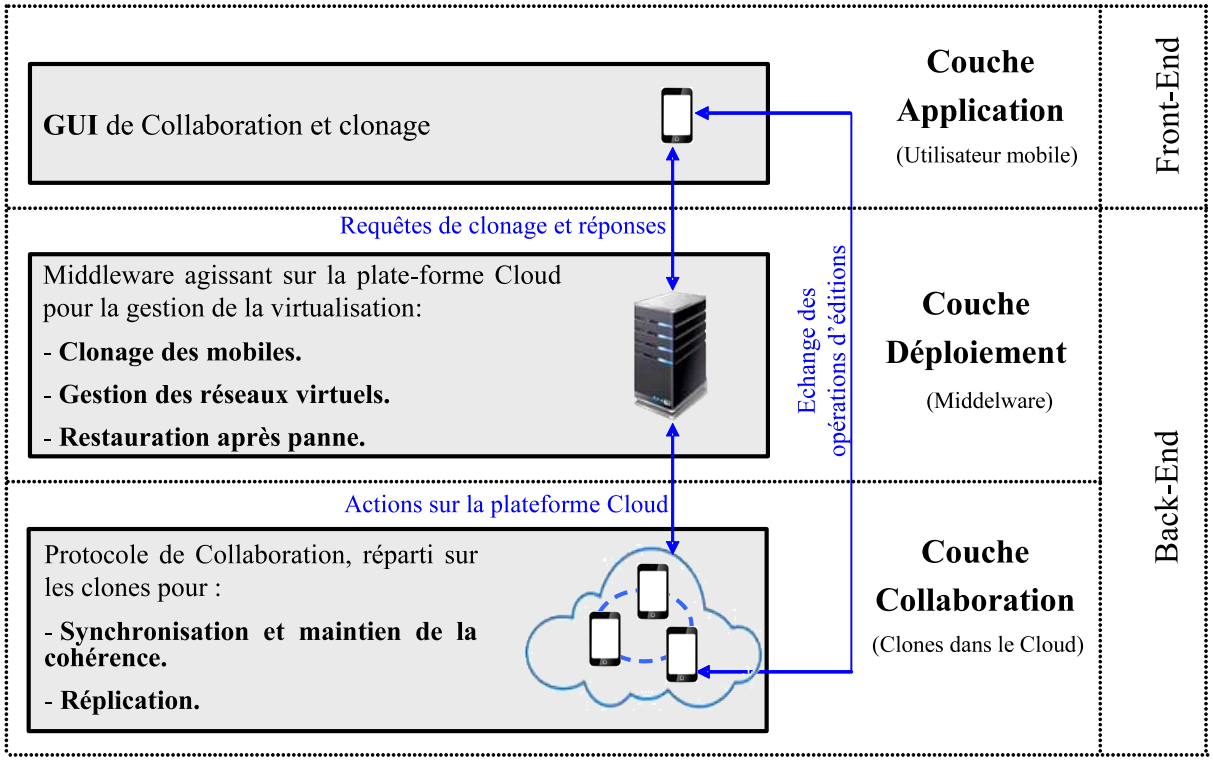


FIGURE 4.1 – Architecture globale.

4.1 Patrons de clonage

Ce modèle consiste à définir des services encapsulant des tâches agissant sur la plateforme de cloud pour le clonage des mobiles, la gestion des réseaux virtuels et la reprise automatique après défaillance des clones.

Dans ce qui suit, les trois principaux patrons de ce modèle de déploiement sont décrits (voir la figure 4.2).

4.1.1 Processus de clonage

Contexte. Avec ses traitements et ses services de déploiement de données, le cloud est considéré comme une solution bien adaptée à l'insuffisance des ressources des mobiles (c-à-d, la courte durée de vie de la batterie et la limitation de la capacité de stockage et des traitements). Dans ce contexte, un processus de clonage va permettre la création des clones qui prendront en charge l'exécution de toutes les tâches de collaboration et de réseautage. En outre, l'absence d'utilisateurs en raison d'une déconnexion intentionnelle ou involontaire (suite à une rupture ou défaillance du réseau sans fil) ne sera pas considérée comme un obstacle pour les travaux de collaboration. En effet, les clones sont toujours actifs dans le cloud et en état de communication permanente les uns avec les autres afin d'assurer la continuité de la collaboration.

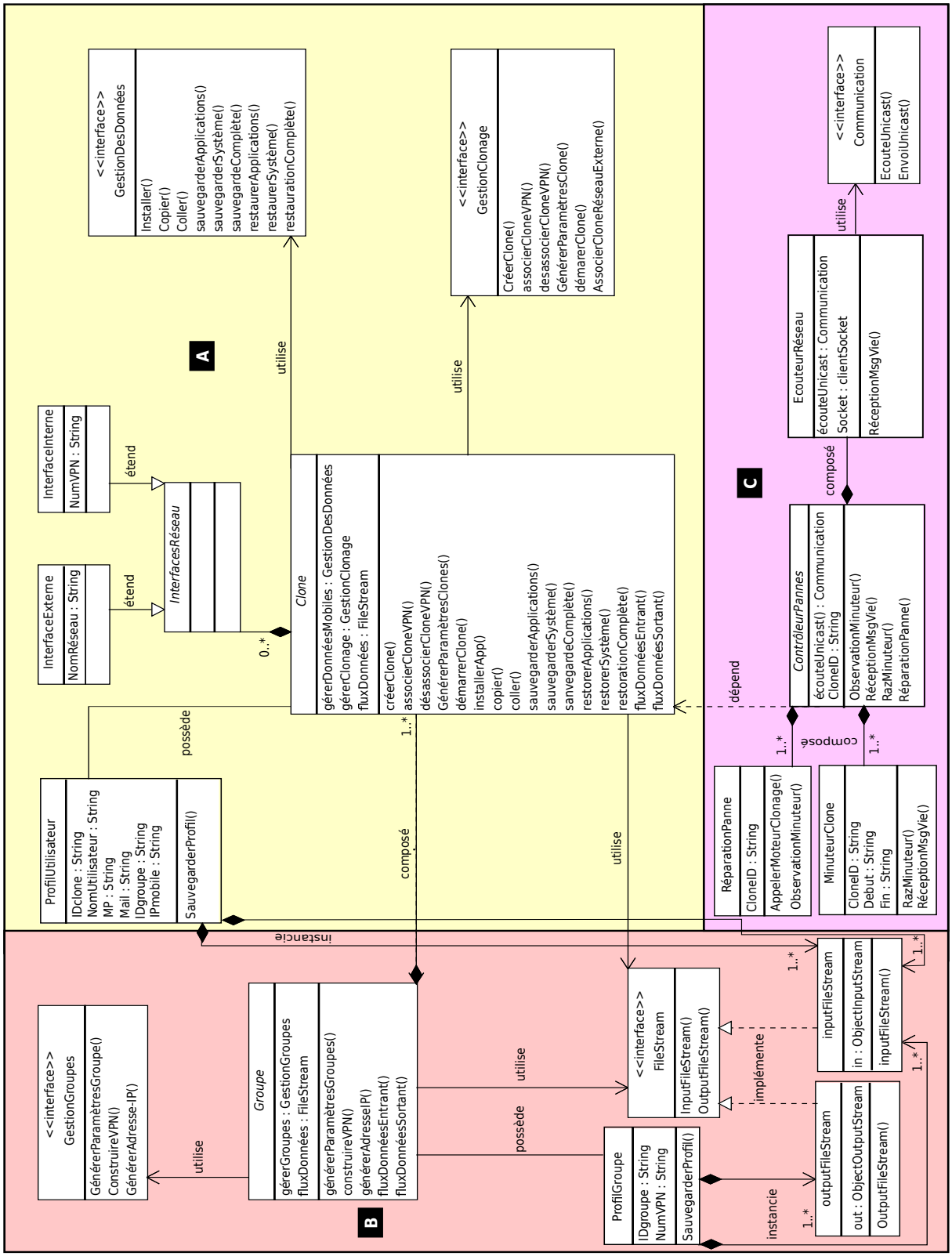


FIGURE 4.2 – Digramme de classes des patrons de clonage.

Problème. Construire des clones n’est pas une tâche simple. La question qui se pose à ce stade est : comment réaliser une interface (entre utilisateurs et le cloud) encapsulant toutes les actions de clonage sur la plateforme cloud ? De plus, il est nécessaire de doter chaque nouveau clone avec un mécanisme autonome pour configurer les paramètres réseau nécessaires à la communication avec d’autres clones ainsi que son mobile.

Solution. Le patron “processus de clonage” remédie à ce problème et consiste à implémenter un service web afin d’encapsuler les différentes actions de clonage qu’il exécute. Ce processus est déclenché suite à une requête utilisateur via une interface web (la couche GUI précédemment décrite dans la section 4) afin d’adhérer au service proposé et créer un clone pour son mobile. La figure 4.2(A) présente les différentes classes, interfaces et relations de ce patron où la classe abstraite *Clone* est considérée comme la première classe mère. Elle utilise trois classes d’interface de base, à savoir *FileStream*, *GestionDesDonnées* et *GestionClonage*. La figure 4.3 résume le principe de réutilisation de ce patron qui consiste à implémenter les différentes interfaces afin de réaliser les actions de clonage suivantes:

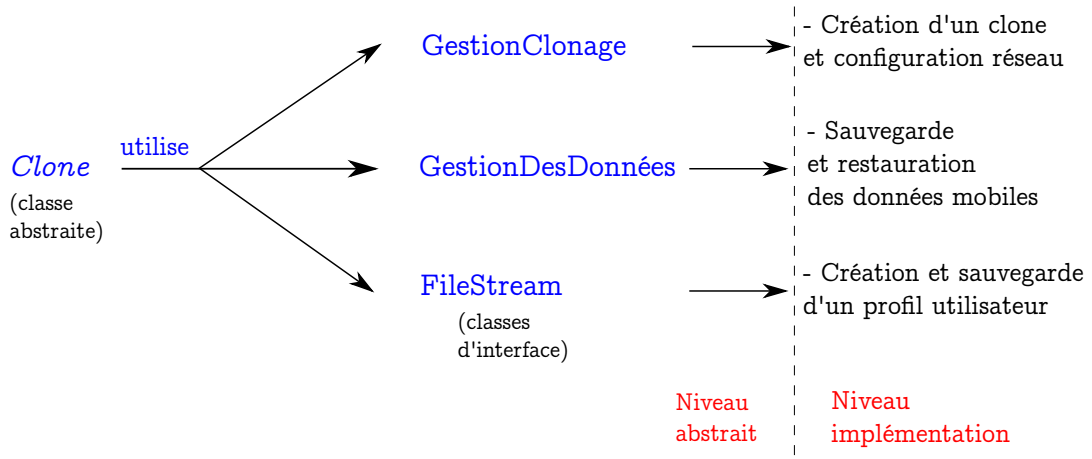


FIGURE 4.3 – Principe de réutilisation du processus de clonage.

— **Création d’un clone et configuration réseau.** Pour la création d’un nouveau clone et sa configuration réseau, les méthodes de la classe abstraite *Clone* sont basées sur une instanciation d’un ensemble de classes implémentant l’interface *GestionClonage* via la redéfinition de ses méthodes. Ceci permet d’implémenter les différentes actions requises pour la création d’une nouvelle machine virtuelle (par exemple, Android) ainsi que sa configuration réseau. Une telle configuration est primordiale afin de bien mener les différentes tâches collaboratives entre clones du même groupe. Ceci consiste à définir des interfaces réseau internes (c-à-d, paramétrer des cartes réseaux virtuelles) qui permettent d’associer chaque clone aux réseaux privés virtuels auxquels il est membre.
Du point de vue implémentation, la création d’un clone avec un système d’exploitation donné est réalisé à travers une redéfinition de la méthode *CréerClone*. Alors que l’initialisation des différentes interfaces réseau d’un clone est achevée à travers une re-

définition des méthodes *AssocierCloneVPN*(), *desassocierCloneVPN* et *AssocierCloneRéseauExterne()*. Cette initialisation permet entre autres de : (i) gérer l'appartenance d'un clone aux VPNs à travers des méthodes d'association et de dissociation et de (ii) définir le mécanisme d'accès au clone depuis l'extérieur.

— **Sauvegarde et restauration des données mobiles.** Afin de sauvegarder les données et les applications mobiles, la classe *Clone* utilise l'interface *GestionDesDonnées*, et par conséquent, elle peut faire appel à d'autres classes qui implémentent cette interface via une redéfinition de ses méthodes. D'une part, la redéfinition des méthodes *installer*(), *copier*() et *coller*() de l'interface *GestionDesDonnées* permet de réaliser des opérations de transfert de fichiers et d'applications. D'autre part, redéfinir les méthodes *sauvegarderApplications()*, *sauvegarderSystème()*, *sauvegardeComplète()*, *restaurerApplications()*, *restaurerSystème()* et *restorationComplète()* fournit des opérations optionnelles pour la sauvegarde et restauration des données mobiles.

— **Création et sauvegarde des profils utilisateurs.** Pour l'adhésion d'un nouvel utilisateur, un profil est créé et enregistré via l'instanciation de la classe *ProfilUtilisateur*. Afin de réaliser cette action, le processus de clonage utilise des informations d'enregistrement en-ligne (nom d'utilisateur, mot de passe, photo et adresse IP mobile) qui sont introduites par l'utilisateur via l'interface web graphique et d'autre part, génère des informations de clonage servant à identifier et paramétrer chaque clone (l'identifiant et l'adresse IP du clone et l'identifiant du groupe). Comme elle découle de la classe *Clone*, la classe *ProfilUtilisateur* utilise à son tour la classe d'interface *FileStream* afin d'instancier la classe *InputFileStream* qui implémente cette interface. Cette instanciation permet d'enregistrer toutes les informations du profil précédemment décrites sous forme d'un fichier d'objets, où chaque objet correspond à une instance d'un profil utilisateur.

4.1.2 Le constructeur VPN

Contexte. Les utilisateurs mobiles forment des groupes de collaboration, partagent des ressources, et veulent toujours protéger leurs données en s'appuyant sur des modes de communication privés. Ces groupes sont dynamiques dans le sens où les utilisateurs peuvent les créer, les joindre ou les quitter à n'importe quel moment. En outre, ils doivent être capables de participer à-la-demande à plusieurs sessions de collaboration.

Problème. La formation de réseaux ad-hoc réels est coûteuse, limitée par des restrictions géographiques et souffre d'instabilité réseau qui est due aux déconnexions fréquentes. La collaboration dans un réseau mobile P2P nécessite des tâches de communication intensives (un trafic réseau important) entre pairs (c-à-d, mobiles) ; cela conduit à épuiser rapidement la charge des batteries.

Solution. L'utilisation du composant constructeur VPN permet de résoudre ce problème. Un réseau privé virtuel est destiné à grouper des clones prenant en charge la totalité des tâches ré-

seau pour alléger les dispositifs réels. D’une manière identique au cas du processus de clonage, le processus de création des réseaux virtuels dans le cloud est implémenté comme un service web. La figure 4.2(B) présente un sous diagramme de classes relatif à ce patron qui permet d’implémenter les différentes actions du constructeur VPN. Ainsi, la classe abstraite *Group* est considérée comme la première classe mère. Elle utilise deux classes d’interface fondamentales, à savoir *GestionGroupes* et *FileStream* permettant respectivement de (i) gérer la construction des réseaux privés virtuels ainsi que leur configuration et de (ii) sauvegarder les informations du profil des groupes créés.

Les actions de ce processus sont déclenchées suite à une requête utilisateur via l’interface web (GUI) ; ceci active l’hyperviseur de virtualisation pour construire un nouveau réseau virtuel. D’autre part, d’autres actions sont nécessaires pour la configuration du réseau créé, telles que la définition de l’espace des adresses IP de ce réseau virtuel ainsi que son adresse de diffusion (*broadcast*) spécifique.

Pour la création d’un nouveau réseau virtuel ainsi que sa configuration, la classe abstraite *Group* utilise la classe d’interface *GestionGroupes* qui contient des méthodes spécifiques à ses tâches. Cette utilisation permet d’instancier d’autres classes qui implémentent l’interface *GestionGroupes* par une redéfinition de ses méthodes :

(i) *GenérerParamètresGroupes* : génère de nouveaux paramètres servant à identifier et paramétrer le VPN à créer, à savoir l’identificateur du groupe, son nom, l’espace d’adressage IP et l’adresse de diffusion servant à la propagation des messages entre les clones au sein du même VPN.

(ii) *ConstruirVPN* : implémente le processus de création du VPN selon l’hyperviseur choisi.

(iii) *GénérerAdresses_IP* : active un serveur DHCP basé sur un espace d’adressage (délimité par des bornes inférieures et supérieures d’adresses IP) unique généré pour ce VPN. Ce serveur a pour rôle d’allouer dynamiquement des adresses IP uniques aux interfaces réseau internes des clones membres de ce VPN.

Afin de gérer le profil du groupe créé, l’instanciation de la classe *ProfilGroupe* permet d’enregistrer les paramètres générés pour ce groupe. Elle fait donc appel à la classe *OutputFileStream* qui implémente la classe d’interface *FileStream*.

À noter que la classe d’interface *FileStream* est communément utilisable par les deux patrons de clonage et de construction VPN afin de gérer la sauvegarde, la consultation et la mise-à-jour des objets contenant les paramètres relatifs aux différents clones et groupes. Un autre lien entre ces deux patrons est présenté à travers la relation père-fils entre les deux classes abstraites *Group* et *Clone* indiquant qu’un groupe peut contenir un ou plusieurs clones.

4.1.3 Contrôleur des pannes

Contexte. Dans notre travail, les clones sont destinés à effectuer les principales tâches de collaboration, telles que la sauvegarde des données, l'échange de requêtes avec d'autres clones et l'exécution des protocoles de maintien de la cohérence des données partagées. Ces machines virtuelles doivent être toujours en ligne pour signaler les modifications des données effectuées par d'autres clones et recevoir des requêtes localement appliquées par leurs mobiles. Ainsi, des mécanismes autonomes de détection des pannes des clones ainsi que leur réparation immédiate est nécessaire afin d'empêcher tout éventuel isolement des utilisateurs de leurs groupes de collaboration.

Problème. Deux questions sont posées à ce niveau : (i) comment détecter les défaillances des clones qui sont déployés dans le cloud sur des serveurs distants ? et (ii) après détection d'un clone en panne, comment effectuer une auto-restauration de son état ?.

Solution. Une solution à ce problème implique l'utilisation d'un contrôleur de panne pour détecter les défaillances qui peuvent affecter le bon fonctionnement des clones. Le patron proposé comme solution pour ce problème est basé sur le principe du "battement de cœur" où chaque clone doit indiquer son bon fonctionnement via un envoi régulier d'un "message de vie".

Comme le montre la figure 4.2(C), le contrôleur des pannes instancie la classe *EcouteurRéseau* pour construire une liste d'écouteurs réseau dont chacune de ses entrées correspond à un clone dans le cloud. Il recevra périodiquement des "messages de vie" depuis les différents clones avec des intervalles de temps égaux à travers ces écouteurs. Un envoi régulier de ces messages indique le bon fonctionnement du clone émetteur. Le contrôleur des pannes utilise également une liste qui contient des minuteurs pour calculer le temps d'attente correspondant à la réception d'un message de vie pour chaque clone. Pour ce faire, il instancie la classe *MinuteurClone* pour chaque clone créé. Chaque minuteur est remis à zéro après chaque réception régulière d'un "message de vie" du clone correspondant. Dans le cas de détection d'une panne, l'instanciation de la classe *RéparationPanne* invoquera le processus de clonage afin de créer une nouvelle instance du clone mis en échec et restaurer ses paramètres et données.

4.2 Patrons de collaboration

Dans notre travail, nous permettons à des mobiles de collaborer via le cloud. Ainsi notre modèle de collaboration supporte des mécanismes de synchronisation utilisés par chaque clone pour partager des ressources. Ces mécanismes permettent la réconciliation des copies divergentes des ressources d'une manière décentralisée (c-à-d, sans recourir à une synchronisation centrale gérée par un serveur central ou un clone maître). Il est nécessaire de mentionner que le modèle de collaboration utilise un schéma de réplication optimiste pour fournir un accès simultané à des ressources partagées. En effet, chaque utilisateur possède deux copies de la ressource partagée : la première copie est stockée dans le mobile, tandis que la seconde est au niveau de

son clone. Afin de maintenir la cohérence des ressources partagées en temps réel, chaque clone doit être simultanément synchronisé avec les autres clones ainsi que son mobile. L'objectif majeur de ce modèle est de déléguer le maximum des tâches intensives (de synchronisation et de communication) aux clones.

Dans ce qui suit, nous présentons une description des principaux patrons de ce modèle de collaboration (voir la figure 4.4).

4.2.1 Intégration d'un nouveau clone

Contexte. Les clones nouvellement créés doivent être intégrés dans leurs groupes de collaboration. Cette intégration consiste à acquérir des identifiants et des paramètres de communication qui sont nécessaires à l'identification des clones membres du même groupe et, par conséquent, l'accomplissement de leurs tâches de collaboration. Pour la manipulation des ressources partagées, la collaboration des clones est basée sur l'échange des requêtes. Chaque requête se compose principalement d'une partie d'identification contenant l'identifiant du clone et de son groupe et une partie dédiée à l'opération ou la mise-à-jour appliquée sur la ressource partagée. L'échange de requêtes entre clones du même groupe est basé sur des primitives de propagation (broadcast). Les clones doivent acquérir l'adresse de propagation leur permettant la diffusion des requêtes à travers leurs réseaux privés virtuels.

Problème. Un clone déployé dans le cloud recevra une adresse IP qui sera affectée à son interface externe par un serveur DHCP juste après son premier démarrage. Les identifiants du clone et de son groupe ainsi que l'adresse de diffusion correspondant à son VPN sont stockés sur le serveur hébergeant le middleware de déploiement. Ainsi, un clone distant déployé dans le cloud doit être capable d'agir de manière autonome pour demander et acquérir ces paramètres depuis le middleware de clonage.

Solution. Une solution est proposée via le composant d'intégration d'un nouveau clone (voir la figure 4.4(A)). Le clone déployé dans le cloud est pré-équipé de la configuration requise permettant un démarrage automatique d'une application collaborative (c-à-d, le protocole de collaboration) juste après son premier démarrage. Les méthodes de la classe *IntégrerClone* (c-à-d, *InitialiserCloneParamètres()*, *DémarrerEcoute()*, *InitialiserRessources()* et *AppelerMiddleware()*) sont utilisées pour lancer les premières tâches effectuées par un nouveau clone juste après son démarrage. Cela inclut l'appel du middleware de déploiement, le lancement d'un processus pour l'envoi des “messages de vie” du clone et le lancement de l'écoute réseau par l'instanciation respective des classes *AppelerMiddleware*, *MessageVie*, *EcouteBroadcast* et *EcouteUnicast*. Pour mieux expliquer cet auto-mécanisme, la figure 4.5 illustre les étapes nécessaires pour intégrer un nouveau clone. Ces étapes sont résumées comme suit :

1. **Démarrage de l'application collaborative.** L'application collaborative est lancée automatiquement avec le premier démarrage du clone créé. Ensuite, un appel du service web *Demander paramètres(IP clone)* du middleware de déploiement est effectué. Cet appel

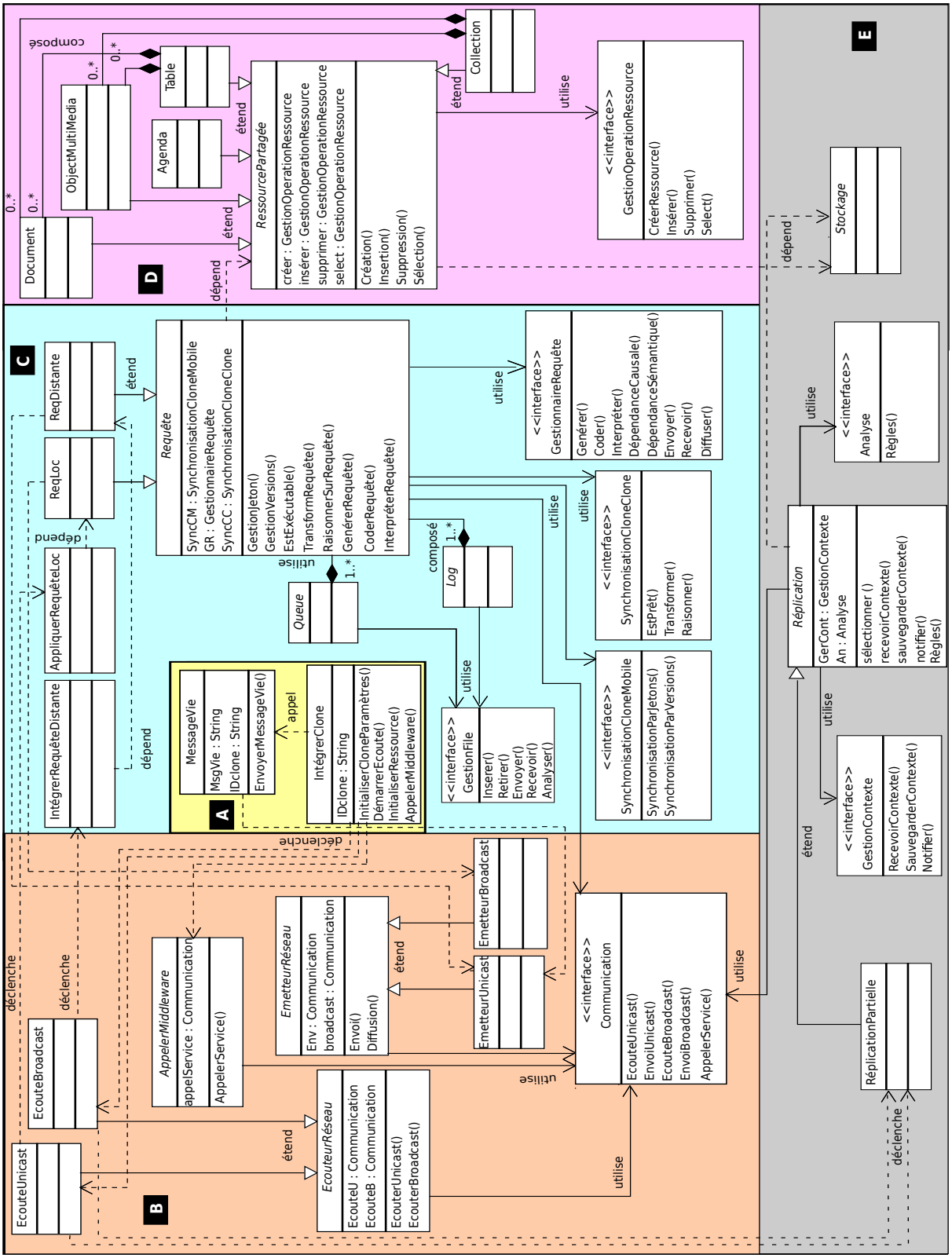


FIGURE 4.4 – Diagramme de classes des patrons de collaboration.

permet de demander les paramètres nécessaires à l’exécution des différentes primitives de réseautage. L’adresse IP (de l’interface externe) du clone est passée comme paramètre à cet appel de service.

2. **Mise à jour des paramètres du clone.** Suite à l’appel précédent, le middleware de déploiement mettra à jour l’adresse IP du clone en utilisant l’adresse fournie via l’appel du service web effectué par le clone. Ensuite, le clone recevra son identifiant unique, l’identifiant du groupe, l’adresse de diffusion au sein de son réseau privé virtuel et l’adresse IP actuelle du mobile.
3. **Initialisation des paramètres et lancement d’écoute en mode diffusion.** Compte tenu des données reçues, le clone met à jour les différents paramètres de l’application collaborative. L’adresse de diffusion sera utilisée pour démarrer un processus d’écoute afin de recevoir des messages (contenant les requêtes distantes) provenant d’autres clones et les stocker dans la file d’attente des requêtes distantes. Le démarrage d’une telle écoute permettra au clone d’entamer une participation anticipée au sein de son groupe (en recevant toutes les modifications en temps réel), le temps qu’il achèvera les différentes étapes nécessaires à son intégration complète. Ensuite, il enverra un message (c-à-d, une requête ou demande) pour obtenir l’état actuel des ressources partagées.

   Le middleware de déploiement enverra au clone une copie des ressources partagées, ainsi que les logs des opérations (contenant l’histoire de toutes les modifications effectuées sur les ressources partagées avant et pendant l’intégration du clone).
4. **Mise à jour des paramètres utilisateur.** Le middleware de déploiement notifie à l’utilisateur mobile un message indiquant l’état prêt du clone pour établir une connexion directe entre le mobile et son clone. L’utilisateur devra alors appeler un service d’authentification afin d’établir cette connexion. Après une authentification réussie, l’utilisateur recevra l’adresse IP clone. En utilisant cette adresse, il peut établir une connexion mobile/clone directe pour demander la copie actuelle des ressources partagées.

4.2.2 Communication

Contexte. La collaboration des différents utilisateurs à travers les clones nécessite un mécanisme d’échange de requêtes (messages) entre les différentes composantes du système de collaboration. Cette communication est considérée comme un package distinct dans le protocole de collaboration. Chaque clone utilise les méthodes de ce package pour échanger des messages avec d’autres clones et avec son mobile et appeler les services du middleware de déploiement. Rappelons que l’échange de messages entre les clones permet l’intégration des requêtes distantes appliquées sur des ressources partagées, tandis que l’appel des services du middleware de déploiement permet aux clones de recevoir les paramètres requis pour l’accomplissement de leurs tâches de collaboration.

Problème. La collaboration est basée sur un module de communication qui devrait permettre

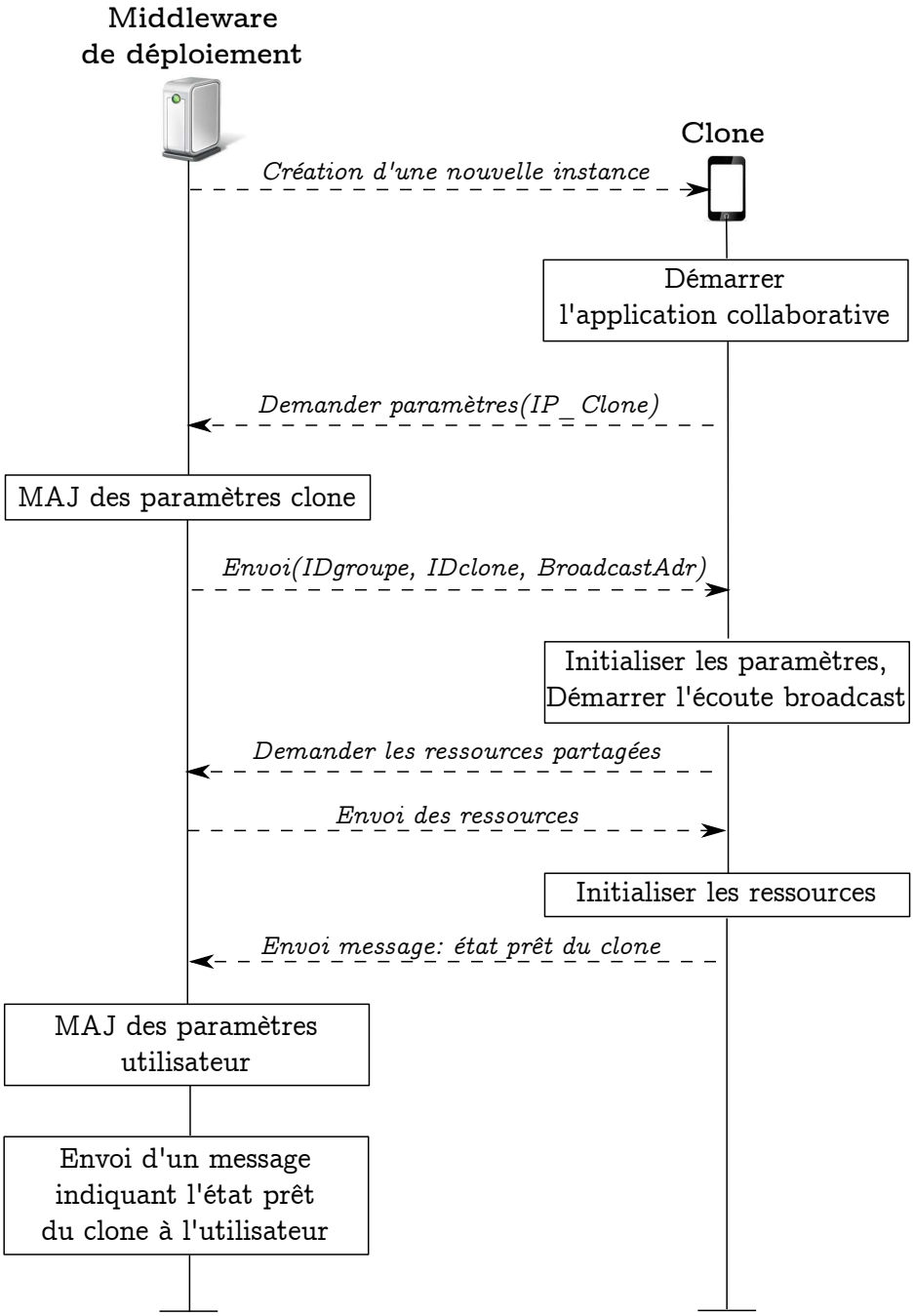


FIGURE 4.5 – Processus d'intégration d'un nouveau clone.

une synchronisation entièrement décentralisée (c-à-d, en mode P2P). Les tâches de communication doivent être optimisées (par exemple, des messages diffusés en mode *broadcast*) afin d'éviter la surcharge des réseaux virtuels et par conséquent améliorer les temps de réponse.

Solution. L'utilisation d'un patron de communication permet de résoudre ce problème. Cet objectif est atteint par l'implémentation d'un ensemble de primitives réseau pour écouter et

envoyer des messages en même temps.

Comme le montre la figure 4.4(B), ces primitives sont accessibles via l'implémentation de la classe d'interface *Communication* par d'autres classes redéfinissant ses méthodes pour coder : (i) des processus d'écoute (en deux modes : unicast et multicast) sur le réseau à travers la redéfinition de la méthode *Communication.EcouteUnicast()* et *Communication.EcouteBroadcast()*, (ii) des processus d'envoi des messages en utilisant les méthodes *Communication.EnvoiUnicast()* et *Communication.EnvoiBroadcast()*, et (iii) des appels des services du middleware en redéfinissant la méthode *Communication.AppelerService()*.

Les classes abstraites *EcouteurRéseau*, *EmetteurRéseau* et *AppelerMiddleware* permettent de regrouper (d'une manière abstraite) les différentes tâches de communication en trois catégories ; à savoir, écoute, envoi et appel de services. Les différents attributs de ces classes sont de type *communication* ; ceci permet d'instancier les différentes classes qui implémentent cette interface. Ainsi, d'une part, les tâches d'écoute sont concrètement définies par les méthodes des classes *EcouteUnicast* et *EcouteBroadcast* qui étendent la classe abstraite *EcouteurRéseau* et bénéficient de cet héritage pour instancier des classes implémentant l'interface *Communication*. Cette instanciation permet de lancer des processus d'écoute en deux modes : unicat et diffusion. Suivant le même principe, les méthodes des classes *EmetteurUnicast* et *EmetteurBroadcast* permettent de concrètement définir les tâches d'envoi des messages (en modes unicast et diffusion) par l'instanciation des classes implémentant l'interface *Communication*.

4.2.3 Synchronisation

Contexte. Un utilisateur peut à tout moment se connecter et manipuler des ressources partagées à travers les répliques stockées au niveau de son mobile réel. Les clones sont toujours en-ligne dans le cloud et coopèrent entre eux par échange des messages afin d'intégrer des requêtes distantes, ou avec leurs mobiles réels pour appliquer des requêtes locales. Ces requêtes sont destinées à envelopper les différentes opérations appliquées sur les ressources partagées.

Problème. Les accès simultanés à des répliques de ressources partagées peuvent produire des vues incohérentes au niveau des différents clones. De plus, un décalage d'exécution des requêtes locales entre le clone et son dispositif réel est inévitable ; ce décalage constitue un autre problème pour maintenir la cohérence des ressources partagées.

Solution. Comme le montre la figure 4.4(C), le patron collaboration offre des solutions aux différents problèmes liés aux interactions simultanées clone/clone et clone/mobile. L'objectif de ce composant est de fournir des mécanismes de synchronisation décentralisés permettant le maintien de la cohérence des ressources partagées (par réplication). Afin de bien comprendre ce composant, nous commençons par une présentation générale des mécanismes de synchronisation qui peuvent être résumés comme suit :

— Le mobile (l'utilisateur) est la source de toute opération localement appliquée.
— Une fois appliquée, l'opération locale sera envoyée par le mobile à son clone dans le

cloud.

— Après réception, le clone appliquera l'opération sur le même état de la ressource partagée. Il procèdera ensuite à la préparation de la propagation de l'opération vers les autres clones (membres du même groupe). Cette préparation consiste à générer une requête pour chaque opération locale et à déduire ses éventuels dépendances avec d'autres opérations préalablement exécutées.
— L'application de n'importe quelle requête impliquera son ajout au *Log* (histoire des requêtes locales et distantes déjà appliquées : *ReqLoc* et *ReqDistante*).
— Une fois que les autres clones ont reçu une requête distante *ReqDistante*, ils procèderont à son intégration en préservant ses dépendances causales. Les requêtes dépendantes et non prêtes (c-à-d, le cas où les autres requêtes dont elles dépendent ne sont pas encore vues ou appliquées par le clone) sont considérées comme "non prêtes" et par conséquent seront misent en attente dans la file *Queue*.

Ce mécanisme est présenté dans sa forme la plus simple, mais la flexibilité de l'architecture proposée permet aux développeurs de créer des mécanismes de synchronisation plus complexes. Par exemple, un modèle spécifique au partage des données RDF peut être basé sur des processus de décomposition des requêtes complexes et de raisonnement.

Ainsi, les différentes tâches de ce patron de collaboration sont regroupées en quatre classes d'interface, permettant de faire abstraction de quatre catégories des méthodes suivantes :

1. **La gestion des requêtes.** La synchronisation entre clones est basée sur l'échange de requêtes. Dans ce contexte, l'implémentation de la classe d'interface *GestionnaireRequête* permet de redéfinir des méthodes de génération, codage et interprétation des requêtes. La génération d'une requête est l'étape qui succède l'application d'une opération locale et elle consiste à formuler un objet *Requête* composé des champs (attributs) : identificateur clone, identificateur groupe, numéro de série de l'opération, l'opération elle même et la liste des dépendances de cette requête. L'objet *opération* est issu de l'implémentation de l'interface *GestionOperationRessource* du patron gestion des ressource qui sera décrit dans la section suivante (voir la figure 4.4(D)). Une fois la requête générée, le clone procèdera à son codage sous forme d'un message composé d'une chaîne de caractères. Le message résultant de ce codage sera diffusé aux autres clones en utilisant les primitives de diffusion (*EmetteurBroadcat* et *EcouteBroadcast*) du patron de communication précédemment décrit (voir la figure 4.4(B)). Notons que pour améliorer les performances du système en matière de latence et trafic réseau, nous avons opté pour l'utilisation des messages dynamiques dont la taille des différents champs est variable (par exemple, le numéro de série d'une requête). Pour ce faire, ces champs sont suivis par un caractère spécial indiquant leur limite de fin. D'une autre part, après réception d'un message, le clone procèdera à son interprétation afin de déduire la requête qu'il contient. La figure 4.6 montre un exemple illustratif simple concernant la gestion des requêtes par un clone.

   Dans le même contexte de gestion des requêtes, les fonctions *DépendanceCausale()* et *DépendanceSémantique()* de l'interface *GestionnaireRequête* permettent de déduire les

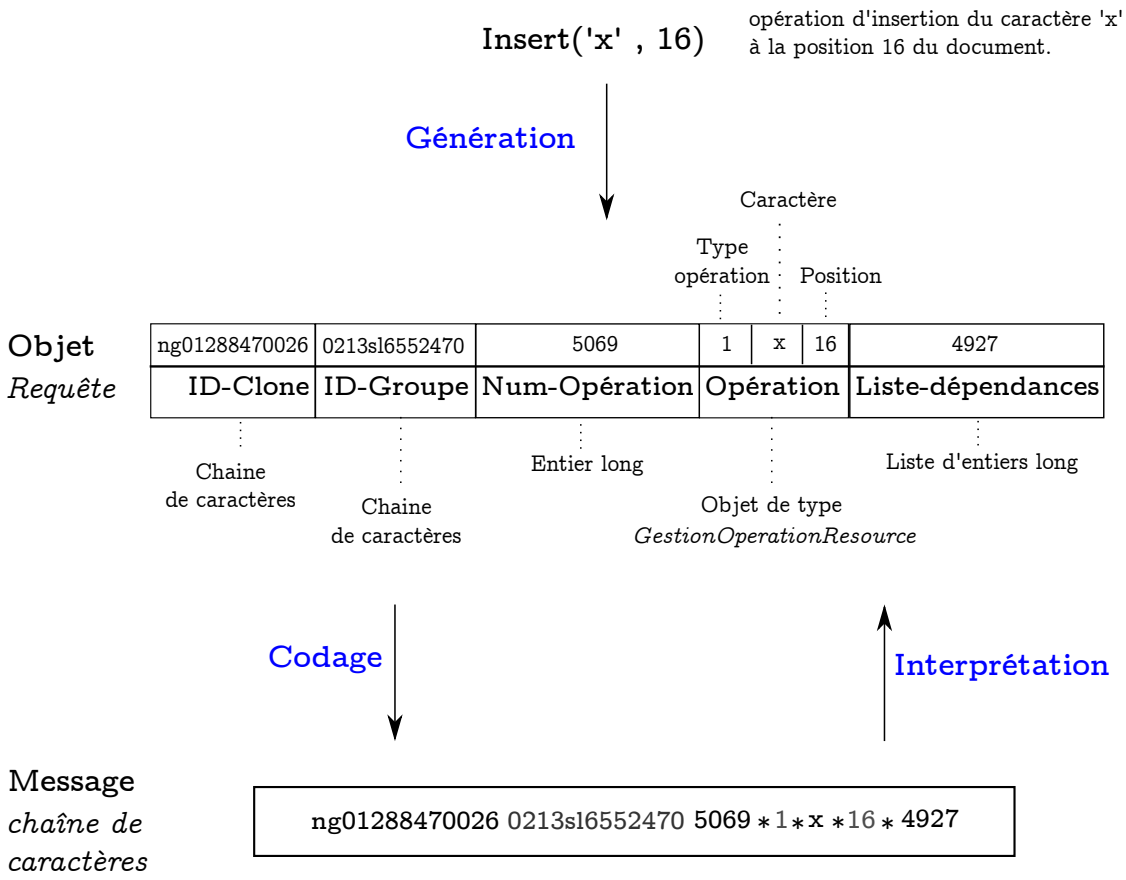


FIGURE 4.6 – Exemple des méthodes de gestion des requêtes.

dépendances causales et sémantiques de chaque requête.

2. **Synchronisation clone/clone.** Les classes implémentant l’interface *SynchronisationCloneClone* permettent une redéfinition des différentes méthodes liées à la synchronisation clone/clone, conduisant ainsi au maintien de la cohérence suite aux divergences causées par les accès concurrents aux ressources partagées. Cette interface peut regrouper des méthodes associées à une ou plusieurs approches de synchronisation par réplication qui sont généralement associées aux structures de données (ressources) bien spécifiques. Par exemple l’approche des Transformées Opérationnelles (OT) [43, 64] est généralement définies sur des structures de données linéaires. Dans notre patron de collaboration, la redéfinition des méthodes *EstPrêt* et *Transformer* de l’interface *SynchronisationCloneClone* permet d’implémenter des mécanismes de synchronisation selon l’approche OT (pour plus de détails concernant cette approche, voir la section 3.4 dans le chapitre 2).
3. **La synchronisation clone/mobile.** Le mobile et son clone sont deux entités qui sont physiquement séparées. Par conséquent, un décalage d’application des mêmes opérations (sur les deux côtés) sur le même état peut avoir lieu. Cela peut mener à des incohérences sur les vues des ressources partagées. Pour résoudre ce problème, deux solutions basées sur une exclusion mutuelle distribuée ou une gestion des versions des ressources peuvent

être utilisées à travers les deux méthodes *SynchronisationParJetons()* et *Synchronisation-ParVersion()* de la classe d'interface *SynchronisationCloneMobile*.

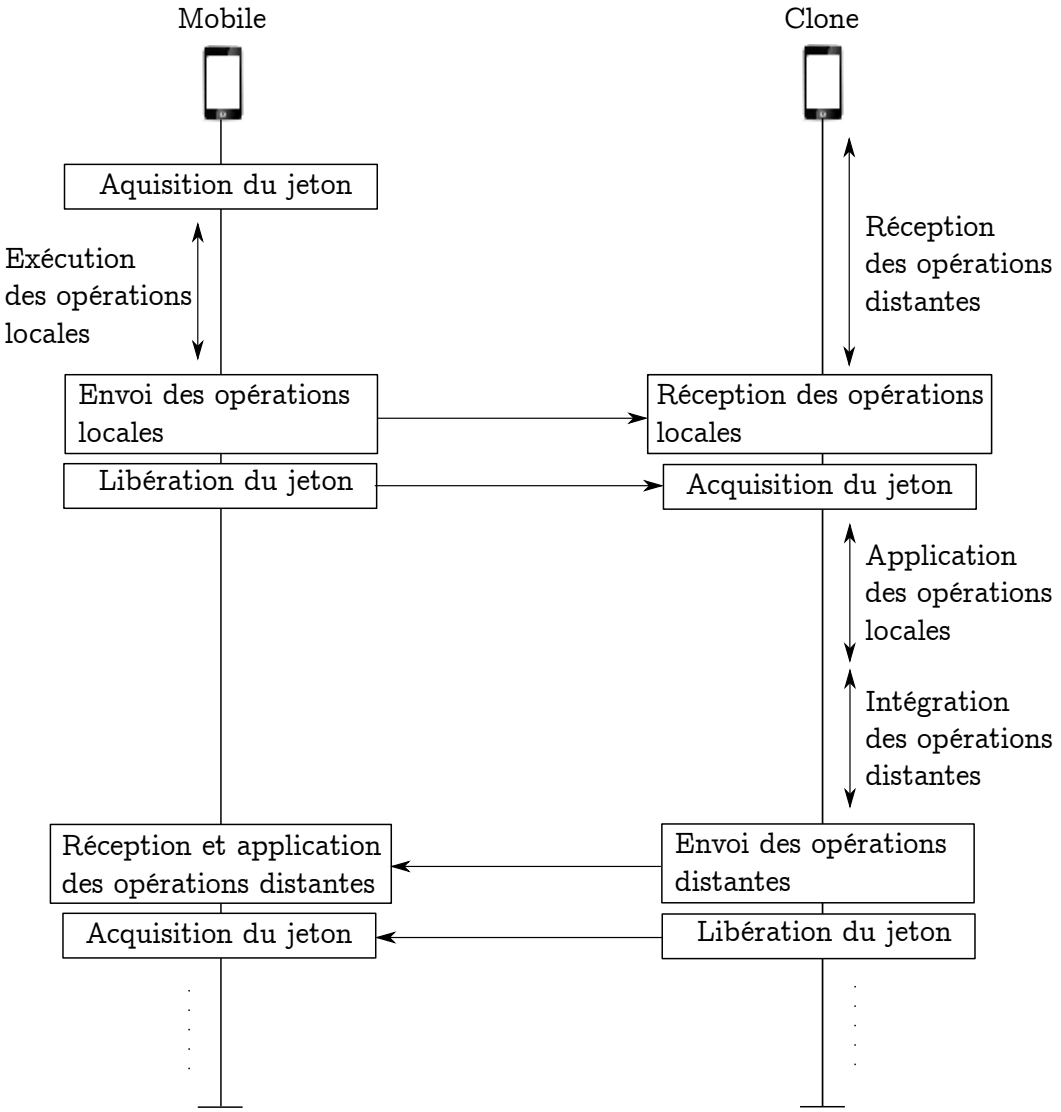


FIGURE 4.7 – Synchronisation clone/mobile : exclusion mutuelle distribuée.

Avec un protocole de synchronisation basé sur une exclusion mutuelle distribuée, la ressource partagée est considérée comme une section critique dont seul le mobile ou son clone aura le droit exclusif d'accéder en mode synchronisation à sa copie de ressource [84]. Comme le montre la figure 4.7, ce protocole est réalisé par échange de messages (c-à-d, jetons). Initialement, le mobile possède le droit d'être le premier à synchroniser avec son clone. Le clone continue de recevoir des opérations distantes émises par d'autres clones afin de les intégrer ultérieurement sur son état local. Mais une fois qu'une décision de synchronisation avec le clone est prise, toutes les opérations locales seront envoyées au clone et le droit d'accès exclusif sera libéré afin de permettre au clone (à son tour) de

démarrer une synchronisation avec son mobile. Ainsi, le clone applique les opérations distantes précédemment reçues sur son état local, et les envoie au mobile qui les appliquera à son tour sur sa copie.

D’autre part, un protocole de synchronisation basé sur une gestion des versions des ressources permet au mobile et son clone de travailler indépendamment sur leurs copies des ressources en mode "on-line" comme en "off-line". La réconciliation des divergences est effectuée côté clone qui doit appliquer les opérations locales sur la même version du mobile. Ce mécanisme est beaucoup plus adapté avec des éditeurs basés sur des opérations commutatives. Plus de détails concernant ce protocole de synchronisation sont illustrés dans le chapitre 6.

4. **La gestion des files d’attentes et logs.** L’interface *GestionFile* est dédiée à la gestion des files d’attente contenant les requêtes en instance d’intégration ou à la gestion des histoires (c-à-d, log) comportant les requêtes déjà appliquées. Cette gestion offre des primitives usuelles pour la manipulation des structures de données linéaires et dynamiques (par exemple, insertion d’une requête reçue dans une file d’attente). Cette interface est utilisée par les classes abstraites *Log* et *Queue* qui peuvent être à leur tour étendues (par les développeurs) pour une définition concrète des structures de données utilisées par le protocole de collaboration.

4.2.4 Gestion des ressources

Contexte. Les utilisateurs collaborent afin de manipuler des données partagées de manière simultanée. Ces données sont stockées localement sur le mobile réel et son clone.

Problème. Les applications collaboratives mobiles doivent prévoir des mécanismes pour partager une variété de types de données (ressources ou fichiers). En outre, la conception proposée doit être réutilisable et facilement extensible pour permettre l’intégration de nouveaux types de données avec leurs éditeurs.

Solution. Le composant gestion des ressources partagées illustré dans la figure 4.4(D) est utilisé pour résoudre le problème précédemment décrit. À travers ce composant, les développeurs peuvent définir des outils permettant d’instancier et éditer plusieurs types de données (par exemple, documents, images, table ou agenda) à travers les sous-classes de la classe abstraite *RessourcePartagée*. Les attributs déclarés au sein de la super-classe sont de type *GestionOperationRessource*. Ce type correspond au nom de la classe d’interface qui permet de définir (d’une manière générale) les différentes opérations (création, consultation et mise-à-jour) destinées à être appliquées aux différents types de ressources.

Comme la classe d’interface est implémentée par un ensemble de classes (regroupés dans un package) redéfinissant ses différentes méthodes, le développeur peut adapter les méthodes redéfinies avec les différents types de données souhaitées.

4.2.5 Réplication partielle

Contexte. Les données manipulées, partagées et échangées entre les différents collaborateurs mobiles sont de plus en plus volumineuses, liées entre elles et étroitement dépendantes du contexte. Par exemple, la participation d'un utilisateur à une conférence peut nécessiter une disponibilité (temporaire mais certaine) d'un ensemble de données, telles que planning, tourisme et cartographie.

Problème. La limitation de l'espace de stockage des mobiles ainsi que l'instabilité des connexions réseaux mobiles peut avoir un impact négatif sur la disponibilité des données.

Solution. Le patron réplication partielle, présenté dans la figure 4.4(E), offre une solution pour le problème cité ci-dessus. Cette solution vise à assurer une haute disponibilité des données mobiles utiles (c-à-d, qui sont effectivement exploitables par l'utilisateur à un instant donné) indépendamment des connexions réseau tout en respectant les limites des ressources mobiles (espace de stockage et durée de vie de la batterie).

En utilisant ce mécanisme, un clone procède à l'envoi d'un sous-ensemble des données utiles à son mobile. Cet ensemble des données répliquées est le résultat d'une sélection appliquée sur les données stockées par un clone et qui sont représentées par la classe abstraite *Stockage*. L'extension de cette classe abstraite permet de préciser le choix de stockage des données (par exemple, bases de données relationnelles ou modèles RDF). La sélection des répliques est basée sur une analyse qui peut être déclenchée suite à la réception des informations contextuelles liées aux utilisateurs et leurs environnements. À cet effet, l'interface *Analyse* regroupe et définit des règles d'analyse basées sur les informations contextuelles reçues et l'interface *GestionContexte* définit des méthodes de réception et sauvegarde du contexte.

4.3 Relations entre patrons

La figure 4.8 illustre les différentes relations pouvant exister entre les patrons précédemment présentés. Ces patrons sont organisés dans un modèle en couches avec deux niveaux. Au niveau supérieur, un processus de création d'un nouveau clone peut être invoqué via le déclenchement du "*processus de clonage*" par un utilisateur afin de cloner son mobile. Ce processus peut à son tour faire appel au "*constructeur VPN*" pour la construction d'un nouveau réseau privé virtuel. Les deux composants "*processus de clonage*" et "*constructeur VPN*" peuvent être, à n'importe quel moment, déclenchés par le "*contrôleur des pannes*" pour la réparation de tout dysfonctionnement lié aux clones ou réseaux virtuels.

Au niveau inférieur, n'importe quelle session de collaboration est déclenchée par le processus de clonage (situé au niveau supérieur). En effet, quand un clone est créé et activé, la composante "*intégration du clone*" procède au démarrage des processus de synchronisation et de réplication des données mobiles après avoir initialisé les paramètres de collaboration requis. La synchronisation du travail collaboratif implique des interactions mobile/clone et clo-

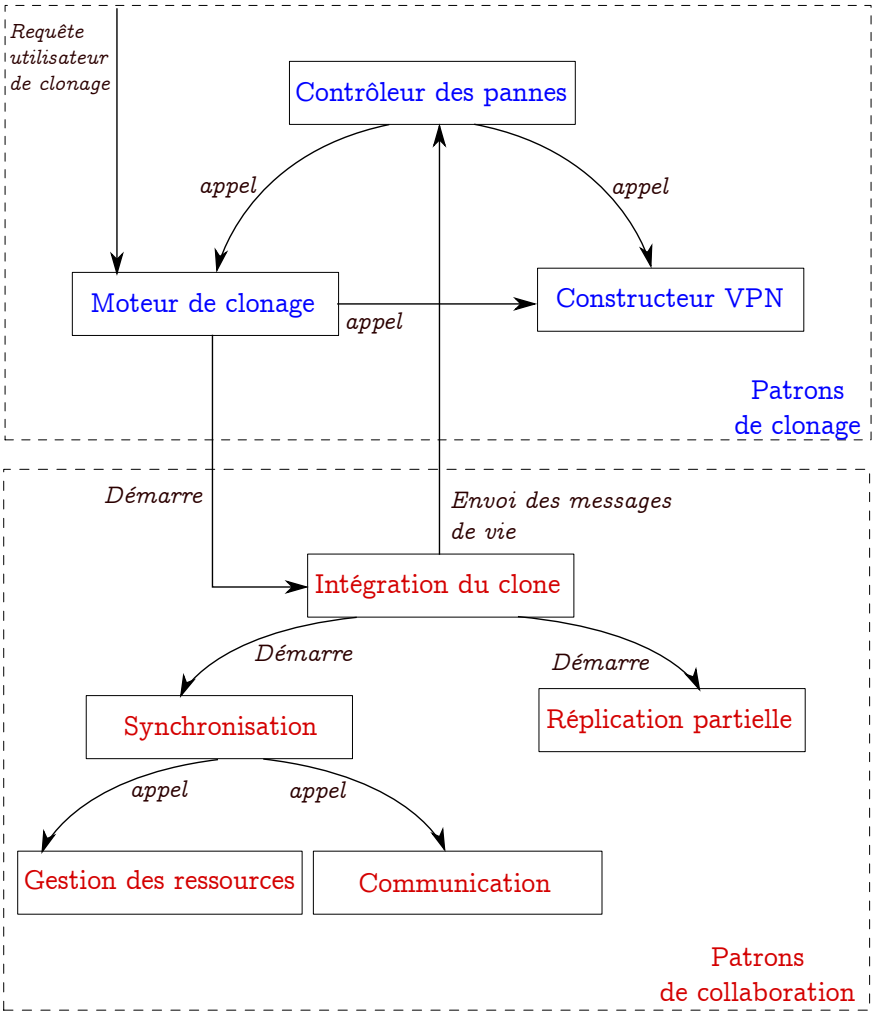


FIGURE 4.8 – Relations entre patrons.

ne/clone et la mise-à-jour des ressources partagées qui sont respectivement assurées par les composantes “*communication*” et “*gestion des ressources*”. D’autre part, la composante “*intégration du clone*” est également responsable de l’envoi des “messages de vie” au “*contrôleur des pannes*” afin d’indiquer le bon fonctionnement du clone.

4.4 Patrons vs. exigences de conception

La matrice présentée dans la figure 4.9 illustre la participation des différents patrons à la satisfaction des exigences des applications collaboratives mobiles dans le cloud (voir la section 3). Cette matrice offre plus de flexibilité en terme de développement. En effet, les développeurs peuvent choisir les modèles qui conviennent aux besoins de leurs applications personnalisées.

	Patterns \ Exigences	Autonomie	Disponibilité des données	Cohérence des données	Communication	Évolutivité	Sensibilisation des utilisateurs	Hétérogénéité	Restauration après panne
Clonage	Moteur de clonage	*	*			*	*	*	*
	Constructeur VPN		*		*	*	*	*	
	Contrôleur des pannes	*	*						*
Collaboration	Intégrateur des clones	*			*		*		*
	Communication	*	*	*	*	*	*	*	*
	Synchronisation	*	*	*					
	Gestionnaire des ressources		*	*				*	
	Réplication partielle		*	*			*		

FIGURE 4.9 – Patrons vs. exigences de conception.

5 Conclusion

Dans ce chapitre nous avons présenté une architecture globale réutilisable pouvant efficacement guider les développeurs d'applications collaboratives s'exécutant dans des environnements cloud. Cette architecture met en évidence deux principales couches, à savoir une couche de déploiement et une couche de collaboration. Dans le chapitre suivant nous allons présenter le middleware de déploiement MIDBox, un système basé sur des services web et issu de l'implémentation des patrons de clonage en utilisant l'hyperviseur de virtualisation VirtualBox [8]. Ce système peut être utilisé comme une plateforme de déploiement et d'exécution des applications collaboratives mobiles dans le cloud.

Chapitre 5

MIDBOX : un middleware de déploiement

Sommaire

1 Introduction

Les plateformes de cloud sont principalement fondées sur la technologie de virtualisation. Comme nous l'avons vu au chapitre 3, les hyperviseurs de virtualisation peuvent être généralement classés en deux types et proposent des kits de développements hétérogènes.

Dans ce chapitre, nous présentons un diagramme de classes issu de la réutilisation de l'architecture globale présentée dans le chapitre précédent afin d'implémenter MIDBox, un **MID**dleware de déploiement agissant sur l'hyperviseur Virtual**BOX** pour répondre aux différentes requêtes des utilisateurs de déploiement mobile sur un cloud privé.

Nous décrivons aussi les principales composantes de notre protocole de déploiement, à travers des diagrammes d'actions implémentant les différentes tâches des différents patrons, à savoir le processus de clonage, la construction des VPNs, l'auto-contrôle des pannes et la gestion des sauvegardes/restaurations des données mobiles. La majorité de ces tâches correspond aux différentes requêtes utilisateurs qui sont exprimées via une interface web donnant accès à des services web encapsulant les différentes actions appliquées sur la plateforme cloud.

Afin de mieux illustrer l'implémentation de ce middleware de déploiement, des classes java courtes implémentant les différentes interfaces seront aussi présentées.

2 Conception de MIDBOX

Conformément à notre objectif général visant à alléger les dispositifs mobiles en délégant le maximum des tâches intenses vers leurs clones, cette section est dédiée à la présentation des patrons de conception de notre middleware de déploiement. Le diagramme des classes associé à MIDBox présenté dans la figure 5.1 est issu de la réutilisation de l'architecture des applications collaboratives dans des environnements cloud qui a été présentée dans le chapitre précédent. Il vise à étendre l'architecture de base dans le but d'offrir un ensemble de packages contenant un ensemble de classes (codées en java) qui implémentent à leur tour les différentes interfaces des différents patrons.

L'objectif de cette réutilisation est de concevoir, développer et mettre en œuvre le middleware de déploiement MIDBox qui agit sur l'hyperviseur de virtualisation VirtualBox[8] (cloud privé) afin de préparer une plateforme virtuelle de collaboration qui est essentiellement composée de clones des mobiles et des réseaux privés virtuels. Afin de mettre en évidence cette extension, les différents packages qui lui sont associés sont représentés avec une couleur de fond grise dans la figure 5.1.

Les classes ajoutées à cette architecture réutilisée implémentent les différentes interfaces par redéfinition de leurs méthodes de base afin de les adapter à notre hyperviseur de virtualisation VirtualBox. Par exemple, la classe *CréerMachineVirtuelleAndroid* redéfinit la méthode *CréerClone* de l'interface *GestionClonage* pour créer une nouvelle machine Android sur VirtualBox.

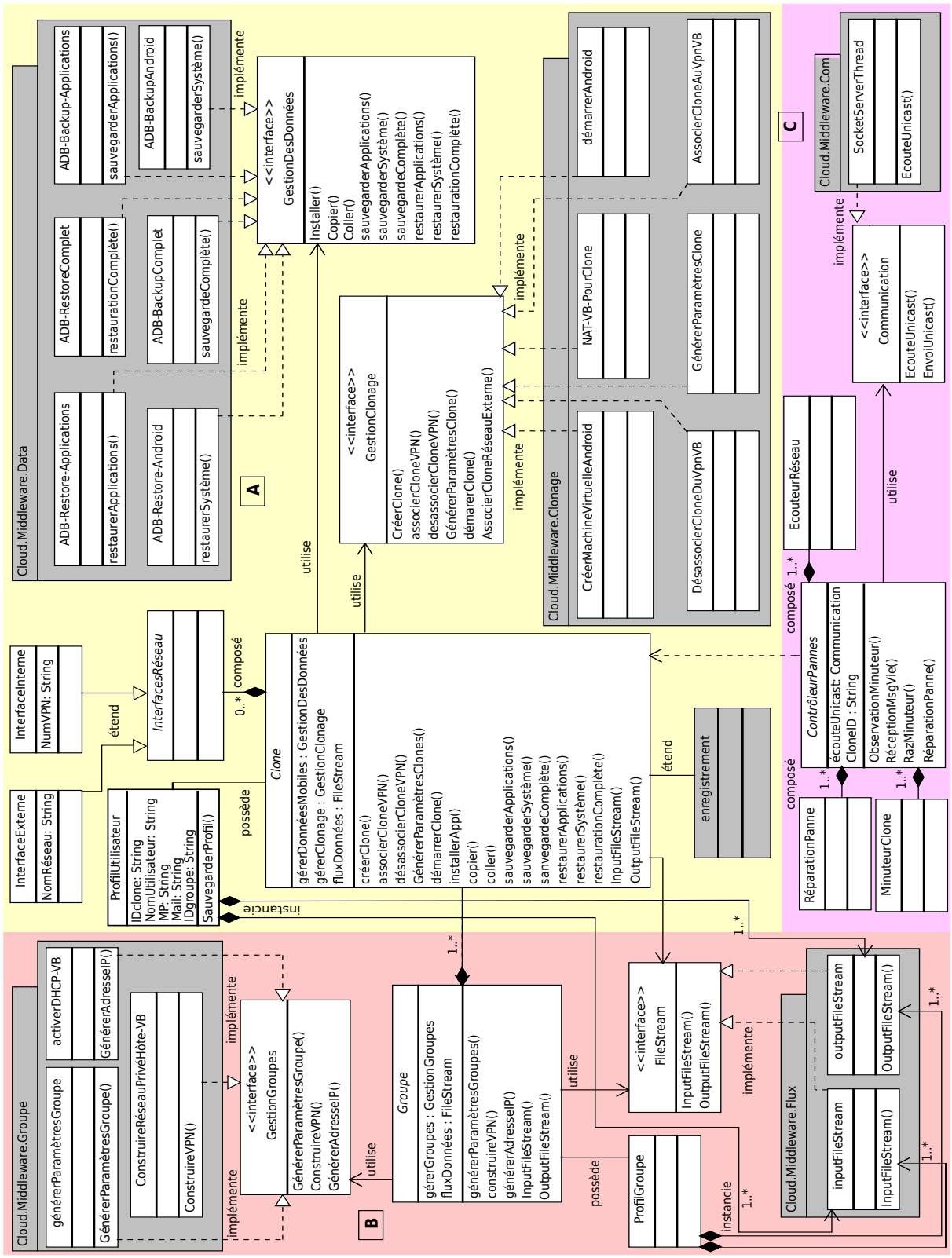


FIGURE 5.1 – Diagramme de classes du middleware de déploiement.

Ainsi, ces classes sont regroupées dans les principaux packages suivants :

(i) *Cloud.Middleware.Clonage*. Offre des classes dédiées à la gestion du cycle de vie des machines virtuelles Android sur VirtualBox. Ceci comprend la création, la configuration réseau, le démarrage et l'intégration du clone à son groupe.

Notons que les processus autonomes associés à la gestion des pannes clones (figure 5.1(C)), sont considérés comme un complément des tâches effectuées par les classes de ce package et contribuent à la gestion d'un cycle de vie complet pour chaque clone. En effet, le contrôleur des pannes veille au bon fonctionnement du système dans sa globalité via un processus d'observation continue menant à la détection de différentes situations de pannes. Dans le cas d'une telle détection de panne, ce contrôleur fait appel aux classes de ce package afin de créer une nouvelle instance du clone en échec.

(ii) *Cloud.Middleware.Groupe*. Les classes de ce package sont utilisées pour gérer les réseaux privés virtuels qui regroupent des machines virtuelles Android du même groupe.

(iii) *Cloud.Middleware.Data*. Comprend des classes destinées à effectuer des sauvegardes et des restaurations optionnelles des données entre les mobiles et leurs clones.

Le tableau 5.1 présente un résumé des différentes tâches dédiées aux différentes classes de ces packages. La description détaillée, ainsi que l'utilisation des différentes classes des packages au niveau des processus du middleware de déploiement seront illustrées par des diagrammes d'actions dans la section suivante.

3 Protocole de déploiement

Notre protocole de déploiement implique un serveur web, un ensemble d'utilisateurs mobiles et un ensemble de clones (ou des machines virtuelles). Chaque utilisateur mobile possède un clone approprié pour son mobile dans le cloud. Un clone est caractérisé par ses caractéristiques de machine (par exemple, nombre de cœurs, la fréquence du processeur, la taille de la mémoire), et une image virtuelle à mettre en œuvre (c-à-d, le système d'exploitation et les applications collaboratives). La communication des clones est effectuée à travers des VPNs. Afin de préparer cette plateforme de collaboration virtuelle, MIDBox offre les services suivants:

3.1 Services de clonage

Le processus de clonage constitue le cœur du protocole de déploiement. Ce processus met en évidence trois services web, à savoir, *enregistrement*, *changementGroupe* et *quitterGroupe* permettant respectivement à un utilisateur de : (i) s'abonner au service cloud proposé afin de cloner son mobile tout en rejoignant un groupe existant ou en créant un nouveau groupe, (ii) changer de groupe de collaboration ou (iii) le quitter. Chaque service web prend en entrée des

paramètres d'appel et retourne des résultats à l'utilisateur. Le tableau 5.2 présente une description générale de ces trois services.

Package	Classes	Description
Cloud.Middleware.Clonage implémente l'interface GestionClonage	*CréerMachineVirtuelleAndroid*	Créer une machine virtuelle Android
	GénérerParamètresClone	Générer les paramètres de profil clone
	NAT-VB-PourClone	Associer une interface réseau d'un clone à un réseau externe
	AssocierCloneAuVpnVB	Associer une interface réseau à un VPN VirtualBox
	DésassocierCloneDuVPN	Désassocier une interface réseau depuis un VPN VirtualBox
	DémarerAndroid	Démarrer une machine virtuelle Android dans VirtualBox
Cloud.Middleware.Groupe implémente l'interface GestionGroupes	*ConstruireRéseauPrivéHôte-VB*	Construire un réseau privé virtuel dansVirtualBox
	GénérerParamètresGroupe	Générer les paramètres du profil groupe d'utilisateurs
	ActiverDHCP-VB	Activer l'allocation dynamique des adresses IP sur un VPN
Cloud.Middleware.Data implémente l'interface GestionDesDonnées	*ADB-BackupApplications*	Sauvegarder des applications mobiles
	ADB-BackupAndroid	Sauvegarder le S.E Android
	ADB-BackupComplet	Réaliser une sauvegarde complète
	ADB-RestaureComplet	Réaliser une restauration complète
	ADB-RestaureApplication	Restaurer des applications mobiles
	ADB-RestaureAndroid	Restaurer le S.E Android
Cloud.Middleware.Flux implémente l'interface FileStream	*InputFileStream*	Récupérer les objets dérivant les profils utilisateurs et groupes
	OutputFileStream	Sauvegarder les objets dérivant les profils utilisateurs et groupes
Cloud.Middleware.Com implémente l'interface Communication	*SocketServerThread*	Écouter en mode Unicast, afin de recevoir les messages de vie des clones

TABLE 5.1 – Description des classes implémentant les interfaces des patrons du middleware de déploiement.

Service web	Paramètres d'entrées		Sortie	
enregistrement Service web d'abonnement	*nomUt*	nom d'utilisateur	*msg-Confirm*	message de confirmation du clonage
	mP	mot de passe d'authentification		
	mail	adresse mail utilisateur	*IP-clone*	adresse IP clone envoyée au mobile servant à une connexion directe
	IP-Mobile	adresse IP actuelle du dispositif mobile		
	choixGroupe	choix du groupe d'appartenance		
changementGroupe Service web pour changer un groupe	*ID-AncienGroupe*	identificateur du groupe à changer	*msg-Confirm*	message de confirmation du changement du groupe
	ID-NouvGroupe	identificateur du nouveau groupe à rejoindre		
quitterGroupe Service web pour quitter un groupe	*ID-Groupe*	identificateur du nouveau groupe à rejoindre	*msg-Confirm*	message de confirmation sur l'abandon du groupe

TABLE 5.2 – Description des services web associés au processus de clonage

3.1.1 Service web d'abonnement

Dans le cas d'abonnement, un nouvel utilisateur peut créer un clone pour son mobile en appelant le service web *enregistrement*. Ainsi, le processus de clonage se déroule dans les étapes suivantes (tâches délimitées par des tirets pointillés bleus dans la figure 5.2) :

(1) Génération des paramètres clone. Cette action est réalisée par une instanciation de la classe *GénérerParametresClone* qui implémente la classe d'interface *GestionClonage*. Ces informations générées seront rassemblées dans un seul objet issu de l'instanciation de la classe *ProfilUtilisateur* et sont en provenance de deux sources. D'une part, une partie des données du profil est introduite par l'utilisateur comme paramètres d'entrée du service web appelé (c-à-d, le service web *enregistrement*), à savoir le nom d'utilisateur, mot de passe, adresse e-mail et l'adresse IP du mobile. L'autre partie des informations concerne les différents paramètres du clone à créer et sont automatiquement générés durant cette phase de ce processus (identificateur du clone, nom de machine virtuelle, adresse IP et numéro du VPN d'appartenance) ou fournies par le clone après son démarrage (adresse IP du clone).

Une fois cette phase achevée, l'utilisateur est invité à confirmer son accord pour une sauvegarde des données et applications de son mobile.

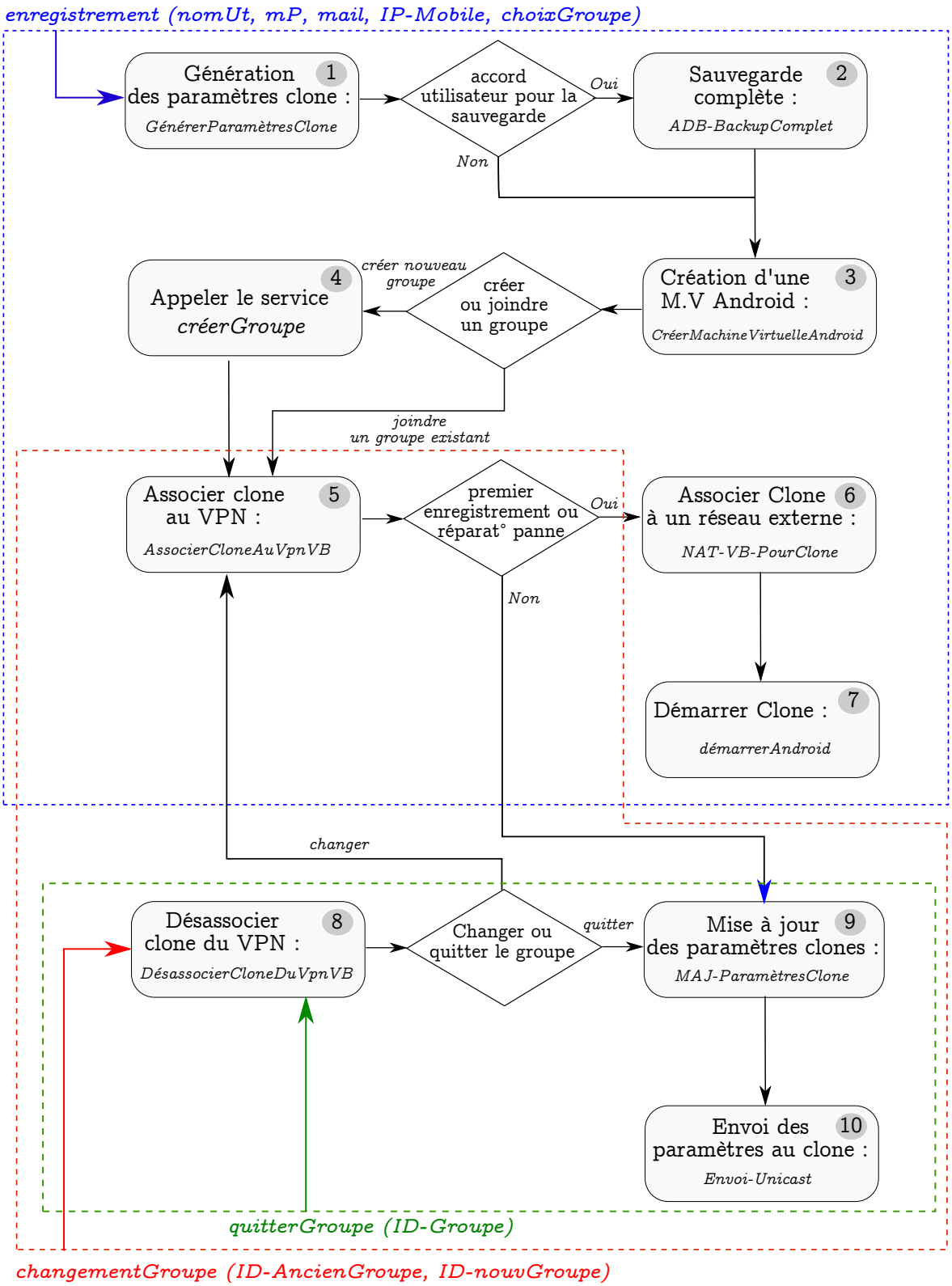


FIGURE 5.2 – Architecture des services web encapsulant le processus de clonage.

(2) Sauvegarde des données et applications du mobile. Ce service optionnel propose plusieurs types de sauvegarde : complète, applications/données mobiles et système android.

Ces sauvegardes sont respectivement effectuées à travers une instanciation des classes *ADB-BackupComplet*, *ADB-Backup-Applications* et *ADB-BackupAndroid* qui implémentent l'interface *GestionDesDonnées*. Dans le cas d'une sauvegarde complète le clone créé sera une copie exacte du dispositif mobile réel. Plus de détails concernant cette phase seront présentés dans la sous-section 3.4.

(3) Création d'une machine virtuelle Android. Pour chaque mobile, MIDBOX construit une nouvelle machine virtuelle Android x86 dans le cloud. Dans notre approche, au lieu de créer cette machine virtuelle à partir de zéro en utilisant un fichier ISO (c-à-d, un fichier contenant une image d'une MV android non configurée), nous avons choisi d'importer une machine virtuelle pré-configurée. Ce choix est motivé par le fait que le déploiement des clones sera facile et très rapide. Une application collaborative initiale est également incorporée dans la machine virtuelle. Cette dernière est équipée d'une pré-configuration permettant le démarrage automatique de l'application collaborative avec le premier démarrage du clone.

Notons que le nouveau clone créé nécessite une configuration du réseau par une association des interfaces réseau virtuelles (c-à-d, cartes réseau virtuelles) d'un clone aux VPNs construits sur VirtualBox. D'une manière générale, cette association permet d'effectuer des tâches de communication internes et privées entre clones à travers des réseaux privés virtuels ou d'accéder aux machines virtuelles depuis l'extérieur (c-à-d, depuis le mobile).

Le listing 5.1 présente la classe *CréerMachineVirtuelleAndroid* qui implémente l'interface *GestionClonage* par redéfinition de sa méthode *CréerClone* pour créer une machine virtuelle Android sur VirtualBox.

(4) Appeler le service *créerGroupe*. Pour associer une interface réseau du clone créé à un réseau privé virtuel (VPN), le processus de clonage prend en considération le choix d'appartenance de l'utilisateur à un groupe de collaboration qui est introduit par l'utilisateur comme paramètre d'appel du service web d'abonnement. Dans le cas correspondant au choix de création d'un nouveau groupe et avant de procéder à la phase d'association d'une interface réseau clone à un réseau privé virtuel interne, le processus de clonage fait appel au service *créerGroupe* afin de construire un nouveau VPN. Plus de détails concernant ce service de construction des VPNs sont illustrés au niveau de la section 3.2.

(5) Associer clone au VPN. Dans le cas d'un choix d'appartenance à un groupe existant, le processus de clonage ignore la phase précédente. Il instancie la classe *AssocierCloneAuVpnVB* qui implémente l'interface *GestionClonage* afin d'associer une interface réseau du clone à un VPN existant et qui correspond au choix de l'utilisateur. Notons que chaque clone possède plusieurs interfaces réseau (cartes réseau virtuelles) qui sont identifiées par des numéros d'ordre (*numInterface*). En l'occurrence, il faut préciser l'interface réseau concernée par une association à un VPN en indiquant son numéro d'ordre comme paramètre d'appel.

Listing 5.1 – Création d'une MV Android sur VirtualBox

```
1  package Cloud.Middleware.Clonage;
2  //Section Import
3  public class CréerMachineVirtuelleAndroid implements
       GestionClonage{
4   public void CréerClone () { //Redéfinition de la méthode
       CréerClone de l'interface GestionClonage
5      //Paramètres du processus de création d'une MV Android.
6          String vbm="VboxManage",
7          imp="import",
8          fichierImport="CloneAndroid.OVA",
9          line;
10     //processus de création d'une MV Android par importation
          d'une machine préconfigurée.
11         try {
12               String[] argVmCreate = {vbm, imp, fichierImport};
13               ProcessBuilder proc =
14              new ProcessBuilder(argVmCreate);
15               Process pr = proc.start();
16               pr.waitFor();
17               BufferedReader br = new BufferedReader (new
                    InputStreamReader(pr.getInputStream()));
18               while ( (line = br.readLine()) != null);
19               } catch (Exception e) {
20                          System.err.println("Error");
21                                 }
22   }
23 }
```

Le listing 5.2 présente un code java permettant d'associer une interface réseau connue par son numéro *numInterface* à un VPN identifié par son identifiant *numVPN*.

(6) Associer clone à un réseau externe. Après achèvement de la phase précédente, le processus de clonage procèdera à l'association d'une autre interface réseau du clone à un réseau externe afin de permettre un accès depuis l'extérieur (depuis le mobile). Cette association est obtenue à travers une instanciation de la classe *NAT_VB_PourClone* qui implémente la classe d'interface *GestionClonage* afin de définir des règles de redirection de ports utilisés par un réseau de type NAT (c-à-d, traduction d'adresse réseau. En anglais, Network Address Translation). Ces règles sont basées sur un mécanisme de traduction d'adresses IP. Avec ce mode réseau, VirtualBox est en écoute permanente sur certains ports sources définis dans les règles de redirection. Suite à l'identification de l'émetteur par son adresse IP, VirtualBox redirige les paquets reçus vers la

machine virtuelle cible identifiée par son adresses IP et le port destination qui sont définis dans ces règles de redirection de ports.

Listing 5.2 – Association d'une interface réseau d'un clone à un VPN

```
1  package Cloud.Middleware.Clonage;
2  //Section Import
3  public class AssocierCloneAuVpnVB implements GestionClonage{
4   public void AssocierCloneVPN (String numVPN, nomClone; int
       numInterface) {
5      String VPN = "VirtualBox Host-Only Ethernet Adapter
          #"+String.valueOf(numVPN); //VPN d'asociation
6      String InterfaceRéseau="--hostonlyadapter"+
          String.valueOf(numInterface); //interface réseau clone, à
          associer au VPN
7        try {
8            //processus d'association d'une interface réseau
                d'un clone à un VPN :
9            String[] argProcess = {vbm, modif, nomClone,
                InterfaceRéseau, VPN};
10           ProcessBuilder proc = new ProcessBuilder(argProcess);
11           Process pr = proc.start();
12           pr.waitFor();
13           BufferedReader br = new BufferedReader(new
                InputStreamReader(pr.getInputStream()));
14           while ((line = br.readLine()) != null)
15           System.out.println(line);
16           } catch (Exception e) {
17               System.err.println("Error");
18           }
19   }
20 }
```

(7) Démarrer clone. Cette action permet de démarrer la machine virtuelle créée, initialiser ses interfaces réseau et auto-démarrer l'application collaborative. Un message de confirmation est alors envoyé à l'utilisateur pour se connecter à son clone mobile. Rappelons que la nouvelle instance du clone déployée, est équipée d'une pré-configuration lui permettant d'exécuter les premières tâches requises juste après son démarrage. Il va appeler un service de MIDBOX afin de fournir son adresse IP et récupérer les paramètres nécessaires au travail collaboratif. Cette action est considérée comme une tâche d'initialisation du protocole collaboratif qui consiste à intégrer le nouveau clone à son groupe de collaboration dans le cloud.

Listing 5.3 – Lancement de la MV Android sur VirtualBox

```
1  package Cloud.Middleware.Clonage;
2  //Section Import
3  public class démarrerAndroid implements GestionClonage{
4   public void démarrerClone (String nomClone) {
5   String start="startvm";
6                   try {
7                       //processus de lancement du clone :
8                       String[] argProcess = {vbm, start, nomClone};
9                       ProcessBuilder proc = new
                           ProcessBuilder(argbackup);
10                      Process pr = proc.start();
11                      pr.waitFor();
12                      BufferedReader br = new BufferedReader(new
                           InputStreamReader(pr.getInputStream()));
13                      while ((line = br.readLine()) != null)
14                          System.out.println(line);
15                  } catch (Exception e) {
16                      System.err.println("Error");
17                  }
18    }
19 }
```

3.1.2 Services web pour changer ou quitter un groupe

Dans le cas d'un service web lié à une requête utilisateur de changement du groupe, un utilisateur peut exprimer sa demande en appelant le service web *changementGroupe(ID_AncienGroupe, ID_nouvGroupe)*. Le processus de clonage est alors déclenché en exécutant les tâches suivantes (tâches délimitées par des tirets pointillés rouges dans la figure 5.2) :

(8) Désassocier clone du VPN. Cette désassociation est effectuée à travers l'instanciation de la classe *DésassocierCloneDuVpnVB* qui implémente l'interface *GestionClonage* et elle permet de libérer l'interface réseau clone du VPN correspondant à l'ancien groupe. Ce dernier est indiqué par l'utilisateur comme paramètre d'entrée dans l'appel du service web via son identificateur (*ID_AncienGroupe*). Une fois l'interface réseau libérée, elle sera à nouveau associée à un autre VPN correspondant au nouveau groupe ayant comme identificateur le paramètre *ID_nouvGroupe* (étape(5)), introduit par l'utilisateur lors de l'appel du présent service web.

(9) Mise à jour des paramètres clones. Cette phase consiste à mettre à jour l'ensemble des différents paramètres enregistrés dans l'objet $ProfilClone$, à savoir l'identificateur du groupe et l'adresse de diffusion au sein de ce nouveau VPN.

(10) Envoi des paramètres au clone. Les nouveaux paramètres seront envoyés au clone afin de servir aux tâches de collaboration au niveau du nouveau VPN.

D'autre part, le service web permettant à un utilisateur de quitter un groupe est presque identique au service précédent, mais il consiste seulement à exécuter les phases 8, 9 et 10 (tâches délimitées par des tirets pointillés verts dans la figure 5.2).

3.2 Service de gestion des réseaux privés virtuels

La deuxième composante de MIDBOX est le service web de gestion des VPNs. Il est utilisé pour configurer la plateforme de collaboration qui permet aux clones de démarrer leurs travaux d'édition collaborative. Comme illustré dans la figure 5.3, trois phases sont nécessaires, à savoir : la préparation d'un nouveau VPN, la définition et l'activation d'un serveur d'allocation dynamique d'adresses IP (ou serveur DHCP) pour le VPN créé, et la génération des paramètres de groupe. Ces phases sont encapsulées dans le service web nommé *CréerGroupe(NomGroupe)*. Il prend en entrée le nom du groupe (*NomGroupe*) à créer (introduit par l'utilisateur) comme paramètre et retourne en sortie un message de notification confirmant le succès de la construction du VPN. Le déclenchement de ce constructeur implique l'enchaînement de l'exécution des phases suivantes :

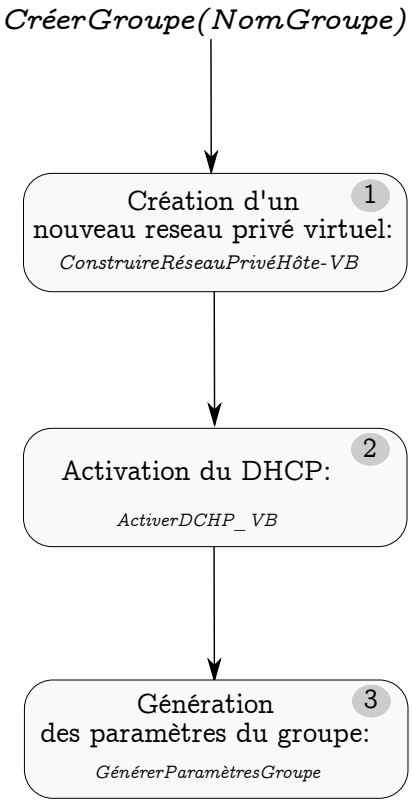


FIGURE 5.3 – Architecture du service web encapsulant le processus de construction des VPNs.

(1) Création d'un nouveau réseau privé virtuel. Le but de cette tâche est de construire un nouveau réseau virtuel dans VirtualBox qui propose les modes de réseautage privé suivants :

— Réseau interne. Ce mode permet de créer un type de réseau basé-logiciel, qui est visible pour les machines virtuelles sélectionnées (membres du VPN) mais pas aux applications exécutées sur l'hôte ou sur le monde extérieur.

— Réseautage privé hôte. Ceci peut être utilisé pour créer un réseau comportant l'hôte et un ensemble de machines virtuelles sans la nécessité d'une interface réseau physique de l'hôte. Comme le mode précédent, les machines virtuelles sont inaccessibles depuis l'extérieur via un tel réseau.

— UDP Tunnel. Ce mode peut être utilisé pour interconnecter des machines virtuelles s'exécutant sur différentes machines hôtes d'une manière directe et transparente. Bien que ce mode soit plus adapté à un système évolutif, il nécessite l'existence d'une infrastructure réseau convenable (c-à-d, plusieurs serveurs et équipements de connexion réseau).

Listing 5.4 – Création d'un VPN de type "hôte privé " dans VirtualBox

```
1  package Cloud.Middleware.Groupe;
2  //Section Import
3  public class ConstruireRéseauPrivéHôte_VB implements
       GestionGroupes{
4   public void ConstruireVpn () {
5       String vbm="VboxManage";
6           String TypeVPN="hostonlyif"; //type VPN : privé hôte
7           String commande="create";
8           String line;
9           try {
10              //processus de construction VPN :
11              String[] argbackup = {vbm, TypeVPN, commande};
12              ProcessBuilder proc = new ProcessBuilder(argbackup);
13              Process pr = proc.start();
14              pr.waitFor();
15              BufferedReader br = new BufferedReader(new
                    InputStreamReader(pr.getInputStream()));
16             numVPN ++; //incrémentation du compteur des VPNs
17                 while ( (line = br.readLine()) != null)
18                         System.out.println(line);
19                       } catch (Exception e) {
20                           System.err.println("Error");
21                       }
22   }
23 }
```

Vu la simplicité de leur création, nous avons opté pour le choix de la construction des VPNs via le mode privé hôte. Ceci est effectué à travers une instanciation de la classe *ConstruireRéseauPrivéHôte_VB* qui implémente l'interface *GestionGroupes*. En l'occurence, VirtualBox créera un nouveau réseau privé virtuel et lui attribuera un nouveau nom de la forme : "*VirtualBox Host-Only Ethernet Adapter* ***#numéro***" où ***numéro*** est un numéro croissant de type entier.

Le listing 5.4 montre le code de la classe *ConstruireRéseauPrivéHôte_VB* qui redéfinit la méthode *ConstruireVpn* de l'interface *GestionGroupes*.

(2) Configuration de l'espace d'adressage. Un serveur DHCP (en anglais, Dynamic Host Configuration Protocol) est activé dans le but de dynamiquement allouer des adresses IP aux interfaces réseaux des machines virtuelles qui sont associées à un VPN donné. Ainsi, il nous permet de définir le masque de réseau et des plages d'adresses IP limitées par des bornes inférieure/supérieure. En outre, il fournit une adresse de diffusion spécifique au VPN afin de faciliter la communication entre les clones affiliés à un même groupe et optimiser le trafic réseau. La génération des plages d'adresses IP et de l'adresse de diffusion associées à chaque VPN est effectuée via l'instanciation de la classe *activerDHCP_VB* qui implémente l'interface *GestionGroupe*. Comme le montre la figure 5.4 et le listing 5.5, dans nos expérimentations nous activons des DHCPs pour des VPNs pouvant accueillir jusqu'à 250 machines virtuelles Android membres sur la même machine hôte. Ceci est réalisé via la variation de la dernière partie de l'adresse IP de 2 à 253 (le nombre 254 de la dernière partie de l'adresse IP est exclusivement réservé à l'adresse de diffusion). Afin d'éviter toute confusion entre les adresses IP des VPNs, la troisième partie de l'adresse IP est fixée au numéro du VPN (*numVPN*).

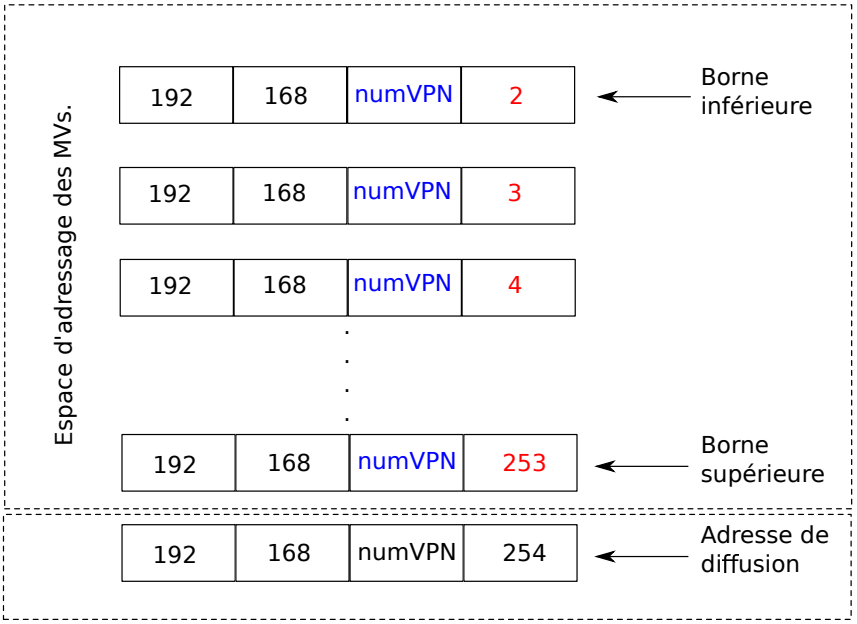


FIGURE 5.4 – Configuration de l'espace d'adressage d'un VPN.

Listing 5.5 – Activation d'un serveur DHCP sur un VPN

```
1  package Cloud.Middleware.Groupe;
2  //Section Import
3  public class ActiveDHCP-VB implements GestionGroupes{
4   public void GénérerAdresse-IP (String numVPN) {
5      String dhcp="dhcpserver";
6      String modif="modify";
7      String name="--ifname";
8      String interf="VirtualBox Host-Only Ethernet Adapter
         #"+String.valueOf(numVPN);//nom du VPN
9      String ip="--ip";
10     String
         adresseBroadcast="192.168."+String.valueOf(numVPN)+".254";
         //adresse de diffusion
11     String option1="--netmask";
12     String maskRéseau="255.255.255.0"; //masque du réseau
13     String option2="--lowerip";
14     String borneINF="192.168."+String.valueOf(numVPN)+".2";
         //borne inférieure des adresses ip
15     String option3="--upperip";
16     String BorneSup="192.168."+String.valueOf(numVPN)+".253";
         //borne supérieure des adresses ip
17     String activationOk="--enable";
18     try {
19         // processus DHCP :
20         String[] argbackup = {vbm, dhcp, modif, name, interf,
             ip, adresseBroadcast, option1, maskRéseau, option2,
             borneINF, option3, BorneSup, activationOk};
21         ProcessBuilder proc = new ProcessBuilder(argbackup);
22         Process prr = proc.start();
23         prr.waitFor();
24         BufferedReader br2 = new BufferedReader(new
             InputStreamReader(prr.getInputStream()));
25         while ((line = br2.readLine()) != null)
26         System.out.println(line);
27         } catch (Exception e) {
28                                 System.err.println("Error");
29                                         }
30   }
31 }
```

(3) Génération des paramètres du groupe. À cette phase, le VPN est entièrement construit. Le constructeur VPN procédera à la génération des paramètres nécessaires à la gestion du groupe de collaboration. Ainsi, le nom du groupe introduit comme paramètre d'appel via ce service, l'adresse de diffusion générée durant la phase précédente et le numéro d'ordre associé à chaque VPN (*numVPN*) seront stockés dans un seul objet issu de l'instanciation de la classe $ProfilGroupe$. Rappelons que chaque clone futur membre de ce groupe créé, recevra ses paramètres pour son identification au niveau du groupe.

3.3 Contrôle autonome des pannes

La troisième composante de MIDBOX est l'auto-contrôleur des pannes. Il sert à détecter et réparer des situations de défaillance qui peuvent affecter le bon fonctionnement de notre service collaboratif. Plus précisément, ce contrôleur prend à sa charge la réparation des dysfonctionnements impactant la réussite du déploiement et la vivacité des clones au sein de leurs groupes de collaboration d'une manière transparente et autonome. Nous traitons trois situations de défaillances visant à maintenir l'intégrité de la plateforme de collaboration :

— Défaillance de l'application collaborative : ce problème est dû à l'occurrence des exceptions ou erreurs logicielles interrompant tout échange des messages depuis et vers le clone concerné.
— Défaillance d'un clone : cet échec provient du logiciel de virtualisation ou du système d'exploitation local ; il affectera le comportement normal du clone qui devient inaccessible à toute interaction avec le middleware de déploiement ou d'autres clones.
— Panne de communication transitoire : ce problème trouble la communication à l'intérieur du réseau pour un temps fini mais arbitraire. Par exemple, cette défaillance peut être due à la perte des messages.

Quelle que soit la situation d'échec décrite ci-dessus, elle mène potentiellement à l'échec du clone. En outre, la détection et la réparation de la panne du serveur MIDBOX est en dehors de la portée de ce travail. Dans ce cas, nous supposons qu'un middleware en échec sera remplacé par un autre middleware esclave. Le principe de cette restauration de MIDBOX est simple. Il consiste à effectuer une réplication instantanée des données et des applications du maître sur un ou plusieurs esclaves. Le maître émet régulièrement des messages aux autres serveurs passifs (c-à-d, les esclaves) afin d'indiquer son bon fonctionnement. Si plus aucun de ces messages n'arrive, un des esclaves prend alors le rôle du middleware actif.

Comme le montre la figure 5.5, ce contrôleur présente deux processus de gestion des pannes, à savoir un écouteur réseau et un détecteur/réparateur des pannes. Ces deux processus utilisent une liste de minuteurs comme une structure de données partagée où chaque entrée correspond à un clone.

Écoute réseau. Pour indiquer un comportement correct, chaque clone doit périodiquement (avec des intervalles de temps égaux) envoyer un message de vie (*msgVie*) au contrôleur des

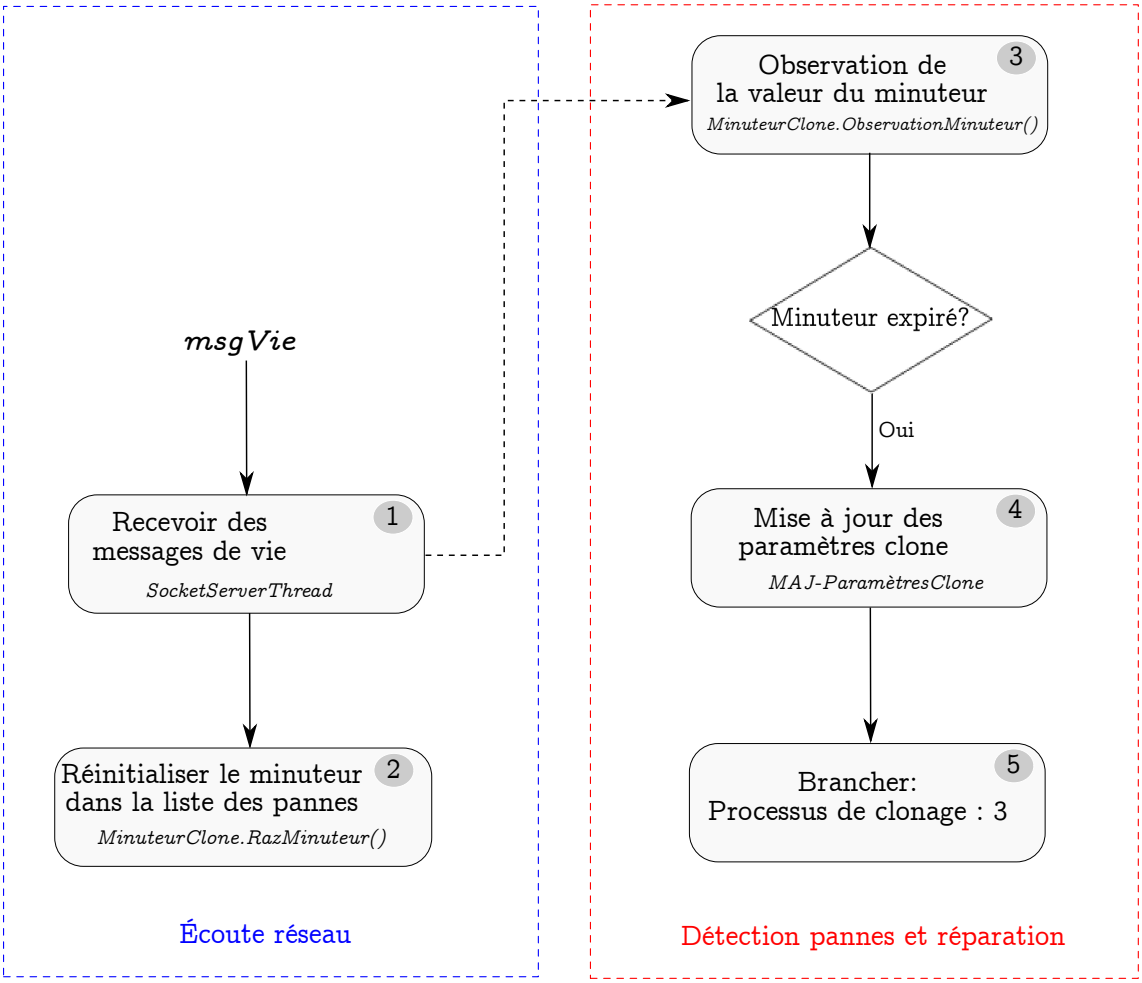


FIGURE 5.5 – Architecture du processus d'auto-contrôle des pannes.

pannes. Ce dernier utilise des instances d'un thread écouteur recevant des messages de vie (par similitude avec un battement de cœur) en provenance des différents clones par instanciation de la classe *SocketServerThred* qui implémente l'interface *communication*. Après chaque réception d'un message de vie, le contrôleur des pannes réinitialisera le minuteur correspondant au clone émetteur dans la liste des minuteurs par instanciation de la classe *MinuteurClone*.

Détection des pannes et réparation. Ce processus est impliqué dans l'observation de la liste des minuteurs afin de détecter tout dépassement de temps d'attente exigé pour chaque clone pour l'envoi de ses messages de vie. À cette fin, un thread observateur est associé à chaque entrée dans la liste des minuteurs par instanciation de la classe *MinuteurClone*. Ainsi, une situation de panne est détectée lorsqu'un minuteur est expiré. Dans ce cas, une mise à jour des paramètres du clone est effectuée afin de repérer le clone en échec et une réparation de la panne est lancée. Cette réparation consiste à faire appel au processus de clonage (par branchement à sa troisième étape) afin de créer une nouvelle instance du clone en échec, tout en restaurant ses paramètres de configuration. En l'occurrence, cette réinstanciation consiste à exécuter les étapes suivantes

du processus de clonage : (3) Création d'une nouvelle machine virtuelle android, (5) réassocier le clone à son ancien VPN, (6) associer le clone au réseau externe et (7) le démarrer.

3.4 Sauvegardes des données mobiles

La création de véritables clones des mobiles dans le cloud n'est pas une tâche facile pour des utilisateurs non-experts. Dans notre approche, le clonage des mobiles ne devrait pas être limité (seulement) à un traitement de déploiement ou de stockage des fichiers utilisateurs. Il doit permettre : (i) la sauvegarde des paramètres du système d'exploitation (c-à-d, Android), (ii) le paramétrage et la sauvegarde des applications mobiles et (iii) la gestion des transferts des fichiers utilisateurs.

Au sein de notre middleware de déploiement, nous proposons d'implémenter une solution pour créer automatiquement une copie complète ou partielle des mobiles. L'utilisateur sera simplement invité à confirmer son accord et choisir l'option de sauvegarde. Dans le cas d'une panne ou une perte du dispositif réel, MIDBOX propose une restauration complète, facile et automatique.
Afin de fournir ce service de sauvegarde, nous avons utilisé le "*Android Debug Pont*" (ADB) [1], qui permet aux utilisateurs de communiquer avec des émulateurs ou dispositifs Android à travers des lignes de commande. C'est un programme client-serveur qui regroupe des outils pour le déploiement/récupération des données depuis/vers des mobiles et comprend trois composantes :

— *Un client*, qui s'exécute sur un serveur web hébergeant le middleware de déploiement. Dans notre implémentation, l'invocation du client *ADB* est effectuée en utilisant des commandes enveloppées dans les services web correspondant aux différentes tâches de sauvegarde.
— *Un serveur* qui s'exécute comme un processus en arrière-plan sur le middleware. Il gère la communication entre un *client ADB* et un *daemon ADB* s'exécutant sur un clone ou sur son mobile.
— *Un daemon* qui s'exécute comme un processus en arrière-plan sur chaque clone, ainsi que sur son mobile. C'est une partie intégrante du système d'exploitation Android.

La sauvegarde des données et des applications mobiles, ainsi que de la version du système d'exploitation Android (installée sur un mobile) est obtenue en exécutant une série de commandes *ADB*. Chaque commande peut être définit avec plusieurs options de sauvegarde. Les processus implémentant ces commandes seront lancés à partir de services web. La figure 5.6 montre les étapes nécessaires pour effectuer une sauvegarde d'un mobile qui sont décrites comme suit :

1. Après avoir effectué une requête de clonage avec sauvegarde des données mobiles comme option, un *client ADB* dédié au mobile est invoqué au niveau du middleware de déploiement pour établir une connexion avec le *Daemon ADB* au niveau de ce mobile.

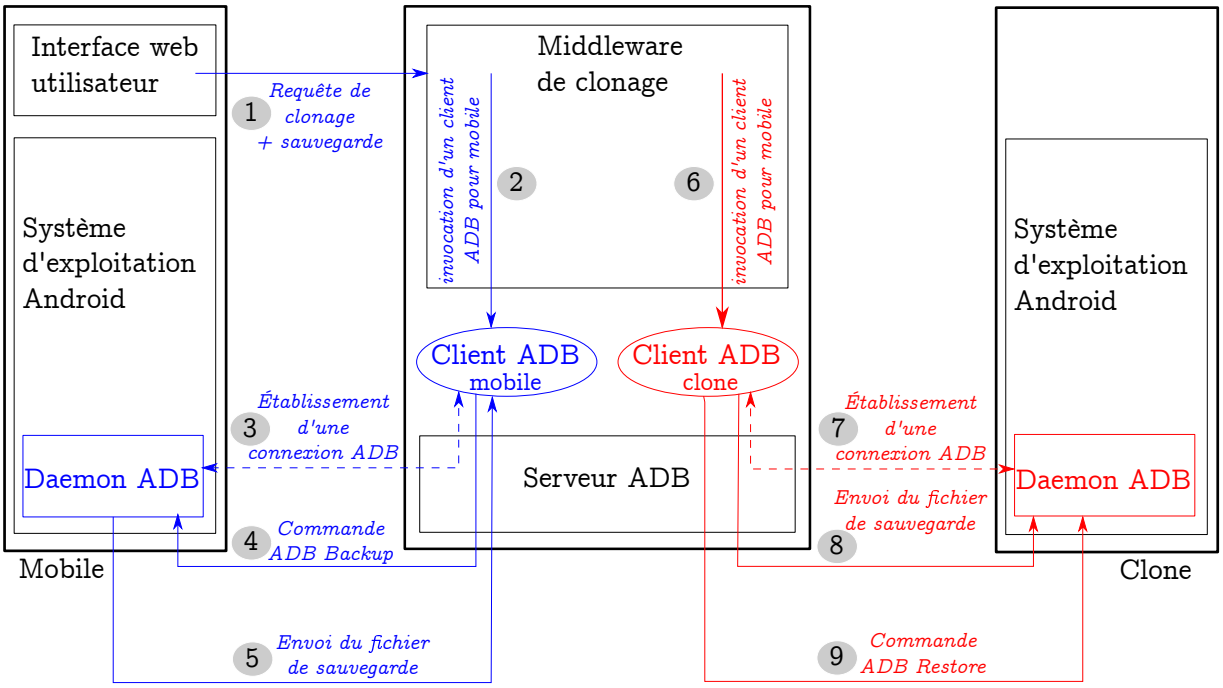


FIGURE 5.6 – La sauvegarde des données mobiles au sein du middleware de clonage.

2. Le *serveur ADB* gère la connexion entre le *client ADB* invoqué au niveau du middleware de déploiement et le *démon ADB* du mobile. Il utilise l'adresse IP du mobile pour établir une connexion TCP sans fil. À noter que l'utilisateur sera invité à modifier certains paramètres du système d'exploitation afin de permettre l'écoute *ADB* en mode TCP.
3. Une commande de sauvegarde est ensuite envoyée au *daemon ADB* du mobile. Cette commande est présentée à l'utilisateur comme un menu contenant les différentes options de sauvegarde disponibles. Il peut inclure des options pour l'enregistrement : des données système, des applications et de leurs statuts, des données SDCard [4] et des fichiers APK [5] des applications mobiles.
4. Selon le choix de l'utilisateur, le *daemon ADB* du mobile réagira par exécution de la commande de sauvegarde et l'envoi du fichier de sauvegarde vers le *serveur ADB* du middleware de déploiement qui va le stocker localement.
5. Un *client ADB* dédié au clone est invoqué au niveau du middleware de déploiement après exécution d'une commande demandant d'initier une connexion avec ce clone.
6. Le *serveur ADB* gère l'établissement de la connexion entre cette deuxième instance du *client ADB* et le *daemon ADB* du clone.
7. Une commande de restauration est ensuite envoyée au *daemon ADB* du clone afin de restaurer le contenu du fichier de sauvegarde qui a été précédemment envoyé par le mobile.

4. SDcard : Secure Digital card, une carte mémoire amovible pour stocker des données numériques.
5. Un APK (par exemple. "NomDuFichier.apk") est une collection de fichiers compressés ("package") du système d'exploitation Android.

Le listing 5.6 correspond à la classe *ADB_BackupComplet* qui implémente l'interface *GestionDesDonnées* par redéfinition de sa méthode *SauvegardeComplète* afin de procéder à une sauvegarde complète des données mobiles. Ceci correspond à la quatrième étape dans le processus de sauvegarde présenté dans la figure 5.6.

Listing 5.6 – Sauvegarde des données mobiles

```
1  package Cloud.Middleware.Data;
2  //Section Import ...
3  public class ADB_BackupComplet implements GestionClonage{
4   public void SauvegardeComplète (String nomClone) {
5              //déclaration des paramètres de sauvegarde complète
6     String adb="adb";
7     String commande="backup";
8     String arg1backup="-f";
9     String fichierSauvegarde="sauv.ab";
10    String arg2back="-apk";
11    String arg3back="-shared";
12    String arg4back="-all";
13    String arg5back="-system";
14    String line;
15               //processus de sauvegarde des données mobiles.
16    try {
17        String[] argbackup = {adb, commande, arg1backup,
             fichierSauvegarde, arg2back, arg3back, arg4back,
             arg5back};
18        ProcessBuilder proc = new ProcessBuilder(argbackup);
19        Process pr = proc.start();
20        pr.waitFor();
21        BufferedReader br = new BufferedReader(new
             InputStreamReader(pr.getInputStream()));
22        while ((line = br.readLine()) != null);
23        } catch (Exception e) {
24                    System.err.println("Error");
25                              }
26
27   }
28 }
```

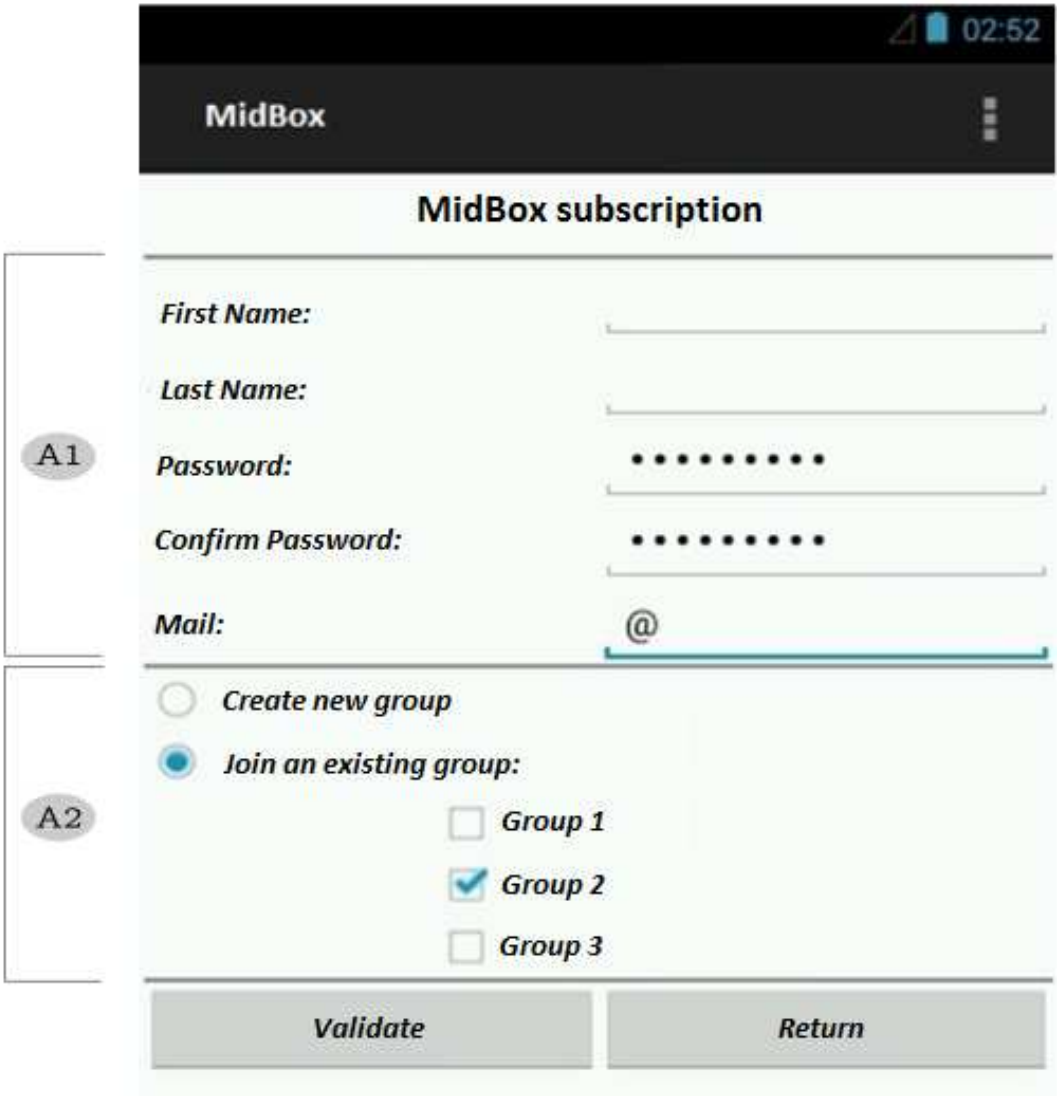


FIGURE 5.7 – Interface web de MIDBOX.

4 Implémentation de MIDBOX

MIDBOX est considéré comme une interface entre les utilisateurs et les applications collaboratives mobiles déployées dans le cloud. Nous avons utilisé le serveur web Axis2 [16] déployé dans le conteneur web tomcat7 [15] afin de mettre en œuvre les différents services de ce middeleware.

Comme le montre la figure 5.7, un service web de clonage est proposé afin de permettre une adhésion utilisateur en ligne via une interface web. Ce service invite l'utilisateur à introduire ses données de profil (partie A1 dans la figure 5.7) et son choix d'appartenance à un groupe de collaboration initial (partie A2). Comme déjà mentionné précédemment, ces informations correspondent aux paramètres d'entrée du service web d'abonnement.

```
B1  localhost:8080/axis2/services/webservfinal?wsdl

- <wsdl:definitions targetNamespace="http://lll.kkk.gg">
    <wsdl:documentation>webservfinal</wsdl:documentation>
  - <wsdl:types>
    - <xs:schema attributeFormDefault="qualified" elementFormDefault="qualified" targetNamespace="http://lll.kkk.gg">
B2    - <xs:element name="insert">
        - <xs:complexType>
          - <xs:sequence>
              <xs:element minOccurs="0" name="u2" nillable="true" type="xs:string"/>
              <xs:element minOccurs="0" name="mp2" nillable="true" type="xs:string"/>
B3            <xs:element minOccurs="0" name="mail2" nillable="true" type="xs:string"/>
              <xs:element minOccurs="0" name="choix2" type="xs:int"/>
              <xs:element minOccurs="0" name="ip2" nillable="true" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
B4    - <xs:element name="insertResponse">
        - <xs:complexType>
          - <xs:sequence>
              <xs:element minOccurs="0" name="return" nillable="true" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
```

FIGURE 5.8 – Fichier WSDL du service d'abonnement de MIDBOX.

Ce service web de clonage est déployé dans le serveur web Axis2 et il est basé sur le standard WSDL (Web Services Description Language). La figure 5.8 montre une partie du fichier WSDL décrivant les différents paramètres de ce service web, à savoir :

B1. L'adresse du service web de clonage : elle implique le chemin de déploiement du service web. Dans le cas de l'implémentation présentée dans la figure 5.8, ce service peut être appelé depuis le serveur web Axis2 via le port 8080 d'une machine locale (localhost:8080).

B2. Le nom de la méthode qui implémente les différentes phases nécessaires à la mise en œuvre du processus de clonage. Dans l'implémentation présentée, la méthode est appelée "*insert*" et elle définit l'ensemble des actions appliquées sur l'hyperviseur de virtualisation VirtualBox (cloud privé) afin de créer des clones et configurer leurs paramètres réseau.

B3. Les paramètres d'entrée du service web de clonage : ces paramètres sont fournis par l'utilisateur et comportent principalement l'identificateur utilisateur, le mot de passe, l'adresse e-mail, et le choix du groupe de collaboration initial.

B4. La réponse du service web de clonage : elle est retournée à l'utilisateur après achèvement du processus de clonage, en l'invitant à se connecter au clone de son mobile, et par conséquent gérer ses propres données et participer aux travaux collaboratifs avec les autres membres de son groupe.

D'autre part, la figure 5.9 présente l'état d'avancement du processus de clonage en répon-

C1
C2
C3
C4

```
C:\Program Files\Apache Software Foundation\Tomcat 7.0\bin\Tomcat7.exe
mars 20, 2015 2:44:39 PM org.apache.catalina.startup.Catalina start
INFOS: Server startup in 5800 ms
3
VirtualBox Host-Only Ethernet Adapter #3192.168.3.255
Interface 'VirtualBox Host-Only Ethernet Adapter #5' was successfully created
already connected to 192.168.1.3:5555
Now unlock your device and confirm the backup operation.
Disks:  vmdisk1 4612042752      -1      http://www.vmware.com/interfaces/specifi
cations/vmdk.html#streamOptimized       clone0403-disk1.vmdk    -1      -1

Virtual system 0:
 0: Suggested OS type: "Other_64"
    (change with "--vsys 0 --ostype <type>"; use "list ostypes" to list all poss
ible values)
 1: Suggested VM name "CloneAndroid_4"
    (change with "--vsys 0 --vmname <name>")
 2: Number of CPUs: 1
    (change with "--vsys 0 --cpus <n>")
 3: Guest memory: 1503 MB
    (change with "--vsys 0 --memory <MB>")
 4: Sound card (appliance expects "", can change on import)
    (disable with "--vsys 0 --unit 4 --ignore")
 5: USB controller
    (disable with "--vsys 0 --unit 5 --ignore")
 6: Network adapter: orig HostOnly, config 2, extra slot=0;type=HostOnly
 7: Network adapter: orig NAT, config 2, extra slot=1;type=NAT
 8: CD-ROM
    (disable with "--vsys 0 --unit 8 --ignore")
 9: IDE controller, type PIIX4
    (disable with "--vsys 0 --unit 9 --ignore")
10: IDE controller, type PIIX4
    (disable with "--vsys 0 --unit 10 --ignore")
11: Hard disk image: source image=clone0403-disk1.vmdk, target path=C:\Users\gue
tmin\VirtualBox VMs\CloneAndroid_4\clone0403-disk1_4.vmdk, controller=9;channel=
0
    (change target path with "--vsys 0 --unit 11 --disk path";
    disable with "--vsys 0 --unit 11 --ignore")
Waiting for VM "nadir" to power on...
VM "nadir" has been successfully started.
```

FIGURE 5.9 – Déroulement de l'exécution du processus de clonage sur la console du conteneur web TOMCAT.

dant à une requête utilisateur. Ce déroulement du processus est illustré sur la console du conteneur web TOMCAT :

C1. Création d'une nouvelle interface réseau Virtualbox de type privé hôte ("host-only"), cela signifie la création d'un nouveau réseau privé virtuel pour communiquer des clones d'un même groupe de collaboration (phase 1 et 2 dans le service de gestion des VPNs, voir la section 3.2).

C2. Sauvegarde des données mobiles (étape 2 dans le processus de clonage, voir la section 3.1.1).

C3. Création et configuration d'un nouveau clone (machine virtuelle Android) sur Virtualbox (étapes 3 à 6 dans le processus de clonage).

C4. Démarrage du clone (étape 7 dans le processus de clonage).

D'autre part, les étapes 1 du processus de clonage et 3 du constructeur VPN concernant la génération et la sauvegarde des différents paramètres relatifs aux profils des clones et des VPNs sont exécutées en arrière-plan et n'apparaissent pas sur la console TOMCAT.

Le listing 5.7 présente le service web *enregistrement* décrit précédemment dans la section 3.1.1 avec sa méthode *insert* permettant de répondre à une requête utilisateur de clonage.

Listing 5.7 – Service web d'abonnement utilisateur via le middleware de clonage

```
1  package Cloud.Middleware.Clonage;
2  public class enregistrement implements Serializable{
3  private static final long serialVersionUID = 1L;
4      public String insert(String u2, String mp2, String mail2,
           int choix2, String ip2){ //méthode du srvice web
           d'abonnement avec ses paramètres d'entrées suivantes ; u2
           : nom d'utilisateur, mp2 : mot de passe, mail2 : adresse
           mail, choix2 : choix d'appartenance et ip2: ip mobile.
5      //Génération des paramètres clone
6      GénérerParamètresClone GPC = new
           GénérerParamètresClone(u2,mp2,mail2,choix2,up2);
7      GPC.générerParamètresClone(u2,mp2,mail2,choix2,up2);
8      //sauvegarde des données mobiles
9      if (accord="oui"){
10     ADB_Connexion ADB_con = new ADB_Connexion(ip2);
11     ADB_BackupComplet sauvegarde=new ADB_BackupComplet();
12     ADB_con.connexion(ip2);
13     sauvegarde.SauvegardeComplète();
14     }
15     //création d'une nouvelle machine virtuelle Android
16     CréerMachineVirtuelleAndroid nouveauClone = new
           CréerMachineVirtuelleAndroid()
17     nouveauClone.CréerClone();
18     //choix d'appartenance à un groupe de collaboration
19     if (choix2!="0") idgr=choix2;  //rejoindre un groupe existant
20                 else if (choix2==0) //créer un nouveau groupe
21                 {//appeler le service web créerGroupe
22                  créerGroupe créerGr = new créerGroupe();
23                  créerGr.ConstruireVPN();
24                 }
25     //configuration d'association au VPN
26     AssocierCloneVpnVB interne= new
           AssocierCloneVpnVB(id_groupe, nomClone);
27     interne.associerCloneVPN(numVPN, nom_clone);
28     //configuration des paramètres de traduction d'adresse IP
29     NAT_VB_PourClone externe = new NAT_VB_PourClone
           externe(numVPN, nomClone);
30     externe.associerCloneRéseauExterne();
31     démarrerAndroid lancer = new démarrerAndroid(nomClone);
32     lancer.démarrerClone(nomClone); //démmarrage Android
33     }
34 }
```

5 Conclusion

Le but de ce chapitre est de montrer l'efficacité de la réutilisation et de la flexibilité du modèle générique présenté dans le chapitre 4 afin de remédier au problème d'hétérogénéité des environnements MCC. Ainsi, l'instanciation de ce modèle nous a permis de concevoir et de développer MidBox, un middleware de déploiement des données et applications mobiles sur VirtualBox. Il offre un ensemble de services web permettant la construction d'une plateforme virtuelle capable d'exécuter des applications collaboratives mobiles en mode P2P. Nous nous sommes intéressé à la présentation de l'architecture fonctionnelle de ces services qui permettent de cloner des mobiles, gérer les groupes de collaboration, configurer des VPNs et surveiller l'intégrité de la plateforme. Nous avons également présenté des scripts codés en java des différents processus constituant ces services et agissant sur l'hyperviseur VirtualBox.

Dans le chapitre suivant, nous allons présenter l'application collaborative mobile MobiRdf qui peut être déployée avec MidBox.

Chapitre 6

Service d'édition collaborative pour des graphes RDF

Sommaire

1 Introduction

Pour le partage des données mobiles publiées sur le web, les données liées sont présentées comme une solution. Le terme de données liées (LD) se réfère à un ensemble de pratiques à mettre en œuvre pour la publication et l'accès à des données structurées et interconnectées (c-à-d, liées) à l'aide d'identificateurs de ressources uniformes (URI). Un URI est une chaîne de caractères normalisée permettant d'identifier les ressources et leurs relations.

Les données liées peuvent représenter toute chose (c-à-d, ressource) imaginable (par exemple, des livres, tâches, pages, processus, voitures, animaux, chiens, personnes, garçons, filles, parents, fils, etc.). Les ressources sont généralement associées à un domaine (par exemple, université, zoo) et peuvent être définies à travers deux niveaux. Le premier niveau regroupe les données qui sont généralement structurées par le modèle standard RDF [71] (en anglais, Resource Description Framework). RDF est une syntaxe générale pour la description de n'importe quelle ressource en utilisant des triplets. Un triplet RDF permet de décrire une ressource (le sujet) qui est liée par une relation (le prédicat) à une valeur de propriété (l'objet). Par exemple, le triplet (Alger, capitale de, Algérie) permet de décrire la ville "Alger". Quant au second niveau, il permet de décrire la sémantique des ressources à l'aide d'un vocabulaire d'interprétation. Les vocabulaires les plus connus sont le schéma RDFS (en anglais, RDF Schema) [30] et le langage d'ontologie OWL (en anglais, Web Ontology Langage) [27]. Cette extension sémantique permet de donner un sens aux ressources ainsi que leurs propriétés et relations. Dans l'exemple précédent, un sens à la relation "capitale" peut être défini avec le triplet (capitale, subClassOf, ville) ; ce qui signifie que la classe "capitale" est une sous-classe de la classe "ville". Cela permet aussi d'éviter les conflits entre des ressources possédant plusieurs sens. Par exemple, le mot "capitale" peut encore désigner une lettre en majuscule dans un autre contexte.

Les données RDF peuvent être stockées en utilisant des bases de données spécifiques appelées triplestore. Elles peuvent être récupérées en utilisant Sparql [56], un langage d'interrogation standard associé au modèle RDF. D'autre part, Sparql-update [51] est une extension de Sparql qui offre un ensemble de requêtes de mise-à-jour permettant la modification d'une base de données RDF. Notons que des moteurs d'inférence basés sur des requêtes Sparql-update peuvent être associés à ces modèles de données afin d'enrichir/minimiser des données RDF ainsi que leurs schémas/ontologies. Les développeurs doivent définir des règles d'inférence (pour ces moteurs) permettant de déduire de nouveaux triplets suite à l'insertion d'un ou plusieurs triplets. Dans notre travail, nous utilisons une syntaxe abstraite simple du langage standard de description des règles sémantiques SWRL [61] (en anglais, Semantic Web Rule Language) pour la définition des dépendances sémantiques inter-ressources. Dans cette syntaxe, une règle possède la forme suivante : *antécédent* $\Longrightarrow$ *conséquent*. Ceci signifie qu'une règle peut être interprétée comme suit : si l'antécédent est vrai alors le conséquent doit être également vrai. Par exemple, la règle *(Pierre,père,John)* $\wedge$ *(Pierre,frère,Bill)* $\Longrightarrow$ *(Bill,oncle,John)* signifie que si *John* a *Pierre* comme père et *Pierre* a *Bill* comme frère alors *John* a *Bill* comme oncle.

Pour le partage des données RDF, toute application collaborative sur un domaine donné doit être en mesure de maintenir et préserver la cohérence des relations sémantiques évolutives inter-ressources en temps réel. Ces relations permettent de décrire un ordre de chaînage entre les triplets. Cependant, cette cohérence sémantique a été négligée par les approches existantes pour l'édition des données RDF partagées [63, 113, 22], qui se sont principalement concentrées sur le maintien de la cohérence syntaxique. Dans notre travail, la définition de cet ordre sémantique est basée sur des règles SWRL de type : $t1 \implies t2$, indiquant une relation d'ordre sémantique entre des couples de triplets $t1$ et $t2$.

D'un point de vue implémentation, le stockage, le maintien de la cohérence, le raisonnement et l'inférence sur les données liées partagées sous contraintes de limitation des ressources mobiles sont considérés comme un grand défi dans des environnements mobiles. Plus précisément, la gestion des travaux de collaboration pour l'édition des données liées tout en supportant leur évolutivité basée-inférence en temps réel et via des réseaux ad-hoc P2P, nécessite des ressources stables et suffisantes. En effet, la mise en œuvre des moteurs d'inférence sur des dispositifs de ressources limitées est très coûteuse en termes de consommation énergétique (comme le cas de JENA [7]), voire impossible (comme le cas de PELLET [103]) [90, 20, 57].

Afin de surmonter les limitations citées ci-dessus, nous proposons MobiRdf, une application d'édition collaborative des données RDF mobiles via le cloud. Le mécanisme de synchronisation est conçu d'une manière permettant d'alléger le fardeau au maximum sur les mobiles durant l'édition des données RDF partagées. Ainsi, afin d'optimiser l'espace de stockage, chaque utilisateur possède uniquement la partie des données RDF utile sans leur schéma sémantique. Il peut explicitement interroger ou mettre à jour sa copie locale des données RDF (c-à-d, son triplestore local) via des requêtes Sparql "light" (c-à-d, sans tenir compte du schéma sémantique et des règles d'inférence). Par exemple, l'implémentation des API JENA : Jena.Model, Jena.Query et Jena.Update permettent respectivement de construire, d'interroger et de mettre à jour des modèles RDF (graphes) sans application d'inférence sur les mobiles [7]. D'autre part, les clones des mobiles possèdent la partie intégrale des données liées (RDF) avec leur schéma (RDFS). Ils sont en mesure d'exécuter toutes les requêtes Sparql (Interrogation et mise-à-jour) tout en supportant des règles d'inférence appliquées par les raisonneurs. À cette fin, chaque clone peut implémenter des APIs Jena pour configurer et instancier des raisonneurs RDFS qui prennent en charge les règles d'inférence RDFS [30].

Le reste de ce chapitre est organisé comme suit : la section 2 présente un cas d'utilisation du modèle collaboratif proposé. La section 3 présente le modèle du service. La section 4 décrit l'architecture du protocole d'édition collaborative. Un mécanisme de réplication partielle des graphes RDF est présenté dans la section 5. La section 6 présente l'éditeur collaboratif utilisé. Un mécanisme de synchronisation de l'application MobiRdf est présenté dans la section 7. Enfin la section 9 conclut ce chapitre.

2 Cas d'utilisation : suivi médical collaboratif

Dans le domaine de la médecine, plusieurs projets de recherche se concentrent sur le suivi médical assisté afin de prendre en charge de nouveaux services, tels que le diagnostic collaboratif, la surveillance à distance et le dossier de santé électronique [60]. Ce type de suivi médical assisté est spécifiquement destiné aux patients qui peuvent souffrir de complications graves à tout moment.

Le personnel de surveillance médicale doit coopérer et agir sans perdre de temps. Pour cela, toute information médicale nécessaire révélant l'état des patients doit être disponible en temps réel, tout en respectant la contrainte de mobilité de tous les participants (patients et personnel médical). Dans ce cas, les mobiles sont présentés comme une solution idéale pour tous ces échanges (informations du patient, diagnostics et traitements). Mais les tâches de collaboration pour maintenir la cohérence des données partagées sont coûteuses en termes de consommation des ressources mobiles (énergie et espace de stockage) [84, 55]. Le recours au cloud surmonte cette limitation.

En outre, un tel système doit être fondé sur un référentiel commun et conventionnel qui peut être utilisé par tout le personnel médical. L'utilisation d'un vocabulaire médical standard basé sur des ressources RDF aide à satisfaire ce besoin [40].

D'un point de vue opérationnel, les données physiologiques détectées chez un patient peuvent varier en fonction de plusieurs critères, comme les activités physiques et l'âge. Par exemple, la fréquence du battement de cœur varie selon l'activité exercée par un patient (marcher, courir, etc.). En l'occurrence, toutes ces relations sémantiques (entre les valeurs physiologiques détectées et le statut du patient) doivent être prises en compte pour éviter les fausses alertes ou la prescription des mauvais traitements. D'autre part, l'ordre sémantique des données médicales recueillies auprès d'un patient, ainsi que des traitements prescrits par les médecins ou déduits par le système doivent être soigneusement respecté.

La figure 6.1 montre un exemple illustratif pour un système collaboratif de suivi médical. Le cas présenté concerne un patient âgé de 80 ans et ayant un diabète de type 2 comme maladie chronique. Ce patient ne doit pas quitter sa maison et dispose d'un robot assistant qui reçoit les instructions d'un staff médical pour intervenir en cas de complications graves. Avec ce système, les patients comme les membres du staff médical possèdent des clones de leurs mobiles dans le cloud. Ces clones raisonnent sur les informations médicales collectées auprès des patients afin de déduire des diagnostics ou diffuser des alertes dans le réseau. Les informations médicales ainsi que leur schéma/ontologie sont répliqués sur des triplestores au niveau des clones. Supposons qu'à un moment donné, les capteurs du patient émettent dans le réseau des données indiquant une situation de malaise : (A) taux de glycémie=0.60, (B) des tremblements et (C) inconscience. Avec un tel ordre de symptômes, le clone associé au patient utilise les règles d'inférence : ($R1 : A \implies B, R2 : B \implies C$ et $R3 : A \wedge B \wedge C \implies$ *diagnostic =hypoglycémie*) du schéma pour déduire un diagnostic d'une hypoglycémie (un taux de sucre anormalement bas) et

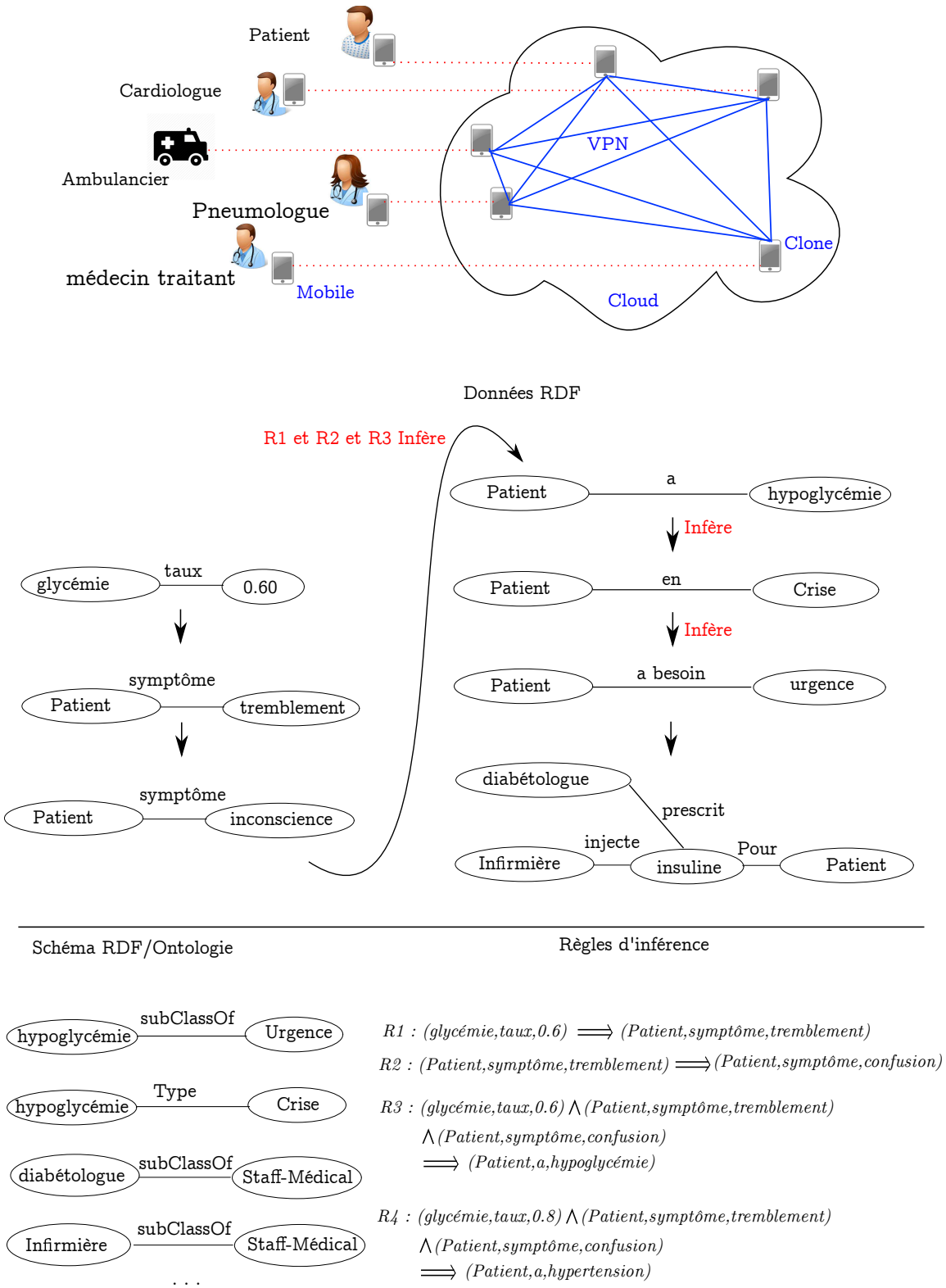


FIGURE 6.1 – Exemple de suivi médical collaboratif via le cloud.

alerte les membres du personnel médical. Par conséquent, le diabétologue indique tout d'abord d'injecter une dose d'insuline au patient, et de donner du sucre après. Cependant, l'ordre d'émission des données peut être perturbé par le trafic et la latence du réseau. Ce désordre peut conduire à des diagnostics et/ou des traitements faux. Supposant, par exemple, qu'un clone reçoit les données précédentes dans l'ordre suivant : (B) tremblements, (C) inconscience, et (A) taux de glycémie=0.60. En se basant sur les deux premiers symptômes (c-à-d, avant l'arrivée du triplet A : taux de glycémie=0.60), le clone utilise les règles d'inférence ($R1 : A \implies B$ et $R4 : A \wedge B \wedge F \implies diagnostic = hypertension$) menant à déduire un faux diagnostic d'hypertension (F est un ancien triplet indiquant un taux normal de glycémie). Afin d'éviter cette situation, chaque donnée révélant l'état du patient doit être annotée avec une métadonnée indiquant la donnée précédente. Ainsi, le protocole collaboratif déployé dans le cloud doit fournir les mécanismes de synchronisation nécessaires pour respecter et assurer la cohérence de ces relations et ordres sémantiques. Notons que chaque clone doit assurer la continuité du service. Il doit être en mesure de prescrire un traitement urgent en cas de déconnexion d'un membre du staff médical.

3 Modèle du service

Dans ce chapitre nous proposons un nouveau service pour le MCC où plusieurs utilisateurs mobiles sont en collaboration afin d'éditer des graphes RDF partagés en mode P2P. La valeur ajoutée de ce modèle est le maintien de la cohérence syntaxique et sémantique des données partagées via une utilisation optimale des ressources de dispositifs mobiles. En effet, les tâches de collaboration, de communication, de sauvegarde et de raisonnement sont parfaitement déployées sur le cloud. Cette section est consacrée à la présentation du modèle de notre service et la conception du système.

3.1 Présentation du modèle

La mise en œuvre de ce système d'édition collaborative vise à permettre à plusieurs utilisateurs qui sont géographiquement dispersés à modifier des graphes RDF partagés à tout moment. L'adoption des smartphones pour une telle collaboration est fortement contrainte par des facteurs essentiels tels que la puissance de calcul, la durée de vie de la batterie et la couverture du réseau. En outre, l'extension de cette collaboration dans l'espace et le temps sera inévitablement perturbée face aux limitations de ces facteurs. Pour surmonter ces limitations, nous proposons de déléguer la plupart des tâches de calcul des mobiles ainsi que leurs tâches de communication vers le cloud. Notre service d'édition collaborative basé sur le cloud permet aux utilisateurs de collaborer comme suit : chaque utilisateur dispose d'une copie locale partielle des graphes RDF partagés, les mises-à-jour de l'utilisateur sont exécutées localement puis elles sont propagées à d'autres utilisateurs afin d'être exécutées sur d'autres copies.

3.2 Architecture globale du système

Comme illustré sur la figure 6.2, le système est structuré en trois couches principales. D'une part, il fournit des interfaces graphiques (GUI) pour interagir avec les deux couches restantes (voir la couche application dans la figure 6.2). L'interaction avec la deuxième couche de clonage, c-à-d, le middleware de déploiement MidBox qui a été présenté dans le chapitre précédent, permet de traiter les requêtes utilisateurs de clonage et de gestion des groupes. Tandis que l'interaction avec la couche de collaboration (c-à-d, l'application MobiRdf) permet aux utilisateurs de synchroniser les mises à jour appliquées sur les graphes RDF partagés entre le mobile et son clone.

La principale couche décrite dans ce chapitre est l'application MobiRdf (voir la couche de collaboration dans la figure 6.2). Il s'agit d'un protocole de collaboration pour l'édition collaborative des graphes RDF partagés (en temps réel) dans une plateforme cloud P2P formée de clones et de VPNs. Cette plateforme virtuelle est construite par le middleware MidBox.

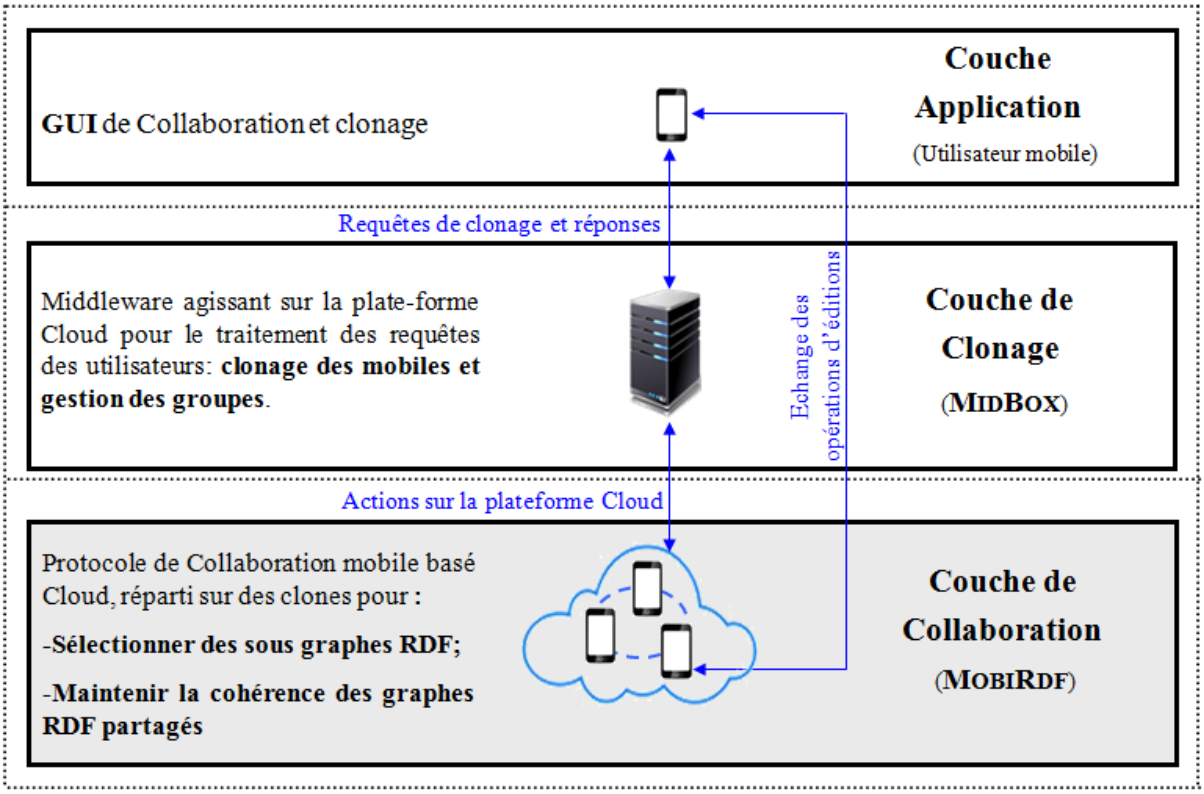


Figure 6.2 – Architecture du système de partage des données RDF dans le cloud.

4 MobiRdf : un protocole d'édition collaborative mobile

4.1 Architecture du protocole

Dans cette section nous présentons l'architecture de notre protocole d'édition collaborative MobiRdf. Cette architecture est issue de la réutilisation des patrons de collaboration précédemment décrits dans le chapitre 4, section 4.2. Les composantes *Sélecteur des graphes partiels*, *communication*, *Synchroniseur* et *Éditeur RDF* de l'application collaborative mobile MobiRdf (voir la figure 6.3) sont respectivement issues du raffinement des patrons de collaboration : *Réplication partielle*, *communication*, *synchronisation* et *Gestion des ressources* (voir la section 4.2).

Comme notre objectif est de déléguer au maximum les tâches consommatrices de ressources (telles que la synchronisation, le raisonnement et la communication) vers le cloud, MobiRdf est exécuté sur deux niveaux (voir la figure 6.3) :

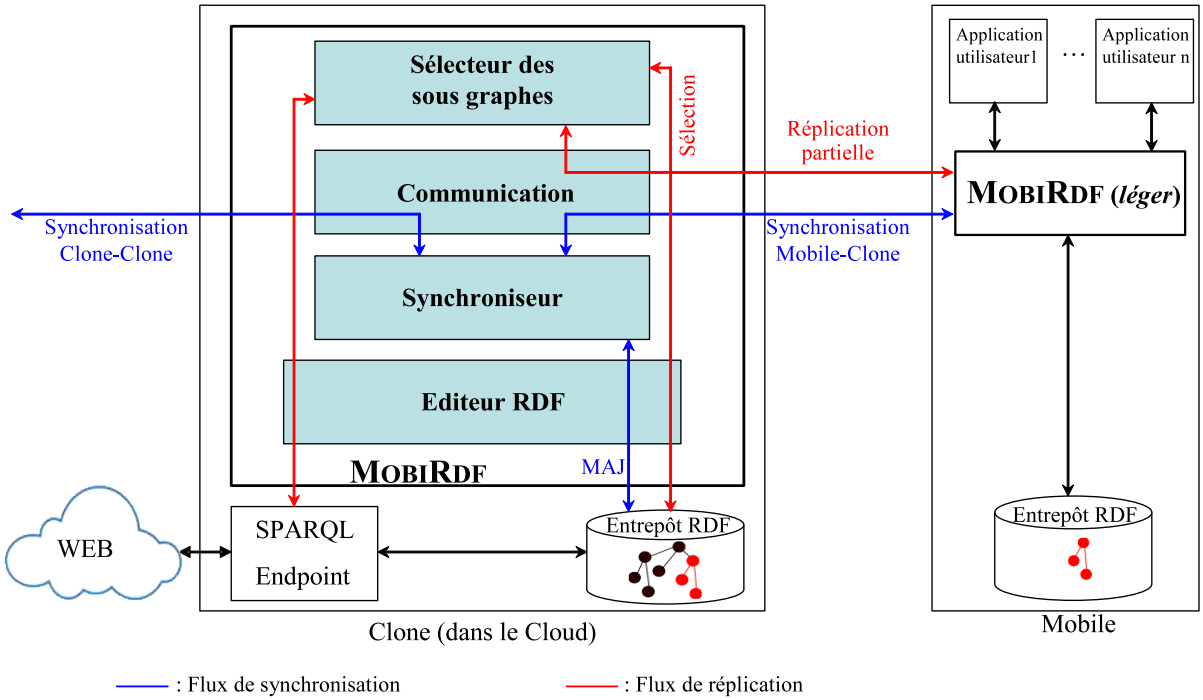


Figure 6.3 – Structure de l'application d'édition collaborative MobiRdf

D'une part, la partie légère du logiciel est exécutée sur le côté du mobile. Elle interagit avec les applications des utilisateurs pour l'exécution des différentes requêtes Sparql-update locales afin de les appliquer sur le triplestore local. En outre, cette application permet d'interagir avec le clone correspondant pour envoyer les requêtes utilisateurs localement exécutées et recevoir/appliquer d'autres requêtes distantes intégrées depuis les autres clones. Cette application est légère dans le sens où elle est libre de toute interaction avec d'autres collaborateurs mobiles,

ne conserve pas d'histoires et n'exécute aucune tâche supplémentaire pour l'intégration des requêtes distantes. D'autre part, la partie lourde de l'application MobiRdf est celle exécutée sur le côté clone dans le cloud. Elle offre les composantes suivantes :

1. *Sélecteur de graphe partiel*. Cette composante a pour objectif d'évaluer les informations de contexte utilisateurs/mobiles pour sélectionner des répliques composées de graphes RDF partiels utiles pour le mobile. Le triplestore du mobile peut être considéré comme un sous-ensemble du triplestore de son clone.
2. *Synchroniseur*. Fournit des mécanismes de synchronisation afin de préserver la cohérence syntaxique et sémantique des graphes RDF partagés. Il est basé sur la commutativité d'un ensemble de requêtes Sparql-update pour assurer la cohérence syntaxique et sur des relations de dépendance sémantique et causale pour forcer la cohérence sémantique.

   La synchronisation est totalement décentralisée et aucun rôle n'est assigné à un serveur central ou clone maître.
3. *Communication*. Ce module offre un support de communication permettant à chaque clone d'effectuer des échanges des données avec : (i) son mobile afin d'envoyer des requêtes Sparql distantes ou recevoir des informations contextuelles ou des requêtes locales générées par ce mobile et (ii) les autres clones dans le cloud afin de diffuser ou intégrer des requêtes Sparql distantes.
4. *Éditeur RDF*. Cet éditeur est basé sur des requêtes commutatives Sparql permettant à la composante synchroniseur d'assurer la cohérence syntaxique des répliques RDF partagées d'une manière simple.

4.2 Principe de fonctionnement du protocole MobiRdf

Avec notre protocole d'édition collaborative en temps réel, chaque utilisateur possède deux copies des graphes RDF partagés ; une copie intégrale est stockée dans le triplestore du clone, tandis qu'une autre copie partielle est stockée dans le triplestore local du mobile. Chaque utilisateur peut exécuter des requêtes Sparql-update en mode on-line comme en off-line. Ainsi, les requêtes localement générées seront reçues et appliquées par le clone correspondant. Ces requêtes donc seront propagées vers les autres clones. Cependant, l'accès concurrent à des répliques des graphes RDF peut entraîner une divergence de leurs états. À cet effet, les clones doivent être synchronisés pour assurer des états corrects et cohérents de leurs copies des graphes RDF répliqués. Dans la littérature, plusieurs approches ont été proposées afin de forcer la convergence des répliques en réseaux P2P, telles que l'ordonnancement des événements [77] et l'approche des transformées opérationnelles [43, 106]. Cependant, ces solutions sont plus adaptées aux structures de données linéaires.

Afin de satisfaire cet objectif, nous avons conçu un nouveau protocole d'édition collaborative P2P dans le cloud. Ce protocole est basé sur l'application et l'échange de requêtes Sparql commutatives qui sont annotées avec des relations de dépendances entre elles pour assurer

le maintien de la cohérence des répliques RDF partagées (les notions de commutativité et de dépendance des requêtes seront détaillées ultérieurement). En outre, il utilise un schéma de réplication optimiste (les graphes RDF partagés sont répliqués au niveau des mobiles et sur leurs clones) permettant ainsi, un accès concurrent aux graphes RDF partagés.

Ainsi, le principe général de ce protocole d'édition collaborative peut être résumé comme suit :

— Chaque mobile peut générer et appliquer une requête Sparql-update d'une manière indépendante.
— Une fois appliquée, les requêtes locales seront envoyées au clone correspondant.
— Après réception, le clone procédera à la décomposition de chaque requête Sparql-update locale en un ensemble de requêtes commutatives. Il les intègre ensuite dans son triplestore local. En outre, il utilise un processus de raisonnement pour déduire les différentes relations de dépendance par-rapport à son histoire des requêtes appliquées avant de les propager aux autres clones.
— Une fois que les autres clones ont reçu ces requêtes commutatives distantes, ils procéderons à leur intégration tout en respectant leurs relations de dépendance. Les requêtes indépendantes seront directement appliquées quel que soit leur ordre d'arrivée.

Dans les sections suivantes nous décrivons les composantes : (i) sélecteur des graphes partiels, à travers une proposition d'un modèle de réplication des graphes RDF partiels, (ii) éditeur RDF, basé sur la commutativité des requêtes Sparql pour assurer la cohérence et (iii) synchroniseur, via un modèle de cohérence comportant les mécanismes de synchronisation clone/mobile et clone/clone.

5 Réplication des graphes RDF partiels

La réplication des données du web sémantique sur des mobiles permet aux applications et services de fonctionner indépendamment de la qualité des connexions réseau. Néanmoins, cette réplication doit prendre en compte les ressources limitées des mobiles telles que la capacité de stockage. Pour cela, une réplication sélective est souhaitable afin de ne mettre à la disposition des utilisateurs que des données utiles. Le processus de sélection est basé sur des informations contextuelles concernant l'utilisateur et son environnement.

Commençons d'abord par l'illustration du contexte dont nous pouvons trouver plusieurs définitions dans la littérature. En général, le contexte peut être défini comme toute information qui peut être utilisée pour caractériser la situation d'une entité [39]. Une autre définition décrit le contexte comme n'importe quelle chose entourant un utilisateur ou un dispositif et donne un sens à quelque chose [111]. De façon plus précise, le contexte mobile peut être défini comme "*toute information décrivant un état de fonctionnement d'un utilisateur ou son mobile*" [121].

D'autre part, l'acquisition de ces informations contextuelles mobiles peut prendre deux formes possibles [52] : (i) sensibilité directe : dans ce cas, le processus d'acquisition du contexte

opère localement sur le mobile. Les informations contextuelles sont implicitement fournies par des capteurs intégrés (par exemple, la localisation fournie par un capteur GPS intégré). (ii) sensibilité indirecte : permet de capter des informations contextuelles provenant des capteurs ou services distants qui sont mis en œuvre dans l'environnant ambiant.

Il faut remarquer que le problème essentiel des systèmes sensibles au contexte est qu'il n'existe pas un modèle général spécialement dédié au domaine MC [121].

Dans cette section, nous présentons un modèle de réplication guidée par le contexte pour les données RDF mobiles [6]. Ce modèle est issu de l'extension du patron de réplication précédemment présenté dans l'architecture générique globale (voir la section 4.2.5). Comme le montre la figure 6.4, les classes ajoutées sont présentées avec une couleur de fond grise. Ainsi, cette extension permet de spécifier les méthodes liées à la gestion du contexte et du processus de réplication d'une manière concrète. Nous proposons de formuler les informations contextuelles sous forme de données RDF, les enregistrer dans des triplestores et d'appliquer un raisonnement basé sur les règles d'inférence. Les processus coûteux en terme de consommation d'énergie tels que l'évaluation des informations contextuelles ainsi que la sélection des graphes partiels s'exécutent sur le clone (pour décharger le mobile). Le mobile est seulement en mesure d'acquérir des informations contextuelles et de les envoyer à son clone sans aucun traitement supplémentaire. Ainsi, nous pouvons distinguer les trois principales tâches suivantes pour la conduite des processus de sélection des graphes RDF partiels :

Collecte du contexte. Ce module est implémenté du côté mobile. Son rôle est limité à la réception et collecte des informations du contexte à partir des capteurs physiques et logiques. Comme le montre la figure 6.4 (Mobile), ce modèle comprend la classe *Collecteur_Contexte* qui utilise les classes d'interfaces suivantes :

(i) l'interface *Capteurs_Hardware* : la redéfinition de la méthode *CapteurHard()* de cette interface permet d'acquérir des informations contextuelles depuis les différents capteurs physiques intégrés dans le mobile (par exemple GPS et caméra).

(ii) L'interface *Capteurs_Ubiquitaires* : redéfinir la méthode *CaptureEnvironnement()* permet de collecter des informations contextuelles à partir des capteurs ou des équipements situés dans l'environnement entourant.

(iii) L'interface *Applications_Web* : la méthode *CaptureDonnéesWeb()* permet d'acquérir des informations contextuelles fournies par des services et applications web (par exemple, facebook, tweeter).

(iv) L'interface *Capteurs_Logiciel* : la méthode *CaptureLogique ()* représente un mécanisme d'observation et de détection des différents changements appliqués par les utilisateurs en utilisant des applications mobiles locales (par exemple, rendez-vous enregistrés dans un calendrier).

(iv) L'interface *Communication* : l'implémentation de cette interface permet de redéfinir la méthode *EnvoiUnicast* pour envoyer les informations contextuelles collectées en mode unicast

6. Les détails ainsi que l'implémentation de ce modèle sont au delà de la portée de ce travail.

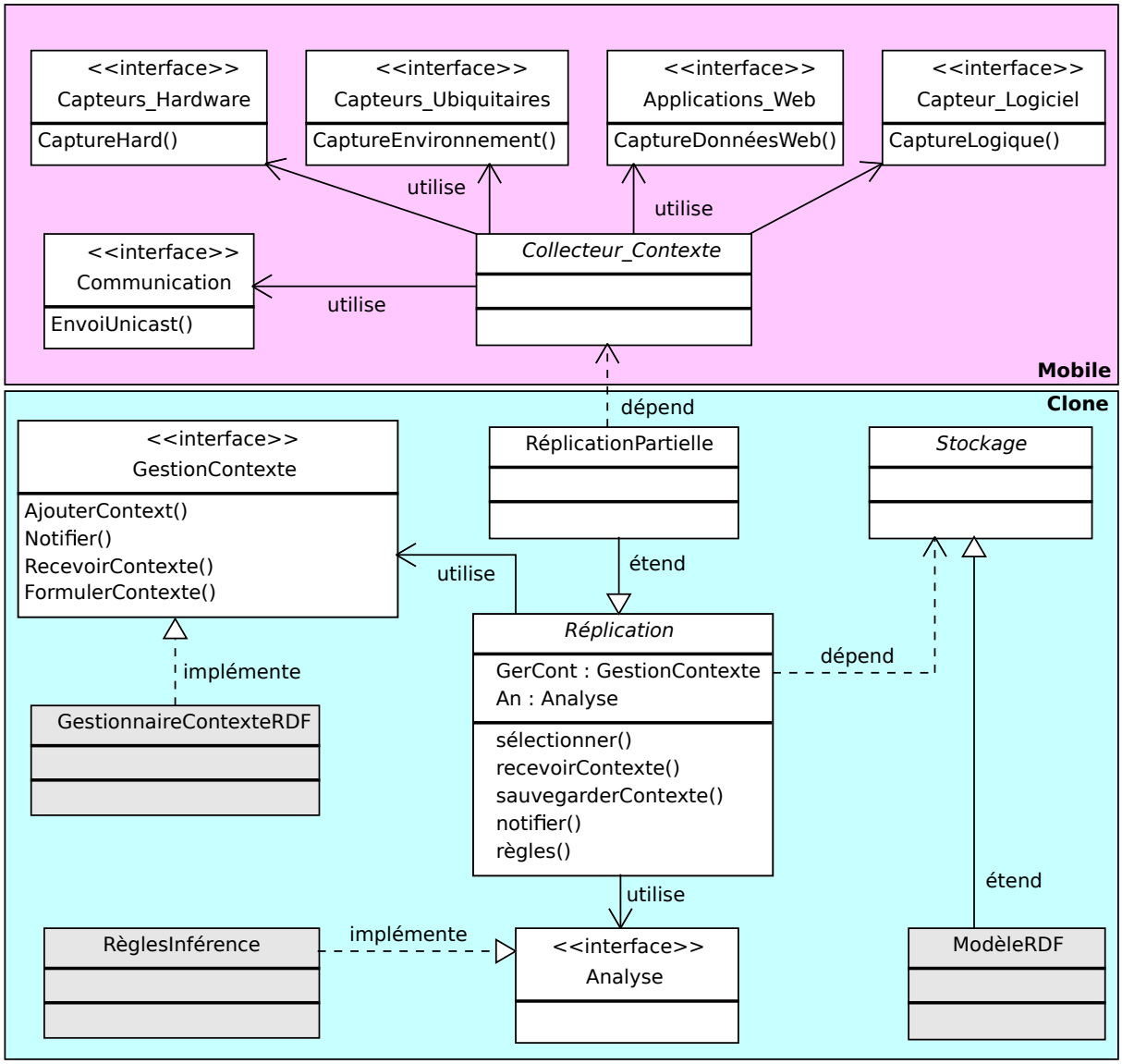


FIGURE 6.4 – Architecture du sélecteur des graphes RDF partiels

au clone correspondant.

Gestion du contexte. Ce module est implémenté côté clone. Son rôle important se concentre dans la réception et l'enregistrement des informations contextuelles envoyées par le mobile. Comme le montre la figure 6.4 (clone), ce modèle comprend la classe abstraite *Réplication* qui utilise les classes d'interface *GestionContexte* et *Analyse*. La classe *RéplicationPartielle* qui étend la classe abstraite *Réplication* profite de son héritage à celle-ci afin d'instancier la classe *GestionnaireContexteRDF* qui implémente l'interface *GestionContexte*. Cette instanciation permet de :

(i) Recevoir des informations contextuelles envoyées par le mobile.

(ii) Coder l'information contextuelle reçue sous forme de triplets RDF.

(iii) Insérer les triplets RDF issus du codage des informations contextuelles reçues dans un triplestore local sur le clone. Ce triplestore est représenté par la classe *ModèleRDF* qui étend la classe abstraite *Stockage* (voir la figure 6.4 (clone)).

(iv) Notifier le processus de réplication partielle (représenté par la classe *RéplicationPartielle*) pour le déclenchement d'une éventuelle réplication (basée sur une nouvelle information contextuelle reçue) d'un sous-graphe RDF pour le mobile.

Sélection des graphes partiels. Ce processus est également implémenté côté clone. Il vise à sélectionner des graphes RDF partiels (depuis le triplestore du clone ou via le web) au profit du mobile. Cette réplication sélective est basée sur un processus de raisonnement (la classe *RèglesInférence* qui implémente l'interface *Analyse*) qui est déclenché suite à la réception de nouvelles informations contextuelles.

Afin d'illustrer ce mécanisme, considérons l'exemple suivant. Supposons qu'un clone a reçu une information contextuelle (coordonnée GPS) depuis le mobile, indiquant que l'utilisateur est présent à un endroit donné de la ville de Paris. Le sélecteur des graphes partiels procèdera à la détermination (sélection) d'un ensemble de points d'intérêt (par exemple, hôtels, restaurants, etc.) proches de cet endroit (depuis son triplestore local ou via le web). Le graphe partiel issu de cette sélection sera proposé à l'utilisateur afin de l'éditer et le partager avec d'autres collaborateurs.

6 Editeur collaboratif pour RDF

Cette section présente notre éditeur RDF défini par un ensemble d'opérations d'édition Sparql-update commutatives. Comme nous l'avons déjà mentionné, l'élaboration d'un tel éditeur est motivé par notre stratégie de synchronisation basée sur la commutativité des requêtes Sparql-update pour le maintien de la cohérence des graphes RDF partagés. Tout d'abord, il est nécessaire de rappeler les principes fondamentaux utilisés pour la conception de notre éditeur, tels que la commutativité et le modèle de graphe RDF (structure de données et opérations d'édition Sparql-update).

6.1 Vue d'ensemble sur les concepts de commutativité et de dépendance

Pour le partage des données RDF, nous présentons un nouveau modèle qui vise à préserver la cohérence via une combinaison de deux techniques : la commutativité et les relations de dépendance. Le mécanisme de la commutativité consiste à définir un ensemble d'opérations qui commutent entre elles sur une structure de données donnée. Il doit assurer la cohérence indépendamment de l'ordre des opérations d'édition à leur réception. Cependant, ces opérations peuvent être dépendantes et leur ordre de génération doit être soigneusement respecté via une

définition de leurs relations de dépendance.

Commutativité. Deux opérations o et o' commutent, si l'exécution de o suivie par o' mène au même état que l'exécution de o' suivie par o. Il existe deux approches de base pour assurer la commutativité, à savoir la coalescence et la précédence [100].

Premièrement, la coalescence (*formation de l'union des événements ayant la même séquence*) [80] signifie que l'exécution d'une opération o en premier ordre doit préserver l'effet attendu de l'exécution d'une opération suivante o' et vice versa. Par exemple, les opérations d'addition et de soustraction commutent d'une manière coalescente sur l'ensemble des entiers naturels.

D'autre part, la précédence est un mécanisme qui préserve la priorité ainsi que les effets liés à l'ordre d'exécution des opérations. Si on suppose que o a la priorité sur o', alors l'effet attendu de l'exécution de o doit être imposé. Une solution intuitive est d'écraser le résultat de o' par o dans l'ordre d'exécution (o', o). Alors qu'avec l'ordre (o, o'), o' sera négligée (remplacée par une opération nulle qui n'aura aucun effet : No_Op).

Dépendance causale. Une opération o est causalement dépendante d'une autre opération o' ($o' \overset{caus}{\rightarrow} o$), si l'effet d'exécution de o sera appliqué à la donnée précédemment générée par l'application de l'opération o'. Une opération ($sel(p, D)$) pour sélectionner un paragraphe p dans un document D est causalement dépendante d'une opération ($cre(p, D)$) pour la création de ce paragraphe dans le document D ; $cre(p, D) \overset{caus}{\rightarrow} sel(p, D)$.

Dépendance sémantique. Une opération o est sémantiquement dépendante d'une opération o' ($o' \overset{sem}{\rightarrow} o$), si l'exécution de l'opération o' suivie par l'exécution de l'opération o possède un sens logique (le cas contraire peut être faux, ou possède un autre sens logique). Par exemple, si on considère les deux opérations *o1 : mettre_le_sucre* et *o2 : mélanger_le_café* pour la préparation d'un café, alors $o2$ est sémantiquement dépendante de $o1$ ($o1 \overset{sem}{\rightarrow} o2$).

Concurrence. Deux opérations o et o' sont dites concurrentes ou commutatives si ($o \overset{caus}{\not\rightarrow} o'$) et ($o' \overset{caus}{\not\rightarrow} o$) et ($o \overset{sem}{\not\rightarrow} o'$) et ($o' \overset{sem}{\not\rightarrow} o$).

6.2 Vue d'ensemble sur le modèle de graphe RDF

Dans ce qui suit, nous rappelons quelques définitions préliminaires autour de la structure de données RDF et des opérations d'édition Sparql-update d'après [86, 71, 51] :

6.2.1 Structure de données

Assumons qu'ils existent des ensembles finis L (Littéraux), U (références URI pour RDF) et B (nœuds vides). Pour une raison de simplicité, nous dénotons les unions de ces ensembles par la concaténation de leurs noms. Par exemple la concaténation des ensembles U, B et L sera notée UBL.

Definition 6.1

URI. *Un identifiant de ressources uniforme (URI) est une chaîne de caractères permettant d'identifier une ressource physique ou abstraite.*

Definition 6.2

URL. *Le terme Localisateur de ressources uniforme (URL) désigne un sous-ensemble des URIs qui, en plus d'identifier une ressource, permet de localiser et d'accéder à une ressource via le web.*

Definition 6.3

Littéral. *Les littéraux sont utilisés pour identifier des valeurs telles que les numéros et les dates au moyen d'une représentation lexicale. N'importe quelle chose représentée par un littéral pourrait être aussi représentée par un URI, mais il est souvent plus commode ou intuitif d'utiliser des littéraux.*

Definition 6.4

Nœud vide. *Les nœud vides représentent un ensemble arbitraire d'identificateurs locaux dans un graphe RDF.*

Definition 6.5

État. *Un état est un triplet* $(s, p, o) \in UBL \times U \times UBL$*, où* s *(Sujet) peut être une référence URI, un littéral ou un nœud vide ,* p *(Prédicat) est une référence URI et* o *(Objet) peut être une référence URI, un littéral ou un nœud vide.*

Definition 6.6

Graphe RDF. *Un graphe RDF est un ensemble d'états (triplets RDF).*

Definition 6.7

RDFS. *RDFS ou RDF-Schéma est un langage extensible qui fournit des éléments de base pour la représentation des connaissances. Cette extension basée sur le vocabulaire permet de définir les relations entre les différentes ressources RDF structurées.*

Definition 6.8

Moteur d'inférence. *Un moteur d'inférence permet de conduire un raisonnement logique pour déduire des nouveaux triplets à partir d'un graphe RDF en se basant sur un ensemble de règles d'inférence.*

Exemple.

La figure 6.5 montre un exemple d'un graphe RDF simple. Il est composé de trois triplets; les sujets sont représentés par des ellipses verts, les prédicats par des flèches et les objets par des rectangles jaunes. Ceux-ci représentent une ressource personne qui peut être décrite par les propriétés suivantes :

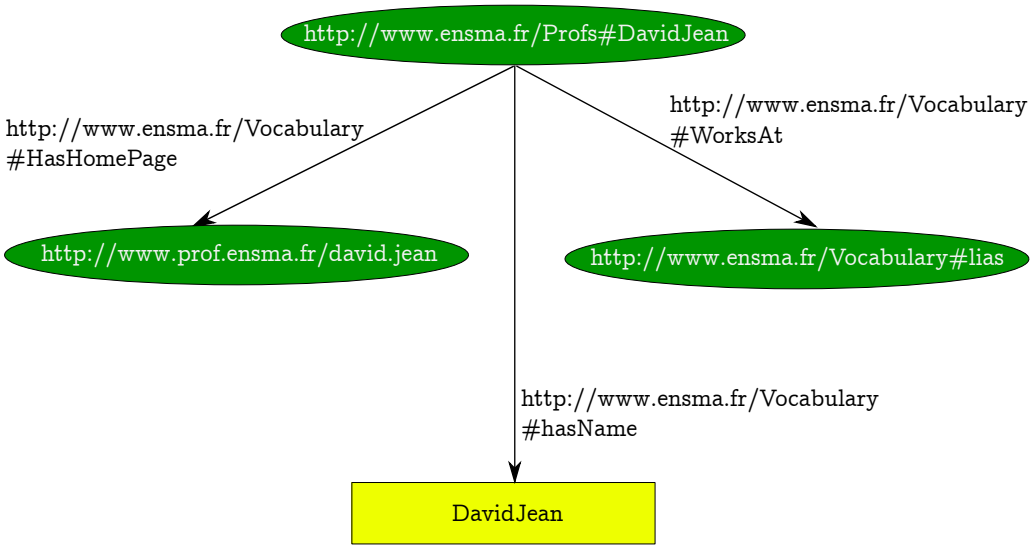


FIGURE 6.5 – Exemple d'un graphe RDF.

(i) un sujet :

- la personne elle-même: `http://www.ensma.fr/Profs#`, ce sujet apparaît dans les trois triplets et il est désigné par un URI.

(i) trois objets :

- Le nom de la personne: "DavidJean", un littéral de type chaîne de caractères.

- Le lieu de travail, désigné par un URI : `http://www.ensma.fr/Vocabulary#lias`. C'est une ressource physique inaccessible via le web.

- La page personnelle, désignée par un URL : `http://www.prof.ensma.fr/david.jean`. C'est une ressource logique qui peut être accédée via le web en utilisant son URL.

D'autre part, le sujet est associé à d'autre objets au moyen des prédicats identifiés par les URI suivants :

- `http://www.ensma.fr/Vocabulary#HasHomePage`.

- `http://www.ensma.fr/Vocabulary#WorksAt`.

- `http://www.ensma.fr/Vocabulary#hasName`.

6.2.2 Opérations d'édition

Comme nous l'avons mentionné précédemment, les opérations d'édition appliquées aux graphes RDF partagés sont basées sur des requêtes Sparql-update [51]. Dans ce qui suit, nous présentons les différentes requêtes existantes :

InsertData. La requête $InsertData(iriGraph, T)$ permet d'insérer un ensemble de triplets $t =$

$(s, p, o) \in T$ dans un graphe RDF identifié par son nom *iriGraph*. *InsertData* n'autorise pas la duplication des triplets existants dans les graphes RDF ; si $InsertData(t)$ tente d'insérer un triplet t qui existe déjà dans un graphe G alors $InsertData(t) = No\_Op$, où No_Op est une requête nulle qui n'aura aucun effet.

DeleteData. La requête $DeleteData(iriGraph, T)$ est utilisée pour supprimer un ensemble de triplets $t = (s, p, o) \in T$ à partir d'un graphe RDF. Dans le cas où le triplet à supprimer n'existe pas dans le graphe cible, la requête *DeleteData* n'aura aucun effet ($DeleteData(t) = No\_Op$).

DeleteInsert. La requête $DeleteInsert(iriGraph, DeleteClause, InsertClause, WhereClause)$ peut être utilisée afin d'apporter des modifications aux triplets d'un graphe RDF. C'est une suppression et/ou insertion d'un ensemble de triplets basée sur la clause *WhereClause*. Le principe d'exécution de la requête *DeleteInsert* est résumée dans les étapes successives suivantes :

(i) Évaluation de la clause *WhereClause* : ceci permet d'identifier l'ensemble des triplets impliqués par le changement ; c'est une requête de sélection basée sur la clause *WhereClause*.

(ii) Suppression : cette étape implique une application de la clause de suppression sur l'ensemble des triplets résultant de la sélection basée sur la clause *WhereClause* décrite à l'étape précédente.

(iii) Insertion : Cette étape permet d'insérer un ensemble de triplets en respectant la clause *WhereClause*.

En conséquence, une requête Sparql-update *DeleteInsert* peut être équivalente à une succession de plusieurs requêtes *DeleteData* et *InsertData*.

Il est également à noter que les deux requêtes $DeleteWhere(iriGraph, DeleteClause, whereClause)$ et $InsertWhere(iriGraph, InsertClause, WhereClause)$ sont deux cas particuliers provenant de la requête *DeleteInsert*. Ces requêtes permettent seulement et exclusivement de supprimer ou insérer des triplets tout en respectant la clause *WhereClause*. En outre, les requêtes $load(iriRef-from, iriRef-To)$ et $clear(iriRef)$, permettent respectivement de charger le contenu d'un graphe vers un autre ou d'effacer son contenu et sont également des cas particuliers de la requête *DeleteInsert*.

6.3 Notre éditeur RDF

Dans ce qui suit, nous démontrons que les deux requêtes Sparql-update *InsertData* et *DeleteData* pour la suppression ou l'insertion d'un seul triplet t à partir d'un graphe G sont exclusivement commutatives. À cette fin, la commutativité de toutes les requêtes Sparql-update précédemment décrites est étudiée par paires de requêtes comme suit :

Proposition 6.1

Les couples des requêtes (DeleteInsert, DeleteData), (DeleteInsert, InsertData) et (DeleteInsert, DeleteInsert) ne sont pas commutatifs.

Démonstration. Les contre-exemples suivants sont suffisants pour démontrer cette non commutativité :

DeleteInsert et DeleteData. Étant donné deux mobiles, Mobile 1 et Mobile 2, commençant une édition collaborative et concurrente à partir d'un état commun d'un graphe RDF qui contient l'ensemble des triplets : ("A", $< uri1 >$, "C"), ("A", $< uri2 >$, "B").

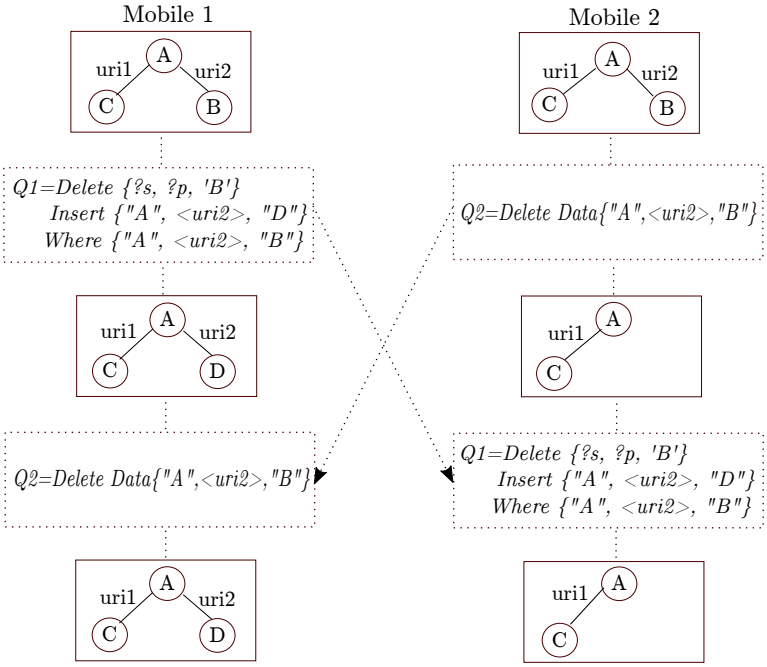


FIGURE 6.6 – Un contre exemple pour la commutativité des requêtes *DeleteInsert* et *DeleteData*.

Comme le montre la figure 6.6, en appliquant et échangeant les deux requêtes : *Q1 =DeleteInsert(Delete(?s,?p,"B") ; Insert("A",<uri2>,"D") ; Where("A",<uri2>,"B"))* et *Q2 =DeleteData(("A",<uri2>,"B"))*, les deux mobiles finissent avec deux états conflictuels. Pourquoi cette divergence ?. Après que le Mobile 2 avait supprimé le triplet *("A", <uri2>, "B")*, il reçoit la requête *Q2* pour changer ce même triplet supprimé !. En effet, cette requête consiste à remplacer le triplet concerné avec un autre (suppression suivie d'une insertion). Mais lors de la sélection de l'ensemble des triplets justifiant la condition exprimée dans la clause *Where("A", <uri2>,"B")* (c-à-d, *s="A",p=<uri2>* et *o="B"*), il aura un ensemble vide comme résultat de cette sélection conditionnée. C'est pour cette raison que le triplet *("A",<uri2>,"D")* n'a pas pu être inséré par Mobile 2 ; les deux requêtes élémentaires *DeleteData* et *InsertData* issues de la requête *DeleteInsert* seront équivalentes à *NO-OP*.

DeleteInsert et InsertData. La figure 6.7 montre un contre-exemple pour la commutativité des requêtes *DeleteInsert* et *InsertData*. L'incohérence est due aux opérations concurrentes d'insertion et de suppression du même triplet *("A", <uri4> "D")*. Le Mobile 1 commence par la suppression de tous les triplets justifiant la condition de *s= "A"* et reçoit ensuite une requête distante pour insérer le triplet *t1 = ("A", <uri4>, "D")* . De son côté, le Mobile 2 commence avec

l'insertion de $t1$ et après, il reçoit une requête *DeleteInsert* ayant comme effet de supprimer tous les triplets justifiant la condition s = *"A"*. Du fait que le Mobile 1 insère et supprime le même triplet qui n'a pas été vu par le Mobile 2 lors de l'exécution de la requête *DeleteInsert*, les deux mobiles vont se retrouver avec deux états conflictuels.

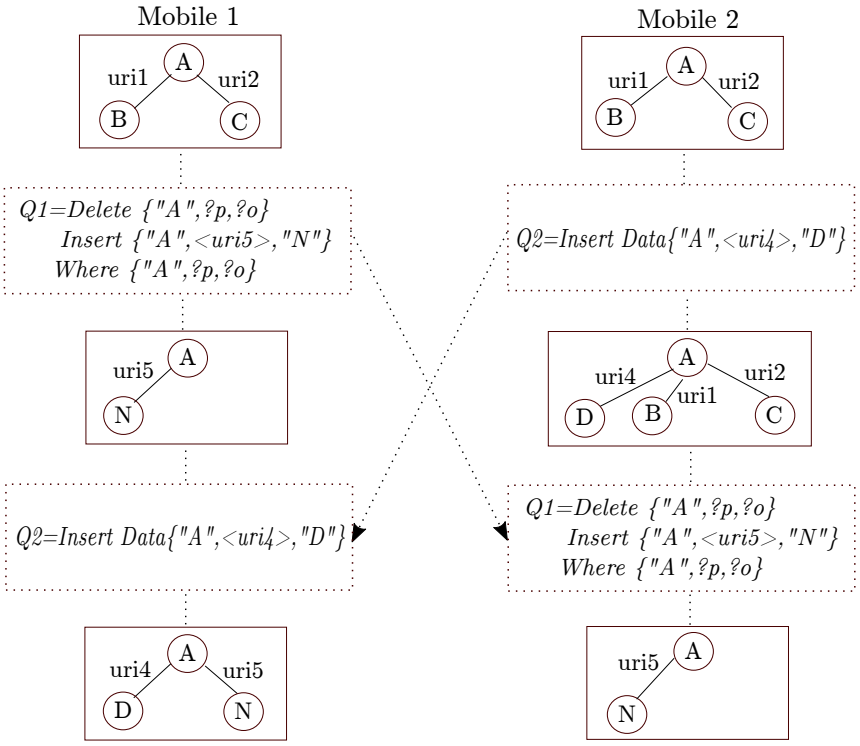


FIGURE 6.7 – Un contre exemple pour la commutativité des requêtes *DeleteInsert* et *InsertData*.

DeleteInsert et DeleteInsert. Comme le montre la figure 6.8, les deux mobiles appliquent et échangent deux requêtes DeleteInsert pour la mise-à-jour d'un graphe partagé et finissent par deux vues incohérentes. Initialement, le Mobile 1 remplace le triplet *("A", <uri2> "B")* par le triplet *("A", <uri3> "K")*. Quant au Mobile 2, il remplace le même triplet avec *("A", <uri4> "M")*. Après échange des deux requêtes, l'évaluation de la clause *Where("A",?p,"B")* donne un ensemble vide au niveau des deux mobiles. Par conséquent les requêtes $Q1$ et $Q2$ seront équivalentes à $No - Op$.

Proposition 6.2

Les différentes paires des requêtes Sparql-update InsertData et DeleteData, appliquées sur un graphe RDF, pour insérer ou supprimer un seul triplet sont commutatives.

Démonstration. Dans ce qui suit, nous donnons une démonstration de la commutativité de ces deux requêtes, selon les différentes paires possibles :

InsertData et InsertData. Il est à démontrer que :

$[InsertData(G, t1); InsertData(G, t2)] \equiv [InsertData(G, t2); InsertData(G, t1)]$.

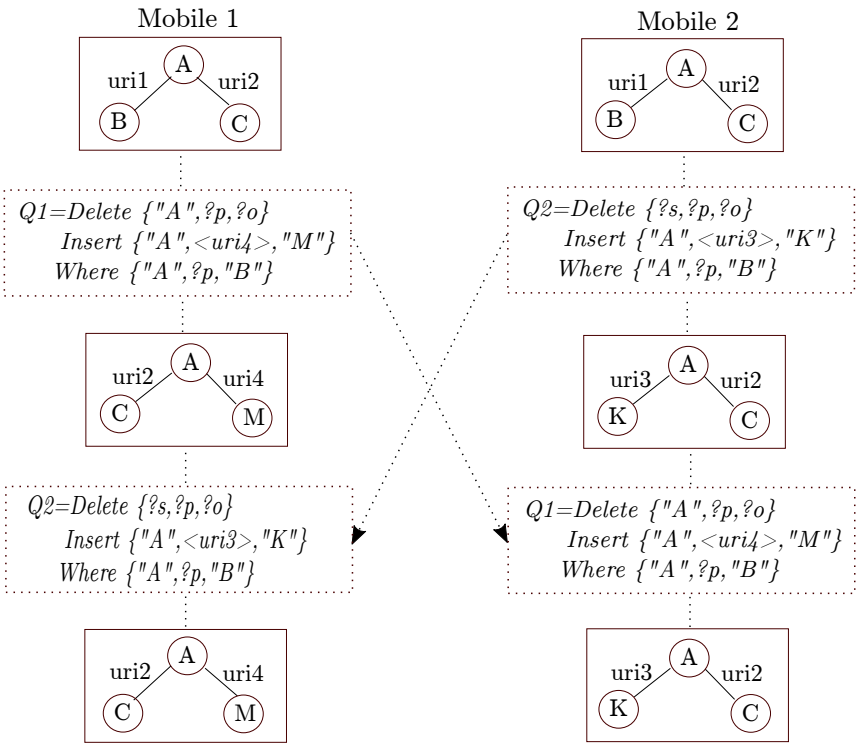


FIGURE 6.8 – Un contre exemple pour la commutativité d'un couple de requêtes *DeleteInsert*.

Pour démontrer l'équivalence de ces deux séquences, nous commençons d'un côté afin d'obtenir l'autre :

$$\begin{aligned}[InsertData(G,t1);InsertData(G,t2)] &\equiv InsertData(InsertData(G,t1),t2)\\ &= InsertData(G\cup t1,t2)\\ &= ((G\cup t1)\cup t2)\end{aligned}$$

$$\begin{aligned}&**\ commutativité\ de\ l'union\ **\\ &= ((G\cup t2)\cup t1)\\ &= InsertData(G\cup t2,t1)\\ &= InsertData(InsertData(G,t2),t1)\\ &\equiv [InsertData(G,t2);InsertData(G,t1)]\end{aligned}$$

InsertData et DeleteData.

Nous considérons deux cas :

(i) $t1 \neq t2$: il est à démontrer que :

$[InsertData(G,t1);DeleteData(G,t2)] \equiv [DeleteData(G,t2);InsertData(G,t1)]$.

<u>Démonstration :</u>

$$\begin{aligned}[InsertData(G,t1);DeleteData(G,t2)] &\equiv DeleteData(G\cup\{t1\},t2)\\ &\equiv \{G\cup\{t1\}\}\backslash\{t2\}\end{aligned}$$

$** A\backslash B = A \cap \bar{B} **$

$$= \{G \cup \{t1\}\} \cap \{\bar{t2}\}$$

$** distributivié\ de \cap\ sur \cup **$

$$= \{G \cap \{\bar{t2}\}\} \cup \{\{t1\} \cap \{\bar{t2}\}\}$$

$** t1 \neq t2 \Rightarrow \{t1\} \cap \{t2\} = \emptyset **$

$$= \{\{G \cap \{\bar{t2}\}\} \cup \{\{\{t1\} \cap \{\bar{t2}\}\}$$
$$\cup \{\{t1\} \cap \{t2\}\}\}$$

$** distributivité\ de \cap\ sur \cup **$

$$= \{G \cap \{\bar{t2}\}\} \cup \{\{t1\} \cap \{\{\bar{t2}\} \cup \{t2\}\}\}$$
$$= \{G \cap \{\bar{t2}\}\} \cup \{\{t1\} \cap U\}$$
$$= \{G \cap \{\bar{t2}\}\} \cup \{t1\}$$
$$= \{G\backslash\{t2\}\} \cup \{t1\}$$
$$\equiv [DeleteData(G, t2); InsertData(G, t1)]$$

(ii) $t1 = t2$: Nous pouvons considérer deux autres cas élémentaires :

cas1 : $t \in G \Rightarrow InsertData(G, t) = NoOp$

$$[InsertData(G, t); DeleteData(G, t)] \equiv [NoOp; DeleteData(G, t)]$$
$$\equiv [DeleteData(G, t); NoOp]$$
$$\equiv [DeleteData(G, t); InsertData(G, t)]$$

cas2 : $t \notin G \Rightarrow DeleteData(G, t) = NoOp$

$$[InsertData(G, t); DeleteData(G, t)] \equiv [InsertData(G, t); NoOp]$$
$$\equiv [NoOp; InsertData(G, t)]$$
$$\equiv [DeleteData(G, t); InsertData(G, t)]$$

DeleteData et DeleteData :

Il est à démontrer que :

$[DeleteData(G, t1); DeleteData(G, t2)] \equiv [DeleteData(G, t2); DeleteData(G, t1)]$.

<u>Démonstration :</u>

$$[DeleteData(G, t1); DeleteData(G, t2)] \equiv DeleteData(G\backslash\{t1\}, t2)$$
$$\equiv \{G\backslash\{t1\}\}\backslash\{t2\}$$

$** A\backslash B = A \cap \bar{B} **$

$$= \{G\backslash\{t1\}\} \cap \{\bar{t2}\}$$
$$= \{G \cap \{\bar{t1}\}\} \cap \{\bar{t2}\}$$
$$= G \cap \{\{\bar{t1}\} \cap \{\bar{t2}\}\}$$
$$= G \cap \{\{\bar{t2}\} \cap \{\bar{t1}\}\}$$

$$= \{G \cap \{\bar{t2}\}\} \cap \{\bar{t1}\}$$
$$= \{G \backslash \{t2\}\} \backslash \{t1\}$$
$$\equiv [DeleteData(G, t2); DeleteData(G, t1)].$$

La table 6.1 récapitule les résultats des démonstrations précédentes pour la commutativité des différentes paires de requêtes Sparql-update.

	DeleteInsert	InsertData	DeleteData
DeleteInsert			
InsertData		✓	✓
DeleteData		✓	✓

TABLE 6.1 – Commutativité des paires de requêtes Sparql-update.

En se basant sur les démonstrations des propositions 6.1 et 6.2, nous pouvons énoncer le théorème suivant :

Theorem 6.1

Les requêtes Sparql-update $InsertData(t)$ et $DeleteData(t)$ sont commutatives sur un graphe RDF.

7 Synchronisation des copies RDF

Le protocole de synchronisation MobiRdf, entièrement exécuté côté clone, vise à maintenir la cohérence des graphes RDF partagés. Il est basé sur l'éditeur commutatif précédemment déduit et utilise un schéma de réplication optimiste afin de permettre un accès concurrent aux différents répliques des graphes RDF partagés sur les mobiles et leurs clones. Cependant, cette synchronisation conduite par un éditeur Sparql-update commutatif, qui est exclusivement basée sur les deux requêtes commutatives $InsertData(t)$ et $DeleteData(t)$, n'empêche pas les utilisateurs d'exécuter toutes les autres requêtes. En effet, chaque requête appliquée côté mobile sera décomposée (par le clone correspondant) en un ensemble de requêtes commutatives qui vont être appliquées et diffusées vers les autres clones.

La commutativité assure la cohérence syntaxique d'une manière simple. Toutefois, le clone fait appel à des moteurs de raisonnement afin de maintenir la cohérence des relations sémantiques des données RDF. Ainsi cet appel permet de :

(i) Déduire les relations de dépendance entre les requêtes.

(ii) Optimiser les requêtes utilisateurs. Par exemple, si un mobile génère deux requêtes *InsertData(t)* et *DeleteData(t)* pour insérer et supprimer le même triplet, ces deux requêtes seront ignorées par le clone.

(iii) Déduire les effets secondaires des requêtes appliquées, c-à-d, inférer de nouveaux triplets suite à une insertion d'un triplet ou supprimer des triplets dépendants d'un autre triplet suite à sa suppression.

Notre mécanisme de synchronisation favorise une gestion optimale même en matière de consommation des ressources cloud. En effet, après diffusion des requêtes par un clone, les autres clones procèderont à leur intégration sans application des processus de raisonnement.

7.1 Modèle de cohérence

7.1.1 Principe

Comme représenté sur la figure 6.9, le protocole de synchronisation à travers l'application mobile MobiRdf est réparti sur deux niveaux :

(i) Le premier niveau représente la synchronisation du côté mobile ; ceci est la partie légère de ce protocole. Dans ce cas, chaque mobile peut appliquer n'importe quelle requête Sparql-update reçue de l'utilisateur sur son triplestore de graphe RDF local. Ensuite, les requêtes appliquées seront enveloppées dans un log (c-à-d, journal ou histoire des requêtes localement appliquées) afin d'être envoyées au clone correspondant. D'autre part, le mobile peut recevoir et appliquer directement des requêtes distantes qui ont été intégrées et envoyées par son clone sans aucun traitement supplémentaire.

(ii) Le second niveau concerne la synchronisation du côté clone ; c'est la partie lourde de ce protocole. Chaque clone supportera les tâches intensives de la synchronisation dans le but du maintien de la cohérence des répliques des graphes RDF partagés.

Dans le cas d'intégration des requêtes Sparql-update locales appliquées et envoyées par un mobile, le clone doit d'abord les décomposer en un ensemble de requêtes commutatives (*InsertData(t)* et *DeleteData(t)*). Un processus de gestion de version est ensuite lancé afin d'appliquer ces requêtes commutatives résultantes sur la même version du graphe RDF du mobile. Un autre processus d'optimisation est ensuite exécuté afin d'éliminer les requêtes inutiles. Après application, le clone utilise un raisonneur pour déduire les différentes dépendances sémantiques et causales liées à chaque requête. Et enfin, un log contenant ces requêtes commutatives annotées sera diffusé aux autres clones.

D'autre part, si un clone reçoit un log distant d'un autre clone, il doit d'abord vérifier l'éligibilité d'exécution des requêtes distantes qu'il contient. Les requêtes indépendantes seront directement appliquées, tandis que les dépendantes seront mises en attente jusqu'à l'exécution des requêtes dont elle dépendent. Cette phase s'achèvera par l'envoi d'un log contenant les requêtes distantes à intégrer sur le mobile.

Il est important de noter, que notre protocole de synchronisation à travers MobiRdf est générique dans le sens où il peut être utilisé par des travaux collaboratifs en mode P2P direct (une

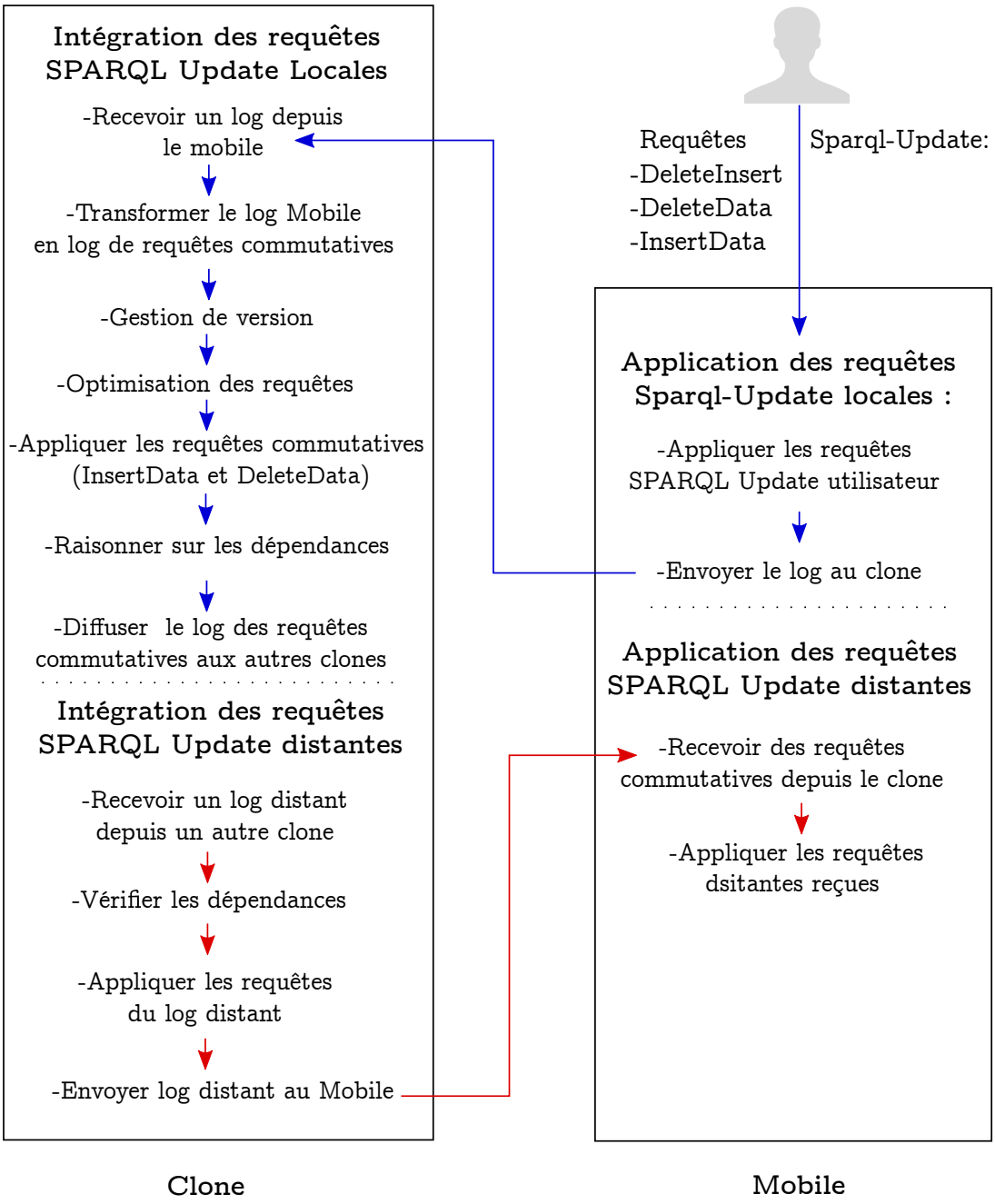


FIGURE 6.9 – Modèle global de synchronisation.

solution non-basée sur le cloud). Dans ce cas, la totalité du protocole de synchronisation peut être facilement implémenté sur le mobile. Mais cette solution ne convient pas à des mobiles ; elle est énormément coûteuse en terme de consommation de ressources (voir les expérimentations dans la section 8).

7.1.2 Notations

Comme notre modèle de cohérence est basé sur la commutativité des requêtes, chaque clone utilise seulement deux types de requêtes Sparql-update pour modifier un graphe RDF partagé : (i) $InsertData(t)$ pour insérer un triplet t et $DeleteData(t)$ pour supprimer un triplet t.

Nous définissons une modification [7] m comme un quadruplet $(ID\_clone, Num\_Modification, List\_Dep, Requête)$ où :

— Id_Clone : l'identificateur du clone émetteur de la modification m.
— $Num\_Modification$: un numéro d'ordre associé à chaque modification.
— $List\_Dep$: une liste contenant les numéros d'ordre des autres modifications desquelles la modification m dépend.
— $Requête$: une requête Sparql-update commutative, ($InsertData(t)$ ou $DeleteData(t)$). Nous définissons aussi une requête Sparql-update commutative Q comme un couple $(Type, Triplet)$ où $Type \in \{InsertData, DeleteData\}$ et $Triplet$ est un triplet RDF.

Nous utilisons les projections $m.ID\_Clone$, $m.Num\_Modification$, $m.List\_Dep$ et $m.Requête$ pour dénoter les composants correspondant de la modification m.

Un tampon (c-à-d, buffer) *log* est composé d'une séquence de modifications pour garder l'historique d'exécution de toutes les requêtes Sparql-update appliquées sur un graphe RDF.

Étant donné un log L, $L[i]$ dénotera la i-ième modification de ce log et $|L|$ est la taille de L.

La table 6.2 récapitule l'ensemble des structures de données utilisées par le modèle de cohérence.

Relations de dépendance. Par convention, les éditeurs collaboratifs utilisent le vecteur d'estampilles afin de déterminer les différentes relations de dépendance entre des opérations appliquées et échangées entre plusieurs collaborateurs d'une manière simultanée [43, 106]. Cependant, la technologie de vecteur d'estampilles n'est pas adéquate aux groupes dynamiques (les entrées de ce vecteur correspondent à un nombre fixe d'utilisateurs).

Au lieu du vecteur d'estampilles, nous utilisons une technique simple pour préserver les relations de dépendance entre les différentes requêtes de mise-à-jour Sparql. Notre technique est minimale dans le sens où elle n'utilise que des informations de dépendance directes entre les requêtes. Elle est indépendante du nombre d'utilisateurs et offre une grande concurrence en comparaison avec les vecteurs d'estampilles [64].

Comme nous utilisons un modèle de cohérence basé sur la commutativité, l'exécution des requêtes commutatives *InsertData* est indépendante de leur ordre de réception. Néanmoins, nous devons traiter deux cas de dépendances :

(i) Les dépendances causales qui concernent chaque requête *DeleteData* qui dépend causalement d'une requête précédente *InsertData*.

7. Nous utilisons le terme modification au lieu de requête afin de le distinguer du terme requête lié à Sparql-update. Toute modification est générée à partir d'une requête Sparql-update.

Structure de données	Description
Ac_Rec_Clone	Indicateur d'accusé de réception utilisé par un mobile, *Ac_Rec_Clone*=vrai : confirmation de la réception d'un log distant depuis le clone.
Ac_Rec_Mobile	Indicateur d'accusé de réception utilisé par un clone ; *Ac_Rec_Mobile*=vrai : confirmation de la réception d'un log local depuis le mobile.
ID_Clone	Identificateur d'un clone.
Cpt_Req_Dist	Compteur des requêtes commutatives distantes appliquées par un mobile.
Queue	Une file d'attente contenant des logs en instance d'intégration par un clone.
Global_Log_Clone	Histoire globale de toutes les requêtes appliquées/intégrées par un clone.
Log_Pour_Mob	Journal des requêtes Sparql-update distantes, intégrées par un clone depuis d'autres clones. Ce journal sera envoyé au mobile.
Log_Pour_Clone	Journal des requêtes Sparql-update locales, appliquées par un clone après réception depuis son mobile. Ce journal sera diffusé aux autres clones.
Etat_Version	Indice pointant sur une entrée dans le journal *Global_log_Clone* afin de repérer ou indiquer la dernière requête commutative distante vue (intégrée) par un mobile.
Commutative_Log	Un journal d'un clone contenant des requêtes commutatives, dérivées du processus d'analyse des requêtes Sparql-update envoyé par un mobile.

TABLE 6.2 – Structures de données utilisées dans l'algorithme de synchronisation d'un clone.

(ii) Les dépendances sémantiques dérivées d'un raisonnement basé sur des règles d'inférence pour décrire des relations sémantiques liant des ressources RDF (par exemple, un ordre sémantique d'un ensemble de faits).

Definition 6.9

(Relation de dépendance causale) *Soit m une modification générée et L un log où $L[i] = m_i$. Nous définissons la relation transitive $\overset{caus}{\rightarrow}$ comme suit :*

$m_i \overset{caus}{\rightarrow} m$ *si : m.Requête.Type=DeleteData , m_i.Requête.Type=InsertData, et m.Requête.Triplet=m_i.Requête.Triplet.*

Definition 6.10

(Relation de dépendance sémantique) *Soit m une modification générée, L un log où $L[i] = m_i$, G un graphe RDF, $Inference\_Set$ et Log_Set deux ensembles de triplets RDF où $Log\_Set \subset G$ et Inference_set = MoteurInférence (Log_Set,Règles,G) et MoteurInférence un raisonneur sur G pour inférer des triplets additionnels. Nous définissons la relation transitive $\overset{sem}{\rightarrow}$ comme suit :*

$m_i \overset{sem}{\rightarrow} m$ *si : m.Requête.Triplet $\in Inference\_set$ et m_i.Requête.Triplet $\in Log\_set$.*

Definition 6.11

(Concurrence) *Soit m une requête générée, L un log où $L[i] = m_i$. Si $m_i \overset{caus}{\not\rightarrow} m$ et $m_i \overset{sem}{\not\rightarrow} m$ alors m_i et m sont* indépendantes *(ou* concurrentes*).*

7.2 Synchronisation mobile/clone

La synchronisation côté mobile est épargnée de toute complication. Le mobile exécutera des tâches minimales tout en assurant une synchronisation efficace avec son clone. À cette fin, il utilisera un journal *Log_Mob* pour regrouper et envoyer des requêtes appliquées à chaque version v de son graphe local G.

Pour respecter l'ordre des requêtes distantes reçues par un mobile, son clone ne lui enverra un nouveau journal, sauf s'il a bien reçu (depuis le mobile) un accusé de réception (*Ret_Rec*) du journal précédent. Ce mécanisme permettra d'éviter des traitements supplémentaires liés à la causalité du côté mobile. D'autre part, les utilisateurs peuvent modifier leurs graphes RDF en mode on-line comme en off-line. Ainsi, les mobiles sont synchronisés avec leurs clones d'une manière non bloquante ; la réconciliation de la cohérence est supportée par les clones en utilisant un mécanisme de gestion des versions. Le mécanisme de synchronisation côté mobile peut être résumé par l'ensemble des étapes suivantes (voir l'algorithme 1) :

Génération des requêtes locales. Lorsqu'un mobile reçoit une requête utilisateur Sparql-upadate, il l'appliquera directement sur son triplestore local. Cette requête sera ensuite ajoutée au log du mobile *Log_Mob* (voir l'algorithme 1, ligne 15).

Envoi des requêtes locales au clone. Cette action consiste à envoyer le journal des requêtes Sparql-update appliquées localement par un mobile au clone correspondant. Ce journal correspond à une version v d'un graphe RDF. À cet effet, le mobile enverra également le numéro d'ordre de la dernière requête distante appliquée avec le journal concerné. De son côté, le clone utilisera ce numéro d'ordre pour appliquer la requête locale reçue sur la même version du graphe RDF (voir l'algorithme 1, ligne 16). Ainsi, chaque journal envoyé au clone ne doit contenir que des requêtes Sparql-update qui ont été localement appliquées sur la même version du graphe RDF.

Il faut remarquer aussi, qu'afin de maintenir le même ordre d'exécution, le mobile ne devrait pas envoyer un nouveau journal au clone sans qu'il soit sûr que l'ancien journal a été bien intégré par son clone. Cet ordre est maintenu via l'envoi d'un accusé de réception.

Réception des requêtes distantes. Lorsqu'un mobile reçoit un journal qui a été intégré par son clone, il l'appliquera directement sur son graphe RDF local. Ce journal contient des requêtes commutatives que le clone avait reçu depuis d'autres clones. Notons qu'après l'exécution de chaque requête, le mobile procèdera à l'incrémentation d'un compteur associé aux requêtes distantes appliquées (voir l'algorithme 1, ligne 24). Cette incrémentation sert à établir une liaison entre la version actuelle du graphe RDF du mobile et la dernière requête distante appliquée.

Algorithme 1 : Algorithme de synchronisation côté mobile

```
1  INITIALISER;
2  tant que non abandonner faire
3      si existe une requête Sparql-update Q alors
4          GÉNÉRER_REQUÊTE;
5      sinon
6          ENVOYER_LOG_AU_CLONE;
7          RECEVOIR_ACCUSÉ_DE_RÉCEPTION_DEPUIS_CLONE;
8          RECEVOIR_LOG_DEPUIS_CLONE;
9  Procédure INITIALISER()
10     Ac_Rec_Clone ← Vrai;
11     Cpt_Req_Dist ← 0;
12     Log_Mobile ← [ ];
13 Procédure GÉNÉRER_REQUÊTE(Q: Requête)
14     Appliquer (Q , G);
15     Log_Mob ← Log_Mob ; Q;
16 Procédure ENVOYER_LOG_AU_CLONE()
17     si Ac_Rec_Clone alors
18         Envoyer Au Clone Log_Mobile et Cpt_Req_Dist;
19         Log_Mobile ← [ ];
20 Procédure RECEVOIR_LOG_DEPUIS_CLONE()
21     Recevoir Log_Clone Depuis Clone;
22     pour tous les requête Q dans Log_Clone faire
23         Appliquer (Q , G);
24         Cpt_Req_Dist + + ;
25     Envoyer Ac_Rec_Mobile Au Clone;
```

7.3 Synchronisation clone/clone et clone/mobile

Pour la deuxième couche associée au mécanisme de synchronisation du protocole MobiRdf, nous présentons notre algorithme de contrôle de la concurrence clone/clone et clone/mobile. Pour cela, et après l'initialisation des différentes structures de données (voir l'algorithme 2, ligne1 et l'algorithme 3), deux types de tâches de synchronisation peuvent être déclenchées via deux événements majeurs : (i) la réception des logs distants contenant des requêtes Sparql-update commutatives qui ont été appliquées par d'autres clones et (ii) la réception des logs locaux contenant des requêtes Sparql-update appliquées par le mobile (voir l'algorithme 2, lignes 3 et 4).

Algorithme 2 : Algorithme de synchronisation côté clone

```
1 INITIALISER;
2 tant que non abandonner faire
3     RECEVOIR_LOG_DEPUIS_MOBILE;
4     RECEVOIR_LOG_DISTANT_DEPUIS_CLONE;
5     RECEVOIR_ACCUSÉ_DE_RÉCEPTION_DEPUIS_MOBILE;
6     INTÉGRER_REQUÊTES_DISTANTES;
```

Algorithme 3 : Procédure d'initialisation

```
1 Procédure INITIALISER()
      Ac_Rec_Mobile ← Vrai;
2     ID_Clone ← Identificateur local du clone;
3     Num_Modification ← 1;
4     Etat_Version ← 0;
5     Queue ← [ ];
6     Global_Log_Clone ← [ ];
7     Log_Pour_Mob ← [ ];
8     Commutative_Log ← [ ];
9     Log_Pour_Clone ← [ ];
```

Réception d'un log local depuis le mobile. Si un journal local de requêtes Sparql-update est reçu à partir d'un mobile, le clone enverra un accusé de réception à son mobile et invoquera la procédure INTEGRER_LOG_LOCAL (voir l'algorithme 4). Cette invocation consiste au lancement du processus d'intégration (présenté dans l'algorithme 5) de ces requêtes locales après leur réception à partir du mobile. Ceci consiste à appeler la fonction *ANALYSER_LOG_MOBILE* qui décompose chacune des requêtes Sparql-update reçue en un ensemble de requêtes commutatives : $InsertData(t)$ et $DeleteData(t)$. Les requêtes résultant de cette décomposition sont rangées dans le log $Commutative\_Log$.

Algorithme 4 : Réception d'un log local depuis le mobile

```
1 Procédure RECEVOIR_LOG_DEPUIS_MOBILE()
2     si existe un log local Log_Mob depuis mobile alors
3         Recevoir (Log_Mob, Cpt_Req_Dist);
4         Envoyer Ac_Rec_Clone au mobile;
5         INTÉGRER_LOG_LOCAL(Log_Mob, Cpt_Req_Dist);
```

Cependant, cette tâche de décomposition des requêtes reçues en requêtes commutatives peut poser un problème d'incohérence. La décomposition d'une requête *DeleteInsert (Wher-*

Algorithme 5 : Intégration des requêtes locales

```
1  Procédure INTÉGRER_LOG_LOCAL (Log_Mob, Cpt_Req_Dist)
2      Commutative_Log ← ANALYSER_LOG_MOBILE (Log_Mob) ;
3      VERSION_LOG_MOBILE (Commutative_Log, Cpt_Req_Dist);
4      OPTIMISER_LOG_MOBILE (Commutative_Log);
5      pour tous les requêtes Q dans Commutative_Log faire
6          DÉDUIRE_EFFETS_SECONDAIRES(Q);
7          Appliquer (Q , G);
8          M ← GÉNÉRER_MODIFICATION (Q, Global_Log_Clone);
9          Log_Pour_Clone ←Log_For_Clone + M;
10         Global_Log_Clone ← Global_Log_Clone + Q;
11         Num_Modification + +;
12     Broadcast aux clones (Log_Pour_Clone);
13     ENVOYER_LOG_AU_MOBILE();
```

Claus, DelClaus, InsClaus) est entamée par une sélection de tous les triplets respectant la clause *WherClaus* afin de déterminer l'ensemble des triplets à supprimer et/ou à insérer. En effet, cette sélection est appliquée sur la version actuelle du graphe RDF du clone qui peut être éventuellement différente de celle du mobile. Un clone pouvait appliquer des requêtes commutatives distantes qui n'ont pas été vues par le mobile au moment de son application de ces requêtes utilisateur, et par conséquent il y aura une sélection sur deux versions différentes du graphe RDF des deux côtés (c-à-d, clone et mobile). En l'occurrence, l'ensemble issu de la sélection peut contenir des triplets qui ne sont pas impliqués par l'ajout ni par la suppression. Pour remédier à ce problème, le clone doit effectuer une vérification des requêtes commutatives issues de la décomposition à partir de son log global en utilisant l'indicateur de version envoyé par le mobile. Pour cela, le processus d'intégration invoquera la procédure *VERSION_LOG_MOBILE* pour éliminer les requêtes commutatives (justifiant la clause *WherClaus*), mais qui n'ont pas été vues (c-à-d, appliquées) par le mobile quand il a appliqué la requête Sparql-update locale (voir l'algorithme 6). À cette fin, le processus d'intégration utilise la valeur du compteur de requêtes distantes (*Cpt_Req_Dist*) qui sont effectivement vues (appliquées) par le mobile lorsqu'il a appliqué la requête Sparql locale. Rappelons que ce compteur est envoyé avec le journal de requêtes Sparql-update locales par le mobile à son clone.

L'étape suivante consiste à optimiser les requêtes. Il s'agit d'exécuter la procédure *OPTIMISER_LOG_MOBILE* pour éliminer les requêtes inutiles suivantes :

(i) Toute requête *InsertData/DeleteData* pour insérer/supprimer un triplet qui a été déjà inséré/supprimé par inférence.

(ii) Tout couple de requêtes (*InsertData*, *DeleteData*) pour une insertion suivie d'une suppression d'un même triplet.

Algorithme 6 : Gestion des versions des requêtes locales

1 **Procédure** *VERSION_LOG_MOBILE(Commutative_Log : Log, Cpt_Req_Dist : entier)*
2 i← *Cpt_Req_Dist*;
3 **soit** *Etat_Version* l'index de la *i*ème requête commutative dans *Global_Log_Clone*;
4 **pour tous les** *requêtes* Q_k *in Commutative_Log* **faire**
j← |*Global_Log_Clone*|-1;
5 **tant que** *j*> *Version_State* **faire**
6 **si** $Q_k.triplet = Q_j.triplet$ **alors**
Commutative_Log ← *Commutative_Log* - Q_k;
7 quitter;
8 j - -;

Algorithme 7 : Effets secondaires d'une requête

1 **Procédure** *DÉDUIRE_EFFETS_SECONDAIRES(Q: requête)*
2 **si** *Q est une requête InsertData* **alors**
3 **soit** *Triple_Set* = *Moteur_Infrence*(*G*, *Q*);
4 **pour tous les** *triplets t dans Triple_Set* **faire**
5 **soit** $Q' = (InertData, t)$;
6 *Commutative_Log* ← *Commutative_Log* + Q';
7 *Log_Pour_Mob* ← *Log_Pour_Mob* + Q';
8 **sinon si** *Q est une requête DeleteData* **alors**
9 **soit** Q' une requête dans *Global_Log_Clone* **tel-que** $Q' \overset{caus}{\rightarrow} Q$;
10 **pour tous les** *requêtes Q" dans Global_Log_Clone qui dépendent de* Q' **faire**
11 **soit** *delQuery* = (*DeleteData*, *Q".triplet*);
12 *Commutative_Log* ← *Commutative_Log* + *delQuery*;
13 *Log_Pour_Mob* ← *Log_Pour_Mob* + *delQuery*;

Après filtration du log mobile et pour chacune de ses requêtes commutatives, le processus d'intégration exécutera un processus de raisonnement en appelant la procédure *DÉDUIRE_EFFETS_SECONDAIRES* (voir l'algorithme 7). Ceci consiste à déduire des effets secondaires causés par l'application d'une requête locale et par conséquent ceci peut mener à : (i) insérer de nouveaux triplets inférés suite à l'application d'une requête *InsertData* ou (ii) supprimer un ensemble de triplets dépendant d'un triplet supprimé par une requête *DeleteData*. Les nouvelles requêtes inférées seront insérées dans le journal commutatif (*Commutative_Log*) en vue d'une application ultérieure au sein du même processus d'intégration. Il est à noter aussi que

les requêtes secondaires issues de ce processus seront ajoutées au log *Log_Pour_Mob* qui sera envoyé au mobile pour l'intégration de ces effets secondaires.

Algorithme 8 : Génération d'une modification à partir d'une requête de mise-à-jour Sparql locale

1 **Fonction** *GÉNÉRER_MODIFICATION(Q:requête, L:Log): modification*
2 $m \leftarrow (ID\_Clone, Num\_Modification, null, Q)$;
3 $m' \leftarrow$ DÉPENDANCE_SÉMANTIQUE (m, L);
4 **si** *m.Requête.Type = DeleteData* **alors**
$m'' \leftarrow$ DÉPENDANCE_CAUSALE(m');
5 **sinon**
$m'' \leftarrow (m')$;
6 retourner (m'');

L'étape succédant le précédent raisonnement déductif consiste à appliquer toutes les requêtes commutatives du log *Commutative_log* sur le graphe RDF (voir l'algorithme 5, ligne 7) et de générer une modification pour chacune d'elles via un appel de la fonction *GÉNÉRER_MODIFICATION*. Cette fonction permet de préparer la propagation des modifications locales vers d'autres clones. Ainsi, pour chaque requête, elle forme une modification *q(ID_Clone, Num_Modification, Liste_Dep, Requête)* (voir l'algorithme 8). En outre, ce processus de génération invoquera deux autres fonctions pour déterminer les relations de dépendance de ces nouvelles modifications générées :

Algorithme 9 : Dépendances sémantiques d'une requête

1 **Fonction** *DÉPENDANCE_SÉMANTIQUE (m: modification , Global_Log_Clone:Log) : modification*
2 $m' \leftarrow m$;
3 **soit** $TripleSet = \emptyset$;
4 **pour tous les** *requêtes* m''_i*.Requête in Global_Log_Clone* **faire**
5 $TripleSet \leftarrow$ Moteur_Inférence $(m''_i$*.Requête,Règles,G*$)$;
si *m.Requête.Triple* $\in TripleSet$ *et m.Requête.Triple* $\notin m'.List\_Dep$ **alors**
$m'.List\_Dep \leftarrow m'.List\_Dep + m''_i.Num\_Requête$
6 retourner (m');

(i) *DÉPENDANCE_SÉMANTIQUE* (voir l'algorithme 9), qui fait appel à un moteur d'inférence pour la déduction des différentes dépendances sémantiques avec les requêtes déjà appliquées (notons que nous pouvons utiliser plusieurs moteurs d'inférence existants, par exemple le raisonneur Pellet [103])

Algorithme 10 : Dépendance causale d'une requête

1 **Fonction** *DÉPENDANCE_CAUSALE (m:modification , Global_Log_Clone:Log): modification*
2 $m' \leftarrow m$;
3 $i \leftarrow Etat\_Version$;
4 **tant que** $i > 0$ **faire**
$m'' \leftarrow$Global_Log_Clone[i];
5 **si** $m'.Requête.Triplet = q''.Requête.Triplet$ **et** $m''.Requête.Type=InsertData$ **alors**
6 $m'.List\_Dep \leftarrow m'.List\_Dep + m''_i.Num\_Requête$;
7 quitter;
i - -
8 retourner (m');

(ii) *DÉPENDANCE_CAUSALE* (voir l'algorithme 10), qui détermine les requêtes *InsertData* précédemment appliquées, dont chaque requête générée *DeleteData* dépend.

Enfin, le journal des requêtes générées sera diffusé aux autres clones. (voir l'algorithme 5, ligne 11).

Exemples sur la synchronisation clone/mobile. Pour mieux comprendre le mécanisme de la synchronisation clone/mobile, nous présentons deux exemples simples pour montrer le déroulement de cet algorithme.

La figure 6.10 montre un exemple sur le maintien de la cohérence syntaxique en utilisant la technique de gestion des versions. Nous supposons que le mobile et son clone démarrent d'un état partagé commun d'un graphe RDF contenant l'ensemble des triplets suivant $\{t1 = (\text{"}A\text{"}, << uri1 >>, \text{"}B\text{"}), t2 = (\text{"}A\text{"}, << uri2 >>, \text{"}C\text{"}), t3 = (\text{"}K\text{"}, << uri3 >>, \text{"}M\text{"})\}$.

À un moment donné et en mode off-line, le mobile exécute la requête Sparql-update $Q = Delete\{\text{"}A\text{"}, ?p, ?o\}Where\{\text{"}A\text{"}, ?p, ?o\}$ consistant à supprimer tous les triplets dont le sujet est "A". Il finira avec un graphe contenant un seul triplet $\{t3\}$. Au cours de cette même période de déconnexion, son clone reçoit la requête distante $Q' = InsertData\{\text{"}A\text{"}, << uri4 >>, \text{"}L\text{"}\}$ depuis un autre clone et il l'exécutera pour insérer le triplet $t4 = (\text{"}A\text{"}, << uri4 >>, \text{"}L\text{"})$. Il finira avec un graphe RDF contenant l'ensemble des triplets $\{t1, t2, t3, t4\}$.

Après rétablissement de la connexion réseau, le clone et son mobile échangent les deux requêtes appliquées. De son côté, le mobile exécutera directement la requête commutative reçue Q' sans aucun traitement additionnel et aura un graphe contenant les triplets $\{t3, t4\}$. D'autre part, après réception de la requête locale Q, le clone procèdera premièrement à sa décomposition en un ensemble de requêtes commutatives et il aura comme résultat le log *Commutative_Log* contenant les requêtes $q1 = DeleteData(\text{"}A\text{"}, << uri1 >>, \text{"}B\text{"})$, $q2 = DeleteData(\text{"}A\text{"}, << uri2 >>, \text{"}C\text{"})$, et $q3 = DeleteData(\text{"}A\text{"}, << uri4 >>, \text{"}L\text{"})$. Il exécutera ensuite un processus de gestion des versions conduisant à l'exclusion de la requête $q3$ du log des requêtes commutatives.

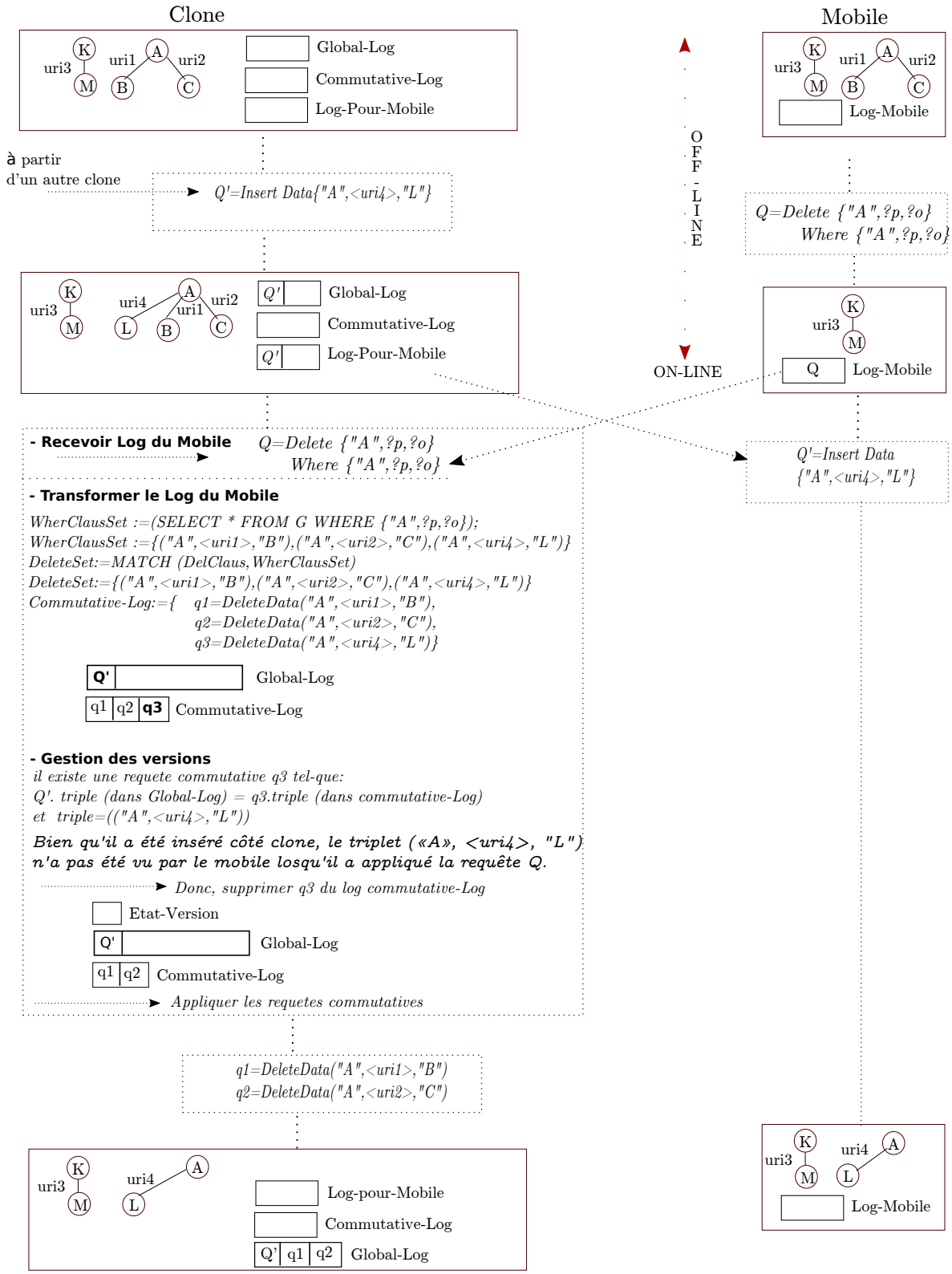


FIGURE 6.10 – Exemple de la synchronisation clone/mobile : cohérence syntaxique

En effet, si elle était naïvement appliquée, cette requête exclue aurait supprimé le triplet $t4$ qui n'a pas été vu par le mobile au moment de son application de la requête locale Q. Enfin, le clone appliquera seulement les deux requêtes commutatives $q1$ et $q2$ et comme son mobile, il finira avec un graphe RDF contenant les deux triplets $t3$ et $t4$.

D'autre part l'exemple présenté dans la figure 6.11 sert à décrire l'utilisation des relations de dépendance pour le maintien de la cohérence sémantique. Afin de simplifier le déroulement de l'algorithme, nous supposons que le clone et son mobile démarrent à partir d'un graphe RDF partagé vierge. Le clone dispose également d'un schéma contenant les deux règles d'inférence $R1$ et $R2$ qui signifient respectivement : (i) un taux de glycémie égal à 0.60 mène à une situation de confusion d'un patient et (ii) un taux de glycémie égal à 0.60 suivi par une situation de confusion implique un diagnostic d'hypoglycémie.

A un moment donné, le mobile applique deux requêtes InsertData $Q1$ et $Q2$ pour l'insertion de deux triplets *t1={glycémie, taux, 0.60}* et *t2={Patient, symptôme,confusion}*. Après réception de ces requêtes par le clone, l'exécution des actions 3, 4 et 5 n'aura aucun effet sur la liste des requêtes commutatives, puisque les deux requêtes $Q1$ et $Q2$ sont de nature commutatives et seront appliquées sur une version de graphe identique à celle du mobile. Le clone exécutera ensuite les actions suivantes :

- Déduction des effets secondaires des requêtes : L'utilisation de la règle $R2$ permet d'inférer un nouveau triplet *t3={Patient, a,hypoglycémie}* qui sera inséré via la requête $Q3$. Cela signifie que le clone a déduit un diagnostic d'hypoglycémie chez le patient en se basant sur les deux informations reçues du mobile.

- Application des requêtes : Le clone met à jour son graphe en appliquant les requêtes $Q1$, $Q2$ et $Q3$. Il utilise les règles de son schéma afin de déduire leurs dépendances. Par exemple, l'utilisation de la règle $R2$ permet de déduire que la requête $Q3$ dépend sémantiquement des deux requêtes $Q1$ et $Q2$. Le clone génèrera également une modification pour chaque requête commutative. Par exemple la modification *M3=(002658444448, 3, (1,2), Q3)* générée par le clone identifié par l'identificateur *Id_Clone=002658444448* correspond à la requête $Q3$ qui dépend des deux requêtes $Q1$ et $Q2$.

- Notification pour le mobile : Il s'agit d'envoyer la requête d'insertion du triplet inféré au mobile afin de lui notifier le diagnostic.

- Notification pour les clones : Diffusion des modifications générées vers les autres clones pour informer l'ensemble du staff médical sur les symptômes et diagnostic du patient.

Réception d'un log distant depuis un autre clone. La réception d'un log distant à partir d'un autre clone impliquera automatiquement sa mise en file d'attente pour une ultérieure intégration des requêtes qu'il contient (voir l'algorithme 11).

Le processus d'intégration de ces requêtes consiste à vérifier l'éligibilité d'exécution des modifications reçues (voir l'algorithme 12, ligne 2). Il exécutera ces requêtes reçues si toutes les requêtes précédentes dont elles dépendent sont déjà vues par le clone (c-à-d, appliquées).

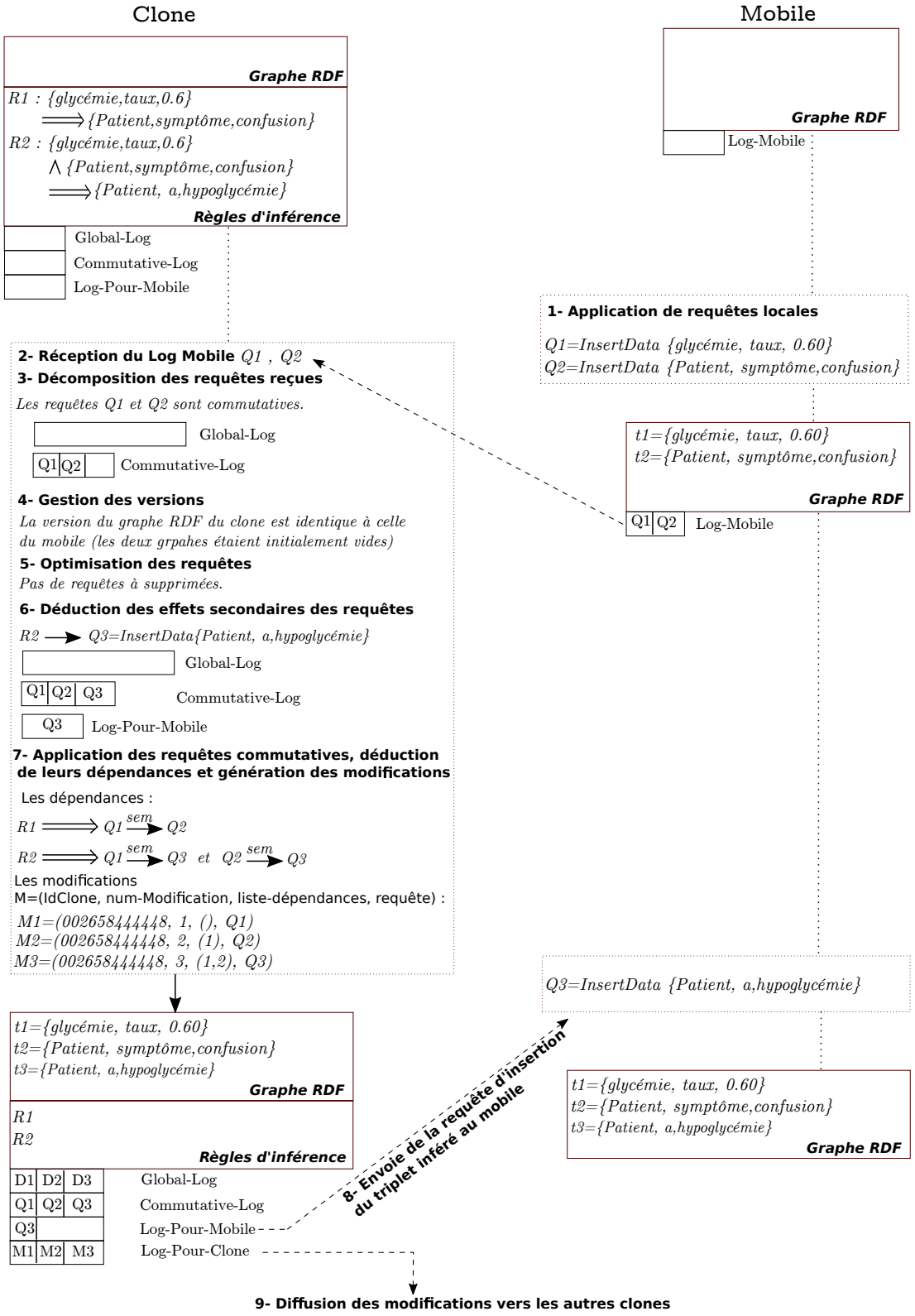


FIGURE 6.11 – Exemple de la synchronisation clone/mobile : cohérence sémantique.

Algorithme 11 : réception d'un log distant depuis un autre clone

1 **Procédure** *RECEVOIR_LOG_DISTANT_DEPUIS_CLONE()*
2 **si** *existe un log distant Log_Cl depuis autre clone* **alors**
3 **pour tous les** *modification m dans Log_Cl* **faire**
4 $Queue \leftarrow Queue + q$;

Algorithme 12 : Intégration des requêtes distantes

1 **Procédure** *INTÉGRER_REQUÊTES_DISTANTES()*
2 **si** *existe une requête q dans Queue* ***telle que*** *EST_EXÉCUTABLE(q)=Vrai* **alors**
3 Appliquer ($q.Query$, G);
4 $Queue \leftarrow Queue - q$;
5 $Global\_Log\_Clone \leftarrow Global\_Log\_Clone + q$;
6 $Log\_Pour\_Mob \leftarrow Log\_Pour\_Mob + q$;

En l'occurrence, si la liste des dépendances d'une modification est vide, la requête qui lui correspond peut être directement appliquée. Sinon, le processus d'intégration doit vérifier que toutes les modifications correspondant aux différentes entrées de la liste des dépendances de la modification reçue sont présentes dans le log *Global_Log_Clone* (voir l'algorithme 13). Si ce n'est le cas, la modification sera mise en attente.

Algorithme 13 : Éligibilité des modifications (requêtes) distantes

1 **Fonction** *EST_EXÉCUTABLE(m:modification): booléen*
2 $Ready \leftarrow Faux$;
3 **pour tous les** $Num\_Modification$ *dans* $m.List\_Dep$ **faire**
4 **si** *existe une modification* m' *dans Global_Log_Clone* ***telle que*** $Num\_Modification = m'.Num\_Modification$ **alors**
5 $Ready \leftarrow Vrai$
6 retourner ($Ready$);

Chaque modification appliquée est immédiatement insérée dans le log (*Log_Pour_Mobile*) qui sera par la suite envoyé au mobile (voir l'algorithme 12, ligne 5 et 6).

Il faut aussi remarquer, qu'afin de maintenir le même ordre d'exécution, le clone ne devrait pas envoyer un nouveau log distant au mobile sans qu'il soit sûr que l'ancien log a été bien intégré par le mobile en recevant un accusé de réception (voir l'algorithme 2, ligne 53). Toutefois, ce mécanisme d'accusé de réception n'est pas utilisé pour la synchronisation clone/clone.

Algorithme 14 : Envoi des requêtes commutatives distantes au mobile

```
1 Procédure ENVOYER_LOG_AU_MOBILE()
2     si Ac_Rec_Mobile alors
3         Envoyer pour mobile (Log_Pour_Mob);
4         Log_Pour_Mob ← [ ];
```

8 Implémentation et évaluation de MobiRdf

8.1 Implémentation

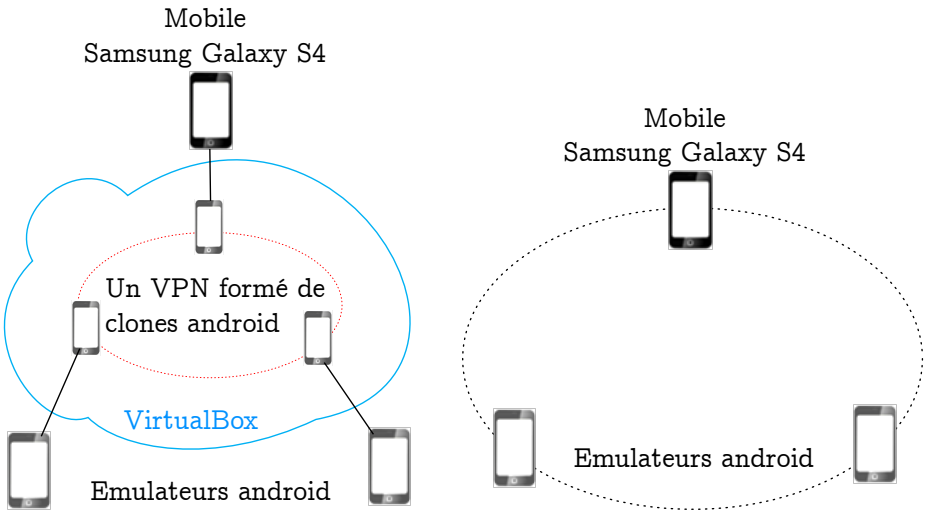
Dans cette section, nous présentons les résultats d'évaluation de notre système MobiRdf. Le but des expérimentations est d'évaluer l'impact de pousser les différentes tâches de synchronisation vers le cloud en matière de consommation énergétique, latence et trafic réseau. Pour ce faire, nous avons implémenté et comparé deux systèmes que nous appelons respectivement : (i) MobiRdf-basé-Cloud et (ii) MobiRdf-sans-Cloud. Ces deux systèmes simulent un simple groupe composé de trois (03) utilisateurs pour l'édition d'un graphe RDF partagé via trois (03) mobiles : un dispositif réel de type Samsung Galaxy S4 et deux émulateurs android.

Avec le système MobiRdf-basé-Cloud, les trois mobiles possèdent des clones sur VirtualBox et le protocole de synchronisation est distribué sur deux niveaux : un premier niveau accueillant la partie légère sur les mobiles (c-à-d, le dispositif réel et les deux émulateurs android) et un autre pour la partie lourde sur les clones. MobiRdf-basé-Cloud représente le système décrit dans ce chapitre (voir la figure 6.12 (a)).

D'autre part avec le système MobiRdf-sans-Cloud, le protocole de synchronisation est entièrement implémenté sur les mobiles qui doivent communiquer entre eux directement sans passer par l'intermédiaire d'un clone dans le cloud (voir la figure 6.12 (b)). Notons qu'à travers ces expérimentations nous avons désactivé la composante relative au raisonnement par inférence afin de pouvoir exécuter les processus de synchronisation sur les mobiles.

Pour le calcul des résultats d'évaluation, nous avons utilisé l'application mobile "PowerTutor" [11] qui permet de mesurer l'énergie consommée par les applications et l'outil "Wireshark" [10] qui détecte les interfaces réseau disponibles et calcul les trafics réseau correspondants. La table 6.3 décrit les différents environnements/outils utilisés pour la mise en œuvre de la plateforme de collaboration ainsi que leur éventuelle utilisation au sein des deux systèmes d'expérimentation.

D'autre part, la figure 6.13 illustre l'architecture utilisée pour l'implémentation de l'application collaborative MobiRdf. Elle résume l'étude effectuée dans ce chapitre tout en respectant les patrons de collaboration présentés précédemment dans le chapitre 4 où les classes d'extension sont présentées avec une couleur de fond grise.



(a) MobiRdf-basé-Cloud. (b) MobiRdf-sans-Cloud.

Figure 6.12 – Systèmes évalués dans les expérimentations.

Environnements/ outils	Description	MobiRdf basé-Cloud	MobiRdf sans-Cloud
Eclipse [14]	Environnement de déloppement Java	✓	✓
SDK ADI Bundle [13]	Kit de développement android installé avec Eclipse	✓	✓
API JENA [7]	Construction et MAJ des modèles RDF	✓	✓
VirtualBox [8]	Hyperviseur de virtualisation	✓	
MidBox	Clonage et construction du VPN	✓	
MV Android [12] X86-4.3-20130725.iso	Création des clones	✓	
PowerTutor [11]	Calcul de l'énergie consommée par les applications mobiles en temps réel	✓	✓
Wireshark [10]	Détection des interfaces réseau disponibles et calcul des trafics réseau correspondant	✓	✓

Table 6.3 – Environnements et outils utilisés pour le développement et l'implémentation de MobiRdf.

8.2 Évaluation

Pour chaque expérimentation, le triplestore du graphe local est initialisé avec 500 triplets RDF et chaque mobile utilise un script pour la génération aléatoire des requêtes Sparql-update : (i) *InsertData{s,p,o}*, (ii) *DeleteData{s,p,o}* et (iii) *Delete{V_s,V_p,V_o} insert{V_s,V_p,V_o} where{V_s,V_p,V_o}*, où :

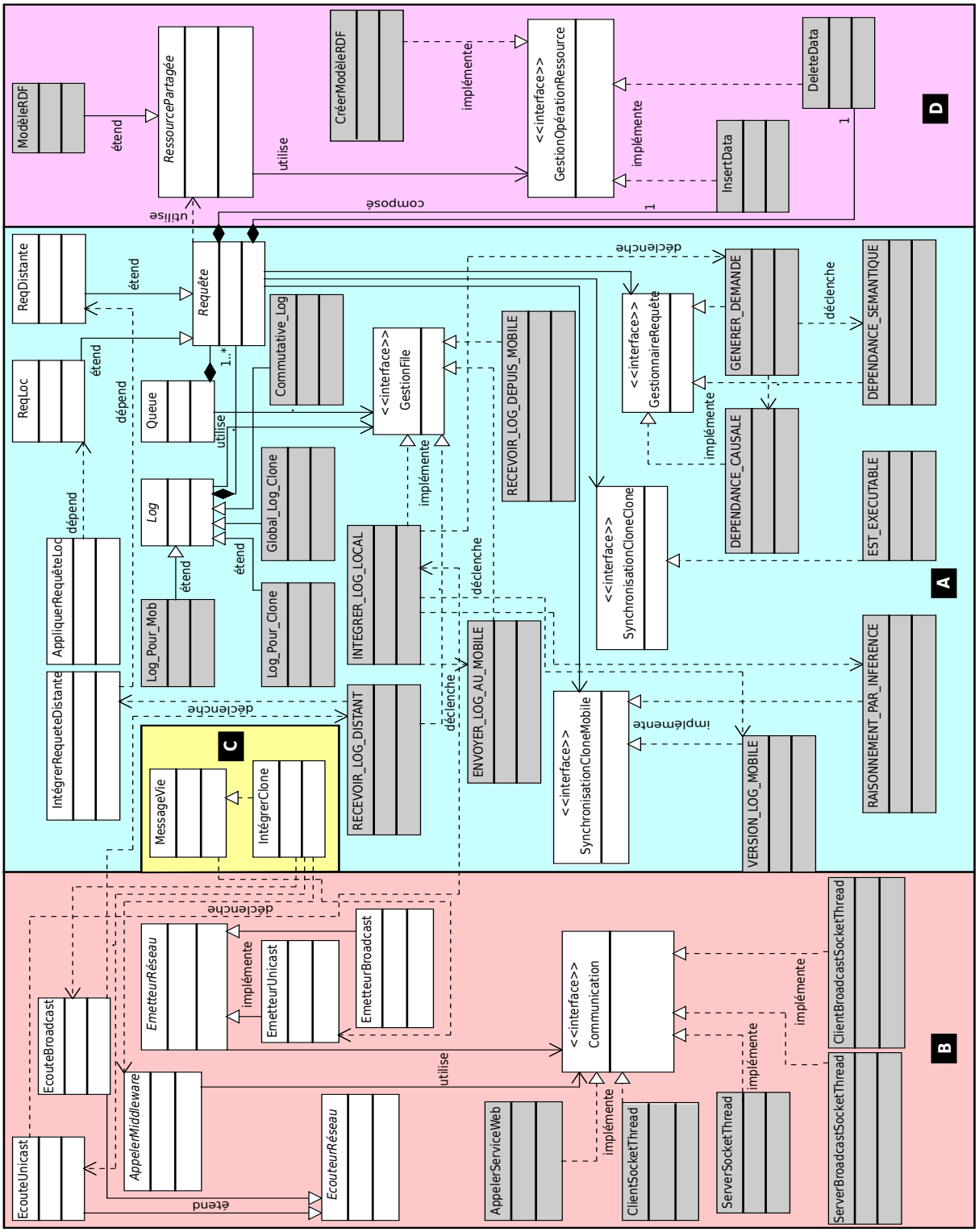


FIGURE 6.13 – Digramme de classes de l'application MOBIRDF.

- les sujets s et les objets o sont considérés comme des littéraux qui peuvent prendre un seul caractère comme valeur. Ces littéraux sont aléatoirement générés à partir de l'ensemble des cinq

caractères suivant: $S1 = \{"A", "B", "C", "D", "E"\}$.

- Les prédicats p sont générés à partir de l'ensemble des uri : $S2 = \{< u1 >, < u2 >, < u3 >, < u4 >, < u5 >\}$.

- Les variables de clauses V_s, V_p et V_o sont respectivement générées à partir des ensembles : $S3 = S1 \cup \{?s\}$, $S4 = S2 \cup \{?p\}$ et $S5 = S1 \cup \{?o\}$.

Les expérimentations sont réalisées selon les deux phases suivantes :

8.2.1 Phase d'édition

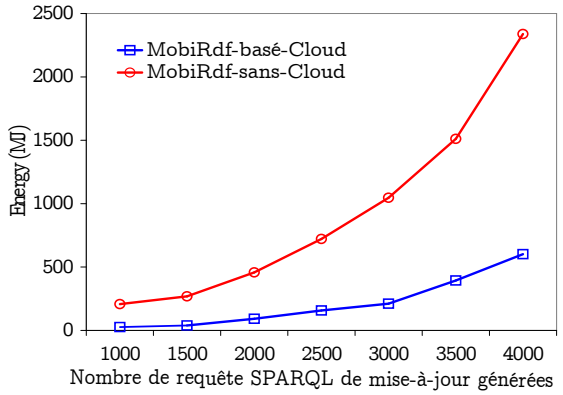
Les expérimentations réalisées durant cette phase consiste à observer et comparer les deux systèmes (MobiRdf-basé-Cloud et MobiRdf-sans-Cloud) en matière de consommation énergétique et trafic réseau au cours de l'édition d'un graphe RDF. Pour ce faire, chaque mobile génère et applique de façon aléatoire (de 1000 à 4000) requêtes Sparql-update. Notons que cette phase concerne uniquement la génération, exécution et diffusion des requêtes locales et par conséquent un seul mobile peut suffire pour l'évaluation des deux systèmes.

Consommation d'énergie. Pour mesurer l'énergie consommée par un mobile durant cette phase, nous avons testé les deux systèmes sous 3 cas de génération et application des requêtes Sparql-update : (i) 100% *InsetData*(t), (ii) 50% *InsertData*(t) et 50% *DeleteData*(t) et (iii) 100% *DeleteInsert*. Comme le montre la figure 6.14 (a,b,c), MobiRdf permet d'avoir 86,22% de gain d'énergie avec une solution basée cloud. L'énorme consommation au cours de cette phase d'un mobile qui implémente le protocole de synchronisation dans son intégralité (c-à-d, MobiRdf-sans-Cloud), est causée par l'exécution des tâches lourdes, à savoir : analyse, décomposition et diffusion des requêtes Sparql-update. Dans l'autre système, ces tâches sont exécutées par le clone du mobile.

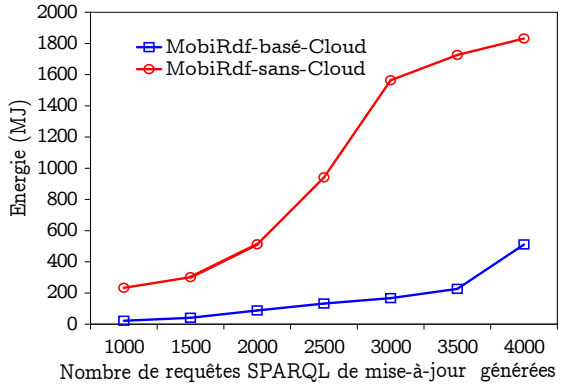
Trafic réseau. Dans cette expérimentation, nous mesurons le trafic réseau sortant du mobile suite à la génération aléatoire et application de 100% de requêtes Sparql-update *DeleteInsert*. Ce trafic sortant correspond à la diffusion des requêtes locales vers les autres mobiles. Comme le montre la figure 6.14 (d), la comparaison des deux systèmes met en évidence un gain de 69,49% à la faveur du système MobiRdf-basé-Cloud. Cette augmentation du trafic dans le système opposé est dû au nombre élevé des requêtes commutatives issues du processus de décomposition avant d'être diffusées aux autres mobiles. Tandis, qu'avec MobiRdf-basé-Cloud, le mobile appliquera la requête et l'enverra directement à son clone qui se chargera du reste (c-à-d, décomposition des requêtes et leur diffusion vers les autres clones).

8.2.2 Phase de mise-à-jour après re-connexion

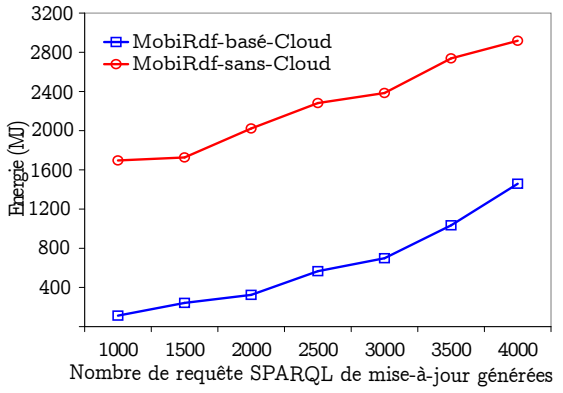
Avec les expérimentations réalisées durant cette phase, le dispositif mobile est en mode off-line (c-à-d, déconnecté), tandis que les deux émulateurs android sont en mode on-line, et tous génèrent et appliquent aléatoirement de 900 à 4500 requêtes Sparql-update (100% *DeleteIn-*



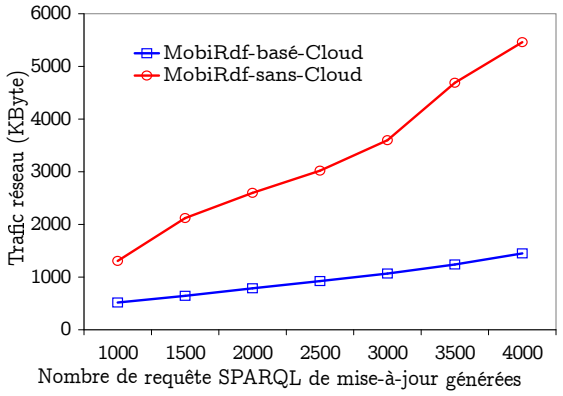
(a) Consommation d'énergie lors de l'édition du graphe RDF avec 100% de requêtes Sparql de mise-à-jour *InserData*.



(b) Consommation d'énergie lors de l'édition du graphe RDF avec 50% de requêtes InserData et 50% de requêtes *DeleteData*.



(c) Consommation d'énergie lors de l'édition du graphe RDF avec 100% de requêtes Sparql de mise-à-jour *DeleteInsert*.



(d) Trafic réseau lors de l'édition du graphe RDF avec 100% de requêtes Sparql de mise-à-jour *DeleteInsert*.

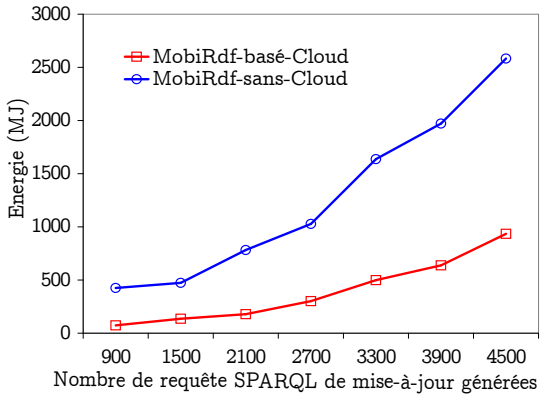
FIGURE 6.14 – Gain d'énergie et du trafic réseau avec MOBIRDF-basé-Cloud lors de la génération et envoi des requêtes Sparql-update.

sert). Nous calculons l'énergie consommée et le temps nécessaire pour que le mobile déconnecté puisse intégrer les requêtes distantes à partir des autres mobiles après sa reconnexion.

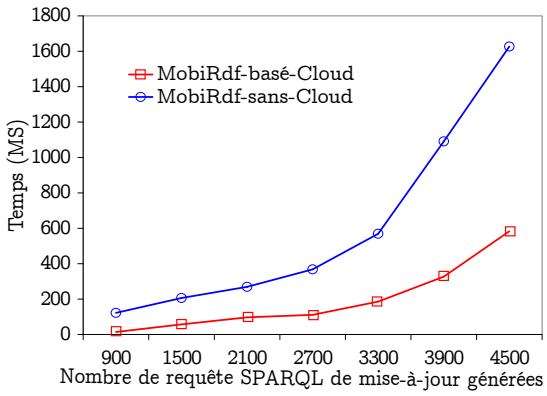
Consommation d'énergie. Comme le montre la figure 6.15(a), l'utilisation de MOBIRDF-basé-cloud permet un gain d'énergie de 76,98%. Avec cette solution, le dispositif mobile est toujours "omniprésent" (par le biais de son clone) même s'il est déconnecté. Le clone exécutera à la place de son mobile les tâches d'intégration intensives telles que la vérification des dépendances et de l'histoire. Contrairement au système opposé, le mobile n'aura qu'à appliquer directement les requêtes reçues depuis son clone sans aucun traitement additionnel.

Latence. Pour cette expérimentation, nous avons mesuré le temps nécessaire pour que le mobile déconnecté arrive à mettre à jour son graphe RDF local après sa reconnexion. La figure 6.15(b) montre un écart important qui donne avantage au système MOBIRDF-basé-cloud. Cet écart est

dû à la différence des tâches exécutées par le mobile dans les deux cas. Le mobile reconnecté du système avantageux (c-à-d, MobiRdf-basé-Cloud) applique directement les requêtes qui ont été intégrées (après vérification de leurs dépendances et de l'histoire) par son clone pendant son absence. Alors que dans le système opposé, le mobile doit exécuter les tâches suivantes : (i) réception des requêtes distantes à partir des autres mobiles actifs, (ii) vérification des dépendances, et (iii) application des requêtes. Ceci nécessite plus de temps.



(a) Consommation d'énergie après reconnexion et intégration des requêtes distantes.



(b) Temps nécessaire pour que le mobile déconnecté puisse mettre à jour son graphe RDF après reconnexion.

Figure 6.15 – Énergie consommée et temps nécessaire pour une mise à jour du graphe RDF après reconnexion.

9 Conclusion

Dans ce chapitre, nous avons présenté une nouvelle approche pour le partage des données liées mobiles via le cloud. Notre approche est une extension des patrons de collaboration détaillés dans le chapitre 4. Ainsi, l'application proposée MobiRdf offre un service de réplication des données utiles et un protocole de synchronisation (pour le partage des graphes RDF) bien adaptés aux réseaux P2P dynamiques dans le cloud.

L'exploitation des services proposés par MobiRdf permet de migrer l'exécution des tâches de collaboration lourdes et le stockage des données volumineuses vers le cloud. Afin d'évaluer l'impact de ce déploiement mobile vers le cloud, les expérimentations réalisées montrent un gain très important en matière de consommation d'énergie, latence et trafic réseau.

Chapitre 7

Conclusion générale

Sommaire

1 Résumé de recherche

Dans cette thèse nous nous sommes intéressés à la conception des applications collaboratives mobiles dans le cloud. Nous avons proposé une nouvelle approche basée sur le déploiement des tâches de collaboration mobile vers des plateformes virtuelles et décentralisées. Notre approche apporte des solutions aux problèmes liés à la virtualisation des groupes de collaboration P2P, à l'hétérogénéité des environnements cloud et mobile et à la centralisation de la synchronisation des données partagées. Dans cette section nous énumérons les principaux apports de nos travaux.

Nous avons d'abord présenté les différents concepts et technologies liés à notre contexte de recherche, à savoir le MCC et les applications collaboratives. Nous avons également fait un tour d'horizon sur les principaux travaux de recherche proposés autour du déploiement des tâches mobiles dans le cloud et de la conception des applications collaboratives. Afin de remédier aux problèmes non traités par les travaux existants, nous avons proposé les contributions suivantes :

Architecture réutilisable. Notre première contribution est la proposition de patrons de conception spécifiques pour les applications collaboratives mobiles s'exécutant sur des plateformes virtuelles distantes. Le modèle proposé s'articule autour de deux principales couches. La première couche concerne le déploiement des tâches mobiles et regroupe trois patrons pour la création des clones, la gestion des VPNs et la reprise après panne. Quant à la deuxième couche, elle est principalement composée des patrons de synchronisation des mises-à-jour appliquées, de réplication des données mobiles et de communication.

Un middleware de déploiement. Notre deuxième contribution est la proposition de MidBox, un middleware de déploiement agissant sur l'hyperviseur de virtualisation VirtualBox. L'objectif de ce middleware issu de l'instanciation de notre modèle générique est la préparation d'une plateforme de collaboration virtuelle composée de machines virtuelles android qui sont regroupées au sein de réseaux VPNs. Outre la préparation de cette plateforme, MidBox assure la dynamique des groupes sans interruption de la collaboration et veille au bon fonctionnement du système.

Un protocole d'édition collaborative des données liées mobiles. MobiRDF est notre troisième contribution pour la gestion du partage des données mobiles RDF. Conformément au modèle générique, l'application MobiRDF assure deux services essentiels : (i) un service d'édition collaborative qui offre un accès simultané à des copies de graphes RDF distribués sur les clones dans le cloud et (ii) un service de réplication permettant aux clones de sélectionner et de copier des données utiles sur les dispositifs mobiles.

L'exécution de MobiRDF est répartie sur deux niveaux. Le premier niveau concerne la partie "lourde" de l'application où la totalité des processus liés aux mécanismes de synchronisation et de réplication est prise en charge par les clones. Quant au deuxième niveau, il correspond à la partie "légère" s'exécutant sur les mobiles.

Les résultats expérimentaux de notre prototype démontrent que MobiRDF apporte un énorme gain en termes de consommation d'énergie, temps de réponse et trafic réseau au profit des mobiles.

2 Travaux futurs

A l'issue de ce travail de thèse, plusieurs pistes de recherche s'avèrent intéressantes à explorer :

2.1 Travaux à court terme

Une couche de sécurité et de contrôle d'accès. Notre proposition de déploiement des tâches de collaboration mobile sur une plateforme virtuelle peut constituer le noyau d'un système performant en matière de sécurité. En effet, l'adoption de la technologie de virtualisation via le clonage des dispositifs mobiles et la construction des VPNs permet d'isoler les données mobiles stockées ainsi que les traitements effectués de l'environnement cloud. Ainsi, ces données et traitements seront invisibles par les serveurs hébergeurs du cloud. D'autre part, l'utilisation des mécanismes de synchronisation totalement décentralisés est considérée comme un autre point fort de ce système en matière de sécurité. Une synchronisation basée sur un serveur d'ordonnancement central est vulnérable à tout type d'attaque. En effet, un serveur malveillant peut causer l'incohérence des différentes vues sur les ressources partagées.

Cependant, les réseaux P2P sont connus par leur vulnérabilité à plusieurs types d'attaques (par exemple, Sybil, intrusion, etc.). Comme extension de notre travail, nous proposons d'ajouter une nouvelle couche de sécurité distribuée sur les différents clones dans le cloud. En utilisant cette extension, les données partagées seront annotées avec des droits d'accès (c-à-d, droits de lecture et de mise-à-jour) et les clones doivent collaborer afin de préserver la cohérence en matière de droits d'accès. Pour la sécurisation des échanges de requêtes entre clones, un mécanisme de chiffrement/déchiffrement est souhaitable afin de préserver la confidentialité au niveau de l'environnement cloud.

Amélioration des mécanismes de réseautage. Nous considérons que nous avons proposé le patron de communication dans sa forme la plus simple. Une étude plus spécialisée de ce patron est souhaitable afin de prendre en compte les différents mécanismes de réseautage existants tels que le routage et les proxys.

Le routage définit des règles permettant d'acheminer les données entre les mobiles et leurs clones en passant par des réseaux différents. Cette solution est recommandée pour des applications déployées sur des clouds fondés sur des réseaux complexes et distribués. Quant aux services proxy, ils permettent d'établir des connexions mobile/clone indirectes. Une telle solution peut être envisagée via la configuration d'un proxy (sur le middleware de déploiement)

pour chaque mobile. Il prendra en charge la communication avec le clone et le mobile restera anonyme pour le cloud (c-à-d, son adresse IP n'est pas connue).

2.2 Travaux à long terme

Réalisation d'une plateforme en ligne de Benchmarking. L'architecture générique proposée par notre travail peut être réutilisée pour développer des applications collaboratives mobiles dans plusieurs domaines (par exemple, jeux, santé, gestion des catastrophes, etc.). Afin d'aider les développeurs, une mise en service d'une plateforme virtuelle de Benchmarking de ces applications peut constituer un outil standard pour l'évaluation des performances (par exemple, latence, trafic réseau, énergie, passage à l'échelle, etc.). En outre, l'élaboration d'un modèle de coût (comme un module de ce benchmark) est souhaitable afin d'estimer la consommation des ressources du côté cloud.

Chaque benchmark doit comporter un minimum de composants standards, à savoir, un scénario, une charge de travail, des métriques et des procédures de test [78]. Pour la conception de la plateforme de Benchmarking suggérée, nous proposons de suivre la même stratégie basée sur les patrons de conception pour la définition de ses composants standards :

— Scénario : Le scénario définit l'environnement de mise en service de ce benchmark (c-à-d, l'environnement MCC). Il décrit aussi les fonctionnalités attendues par les applications collaboratives à évaluer et leurs interfaces de déploiement.
— Charge de travail : Il s'agit de l'application collaborative à déployer et évaluer sur la plateforme de Benchmarking dans le cloud. Ce composant doit être synthétique dans le sens où il doit supporter plusieurs types d'applications. Nous proposons la réutilisation de nos patrons de collaboration pour la conception de la charge de travail.
— Métriques : L'évaluation des applications est basée sur un ensemble de métriques. Un modèle de coût peut être proposé pour estimer la consommation des ressources des deux côtés, mobile et cloud, en termes de métriques (tels que, énergie, temps de réponse, stockage, etc.).
— Procédures de test : Ce composant définit un jeu d'instruction pour le test et l'évaluation de l'application. Il est généralement basé sur un paramètre de passage à l'échelle permettant d'évaluer la qualité de l'application et ses limites. Par exemple, une procédure de test d'une application d'édition collaborative d'un texte partagé peut être définie à travers un script de génération aléatoire des opérations de mise-à-jour. La variation du nombre d'opérations générées permet d'évaluer cette application en terme de passage à l'échelle.

Bibliographie

[1] Android Debug Pont. http://developer.android.com/tools/help/adb.html.

[2] HADOOP. http://hadoop.apache.org.

[3] IDS1000-A All-in-one container data center. http://www1.huawei.com/enmobile/enterprise/products/itapp/dataCenterInfrastructure/All-in-oneContainerDataCenter/hw-204419.htm.

[4] Mobile Cloud Computing Forum. http://www.mobilecloudcomputingforum.com/.

[5] Page d'accueil Android. https://www.android.com/.

[6] Page d'accueil Apple. http://www.apple.com/.

[7] Page d'accueil de Apache Jena. https://jena.apache.org/.

[8] Page d'accueil de l'hyperviseur VirtualBox. https://www.virtualbox.org/.

[9] Page d'accueil de l'hyperviseur Xen. http://www.xen.org/products/xenhyp.html.

[10] Page d'accueil de Wireshark. https://www.wireshark.org/.

[11] Page d'accueil du moniteur PowerTutor. http://ziyang.eecs.umich.edu/projects/powertutor/.

[12] Page de téléchargement Android. http://www.android-x86.org/download.

[13] Page de téléchargement du kit de développement Android. http://developer.android.com/sdk/index.html.

[14] Page web d'accueil d'Eclipse. https://eclipse.org/.

[15] Page web d'accueil du conteneur Web TOMCAT. http://tomcat.apache.org/.

[16] Page web d'accueil du service Web AXIS. http://axis.apache.org/.

[17] Saleforce. http://www.salesforce.com/.

[18] Système de virtualisation IBM 2005. http://publib.boulder.ibm.com/infocenter/eserver/v1r2/topic/eicay/eicay.pdf.

[19] Rosa Alarcon, Luis A Guerrero, Sergio F Ochoa, and José A Pino. Analysis and design of mobile collaborative

applications using contextual elements. *Computing and Informatics*, 25(6):469–496, 2012.

[20] Safdar Ali and Stephan Kiefer. μor–a micro owl dl reasoner for ambient intelligent devices. In *Advances in Grid and Pervasive Computing*, pages 305–316. Springer, 2009.

[21] Michael Armbrust, O Fox, Rean Griffith, Anthony D Joseph, Y Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. M.: Above the clouds: a berkeley view of cloud computing. 2009.

[22] Khaled Aslan, Pascal Molli, Hala Skaf-Molli, and Stéphane Weiss. C-set: a commutative replicated data type for semantic stores. In *RED: Fourth International Workshop on REsource Discovery*, 2011.

[23] Mirza Basim Baig, Connor Fitzsimons, Suryanarayanan Balasubramanian, Radu Sion, and Donald E Porter. Cloudflow: Cloud-wide policy enforcement using fast vm introspection. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 159–164. IEEE, 2014.

[24] Rajesh Krishna Balan, Mahadev Satyanarayanan, So Young Park, and Tadashi Okoshi. Tactics-based remote execution for mobile computing. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 273–286. ACM, 2003.

[25] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[26] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta informatica*, 1(4):290–306, 1972.

[27] Sean Bechhofer. Owl: Web ontology language. In *Encyclopedia of Database Systems*, pages 2008–2009. Springer, 2009.

[28] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[29] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(2007):21, 2007.

[30] Dan Brickley and Ramanathan V Guha. {RDF vocabulary description language 1.0: RDF schema}. 2004.

[31] Jesse Chang, Rajesh Krishna Balan, and Mahadev Satyanarayanan. *Exploiting rich mobile environments*. PhD thesis, thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, 2005.

[32] Jason H Christensen. Using restful web-services and cloud computing to create next generation mobile applications. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 627–634. ACM, 2009.

[33] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execu-

tion between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[34] Lawrence Chung, Brian A Nixon, Eric Yu, and John Mylopoulos. *Non-functional requirements in software engineering*, volume 5. Springer Science & Business Media, 2012.

[35] A Covert. Google drive, icloud, dropbox and more compared: What is the best cloud option? *Technical Review*, 2012.

[36] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.

[37] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[38] Prasun Dewan. Architectures for collaborative applications. *Computer Supported Co-operative Work*, 7:169–193, 1999.

[39] Anind K Dey. Understanding and using context. *Personal and ubiquitous computing*, 5(1):4–7, 2001.

[40] Rose Dieng-Kuntz, Marek Minier, Olivier Corby, and Laurent Alamarguy. Building and using a medical ontology for knowledge management and cooperative work in a health care network. *Computers in Biology and Medicine*, 36(7):871–892, 2006.

[41] Hoang T Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing*, 13(18):1587–1611, 2013.

[42] Hoang T Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing*, 13(18):1587–1611, 2013.

[43] Clarence A Ellis and Simon J Gibbs. Concurrency control in groupware systems. In *Acm Sigmod Record*, volume 18, pages 399–407. ACM, 1989.

[44] Clarence A Ellis, Simon J Gibbs, and Gail Rein. Groupware: some issues and experiences. *Communications of the ACM*, 34(1):39–58, 1991.

[45] Ariel J Feldman, William P Zeller, Michael J Freedman, and Edward W Felten. Sporc: Group collaboration using untrusted cloud resources. In *OSDI*, volume 10, pages 337–350, 2010.

[46] Niroshinie Fernando, Seng W Loke, and Wenny Rahayu. Mobile cloud computing: A survey. *Future Generation Computer Systems*, 29(1):84–106, 2013.

[47] Jason Flinn, So Young Park, and Mahadev Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 217–226. IEEE, 2002.

[48] APACHE FOUNDATION. Hadoop Distributed File System. `https://hadoop.apache.org/`.

[49] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, pages 1–1, 2004.

[50] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: elements of, 1994.

[51] Paul Gearon, Alexandre Passant, and Axel Polleres. Sparql 1.1 update. *Working draft WD-sparql11-update-20110512, W3C (May 2011)*, 2012.

[52] Hans W Gellersen, Albercht Schmidt, and Michael Beigl. Multi-sensor context-awareness in mobile devices and smart artifacts. *Mobile Networks and Applications*, 7(5):341–351, 2002.

[53] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[54] INC. GOOGLE. Google App Engine. `https://cloud.google.com/appengine/`.

[55] Nadir Guetmi, Moulay Driss Mechaoui, Abdessamad Imine, and Ladjel Bellatreche. Mobile collaboration: a collaborative editing service in the cloud. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 509–512. ACM, 2015.

[56] Steve Harris, Andy Seaborne, and Eric Prudhommeaux. Sparql 1.1 query language. *W3C Recommendation*, 21, 2013.

[57] Cory Henson, Krishnaprasad Thirunarayan, and Amit Sheth. An efficient bit vector approach to semantics-based machine perception in resource-constrained devices. In *The Semantic Web–ISWC 2012*, pages 149–164. Springer, 2012.

[58] Valeria Herskovic, Sergio F Ochoa, José Pino, et al. Modeling groupware for mobile collaborative work. In *Computer Supported Cooperative Work in Design, 2009. CSCWD 2009. 13th International Conference on*, pages 384–389. IEEE, 2009.

[59] Valeria Herskovic, Sergio F Ochoa, José A Pino, and H Andrés Neyem. The iceberg effect: Behind the user interface of mobile collaborative systems. *J. UCS*, 17(2):183–201, 2011.

[60] Doan B Hoang and Lingfeng Chen. Mobile cloud for assistive healthcare (mocash). In *Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific*, pages 325–332. IEEE, 2010.

[61] Ian Horrocks, Peter F Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, Mike Dean, et al. Swrl: A semantic web rule language combining owl and ruleml. *W3C Member submission*, 21:79, 2004.

[62] Gonzalo Huerta-Canepa and Dongman Lee. A virtual cloud computing provider for mobile devices. In *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, page 6. ACM, 2010.

[63] Luis-Daniel Ibáñez, Hala Skaf-Molli, Pascal Molli, and Olivier Corby. Live

linked data: synchronising semantic stores with commutative replicated data types. *International Journal of Metadata, Semantics and Ontologies 4, 8*, 8(2):119–133, 2013.

[64] Abdessamad Imine. Coordination model for real-time collaborative editors. In *Coordination Models and Languages*, pages 225–246. Springer, 2009.

[65] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. Cuckoo: a computation offloading framework for smartphones. In *Mobile Computing, Applications, and Services*, pages 59–79. Springer, 2012.

[66] Rabia Khan and Amjad Mehmood. Realization of interoperability and portability among open clouds by using agent's mobility and intelligence. *International Journal of Multidisciplinary Sciences and Engineering*, 3(7):7–11, 2012.

[67] Michael Kircher and Prashant Jain. *PATTERN ORIENTED SOFTWARE ARCHITECTURE*, volume 3. John Wiley & Sons, 2005.

[68] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[69] A Klein, C Mannweiler, and HD Schotten. A framework for intelligent radio network access based on context models. In *Proceedings of the 22nd WWRF meeting*, 2009.

[70] Andreas Klein, Christian Mannweiler, Joerg Schneider, and Hans D Schotten. Access schemes for mobile cloud computing. In *Mobile Data Management (MDM), 2010 Eleventh International Conference on*, pages 387–392. IEEE, 2010.

[71] Graham Klyne and Jeremy J Carroll. Resource description framework (rdf): Concepts and abstract syntax. 2006.

[72] Sokol Kosta, Vasile Perta, Julinda Stefa, Pan Hui, and Alessandro Mei. Clonedoc: exploiting the cloud to leverage secure group collaboration mechanisms for smartphones. In *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*, pages 19–20. IEEE, 2013.

[73] Sokol Kosta, Vasile Claudiu Perta, Julinda Stefa, Pan Hui, and Alessandro Mei. Clone2clone (c2c): Peer-to-peer networking of smartphones on the cloud. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud13)*, 2013.

[74] F John Krautheim. Private virtual infrastructure for cloud computing. *Proc of HotCloud*, 2009.

[75] Charles W Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.

[76] Karthik Kumar, Jibang Liu, Yung-Hsiang Lu, and Bharat Bhargava. A survey of computation offloading for mobile systems. *Mobile Networks and Applications*, 18(1):129–140, 2013.

[77] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[78] Max Lehn, Tonio Triebel, Christian Gross, Dominik Stingl, Karsten Saller,

Wolfgang Effelsberg, Alexandra Kovacevic, and Ralf Steinmetz. Designing benchmarks for p2p systems. In *From active data management to event-based systems and more*, pages 209–229. Springer, 2010.

[79] Leslie Liu, Randy Moulic, and Dennis Shea. Cloud service portal for mobile device management. In *e-Business Engineering (ICEBE), 2010 IEEE 7th International Conference on*, pages 474–478. IEEE, 2010.

[80] R Luce. Coalescing, event commutativity, and theories of utility. *Journal of Risk and Uncertainty*, 16(1):87–114, 1998.

[81] Eugene E Marinelli. Hyrax: cloud computing on mobile devices using mapreduce. Technical report, DTIC Document, 2009.

[82] Vidal Martins, Esther Pacitti, and Patrick Valduriez. Survey of data replication in p2p systems. 2006.

[83] Oleksiy Mazhelis, Jouni Markkula, and Markus Jakobsson. Specifying patterns for mobile application domain using general architectural components. pages 157–172, 2005.

[84] Moulay Driss Mechaoui, Nadir Guetmi, and Abdessamad Imine. MiCa: Lightweight and mobile collaboration across a collaborative editing service in the cloud. *Peer-To-Peer Networking and Applications*, 2016.

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

[86] Sergio Munoz, Jorge Pérez, and Claudio Gutierrez. Minimal deductive systems for rdf. In *The Semantic Web: Research and Applications*, pages 53–67. Springer, 2007.

[87] Dirk Neumann, Christian Bodenstein, Omer F Rana, and Ruby Krishnaswamy. Stacee: enhancing storage clouds using edge devices. In *Proceedings of the 1st ACM/IEEE workshop on Autonomic computing in economics*, pages 19–26. ACM, 2011.

[88] Andrés Neyem, Sergio F Ochoa, José A Pino, and Rubén Darío Franco. A reusable structural design for mobile collaborative applications. *Journal of systems and software*, 85(3):511–524, 2012.

[89] H Andrés Neyem, Sergio F Ochoa, and José A Pino. Integrating service-oriented mobile units to support collaboration in ad-hoc scenarios. *J. UCS*, 14(1):88–122, 2008.

[90] Evan W Patton and Deborah L McGuinness. A power consumption benchmark for reasoners on mobile devices. In *The Semantic Web–ISWC 2014*, pages 409–424. Springer, 2014.

[91] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 288–297. ACM, 1996.

[92] Juan Rodríguez-Covili, Sergio F Ochoa, José A Pino, Valeria Herskovic, Jesus Favela, David Mejía, and Alberto L Morán. Towards a reference architecture for the design of mobile shared workspaces. *Future Generation Computer Systems*, 27(1):109–118, 2011.

[93] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys (CSUR)*, 37(1):42–81, 2005.

[94] Johannes Sametinger. *Software engineering with reusable components*. Springer Science & Business Media, 1997.

[95] Farshad A Samimi, Philip K McKinley, and S Masoud Sadjadi. Mobile service clouds: a self-managing infrastructure for autonomic mobile computing services. In *Self-Managed Networks, Systems, and Services*, pages 130–141. Springer, 2006.

[96] Brahima Sanou. The world in 2014: Ict facts and figures, 2014.

[97] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, 8(4):14–23, 2009.

[98] Lutz Schubert, Keith G Jeffery, and Burkard Neidecker-Lutz. *The Future of Cloud Computing: Opportunities for European Cloud Computing Beyond 2010:–expert Group Report*. European Commission, Information Society and Media, 2010.

[99] Jeffrey Shafer, Scott Rixner, and Alan L Cox. The hadoop distributed filesystem: Balancing portability and performance. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 122–133. IEEE, 2010.

[100] Marc Shapiro and Nuno Preguiça. Designing a commutative replicated data type. *arXiv preprint arXiv:0710.1784*, 2007.

[101] Marc Shapiro and Nuno Preguiça. Designing a commutative replicated data type. *arXiv preprint arXiv:0710.1784*, 2007.

[102] Amit K Sharma and Priyanka Soni. Mobile cloud computing (mcc): Open research issues. *International Journal of Innovations in Engineering and Technology (IJIET)*, pages 24–27, 2013.

[103] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007.

[104] Subashini Subashini and Veeraruna Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of network and computer applications*, 34(1):1–11, 2011.

[105] Maher Suleiman, Michele Cart, and Jean Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In *Proceedings of the international ACM SIGGROUP conference on Supporting group work: the integration challenge*, pages 435–445. ACM, 1997.

[106] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 59–68. ACM, 1998.

[107] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation

in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(1):63–108, 1998.

[108] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(1):63–108, 1998.

[109] Chengzheng Sun, Steven Xia, David Sun, David Chen, Haifeng Shen, and Wentong Cai. Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 13(4):531–582, 2006.

[110] David Sun, Steven Xia, Chengzheng Sun, and David Chen. Operational transformation for collaborative word processing. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 437–446. ACM, 2004.

[111] Hong-Siang Teo. An activity-driven model for context-awareness in mobile computing. In *Proceedings of the 10th international conference on Human computer interaction with mobile devices and services*, pages 545–546. ACM, 2008.

[112] Will Tracz. *Software reuse: emerging technology*. IEEE Computer Society Press, 1988.

[113] Giovanni Tummarello, Christian Morbidoni, Reto Bachmann-Gmür, and Orri Erling. *RDFSync: efficient remote synchronization of RDF models*. Springer, 2007.

[114] Narseo Vallina-Rodriguez and Jon Crowcroft. Erdos: achieving energy savings in mobile os. In *Proceedings of the sixth international workshop on MobiArch*, pages 37–42. ACM, 2011.

[115] Rob V Van Nieuwpoort, Jason Maassen, Gosia Wrzesińska, Rutger FH Hofman, Ceriel JH Jacobs, Thilo Kielmann, and Henri E Bal. Ibis: a flexible and efficient java-based grid programming environment. *Concurrency and Computation: Practice and Experience*, 17(7-8):1079–1107, 2005.

[116] Luis M Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008.

[117] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[118] Timothy Wood, Alexandre Gerber, KK Ramakrishnan, Prashant Shenoy, and Jacobus Van der Merwe. The case for enterprise-ready virtual private clouds. *Usenix HotCloud*, 2009.

[119] Huanhuan Xia, Tun Lu, Bin Shao, Xianghua Ding, and Ning Gu. Hermes: On collaboration across heterogeneous collaborative editing services in the cloud. In *Computer Supported Cooperative Work in Design (CSCWD), Proceedings of the 2014 IEEE 18th International Conference on*, pages 655–660. IEEE, 2014.

[120] Sheng-Cheng Yeh, Ming-Yang Su, Hui-Hui Chen, and Chun-Yuen Lin. An

efficient and secure approach for a cloud collaborative editing. *Journal of Network and Computer Applications*, 36(6):1632–1641, 2013.

[121] Stefan Zander and Bernhard Schandl. Context-driven rdf data replication on mobile devices. *Semantic Web Journal Special Issue on Real-time and Ubiquitous Social Semantics*, 3(2):131–155, 2012.

[122] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[123] Zehua Zhang and Xuejie Zhang. Realization of open cloud computing federation based on mobile agent. In *Intelligent Computing and Intelligent Systems, 2009. ICIS 2009. IEEE International Conference on*, volume 3, pages 642–646. IEEE, 2009.

[124] Weimin Zheng, Pengzhi Xu, Xiaomeng Huang, and Nuo Wu. Design a cloud storage platform for pervasive computing environments. *Cluster Computing*, 13(2):141–151, 2010.

Table des figures

Liste des tableaux

Listings

Glossaire

P2P : Pair-à-Pair
MCC : Mobile Cloud Computing
MC : Mobile Computing
Données mobile : Données du dispositifs mobiles
Données clone : Données du clone
Mobile : Dispositif mobile
Clone : Clone du dispositif mobile
Déploiement mobile : déploiement des tâches mobiles vers le cloud
VPN : Réseau privé virtuel
OT : Transformées Opérationnelles
CRDT : Type de données réplicatif et commutatif
SLA : Service-Level Agreement
DHCP : Protocole de Configuration Dynamique d'Hôte
MV : Machine Virtuelle
Triplestore : Base de données pour le stockage et la récupération de données RDF

Résumé

De nos jours, nous assistons à une énorme avancée des applications collaboratives mobiles. Ces applications tirent parti de la disponibilité croissante des réseaux de communication et de l'évolution impressionnante des dispositifs mobiles. Cependant, même avec un développement en accélération, ils demeurent toujours pauvres en ressources (une courte durée de vie des batteries et une connexion réseau instable) et moins sécurisés. Dans le cadre de notre travail, nous proposons une nouvelle approche basée sur le déploiement des tâches de collaboration mobile vers le cloud. La gestion d'une virtualisation efficace assurant la continuité de la collaboration pour des réseaux pair-à-pair est une tâche très difficile. En effet, l'aspect dynamique des groupes (où les utilisateurs peuvent joindre, quitter ou changer de groupes) ainsi qu'une vulnérabilité aux pannes peuvent affecter la collaboration. En outre, la conception de telles applications doit prendre en compte l'hétérogénéité des environnements cloud et mobile. Contrairement aux travaux existants , nous proposons une architecture réutilisable de haut niveau basée sur les patrons de conception et qui peut être facilement adaptée à plusieurs environnements clouds et mobiles hétérogènes. Nos modèles ont été utilisés comme base pour la conception de : (i) MidBox, une plate-forme virtuelle pour exécuter des applications collaboratives mobiles sur un cloud privé et (ii) MobiRDF, un service de cloud décentralisé pour la manipulation en temps réel des connaissances via des documents RDF partagés.

Mots clés : Patrons de Cloud, Collaboration Mobile, Middleware de Clonage, Synchronisation, Usage du Cloud pour les Mobiles.

Abstract

Nowadays we assist to an enormous progress of mobile collaborative applications. These applications take advantage of the increasing availability of communication networks and the impressive evolution of mobile devices. However, even with a developing acceleration, they are still poor in resources (short life of batteries and unstable network connections) and less secure. In the context of our work, we propose a new approach based on the deployment of mobile collaboration tasks to the cloud. The management of efficient virtualization ensuring continuity of collaboration in peer-to-peer networks is a very difficult task. Indeed, the dynamic aspect of the groups (where users can join, leave or change groups) and a vulnerability to failures can affect the collaboration. In addition, the design of such applications must consider the heterogeneity of cloud and mobile environments. Unlike existing works, we propose a reusable high-level architecture based on patterns design, which can be easily adapted to heterogeneous clouds and mobile environments. Our models have been used as basis for the design of: (i) MidBox, a virtual platform for running mobile collaborative applications on a private cloud and (ii) MobiRDF a decentralized cloud service for real-time manipulation of knowledge via shared RDF documents.

Key words : Cloud Patterns, Mobile Collaboration, Cloning Middleware, Synchronization, Mobile Cloud Computing.
