La décision répartie pour le déploiement distribué

Chafik Merkak

▶ **To cite this version:**

Chafik Merkak. La décision répartie pour le déploiement distribué. Calcul parallèle, distribué et partagé [cs.DC]. 2011. dumas-00636750

HAL Id: dumas-00636750

https://dumas.ccsd.cnrs.fr/dumas-00636750

Submitted on 28 Oct 2011

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

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

La décision répartie pour le déploiement distribué

Auteur : **Chafik Merkak**
Responsable : **Fabien Dagnat**

Master Recherche en Informatique

– Systèmes et Objets Communiquant Mobiles –

TELECOM Bretagne
2011

Remerciements

Mes remerciements s'adressent en premier lieu à mon encadreur Monsieur Fabien Dagnat pour sa confiance et ses conseils qui m'ont permis de progresser sans cesse durant ces six mois de stage.
Je tiens à exprimer toute ma reconnaissance à Monsieur Antoine Beugnard, Responsable du Master Recherche en Informatique pour son accueil au sein de la formation.
J'exprime également ma gratitude à l'égard de l'ensemble du personnel du département informatique de Télécom Bretagne pour leur précieuse aide ainsi que leur sympathie qui ont favorisées mon intégration.
Sans oublier tous les membres de jury qui ont accepté de juger mon travail.
Et enfin mes derniers remerciements qui ne sont pas les moindres sont pour les personnes les plus chères de ma vie : ma maman, mon épouse et ma famille qui m'ont toujours soutenu.

Résumé

Actuellement, avec l'émergence des nouveaux systèmes répartis et complexes, le déploiement des composants logiciels est devenu difficile. Et ce principalement à cause de l'hétérogénéité et de la répartition des nœuds qui participent au déploiement. D'où la nécessité d'intégrer un processus de décision dans le déploiement, pour l'adapter au contexte et aux changements qui peuvent survenir au sein du système. Dans notre rapport, nous allons présenter les aspects de déploiement, de décision et des composants logiciels, comme introduction à mon travail. Ce travail a été réalisé durant la période de mon stage au sein de l'équipe CAMA au département informatique de Télécom Bretagne.

Mots clé :

déploiement réparti, composants logiciels, décision.

Table des matières

Table des figures

Problématique

Le déploiement des composants logiciels devient de plus en plus complexe du fait de la diversité des terminaux d'accès et des infrastructures de communication. La plupart des outils de déploiement existants utilisent des techniques qui ne sont pas à la mesure de la complexité des problèmes rencontrés, tels que :

- L'hétérogénéité des machines destinations
- Les conflits de déploiement entre les composants
- Les exigences requises des composants
- Le choix de placement des composants sur les nœuds de déploiement
- La nature et la topologie des réseaux sur lequels s'effectuera le déploiement

Surtout avec l'explosion du réseau internet et l'apparition de multiples terminaux mobiles (Smartphone, PDAs, Tablette PC, etc.) qui nécessitent des opérations de déploiements spécifiques, légères, répartis et dynamiques.

Le placement de composants logiciels sur un ensemble de nœuds répartis est devenu une étape centrale dans le déploiement complet d'une application sur ces machines destinations. En effet, le choix du placement influence énormément les performances de l'application, ainsi que le rendement et la consommation des différentes ressources utilisées par l'application.(influence aussi la possibilité d'installation).
Le déploiement doit prendre en compte les besoins, les conditions et les contraintes fonctionnelles de l'application à déployer, ainsi, que les propriétés non fonctionnelles et la disponibilité des ressources qui sont souvent partagées avec des accès réglementés. Il faut aussi respecter les exigences du réseau sur lequel on déploie, par exemple les conflits entre les composants ne doivent pas influencer les autres composants déjà installés sur les machines. Le déploiement doit aussi assurer un meilleur rendement et des performances optimales du système global qui sera déployé.

C'est là où la notion de décision et les mécanismes associés interviennent et s'avèrent utiles. Ces mécanismes de décisions peuvent contribuer largement afin de permettre un déploiement distribué optimal, selon des critères qui restent à définir par l'utilisateur.

Introduction

Après la terminaison du cycle de vie du développement d'un logiciel, un autre cycle commence et prend le relais : c'est le cycle du déploiement. Le déploiement va accompagner le logiciel développé dans toute sa phase d'intégration dans l'entreprise et jusqu'à sa mise hors service. Le déploiement couvre un nombre important d'opérations dont les principales sont :

- L'*installation* qui rend utilisable l'application est la première étape du déploiement.
- La *mise à jour* qui permet de modifier une application installée pour passer à une version plus récente que celle précédemment installée.
- La *reconfiguration* qui consiste à modifier les paramétres des composants où leur topologie de connexion.
- La *désinstallation* qui permet de retirer le logiciel et ses fichiers associés, est la dernière étape dans les cycle de vie du déploiement d'une application.

À travers la problématique, on s'aperçoit que le déploiement d'une application à base de composants logiciels n'est pas une opération simple. Pour surmonter ces différents problèmes, nous allons, à travers ce rapport de stage, introduire les différents concepts de déploiement dans les environnements et les systèmes répartis et hétérogènes et les mécanismes de prise de décision. Le contexte est vaste et plusieurs solutions plus au moins adaptées au cas par cas sont disponibles dans la littérature. Nous allons introduire les solutions existantes dans la suite de ce rapport. Nous allons commencer par introduire le concept de composant, le déploiement, ensuite les différentes formes de décisions existantes et nous présenterons notre contribution.

Nous terminerons ce rapport en parlant des pistes à suivre et ce qui reste à faire.

Mon stage s'est déroulé sous la supervision de Fabien Dagnat au sein de l'équipe CAMA du département informatique de Télécom Bretagne.

1 Les composants logiciels

Dans cette section, les notions et les concepts liés aux composants logiciels sont présentés.

1.1 La notion de composant

Le composants logiciels s'imposent de plus en plus dans l'industrie du logiciel. Ils proposent une approche du développement basée sur l'assemblage d'entités logicielles réutilisables. Ils sont le successeur logique de l'objet du fait de la solidité de leur base théorique [1, 2].
La définition [1] la plus reconnue est celle de *Clemens Szyperski* [1]

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

Dans cette définition le composant est considéré comme une boite noire, communiquant avec l'environnement extérieur par le biais de ses interfaces. Elle implique aussi l'existence des notions d'interface, de composition et de dépendance.

- ***Interface*** :
  Elle définit le comportement du composant visible de l'extérieur, à travers les services qu'elle fournit et les services requis. Elle définit aussi les contrats qui doivent être respectés par les composants.
  Elle permet aussi au composant de communiquer avec son environnement extérieur.
- ***Composition*** :
  Un composant peut être lui-même composé à partir d'autres composants primitifs où composés à leur tour.
- ***Dépendances*** :
  Chaque composant a un ensemble de contraintes et des conditions qui doivent être respectées pour assurer son fonctionnement correct. Les dépendances expriment aussi les exigences des composants en matière de ressources système, par exemple : l'espace disque requis pour l'installation, la taille de la mémoire RAM, la bande passante ou le type et la vitesse du processeur.

La figure 1 page 8 illustre qu'un composant est constitué d'autres composants reliés entre eux par des interfaces (les petits T).

1. La définition originale en anglais est conservée pour ne pas perdre une partie de son sens.

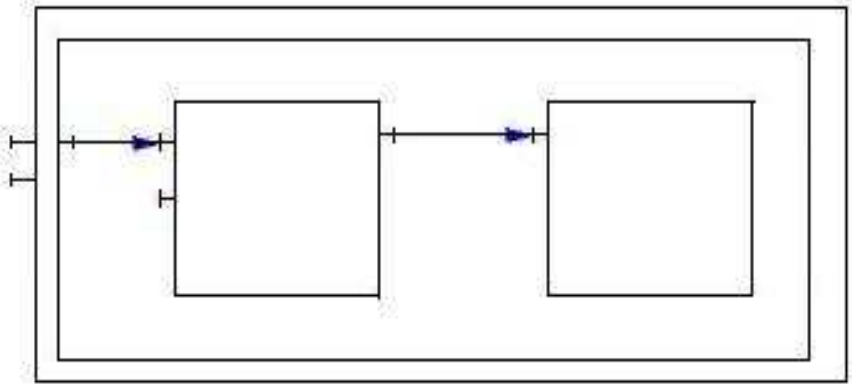

FIGURE 1 – Le concept de composant

1.2 Les modèles de composants

Différents modèles de composants existent dans l'industrie. On peut citer, les plus utilisés :

1.2.1 CCM

Corba Component Model est un modèle de composants qui permet de faire communiquer des composants hétérogènes et développés séparément dans des langages de programmations différents [3]. Le modèle de *packaging* et de déploiement spécifié par Corba 3.0 [4] définit un descripteur de composant via un fichier XML et associé au composant lui-même. Un tel descripteur contient la description du composant, son type en termes de ports d'entrées et de sorties (les interfaces). Ces descripteurs sont générés lors de la compilation de la description des composants, mais peuvent être modifiés pour paramétrer la gestion des aspects non-fonctionnels. Les composants sont ensuites empaquetés avec leurs descripteurs dans une archive appelée *paquetage de composant*.

1.2.2 .NET

.Net est l'architecture de composants fournie par Microsoft pour la plate-forme de développement Windows [5]. L'objectif est de fournir un langage indépendant de la plate-forme, aussi bien pour le développement que pour l'exécution. Le framework .Net repose sur le CLI(*Common Language Infrastructure*) [6].
L'inconvénient est que Microsoft fournit le framework .Net que pour ses systèmes d'exploitations Windows et Windows CE et le code source du framework n'est pas disponible.

1.2.3 Fractal

Fractal est un modèle de composants développé par l'INRIA et France Télécom [7]. Il permet la reconfiguration dynamique des applications. La figure 2 page 9

illustre la structure d'un composant fractal.
Fractal permet :

- la reconfiguration dynamique des applications à chaud (en cours d'exécution),
- le partage d'un composant par plusieurs autres composants composites,
- de gérer le cycle de vie d'un composant depuis sa création jusqu'à son utilisation finale par des interfaces spécifiques.

De plus, il existe plusieurs implémentations de Fractal pour les langages de programmations. On peut citer par exemple :

- l'implémentation Julia ou AOKell pour Java
- Think pour les programmeurs en C/C++
- Fractnet pour les développeurs .Net
- FracTalk pour SmallTalk
- Julio pour les développeurs Python

Le modèle de composants Fractal est accompagné d'un langage de description d'architecture permettant de spécifier une architecture de composants hiérarchiques. Ce langage est mis en œuvre par *FractalADL*. La description d'une architecture à base de composants Fractal est souvent spécifiée via le langage XML même si *FractalADL* n'impose pas une syntaxe particulière. Pour les architectures distribuées, *FractalRMI* propose un ensemble de composants permettant de faire la liaison entre des composants distants, ce qui n'est pas possible avec *FractalADL*.

1.2.4 EJB

EJB [8] est une architecture de composants distribués développée par Sun pour la communauté J2EE. Les EJB sont très utilisés dans le développement des applications et des services web.

Les composants EJB sont archivés à l'aide d'un descripteur de déploiement. Ce descripteur se présente sous la forme d'un fichier XML qui indique aux serveurs d'applications comment déployer les *beans* contenus dans l'archive en définissant leurs caractéristiques (e.g. les noms des composants, les types de transaction, les méthodes d'accès, etc.). La figure 3 page 10 illustre un exemple de descripteur EJB.

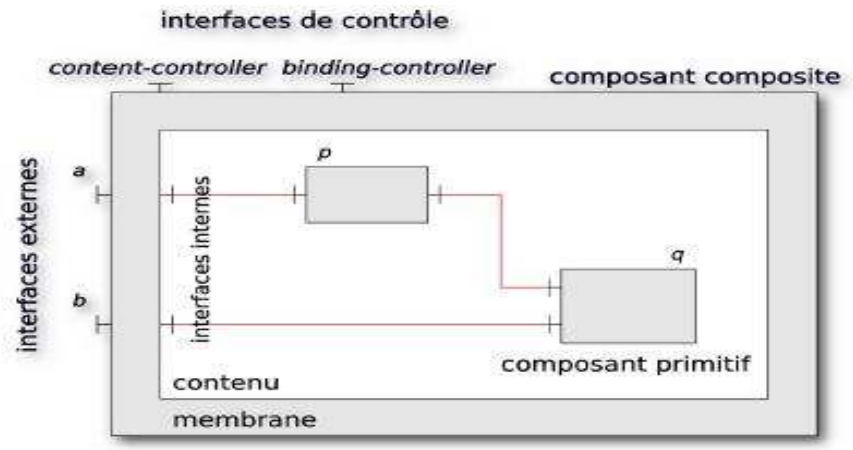

FIGURE 2 – Structure d'un composant Fractal

```
<? xml version='1.0' encoding='UTF-8'?>
<ejb-jar
version="2.1"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"
>
<display-name>SimpleSessionJar</display-name>
<enterprise-beans>
<session>
<ejb-name>SimpleSessionEjb</ejb-name>
<home>beans.SimpleSessionHome</home>
<remote>beans.SimpleSession</remote>
<ejb-class>beans.SimpleSessionBean</ejb-class>
<session-type>Stateless</session-type>
<transaction-type>Bean</transaction-type>
<security-identity>
<use-caller-identity>
</use-caller-identity>
</security-identity>
</session>
</enterprise-beans>
</ejb-jar>
```

FIGURE 3 – Exemple d'un descripteur de déploiement d'un composant EJB

La définition des serveurs cibles pour les composants beans constituant l'application se fait manuellement, en indiquant dans le descripteur de déploiement le nom des serveurs d'applications hôtes.

Dans ce chapitre, nous avons décrit quelques modèles de composants et leurs principaux concepts. Chaque modèle à ses avantages et inconvénients. Nous aborderons dans le chapitre suivant le concept du déploiement.

2 Le déploiement

Dans cette section, nous présentons les différents concepts liés au déploiement.

2.1 Le concept du déploiement

Le mot *déploiement* [2] correspond à l'expression de mise en œuvre. Ainsi, lorsqu'il porte sur une entité logicielle que l'on souhaite utiliser, il couvre toutes les activités nécessaires : l'installation, la reconfiguration, la mise à jour et la désinstallation [9].
Le déploiement logiciel est une activité complexe, qui couvre toutes les étapes depuis la validation du logiciel par le producteur jusqu'à son installation, puis sa désinstallation sur les sites utilisateurs. Toutes ces étapes représentent le cycle de vie du déploiement du logiciel.
Cette étape est cruciale pour que l'entité logicielle soit utilisable, elle doit se dérouler dans de bonnes conditions pour ne pas perturber le site du client.

2.2 Les acteurs du déploiement

Le déploiement d'une entité ne peut être effectué sans l'intervention de trois acteurs qui collaborent pour cette tâche :

- Le ***producteur*** qui produit l'entité à déployer et qui la met à la disposition de ses deux autres collaborateurs.
- L'***entreprise*** qui est responsable de l'élaboration de la politique de déploiement sur chaque site utilisateur.
- L'***utilisateur*** qui représente le site sur lequel s'effectue le déploiement.

Chaque acteur a ses propres exigences et ses contraintes. Le producteur par exemple, cherche à réduire les coûts de production de l'entité à déployer. Souvent les acteurs ont des objectifs contradictoires ; un déploiement fiable doit satisfaire l'ensemble des exigences et les contraintes des acteurs et trouver le bon compromis pour satisfaire l'ensemble.

2.3 Le cycle de vie du déploiement

Le déploiement d'un logiciel ne consiste pas seulement à l'installer sur un site, il accompagne le logiciel durant toute sa durée de vie chez le client jusqu'à la désinstallation. Il fait référence aussi à toutes les activités durant le cycle de vie d'une application allant de l'empaquetage, l'installation, l'activation, la désactivation, la mise à jour jusqu'à la désinstallation.

1. L'*empaquetage*[2] : cette étape précéde l'installation de l'entitée logicielle chez le client [2]. Le but de cette étape est de mettre à la disposition des clients du producteur l'entitée souhaitée sous forme d'archive selon les contraintes du client. (Par exemple, l'architecture utilisée par le client, le système d'exploitation).
2. L'*installation* : cette phase du cycle de vie consiste à installer sur une machine une entité logicielle. Avant l'installation, il faut vérifier que les exigences

2. Ce terme est utilisé pour traduire le terme anglais *packaging*

et les dépendances de l'application sont satisfaites dans l'environnement de déploiement.

3. La *reconfiguration* : il s'agit de paramétrer l'application avec d'autres paramétres différents, afin d'influencer le comportement de l'application. Cette opération est offerte par certains modèles de composants comme par exemple Fractal. Elle peut aussi modifier l'architecture de l'application.
4. La *mise à jour* : elle consiste à installer une autre version de l'entité logicielle déjà installée, pour corriger des erreurs par exemple ou ajouter des fonctionnalités. Il existe deux types de mise à jour :
   - La *mise à jour statique* nécessite la désactivation de l'entitée logicielle sur le site client pour effectuer les modifications nécessaires.
   - La *mise à jour dynamique* certaines applications critiques ne peuvent pas être interrompues (par exemple : en temps réels, embarquées, etc.)
5. La *désinstallation* : c'est la dernière étape du cycle de vie d'une entité logicielle, elle n'est pas obligatoire. Elle consiste à supprimer le logiciel du site de déploiement sans perturber les autres entités.

D'autres étapes intermédiaires peuvent exister dans le cycle de vie du déploiement, telles que : l'activation, l'adaptation, et la désactivation [10].

2.4 Le déploiement distribué

Avec l'explosion du réseau Internet et l'apparition des terminaux mobiles comme les smartphones et les PDAs, le déploiement des composants logiciels a pris une autre dimension.
L'entité logicielle peut contenir plusieurs « bouts »qui doivent être déployés sur plusieurs nœuds éparpillés sur le réseau et sur des stations (hôtes) hétérogènes. Une application distribuée met en jeu des composants répartis sur les machines du réseau. Il existe différents modèles d'architecture distribuée dont nous allons présenter un échantillon. Des architecture logicielles (3-tiers et P2P) et des architectures matérielles (Grille de calcul, Cloud computing)

2.4.1 Les architectures 3-tiers

Connus plus fréquemment sous le terme de client-serveur, ce sont des architectures qui contiennent généralement trois tiers : le client, l'application et le tiers base de données (ressource) [11]. Elles sont très utilisées dans les applications Web. Cette architecture est définie dans la figure 4 page 13 où l'on voit bien les trois tiers.

- le ***client***, c'est la partie visible par les clients (présentation), elle sert d'interfaces entre l'utilisateurs et la couche métier (milieu). Des composants graphiques sont utilisés pour l'implémentation de cette couche.
- la couche ***milieu***, le plus souvent connue sous le nom d'*application métier*, correspond à la partie fonctionnelle de l'application. Elle sert de relai entre la couche client et la couche *ressource*.
- la ***base de données*** fournit l'accés aux données de l'application, elle est souvent hébérgée sur une machine différente de celle qui accueille la couche *métier*.

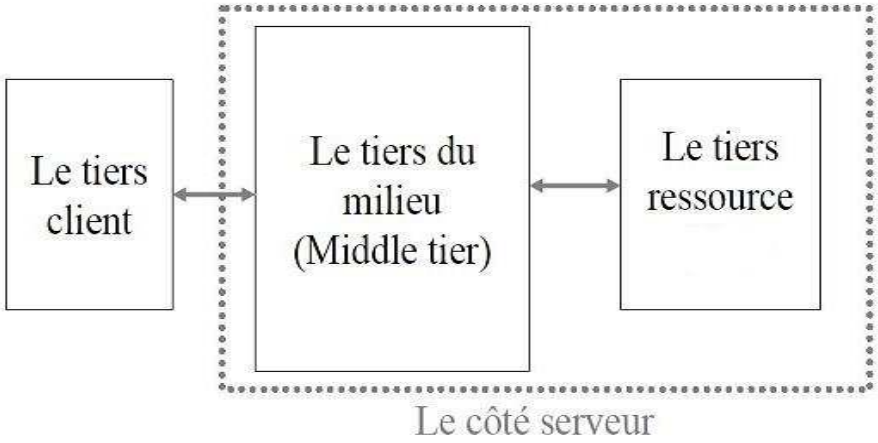

FIGURE 4 – Architecture 3-tiers

Chaque tiers est déployé séparément sur un nœud distinct. L'inconvénient est que tous les composants clients dépendent des composants serveurs. Ceci s'avére un énorme problème pour la mise à jour. Si on met à jour le composant *Serveur*, il peut être nécessaire de mettre à jour tout les autres composants *Clients* qui se connectent sur ce serveur.

2.4.2 Les architectures *peer-to-peer*

Ce sont des modèles d'architecture proche du modèle client-serveur mais où chaque client est aussi un serveur [12]. De plus en plus d'applications de ce genre sont déployées et utilisées sur internet, comme par exemple les applications de partages de fichiers et de téléchargement telles que : BitTorrent, eDonkey, etc.
Il faut prendre en compte dans le processus de déploiement le changement continuel de topologie de ces types de réseau.

2.4.3 Les grilles de calculs

Elles sont utilisées dans le calcul parrallèlle.Une grille de calcul exploite la puissance de calcul(processeurs, mémoires, etc) de milliers de machines connectées entre elles. Une grille de calcul est une infrastructure virtuelle constituée d'un ensemble de ressources de calcul potentiellement partagées, distribuées et hétérogènes [13].

- ***partagées*** peuvent être partagées par plusieurs composants et utilisateurs.
- ***distribuées*** sur le réseau, dans des lieux géographiques qui peuvent être éloignés.
- ***hétérogènes*** dans un réseau les ressources ne sont pas toutes homogènes. Par exemple le système d'exploitation.

Les grilles de calculs s'avérent intéressantes dans le cas de déploiement de composants logiciels parrallèlles.

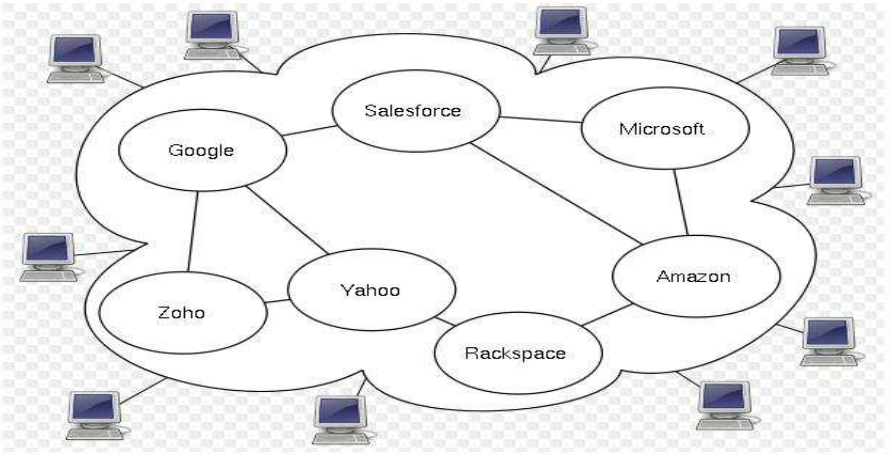

FIGURE 5 – Le concept du cloud computing

2.4.4 Le *cloud computing* – informatique dans le nuage

Dans ce genre d'architecture les traitements informatiques localisés traditionnellement sur le poste utilisateur sont déportés sur des serveurs distants. Il permet de ne déployer que des composants légers sur le poste du client (un navigateur web, par exemple). Google offre plusieurs applications en cloud computing [14]. La figure 5 illustre le concept de l'architecture du cloud.

2.5 Les difficultés du déploiement répartis

Les différentes architectures vues dans la section précédente rendent le déploiement distribué plus difficile et plus complexe. Lorsqu'il s'agit d'un déploiement réparti sur plusieurs machines [13], beaucoup de difficultés qui n'existent pas dans le cas du déploiement monoposte surgissent [10]. On peut les résumer comme suit :

- Le choix du placement des composants sur les différents nœuds [15].
- La connaissance de la topologie du réseau et les nœuds qui le composent.
- L'hétérogénéité des machines et des réseaux sur lesquelles on réalise le déploiement [16].
- La communication entre les différents composants de l'application à déployer [17].

3 La décision pour le déploiement

Dans cette section, nous décrivons les notions relatives à la décision dans le déploiement de logiciels.

3.1 Notions et concepts

En informatique, l'aide à la décision est un domaine qui vise à concevoir des outils informatiques pour aider un décideur à analyser un problème ou une situation, et à lui fournir des solutions [18].

La décision prise peut être multicritères, c'est-à-dire qu'elle doit satisfaire plusieurs contraintes et critères, par exemple : de performances, d'organisation et de gestion de ressources, d'où la notion de l'*agrégation multi-critère* [19].
Les critères à satisfaire dans notre cas, sont :

- les performances du système,
- la topologie du réseau,
- la bande passante et la gestion des ressources,
- il ne faut pas perturber le système aussi.

Ainsi que des critères propres à chaque nœuds, comme :

- les contraintes conflictuelles (par exemple : interdiction de déployer le composant `B` avec le composant `A` sur le même nœud).
- exigences de préinstallation de certains services,
- exigences de ressources,
- la mobilité et la disponibilité des nœuds sur lesquels on déploie.

3.2 Les approches de décision dans le déploiement

Les modèles de composants et leurs outils de déploiement qu'on a vu jusqu'ici, sont conçus pour des environnements peu tolérants aux pannes et aux changements de ressources. Ainsi, les reconfigurations et les adaptations qui s'imposent nécessitent l'intervention d'un administrateur, voire même de l'utilisateur et généralement sont des interventions ad-hoc au cas par cas. Différents algorithmes et approches qui traitent le problème du déploiement distribué existent dans la littérature. Ces approches ont été conçues dans le but d'accroitre l'autonomie et l'optimisation du déploiement. Elles sont présentées dans le reste de la section.

3.2.1 L'optimisation combinatoire

Dans cette approche, les auteurs considèrent le déploiement comme étant un problème de placement des composants de l'application à déployer sur l'ensemble des nœuds distribués sur le réseau [20].

Il se propose de modéliser le problème sous forme combinatoire (un modèle de recherche opérationnelle) et trouver le meilleur placement possible des composants sur les nœuds en fonction de :

- la topologie du réseau,
- les connexions entre les nœuds,
- les charges sur les nœuds,
- les coûts de déploiement sur les nœuds.

En effet, dans cette approche :

- chaque composant posséde des contraintes que la machine cible devra vérifier ou des préférences (optionnelles) sur des ressources qu'offrent les nœuds.
- une contrainte est caractérisée par un intervalle de valeurs et un poids qui permet d'accorder plus d'importance à telle ou telle contrainte.

La figure 6 page 16 illustre un exemple de quarante composants placés sur cinq nœuds du réseau. L'intérêt principal de cette approche est que l'on obtient un placement optimal des composants sur les nœuds selon les contraintes définies. Par

contre, la décision est centralisée dans un nœud et il n'y a pas de possibilité de reconfiguration de l'application.

3.2.2 Le placement des composants sur les grilles

Il existe des difficultés pour déployer des applications sur les grilles de calcul, dans sa thése S.Lacour [21] expose ses problémes dont :

- la nécessité d'une intervention manuelle de l'utilisateur dans le sélection des ressources
- le choix des implémentations de l'application pour chaque ressource (nœud)
- le transfert de fichier à distance et la configuration des applications imposent une certaine connaissance de la part de l'utilisateur.

La proposition faite dans [21] est d'automatiser le déploiement d'applications. Afin d'automatiser cette opération, l'outil de déploiement à besoin de la description de l'architecture à déployer, ainsi que celles des ressources disponibles sur la grille. L'utilisateur peut également exprimer ces paramétres tel que : le degré de parrallélisme ou le réseau auquel les machines appartiennent. À partir de là un planificateur de déploiement sélectionne les ressources, les protocoles de lancement des tâches, le placement des différents composants de l'application et la séléction de l'implémentation adéquate pour chaque composant. Le planificateur produit en sortie un plan de déploiement qui respecte toutes les exigences et les contraintes qui ont été faites lors de la phase de planification.

3.2.3 Les algorithmes basés sur le principe des enchères

Cette solution est basée sur le principe des enchères. Elles permettent le redéploiement dynamique des composants afin de prendre en compte l'évolution des ressources du système [22]. Le but est de replacer les composants dans le système, en fonction de la fiabilité des liaisons inter-nœuds, la fréquence de communication entre les composants et la disponibilité des ressources du nœud (mémoire, CPU et la bande passante). Il cherche à redéployer les composants d'une manière homogène dans le système, en fonction des ressources disponibles. Chaque nœud qui se porte

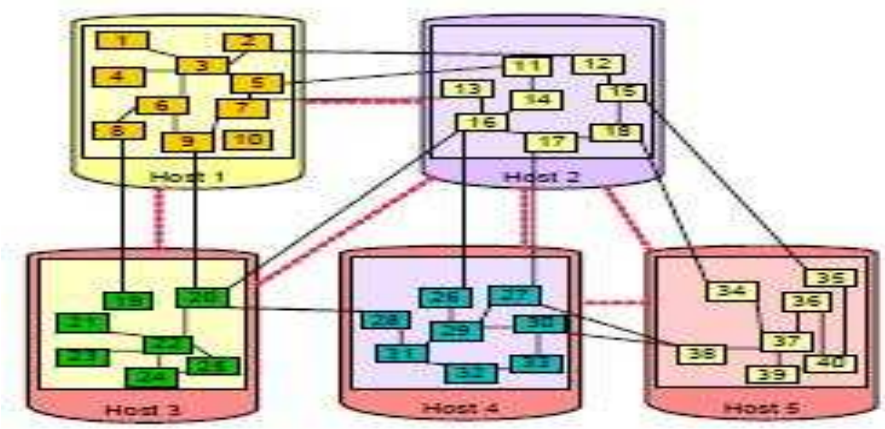

FIGURE 6 – Déploiement de 40 composants sur cinq nœuds

candidat au déploiement du composant envoie une enchère, sans connaître celles des autres.Chaque nœud a son propre poids de déploiement, en fonction de la disponibilités de ces ressources, sa fréquence de communication avec les autres nœuds et la fiabilité des liaisons. Le choix de la mise en dépend. Le composant sera déployé sur le nœud qui a émis la plus grande valeur. Des évenements extérieurs initient l'enchère, tels que :

- la disparition ou l'apparition d'un nœud ;
- la dégradation ou le rétablissement d'une liaison de connexion entre deux nœuds ;
- l'ajout ou la suppression de composants ;
- la réévaluation du placement d'un ou plusieurs composants.

L'intérêt de cette approche est que tous les nœuds sont identiques et se déclarent candidats au déploiement. Cette idée sera reprise dans notre travail. Par contre, la décision est centralisée sur un nœud qui décide du placement.

3.2.4 La programmation par contraintes

Les auteurs [10, 23] de cette solution proposent la résolution du probléme de placement des composants logiciels selon le paradigme de la programmation par contraintes. On génére un ensemble de besoins (les contraintes du problémes à résoudre), leurs résolution étant prise en charge par des solveurs logiques (Prolog par exemple). Il suffit de déclarer les contraintes pour modéliser le système.

Les avantages de cette solution sont :

- le placement est calculé automatiquement,
- on peut obtenir plusieurs solutions de placement qui peuvent être comparées,
- plusieurs algorithmes de résolutions existent.

Par contre, toutes les contraintes doivent être spécifiées au départ et la décision est faite par un seul nœud qui décide pour les autres. L'ensemble de ces contraintes définit l'objectif du déploiement. Toutes les contraintes sont ensuite utilisées par un résolveur de contraintes qui peut générer une ou plusieurs solutions. Chaque solution représente une configuration (interconnexions de composants et les adresses des machines pour chacun d'entre eux) qui vérifie les contraintes exprimées.

4 Notre contribution

Un langage pour la vérification formelle du déploiement monoposte a été défini dans [24], ainsi que des mécanismes et des règles de déploiement au sein de l'équipe CAMA. Nos deux principales contributions à ce travail sont :

1. Le déploiement d'application sur plusieurs nœuds.
2. Les exigences qui peuvent porter sur des services d'autres nœuds.

4.1 Le méta-modèle de déploiement

Le méta-modèle de la figure 7 est une représentation UML des différents concepts que nous allons utiliser dans notre travail.
Le système `(system)` au sein duquel on déploie peut contenir plusieurs applications distribuées `(application)` et plusieurs nœuds `(Node)`. Une application est composée d'un ou plusieurs composants `(Component)` qui peuvent être déployés sur un ou plusieurs nœuds. Les composants communiquent entre eux par le biais des liens *(Link)* qui relient les nœuds. Par exemple, une application de carnet d'adressse réparti contient plusieurs composants déployés sur différents nœuds dans le réseau. Le composant base de données, le composant serveur d'application et le composant client seront déployés sur des nœuds différents.

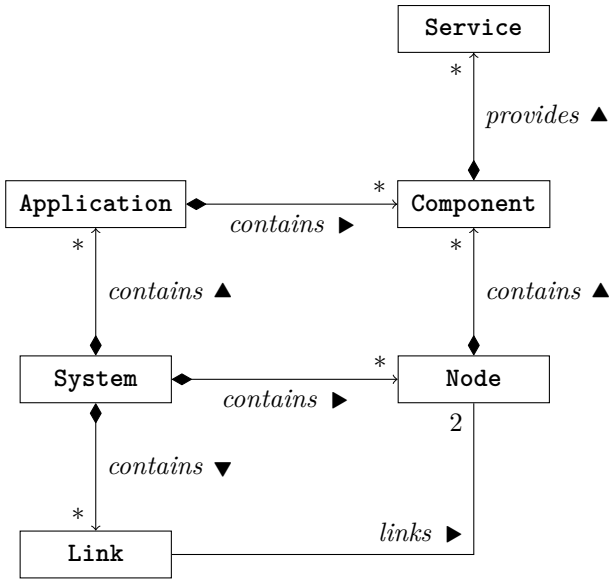

FIGURE 7 – Le méta-modéle du déploiement réparti

4.2 Description du contexte du déploiement réparti

Le contexte représente l'état (caractéristiques et contraintes) des nœuds et du système sur lesquel on déploie. Il contiendera par exemple :

– L'architecture du système et les liens entre les différents nœuds qui le composent.

– Les ressources fournies par le système, par exemple : la mémoire disponible, le type du système d'exploitation, l'espace disque, la bande passante.
– Les services qui peuvent être fournis où interdits.
– La liste des composants logiciels interdits, par exemple : les logiciels malveillants (les virus et les vers), les logiciels espions. Ou un logiciel qui rend un service en utilisant une ressource unique peut exiger d'être le seul à le faire en signalant un conflit. Par exemple, `Sendmail` interdit tout autre serveur de mail (`postfix` par exemple).

Dans le cas du déploiement réparti, on distingue deux types de contexte : un contexte local (Ctx_{local}) spécifique à chaque nœud et le contexte global (Ctx_{global}).

4.2.1 Contexte local

Chaque nœud de déploiement a son propre contexte qui contient :
– les ressources disponibles, par exemple : mémoire vive(RAM), la vitesse et le type du processeur, l'espace disque disponible, le type du système d'exploitation.

et $Ctx.\mathcal{C}$ représente le quadruplet $(c, \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c)$ qui décrit le composant c avec :
– $\mathcal{P}_s$ l'ensemble des services fournis par c,
– $\mathcal{F}_s$ l'ensemble des services interdits,
– $\mathcal{F}_c$ l'ensemble des composants intérdits.

A partir de ces données on peut définir :
– l'ensemble des composants disponibles (AC)
– l'ensemble des composants interdits (FC)
– l'ensemble des services disponibles (AS)
– l'ensemble des services interdits (FS)

qui sont calculés à partir du contexte :

$$\begin{cases} AC(Ctx) = \cup \{c \mid (c,-,-,-) \in Ctx.\mathcal{C}\} \\ AS(Ctx) = \cup \{\mathcal{P}_s \mid (-,\mathcal{P}_s,-,-) \in Ctx.\mathcal{C}\} \\ FS(Ctx) = \cup \{\mathcal{F}_s \mid (-,-,\mathcal{F}_s,-) \in Ctx.\mathcal{C}\} \\ FC(Ctx) = \cup \{\mathcal{F}_c \mid (-,-,-,\mathcal{F}_c) \in Ctx.\mathcal{C}\} \end{cases}$$

4.2.2 Contexte global

Le contexte global caractérise le système réparti de déploiement et ces caractéristiques, cependant ce contexte est très difficile à calculer ou à modéliser du fait de son changement rapide d'un instant t à $t+1$. Il se compose de l'ensemble des contextes locaux de chacun des nœuds du système. Pour remedier à ce problème et pour pouvoir utiliser le contexte global dans nos régles sans avoir à le calculer, on définit la fonction suivante qui retourne le contexte d'un nœud i :

$$\begin{aligned} Ctx_{global} : \ & \mathcal{N} \longrightarrow Ctx \\ & i \ \longrightarrow Ctx_{local} \text{ de } i \end{aligned}$$

4.3 Un langage formel pour le déploiement réparti

Afin de prendre en compte l'aspect réparti du déploiement, la grammaire de [24] est étendue. Les constructions ajoutées sont en rouge :

$$
\begin{array}{ll}
\mathcal{D} & ::= \mathcal{P} \Rightarrow s \mid \mathcal{D} \bullet \mathcal{D} \mid \mathcal{D} \# \mathcal{D} \mid ? \mathcal{D} \\
\mathcal{P} & ::= true \mid \mathcal{P} \wedge \mathcal{P} \mid \mathcal{Q} \\
\mathcal{Q} & ::= \mathcal{Q} \vee \mathcal{Q} \mid [v\ O\ val] \mid \mathcal{X}\ [O_d\ \mathcal{N}] \mid \mathcal{Y}[\in \mathcal{N}] \\
\mathcal{X} & ::= s \mid c.s \\
\mathcal{Y} & ::= \neg\ c \mid \neg\ s \\
O & ::= >|\geq|<|\leq|=|\neq \\
O_d & ::= \in|\notin
\end{array}
$$

où, $\mathcal{N}$ est un ensemble de nœuds du système.

Un composant peut avoir des dépendances ($\mathcal{D}$) qui sont :

- soit des ressources requises, par exemple : $(FDS \geq 50Go)$ où $OS = "WIN"$
- soit des services requis ou des services interdits.
- les dépendances peuvent être simples par exemple : $P \Rightarrow s$ qui signifie : si le prédicat P est valide, alors le service s sera fourni ou composé d'autres dépendances.

Une grammaire pour décrire les applications distribuées peut ainsi être proposée :

$$
\begin{array}{ll}
\mathcal{A} & ::= c : \mathcal{DLOC} \mid \mathcal{A}, \mathcal{A} \\
\mathcal{LOC} & ::= \in \mathcal{N} \mid \in^{n..m} \mathcal{N}
\end{array}
$$

Exemple :

L'exemple suivant, sert à expliquer la grammaire précédente :

Contexte : Un serveur et deux clients

- Serveur qui offre deux services :
  - http
  - ftp
- Deux Clients sur deux nœuds qui utilisent les services fournis par un composant serveur c_0

On dispose de trois nœuds pour le déploiement :

- le serveur c_0 sera installé sur le nœud n_0
- aucun composant de type client ne doit être installé avec le serveur
- les deux composants clients c_1 *et* c_2 fournissent les services *httpc* et *ftpc* et seront installés sur les nœuds n_1 et n_2. avec $\mathcal{N} = \{n_1, n_2\}$

Les dépendances de ces composants sont :

Coté serveur :

$(\neg\ c_1 \in n_0\ \wedge\ \neg\ c_2 \in n_0)\ \Rightarrow\ http$

$(\neg\ c_1 \in n_0\ \wedge\ \neg\ c_2 \in n_0)\ \Rightarrow\ ftp$

Coté client :

$(\neg c_0 \in \mathcal{N}) \Rightarrow httpc$

$(\neg c_0 \in \mathcal{N}) \Rightarrow ftpc$

4.4 Les règles d'installations distribuées

L'installabilité des applications à base de composants est vérifiée par les règles logiques décrites dans la suite (4.4.1). Ces règles utilisent des règles sur les composants qui sont décrites en 4.4.2. et les règles sur les prédicats en 4.4.3.
Chaque règle contient deux parties : une partie conclusion et la partie prémice :

$$\textbf{Nom de la regle:}\ \frac{Premices_1 \quad Premices_2 \quad \ldots Premices_n}{Conclusion}$$

Si la partie supérieure est vérifiée, alors la partie inférieure est valide. Si toutes les prémices sont vraies alors la conclusion sera valide. Nous avons défini trois types de règles :

- $\vdash_A$: pour le déploiement des applications. Ces régles prennent en paramètre le Contexte global et l'application à installer, elles ont la forme :
  $Ctx_{global} \vdash_A \mathcal{A} \Rightarrow \mathcal{N}\}$ avec $\mathcal{N}$, l'ensemble des nœuds sur lesquels on peut déployer.
- $\vdash_C$: pour la vérification des dépendances des composants, elles ont en paramètres d'entrés : contexte global, contexte local et la dépendance du composant à installer. Elles ont la forme suivante :
  $$Ctx_{local}, Ctx_{global} \vdash_C c : \mathcal{D}$$
- $\vdash_P$: pour la vérification des prédicats dans le contexte. Elles ont la forme :
  $$Ctx_{local}, Ctx_{global} \vdash_P Predicat$$
- $\vdash_G$ pour le calcul du graphe de dépendances. Elles ont la forme :
  $$Ctx_{local}, Ctx_{global}, c, s \vdash_G Predicat \Rightarrow Graphe\ local,\ Graphe\ distant$$
- $\vdash_I$ pour le calcul des effets d'installations. Elles ont la forme :
  $$Ctx_{local}, Ctx_{global} \vdash_I c : \mathcal{D} \Rightarrow Effets$$
- $\vdash_I^{\mathcal{A}}$ pour l'installation des applications. Elles ont la forme :
  $$Ctx_{global} \vdash_I^{\mathcal{A}} \mathcal{A} \Rightarrow Effets$$

4.4.1 Les règles sur les applications

Pour qu'un composant c soit installable dans le système réparti, il suffit qu'il existe au moins un nœud dans le système qui peut l'héberger.

$$\text{CD:}\ \frac{i \in N \quad Ctx_{global}(i), Ctx_{global} \vdash_C c : \mathcal{D}}{Ctx_{global} \vdash_A c : \mathcal{D} \Rightarrow \{i\}}$$

Si un ensemble de nœuds $\mathcal{N}$ est exigé, il faut que le composant c soit installé sur tous les nœuds de $\mathcal{N}$.

$$\text{DApp1:}\ \frac{\forall\ i \in \mathcal{N},\ Ctx_{global}(i), Ctx_{global} \vdash_C \ c : \mathcal{D}}{Ctx_{global} \vdash_A \quad c : \mathcal{D} \in \mathcal{N} \Rightarrow \mathcal{N}}$$

Lorsqu'en plus, il y a une contrainte sur le nombre d'instance, le composant c doit pouvoir installer sur un sous-ensemble de $\mathcal{N}$ ayant entre n et m eléments.

$$\text{DApp2:}\ \frac{n \leq \ card\{i \in \mathcal{N} \mid Ctx_{global}(i), Ctx_{global} \vdash_C \ c : \mathcal{D}\} \ \leq \ m}{Ctx_{global} \vdash_A \ c : \mathcal{D} \ \in^{n..m} \ \mathcal{N} \Rightarrow \{n_n, n_{n+1}, .., n_m\}}$$

Pour que deux applications $\mathcal{A}_1$ et $\mathcal{A}_2$ soient déployées, il faut que les exigences des deux applications soient satisfaites.

$$\text{DApp3:}\ \frac{Ctx_{global} \vdash_A \mathcal{A}_1 \Rightarrow \mathcal{N} \quad Ctx_{global} \vdash_A \mathcal{A}_2 \Rightarrow \mathcal{N}}{Ctx_{global} \vdash_A \mathcal{A}_1, \mathcal{A}_2 \Rightarrow \mathcal{N}_1 \cup \mathcal{N}_2}$$

4.4.2 Les règles sur les composants

Une dépendance optionnelle $(c :? \mathcal{D})$ est toujours vérifiée dans le contexte.

$$\text{COpt:}\ Ctx_{local}, Ctx_{global} \vdash_C c :? \mathcal{D}$$

Une dépendance simple de type $(P \Rightarrow s)$ est vérifiée dans le contexte, si son prédicat P est satisfait dans un contexte local et le service s n'est pas interdit dans ce même contexte.

$$\text{CTriv:}\ \frac{Ctx_{local}, Ctx_{global} \vdash_P P \qquad s \notin FS(Ctx_{local})}{Ctx_{local}, Ctx_{global} \vdash_C \ c : (P \Rightarrow s)}$$

La disjonction de deux dépendances est satisfaite, si l'une des deux dépendances est satisfaite dans le contexte.

$$\text{COr}\mathbf{L}\text{:}\ \frac{Ctx_{local}, Ctx_{global} \vdash c : \mathcal{D}_1}{Ctx_{local}, Ctx_{global} \vdash_C c : \mathcal{D}_1 \# \mathcal{D}_2}$$

$$\text{COr}\mathbf{R}\text{:}\ \frac{Ctx_{local}, Ctx_{global} \vdash c : \mathcal{D}_2}{Ctx_{local}, Ctx_{global} \vdash_C c : \mathcal{D}_1 \# \mathcal{D}_2}$$

La conjonction de deux dépendances est satisfaite, si les deux dépendances sont satisfaites dans le contexte.

$$\text{CAnd:}\ \frac{Ctx_{local}, ctx_{global} \vdash c : \mathcal{D}_1 \quad Ctx_{local}, Ctx_{global} \vdash c : \mathcal{D}_2}{Ctx_{local}, Ctx_{global} \vdash_C c : \mathcal{D}_1 \bullet \mathcal{D}_2}$$

4.4.3 Les règles sur les prédicats

Le prédicat *true* est toujours valide dans le contexte.

$$\text{PTrue:}\ Ctx_{local}, Ctx_{global} \vdash_P true$$

La conjonction de deux prédicats est vérifiée dans le contexte, si les deux prédicats sont vérifiés dans le contexte.

$$\text{PAnd:}\ \frac{Ctx_{local}, Ctx_{global} \vdash_P P_1 \quad Ctx_{local}, Ctx_{global} \vdash_P P_2}{Ctx_{local}, Ctx_{global} \vdash_P P_1 \wedge P_2}$$

La disjonction de deux prédicats est satisfaite, si l'un des deux prédicats est satisfait dans le contexte.

$$\text{POrL:}\ \frac{Ctx_{local}, Ctx_{global} \vdash_P Q_1}{Ctx_{local}, Ctx_{global} \vdash_P Q_1 \vee Q_2}$$

$$\textsc{POrR:}\ \frac{Ctx_{local}, Ctx_{global} \vdash_P Q_2}{Ctx_{local}, Ctx_{global} \vdash_P Q_1 \vee Q_2}$$

Si la variable v de l'environnement du contexte Ctx_{local} vérifie la contrainte par rapport à la valeur V, alors le prédicat $[v\ O\ V]$ est vérifié.

$$\textsc{PVar:}\ \frac{Ctx_{local}.\xi(v)\ O\ V}{Ctx_{local}, Ctx_{global} \vdash_P [v\ O\ V]}$$

Un service s peut être interdit, s'il n'appartient pas à l'ensemble des services disponibles du contexte.

$$\textsc{PNotS:}\ \frac{s \notin AS(Ctx_{local})}{Ctx_{local}, Ctx_{global} \vdash_P \ \neg s}$$

Un composant c peut être interdit, s'il n'appartient pas à l'ensemble des composants disponibles dans le contexte.

$$\textsc{PNotC:}\ \frac{c \notin AC(Ctx_{local})}{Ctx_{local}, Ctx_{global} \vdash_P \ \neg c}$$

L'exigence d'un service s est satisfaite s'il appartient à l'ensemble des services disponibles dans le contexte.

$$\textsc{PServ:}\ \frac{s \in AS(Ctx_{local})}{Ctx_{local}, Ctx_{global} \vdash_P \ s}$$

L'exigence d'un service s fourni par un composant c est satisfaite, si c est disponible et le service s appartient à l'ensemble des services fournis par c

$$\textsc{PComp:}\ \frac{(c, \mathcal{P}_s, -, -) \in Ctx_{local}.\mathcal{C} \quad s \in \mathcal{P}_s}{Ctx_{local}, Ctx_{global} \vdash_P c.s}$$

Si le composant c n'appartient à aucun ensemble de composants disponibles des différents nœuds de $\mathcal{N}$, alors c peut être interdit dans $\mathcal{N}$.

$$\textsc{PNotC:}\ \frac{\forall\ i \in \mathcal{N},\quad c \notin AC(Ctx_{global}(i))}{Ctx_{local}, Ctx_{global} \vdash_P \quad \neg c\ \in \mathcal{N}}$$

Si le service s n'est disponible dans aucun des nœuds de $\mathcal{N}$, alors s peut être interdit dans $\mathcal{N}$.

$$\textsc{PNotS:}\ \frac{\forall\ i \in \mathcal{N},\quad s \notin AS(Ctx_{global}(i))}{Ctx_{local}, Ctx_{global} \vdash_P \quad \neg s\ \in\ \mathcal{N}}$$

Si le service s est disponible dans un des nœuds de $\mathcal{N}$, alors son exigence peut être satisfaite dans $\mathcal{N}$.

$$\textsc{PServ:}\ \frac{i \in \mathcal{N}\quad s \in AS(Ctx_{global}(i))}{Ctx_{local}, Ctx_{global} \vdash_P \quad s\ \in\ \mathcal{N}}$$

Si le service s n'est disponible dans aucun des nœuds de $\mathcal{N}$, alors s n'appartient à aucun nœud de $\mathcal{N}$.

$$\textsc{PNotServ:}\ \frac{\forall\ i \in \mathcal{N},\quad s \notin AS(Ctx_{global}(i))}{Ctx_{local}, Ctx_{global} \vdash_P\ \ s\ \notin\ \mathcal{N}}$$

S'il existe un nœud i où le composant c est disponible et fournit le service s, alors l'exigence $c.s \in \mathcal{N}$ est satisfaite.

$$\textsc{PComp:}\ \frac{i \in \mathcal{N}\ \ (c, \mathcal{P}_s, -, -) \in Ctx_{global}(i)\ \ s \in \mathcal{P}_s}{Ctx_{local}, Ctx_{global} \vdash_P\quad c.s\ \ \in \mathcal{N}}$$

Si le composant c ne fournit pas le service s, alors l'exigence $c.s \in \mathcal{N}$ ne sera pas satisfaite.

$$\textsc{PNotC.S:}\ \frac{\forall\ i \in \mathcal{N}, ((c, \mathcal{P}_s, -, -) \in Ctx_{global}(i)\quad\ s \notin \mathcal{P}_s)}{Ctx_{local}, Ctx_{global} \vdash_P\quad c.s\ \ \notin \mathcal{N}}$$

4.5 Exemple d'installabilité réparti

Exemple 2 : Carnet d'adresse réparti

Un carnet d'adresse réparti permet à un ensemble d'utilisateurs de maintenir un carnet d'adresses de leurs contacts dans une base de données accessible à travers un serveur d'application. Le système comportera :

- trois composants :
  - c_{bd} : Le composant base de données qui fournit le service base de données s_{bd} sera installé sur le nœud n_0
  - c_{sa} : Le serveur d'application sera installé sur le nœud n_1 qui fournit un service d'accès s_a
  - c_{ic} : Le composant interface client qui sera installé dans le sous-ensemble des nœuds des utilisateurs qu'on appelera $\mathcal{N}' = \{n_2, n_3\}$
- quatre nœuds $\mathcal{N} = \{n_0, n_1, n_2, n_3\}$

On exige aussi qu'aucun logiciel malveillant par exemple : virus ou ver (C_v) ne soit installé sur aucun des nœuds du système. La figure 8 illustre l'architecture de l'application à installer et la localisation des différents composants sur les nœuds.

Chaque nœud a son propre contexte :

$\mathcal{E}_{n0} = \{FDS = 500, OS = LINUX, RAM = 4\}$

$\mathcal{E}_{n1} = \{FDS = 300, OS = WINDOWS\ XP, RAM = 2\}$

$\mathcal{E}_{n2} = \{FDS = 32, OS = Android, RAM = 0.064\}$

$\mathcal{E}_{n3} = \{FDS = 8, OS = Symbian, RAM = 0.5\}$

$AC_0 = AC_1 = AC_2 = AC_3 = \phi$

$AS_0 = AS_1 = AS_2 = AS_3 = \phi$

$FS_0 = FS_1 = FS_2 = FS_3 = \phi$

$FC_0 = FC_1 = FC_2 = FC_3 = \{c_v\}$

Spécification des dépendances de composants :

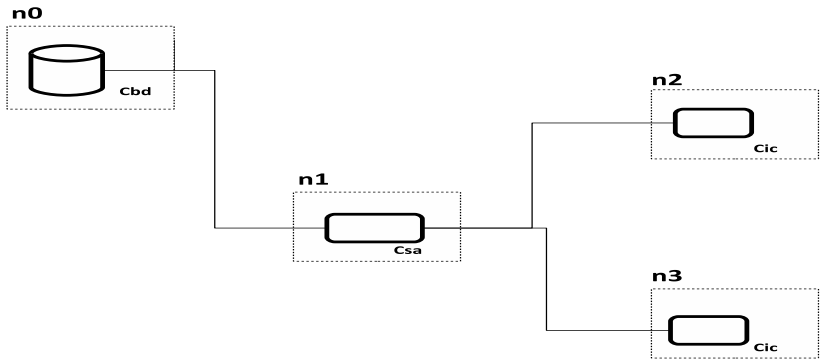

FIGURE 8 – L'architecture du carnet d'adresse réparti

Pour le composant c_{bd} :

$(\neg c_v \in \mathcal{N}) \wedge (FDS \geq 20) \wedge (RAM \geq 3) \Rightarrow s_{bd}$

Pour le serveur d'application c_{sa}

$(c_{bd}.s_{bd} \in n_0) \wedge (\neg c_v \in \mathcal{N}) \wedge (FDS \geq 6) \Rightarrow s_a$

Les composants clients fournissent un service intéraction s_{ic} :

$(c_{sa}.s_{sa} \in n_1) \wedge (\neg c_v \in \mathcal{N}) \Rightarrow s_{ic}$

Preuve d'installabilité :

Pour prouver l'installabilité de l'application, il faut vérifier que les trois dépendances sont valides dans le contexte :

$$\textbf{Cbd:}\ \frac{\dfrac{c_v \notin \mathcal{N}}{Ctx_l, Ctx_g \vdash_C \neg c_v \in \mathcal{N}} : \quad \dfrac{\dfrac{500 \geq 20}{Ctx_l, Ctx_g \vdash_P FDS \geq 20} : \quad \dfrac{4 \geq 3 \qquad s_{bd} \notin FS(Ctx_l)}{Ctx, Ctx_g \vdash_P \ (RAM \geq 3) \Rightarrow s_{bd}}}{Ctx_l, Ctx_g \vdash_C (FDS \geq 20) \wedge (RAM \geq 3) \Rightarrow s_{bd}}}{Ctx_l, Ctx_g \vdash_C c_{bd} : (\neg c_v \in \mathcal{N}) \wedge (FDS \geq 20) \wedge (RAM \geq 3) \Rightarrow s_{bd}}$$

$$\textbf{Csa:}\ \frac{\dfrac{c_{bd} \in n_0 \qquad s_{bd} \in \mathcal{P}_{s_{Cbd}}}{Ctx_l, Ctx_g \vdash_C c_{bd}.s_{bd} \in n_0} : \quad \dfrac{\dfrac{c_v \notin AC(Ctx_l)}{Ctx_l, Ctx_g \vdash_P \neg c_v \in \mathcal{N}} : \quad \dfrac{300 \geq 6 \qquad s_a \notin FS(Ctx_l)}{Ctx_l, Ctx_g \vdash_P (FDS \geq 6) \Rightarrow s_a}}{Ctx_l, Ctx_g \vdash_C (\neg c_v \in \mathcal{N}) \wedge (FDS \geq 6) \Rightarrow s_a}}{Ctx_l, Ctx_g \vdash_C c_{sa} : (c_{bd}.s_{bd} \in n_0) \wedge (\neg c_v \in \mathcal{N}) \wedge (FDS \geq 6) \Rightarrow s_a}$$

$$\textbf{Cic:}\ \frac{\dfrac{c_v \notin AC(Ctx_l)}{Ctx_l, Ctx_g \vdash_C \neg c_v \in \mathcal{N}} : \quad \dfrac{\dfrac{c_{sa}.s_{sa} \in n_1 \qquad s_{sa} \in \mathcal{P}_{s_{Csa}}}{Ctx_l, Ctx_g \vdash_P c_{sa}.s_{sa} \in n_1} : \qquad s_{ic} \notin FS(Ctx_l)}{Ctx_l, Ctx_g \vdash_C (c_{sa}.s_{sa} \in n_1) \Rightarrow s_{ic}}}{Ctx_l, Ctx_g \vdash_C c_{ic} : (\neg c_v \in \mathcal{N}) \wedge (c_{sa}.s_{sa} \in n_1) \Rightarrow s_{ic}}$$

On peut en déduire que les composant c_{bd}, c_{sa}, c_{ic} sont installables dans le système

4.6 Les règles avec le calcul des effets d'installations

Pour calculer les effets de l'installation, un graphe a été conçu dans [24], nous avons réadapté ce graphe pour les dépendances réparties et le déploiement distribué.

4.6.1 Définition du graphe de dépendance

Comme définit dans [24], un graphe de dépendance $\mathcal{G}$ mémorise les liens entre les services fournis par un composant et les services qu'il requiert. Nous avons réadapté les règles de calcul du graphe de dépendance définit dans [24], pour prendre en compte l'aspect réparti du déploiement. Il est utilisé pour mémoriser les dépendances entre les composants installés et donc permet de réaliser des desinstallations

correctes (de composants non indispensables). L'unique changement dans la forme d'un graphe de dépendance est que chaque nœud est maintenant de la forme $n.c.s$. En effet, un service fourni par un composant d'un nœud peut dépendre d'un service fourni par un composant d'un autre nœud.
Ainsi, par exemple, si un composant c_1 existe sur un nœud n_1 et fournit un service s_1, il peut satisfaire une exigence du services s_2 du composant c_2 du nœud n_2. Cette dépendance $n_1.c_1.s_1 \rightarrow n_2.c_2.s_2$ sera mémorisée dans le graphe des deux nœuds.
Notons que, pour simplifier, dans le graphe d'un nœud, on notera $c.s$ pour un composant c fourni localement. Ses nouvelles règles sont présentées dans la prochaine section.

4.6.2 Les règles de calcul du graphe de dépendance

Le principe des règles de calcul du graphe est de s'assurer que l'ensemble des exigences de chaque service fourni est disponible dans le contexte. Dans nos règles, nous avons ajouté une deuxième fonction **G** qui a un nom de nœud associé un graphe de dépendance. Les règles sur le calcul du graphe sont décrites ci-dessous : La règle GAnd calcule le graphe d'une conjonction de deux prédicats en faisant l'union des graphes correspondants à ces prédicats.

$$\textsc{GAnd:}\ \frac{Ctx_{local}, Ctx_{global}, c, s \vdash_G P_1 \Rightarrow \mathcal{G}_1 \qquad Ctx_{local}, Ctx_{global}, c, s \vdash_G P_2 \Rightarrow \mathcal{G}_2}{Ctx_{local}, Ctx_{global}, c, s \vdash_G P_1 \wedge P_2 \Rightarrow \mathcal{G}_1 \cup \mathcal{G}_2, G_1 \cup G_2}$$

Cette règle calcule le graphe d'une disjonction de deux prédicats.

$$\textsc{GOrL:}\ \frac{Ctx_{local}, Ctx_{global} \vdash_P \mathcal{Q}_1 \qquad Ctx_{local}, Ctx_{global}, c, s \vdash_G \mathcal{Q}_1, G_1 \Rightarrow \mathcal{G}_1}{Ctx_{local}, Ctx_{global}, c, s \vdash_G \mathcal{Q}_1 \vee \mathcal{Q}_2 \Rightarrow \mathcal{G}_1, G_1}$$

$$\textsc{GOrR:}\ \frac{Ctx_{local}, Ctx_{global} \nvdash_P \mathcal{Q}_1 \qquad Ctx_{local}, Ctx_{global}, c, s \vdash_G \mathcal{Q}_2 \Rightarrow \mathcal{G}_2, G_2}{Ctx_{local}, Ctx_{global}, c, s \vdash_G \mathcal{Q}_1 \vee \mathcal{Q}_2 \Rightarrow \mathcal{G}_2, G_2}$$

Les règles GServ, GServC, GSinN, GCSinN, permettent de lier les nœuds du service disponible $c^{'}.s^{'}$ au service fourni $c.s$

$$\textsc{GServ:}\ \frac{s^{'} \in AS(Ctx_{local})}{Ctx_l, Ctx_g, c, s \vdash_G s^{'} \Rightarrow \{c^{'}.s^{'} \rightarrow c.s \mid (c^{'}, \mathcal{P}_s, -, -) \in Ctx_l.\mathcal{C} \wedge s^{'} \in \mathcal{P}_s)\}, \phi}$$

$$\textsc{GServC:}\ \frac{(c^{'}, \mathcal{P}_s, -, -) \in Ctx_{local}.\mathcal{C} \quad s^{'} \in \mathcal{P}_s}{Ctx_{local}, Ctx_{global}, c, s \vdash_G c^{'}.s^{'} \Rightarrow \{c^{'}.s^{'} \rightarrow c.s\}, \phi}$$

$$\text{GSinN:}\ \frac{FPot = \{(i,c') \mid i \in \mathcal{N} \wedge s \in AS(Ctx_g(i)) \wedge (c', \mathcal{P}_s, -, -) \in Ctx_g(i).\mathcal{C} \wedge s' \in \mathcal{P}_s\} \quad G_i = \{c'.s' \rightarrow Ctx_l.n.c.s \mid (i,c') \in F}{Ctx_l, Ctx, c, s \vdash_G s' \in \mathcal{N} \Rightarrow \{Ctx_g(j).n.c'.s' \rightarrow c.s \mid (j,c') \in FPot\}, \{i \rightarrow G_i\}}$$

$$\text{GCSinN:}\ \frac{FPot = \{(i,c') \mid i \in \mathcal{N} \wedge s \in AS(Ctx_g(i)) \wedge (c', \mathcal{P}_s, -, -) \in Ctx_g(i).\mathcal{C} \wedge s' \in \mathcal{P}_s\} \quad G_i = \{c'.s' \rightarrow Ctx_l.n.c.s \mid (i,c') \in I}{Ctx_l, Ctx, c, s \vdash_G c.'s' \in \mathcal{N} \Rightarrow \{Ctx_g(j).n.c'.s' \rightarrow c.s \mid (j,c') \in FPot\}, \{i \rightarrow G_i\}}$$

Les règles GTrue, GVar, GNots, GnotC, GNotSinN, GNotCinN, GSNotinN et GCS-NotinN ne génèrent aucun effet (ni nœud ni arc).

$$\text{GTrue:}\ Ctx_{local}, Ctx_{global}, c, s \vdash_G true \Rightarrow \phi, \phi$$

$$\text{GVar:}\ Ctx_{local}, Ctx_{global}, c, s \vdash_G [v\ O\ V] \Rightarrow \phi, \phi$$

$$\text{GNotS:}\ Ctx_{local}, Ctx_{global}, c, s \vdash_G \neg s' \Rightarrow \phi, \phi$$

$$\text{GNotC:}\ Ctx_{local}, Ctx_{global}, c, s \vdash_G \neg c' \Rightarrow \phi, \phi$$

$$\text{GNotSinN:}\ Ctx_{local}, Ctx_{global}, c, s \vdash_G \neg s' \in \mathcal{N} \Rightarrow \phi, \phi$$

$$\text{GNotCinN:}\ Ctx_{local}, Ctx_{global}, c, s \vdash_G \neg c' \in \mathcal{N} \Rightarrow \phi, \phi$$

$$\text{GSNotinN:}\ Ctx_{local}, Ctx_{global}, c, s \vdash_G s' \notin \mathcal{N} \Rightarrow \phi, \phi$$

$$\text{GCNotinN:}\ Ctx_{local}, Ctx_{global}, c, s \vdash_G c'.s' \notin \mathcal{N} \Rightarrow \phi, \phi$$

4.6.3 Les règles d'installations des composants

Une fois l'installabilité des composants vérifiée, nous utilisons les règles d'installations pour calculer les effets sur le contexte et le système. La forme des règles d'installations diffère un peu des règles d'installabilités, car elle calcule les effets d'installation. La dépendance peut être invalide, elle est notée par $\perp$.Les règles d'installations sont décrites dans la suite.
Si le prédicat P n'est pas vérifié (INot1) ou le service s est interdit (INot2), la dépendance $P \Rightarrow s$ n'est pas vérifiée. Dans le cas contraire, les services et les composants interdits sont calculés par la fonction *CalcF* (ITriv)

$$\text{INot1:}\ \frac{Ctx_{local}, Ctx_{global} \nvdash_P P}{Ctx_{local}, Ctx_{global} \vdash_I c : (P \Rightarrow s) \Rightarrow \perp}$$

$$\text{INot2:}\ \frac{s \in FS(Ctx_{local})}{Ctx_{local}, Ctx_{global} \vdash_I \ c : (P \Rightarrow s) \Rightarrow \perp}$$

$$\text{ITriv:}\ \frac{Ctx_{local}, Ctx_{global} \vdash_P P \qquad s \notin FS(Ctx_{local})}{Ctx_{local}, Ctx_{global} \qquad CalcF = \mathcal{F}_s, \mathcal{F}_c \vdash_I c : (P \Rightarrow s) \Rightarrow \{s\}, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}, G}$$

Si la dépendance optionnelle $?\mathcal{D}$ n'est pas vérifiée, rien n'est fourni (IOpt1). Dans le cas contraire, les services sont calculés et le graphe est calculé. L'ensemble des services et composants interdits, ainsi que l'ensemble des services fournis, sont calculés au niveau de la dépendance $\mathcal{D}$ (IOpt2).

$$\text{IOpt1:}\ \frac{Ctx_{local}, Ctx_{global}, c \vdash_I \mathcal{D} \Rightarrow \bot}{Ctx_{local}, Ctx_{global} \vdash_I c?\mathcal{D} \Rightarrow \phi, \phi\phi\phi}$$

$$\text{IOpt2:}\ \frac{Ctx_{local}, Ctx_{global}, c \vdash_I \mathcal{D} \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}, G}{Ctx_{local}, Ctx_{global} \vdash_I c?\mathcal{D} \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \{s \to s^{'} \mid s \to s^{'} \in \mathcal{G}\}}$$

Dans le cas où, la conjonction de dépendance $\mathcal{D} \bullet \mathcal{D}$ est vérifiée, l'effet calculé est l'union des effets de $\mathcal{D}$ et $\mathcal{D}$ (IAnd3). Dans les autres cas, l'effet n'est pas calculé (IAnd3) et (IAnd2)

$$\text{IAnd1:}\ \frac{Ctx_{local}, Ctx_{global}, c \vdash \mathcal{D}_1 \Rightarrow \bot}{Ctx_{local}, Ctx_{global} \vdash_I c : \mathcal{D}_1 \bullet \mathcal{D}_2 \Rightarrow \bot}$$

$$\text{IAnd2:}\ \frac{Ctx_{local}, Ctx_{global}, c \vdash \mathcal{D}_2 \Rightarrow \bot}{Ctx_{local}, Ctx_{global} \vdash_I c : \mathcal{D}_1 \bullet \mathcal{D}_2 \Rightarrow \bot}$$

$$\text{IAnd3:}\ \frac{Ctx_l, Ctx_g, c \vdash \mathcal{D}_1 \Rightarrow \mathcal{P}_s^1, \mathcal{F}_s^1, \mathcal{F}_c^1, \mathcal{G}_1, G_1 \quad Ctx_l, Ctx_g, c \vdash \mathcal{D}_2 \Rightarrow \mathcal{P}_s^2, \mathcal{F}_s^2, \mathcal{F}_c^2, \mathcal{G}_2, G_2}{Ctx_l, Ctx_g \vdash_I c : \mathcal{D}_2 \bullet \mathcal{D}_1 \Rightarrow \mathcal{P}_s^1 \cup \mathcal{P}_s^2, \mathcal{F}_s^1 \cup \mathcal{F}_s^2, \mathcal{F}_c^1 \cup \mathcal{F}_c^2, \mathcal{G}_1 \cup \mathcal{G}_2, G_1 \cup G_2}$$

L'effet de la disjonction représente l'effet de $\mathcal{D}$ (IOrL) si $\mathcal{D}$ est vérifiée, soit l'effet de $\mathcal{D}$, si $\mathcal{D}$ est vérifiée. Sinon $\bot$, si aucune des dépendances $\mathcal{D}, \mathcal{D}$ n'est vérifiée.

$$\text{IOrL:}\ \frac{Ctx_{local}, Ctx_{global}, c \vdash \mathcal{D}_1 \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}, G}{Ctx_{local}, Ctx_{global} \vdash_I c : \mathcal{D}_1 \# \mathcal{D}_2 \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}, G}$$

$$\text{IOrR:}\ \frac{Ctx_l, Ctx_g(i), c \vdash \mathcal{D}_1 \Rightarrow \bot \quad Ctx_l, Ctx_g(j), c \vdash \mathcal{D}_2 \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}, G}{Ctx_l, Ctx_g \vdash_I c : \mathcal{D}_1 \# \mathcal{D}_2 \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}, G}$$

$$\text{IOr3:}\ \frac{Ctx_{local}, Ctx_{global}, c \vdash \mathcal{D}_1 \Rightarrow \bot \quad Ctx_{local}, Ctx_{global}, c \vdash \mathcal{D}_2 \Rightarrow \bot}{Ctx_{local}, Ctx_{global} \vdash_I c : \mathcal{D}_1 \# \mathcal{D}_2 \Rightarrow \bot}$$

Les ensembles des services interdits $\mathcal{F}_s$ et des composants interdits $\mathcal{F}_c$ sont calculés par la fonction $Calc\mathcal{F}$ définit comme suit :

$$\begin{cases} CalcF(true) = \varnothing, \varnothing \\ CalcF(P_1 \wedge P_2) = \mathcal{F}_s^1 \cup \mathcal{F}_s^2, \mathcal{F}_c^1 \cup \mathcal{F}_c^2 \text{ où } CalcF(P_i) = \mathcal{F}_s^i, \mathcal{F}_c^i \\ CalcF(Q_1 \vee Q_2) = \mathcal{F}_s^1 \cup \mathcal{F}_s^2, \mathcal{F}_c^1 \cup \mathcal{F}_c^2 \text{ où } CalcF(Q_i) = \mathcal{F}_s^i, \mathcal{F}_c^i \\ CalcF(s) = CalcF(c.s) = CalcF([v\, O\, val]) = \varnothing, \varnothing \\ CalcF(\neg\, s) = \begin{cases} \{s\}, \varnothing \texttt{ si } s \notin AS(Ctx) \\ \varnothing, \varnothing \texttt{ sinon} \end{cases} \\ CalcF(\neg\, c) = \begin{cases} \varnothing, \{c\} \texttt{ si } c \notin AC(Ctx) \\ \varnothing, \varnothing \texttt{ sinon} \end{cases} \\ CalcF(s \in \mathcal{N}) = CalcF(c.s \in \mathcal{N}) = \varnothing, \varnothing \\ CalcF(\neg\, s \in \mathcal{N}) = CalcF(\neg\, c.s \in \mathcal{N}) = \begin{cases} \{s\}, \varnothing \texttt{ si } \forall i \in \mathcal{N}\ s \notin AS(Ctx_{global}(i)) \\ \varnothing, \varnothing \texttt{ sinon} \end{cases} \\ CalcF(s \notin \mathcal{N}) = CalcF(c.s \notin \mathcal{N}) = \begin{cases} \varnothing, \{c\} \texttt{ si } \forall i \in \mathcal{N}\ s \notin AC(Ctx_{global}(i)) \\ \varnothing, \varnothing \texttt{ sinon} \end{cases} \end{cases}$$

4.6.4 Les règles d'installations des applications

Durant la vérification de l'installabilité des composants, on obtient l'ensemble des nœuds candidats pour chaque composant installable on le dénote : $(c : \mathcal{D})_{\mathcal{N}}$ avec $\mathcal{N}$: l'ensemble des nœuds candidats. Pour qu'une application soit installée, il faut que l'ensemble des ses composants soient installés dans le système, les règles sur les applications sont décrites ci-dessous :

$$\text{IAApp1:}\ \frac{\forall i \in \mathcal{N} : Ctx_{global}(i), Ctx_{global} \vdash_I c : \mathcal{D} \Rightarrow \mathcal{P}_s^i, \mathcal{F}_s^i, \mathcal{F}_c^i, \mathcal{G}_i, G^i}{Ctx_{global} \vdash_I^{\mathcal{A}} (c : \mathcal{D})_{\mathcal{N}} \Rightarrow \bigcup_{\mathcal{N}} \mathcal{P}_s^i, \bigcup_{\mathcal{N}} \mathcal{F}_s^i, \bigcup_{\mathcal{N}} \mathcal{F}_c^i \bigcup_{\mathcal{N}} \mathcal{G}_i, \bigcup_{\mathcal{N}} G^i}$$

$$\text{IAApp2:}\ \frac{\forall i \in \mathcal{N} : Ctx_{global}(i), Ctx_{global} \vdash_I c : \mathcal{D} \Rightarrow \mathcal{P}_s^i, \mathcal{F}_s^i, \mathcal{F}_c^i, \mathcal{G}_i, G^i}{Ctx_{global} \vdash_I^{\mathcal{A}} (c : \mathcal{D} \in \mathcal{N})_{\mathcal{N}} \Rightarrow \bigcup_{\mathcal{N}} \mathcal{P}_s^i, \bigcup_{\mathcal{N}} \mathcal{F}_s^i, \bigcup_{\mathcal{N}} \mathcal{F}_c^i \bigcup_{\mathcal{N}} \mathcal{G}_i, \bigcup_{\mathcal{N}} G^i}$$

$$\text{IAApp3:}\ \frac{\forall i \in \mathcal{N}' : Ctx_{global}(i), Ctx_{global} \vdash_I c : \mathcal{D} \Rightarrow \mathcal{P}_s^i, \mathcal{F}_s^i, \mathcal{F}_c^i, \mathcal{G}_i, G^i}{Ctx_{global} \vdash_I^{\mathcal{A}} (c : \mathcal{D} \in^{n..m} \mathcal{N})_{\mathcal{N}'} \Rightarrow \bigcup_{\mathcal{N}'} \mathcal{P}_s^i, \bigcup_{\mathcal{N}'} \mathcal{F}_s^i, \bigcup_{\mathcal{N}'} \mathcal{F}_c^i \bigcup_{\mathcal{N}'} \mathcal{G}_i, \bigcup_{\mathcal{N}'} G^i}$$

$$\text{IAApp4:}\ \frac{Ctx_g \vdash_A \mathcal{A}_1, G_{A_1} \qquad Ctx_g \vdash_A \mathcal{A}_2, G_{A_2}}{Ctx_g \vdash_I^{\mathcal{A}} \quad \mathcal{A}_1, \mathcal{A}_2 \Rightarrow \mathcal{P}_s^{\mathcal{A}_1} \cup \mathcal{P}_s^{\mathcal{A}_2}, \mathcal{F}_s^{\mathcal{A}_1} \cup \mathcal{F}_s^{\mathcal{A}_2}, \mathcal{F}_c^{\mathcal{A}_1} \cup \mathcal{F}_c^{\mathcal{A}_2}, \mathcal{G}_{\mathcal{A}_1} \cup \mathcal{G}_{\mathcal{A}_2}, G_{\mathcal{A}_1} \cup G_{\mathcal{A}_2}}$$

4.7 L'architecture fonctionnelle

Notre architecture d'implémentation comporte des solveurs locaux installés sur chacun des nœuds du système. Les solveurs peuvent échanger des messages entre eux. Chaque solveur utilise les règles de déploiement définies précédemment pour prendre ces propres décisions d'installation. Quand il s'agit d'une dépendance distribuée, un solveur peut communiquer avec un autre pour vérifier si la dépendance est vérifiée La figure 9 illustre une architecture avec trois nœuds qui communiquent à travers des canaux de communications qui peuvent être soit des sockets, soit un appel de procédure à distance(RMI) ou du *XmlRpc*, par exemple. Un solveur peut jouer le rôle de chef d'orchestre pour coordonner les décisions locales prises par les différents solveurs locaux, comme le cas du **solveur2** dans la figure 9.

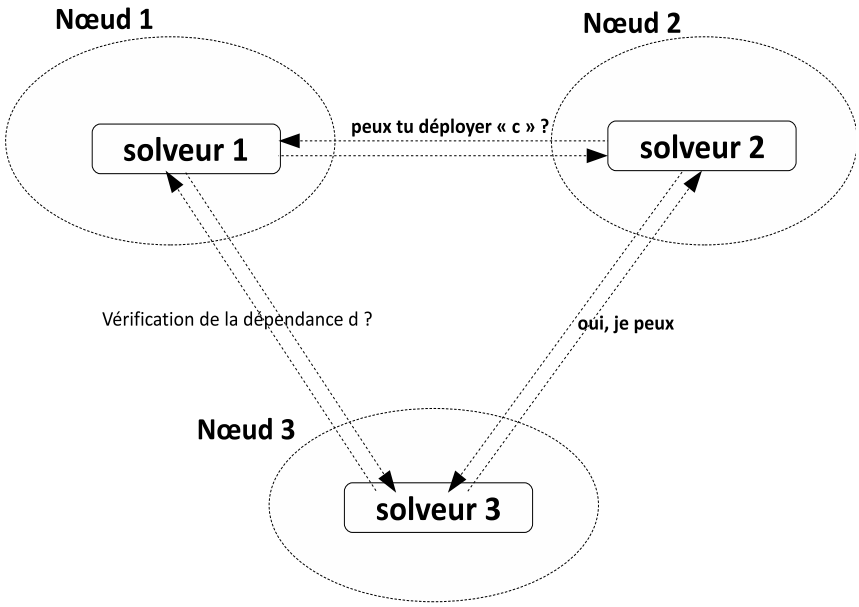

FIGURE 9 – L'architecture fonctionnelle des solveurs

Conclusions et perspectives

Ce travail s'inscrit dans le continuation du travail réalisé par [24] dans le domaine de déploiement des composants logiciels. Ce stage m'a permis d'avoir une première expérience significative dans la recherche scientifique, à travers une démarche méthodique. En commençant par définir et comprendre le problème, ensuite par voir les solutions qui ont été proposées dans la littérature. À travers l'état de l'art et ma compréhension du problème, nous avons proposé une solution pour :

- l'expression des dépendances réparties des entités à déployer.
- la vérification de l'installabilité et l'installation des composants logiciels dans un environnement de déploiement réparti.
- un papier est en cours de préparation.

Actuellement,

- sur la base de la grammaire et des règles conçues un prototype a été réalisé et a été testé sur des exemples de dépendances par un stagiaire du département informatique de Télécom Bretagne pour valider notre solution.
- l'élaboration des règles qui permettent le calcul des effets d'installations sur les nœuds et le système.

Et dans la perspective de mon stage :

1. Désinstallation et la mise à jour
2. Problèmes liés à la dynamique du système (gestion dynamique des nœuds, transaction d'installation pour défaire en cas d'erreur,...)
3. Trouver la formulation logique pour décentraliser le contexte global (Ctx_{global})

Références

[1] C.Szyperski, *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[2] M.Belguidoum, *Conception d'une infrastructure pour un déploiement sûr et flexible des composants logiciels*. Thèse de doctorat, Telecom Bretagne, 2008.

[3] S.Ahmed, *CORBA Programming Unleashed*. Sams Publishing, December 1998.

[4] "Deployment and configuration of component-based distributed applications," Object Management Group, Inc, juin 2003.

[5] Wikipédia. `http://fr.wikipedia.org/wiki/Microsoft-.NET`.

[6] Wikipédia. `fr.wikipedia.org/wiki/Common-Language-Infrastructure`.

[7] E.Bruneton, T.Coupaye, and J-B.Stefani, "Fractal component model," *France Telecom RD, INRIA*, 2003.

[8] G.Anderson and P.Anderson, *Entreprise JavaBeans Component Architecture Designing and Coding Entreprise Applications*. Sun Microsystems Press Series, 2002.

[9] A. Dearle, "Software deployment, past, present and future," in *2007 Future of Software Engineering*, FOSE '07, (Washington, DC, USA), pp. 269–284, IEEE Computer Society, 2007. `http://dx.doi.org/10.1109/FOSE.2007.20`.

[10] A. Dearle, G. N. C. Kirby, and A. J. McCarthy, "A framework for constraint-based deployment and autonomic management of distributed applications," in *Proceedings of the First International Conference on Autonomic Computing*, (Washington, DC, USA), pp. 300–301, IEEE Computer Society, 2004. `http://portal.acm.org/citation.cfm?id=1078026.1078439`.

[11] D. E.Comer and D. L.Stevens, *Internetworking with TCP/IP : Client-Server Programming and Applications*. Prentice-Hall, 2001.

[12] M. Q.Hieu Yu and B. Ooi, "Peer-to-peer computing principles and applications," in *Peer-To-Peer Computing*, vol. 2370 of *Lecture Notes in Computer Science*, p. 309, Springer Heidelberg, 2009. `http://dx.doi.org/10.1007/3-540-45440-3_3`.

[13] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici, "Programming, Composing, Deploying for the Grid," in *Grid Computing : Software Environments and Tools* (O. F. Cunha, Jose C. ; Rana, ed.), pp. 205 – 229, Springer, 2006. `http://hal.archives-ouvertes.fr/inria-00486114/en`.

[14] Wikipédia. `http://fr.wikipedia.org/wiki/Cloud-computing`.

[15] M. Mikic-Rakic and N. Medvidovic, "Architecture-level support for software component deployment in resource constrained environments," in *Component Deployment* (J. Bishop, ed.), vol. 2370 of *Lecture Notes in Computer Science*, pp. 493–502, Springer Berlin / Heidelberg, 2002. `http://dx.doi.org/10.1007/3-540-45440-3_3`.

[16] A. Flissi, J. Dubus, N. Dolet, and P. Merle, "Deploying on the Grid with DeployWare," in *Proceedings of the 8th International Symposium on Cluster Computing and the Grid (CCGRID'08)*, (Lyon, France), pp. 177–184, IEEE, may 2008. `http://hal.inria.fr/hal-00259836`.

[17] D. Hoareau and Y. Mahéo, "Middleware support for the deployment of ubiquitous software components," *Personal Ubiquitous Comput.*, vol. 12, pp. 167–178, January 2008. http://dx.doi.org/10.1007/s00779-006-0110-7.

[18] G. P.Zarate, J.P.Belaud, *Collaborative Decision Making : Perspectives And Challenges*. Ios Press, Mars 2008.

[19] M.Grabisch, R. J-L.Marichal, and E.Pap, *Agregation Functions*. Cambridge University Press, 2009.

[20] S.Malek, M.Mikic-Rakic, and N.Medvidovic, "A decentralized redeployment algorithm for improving the availability of distributedsystems," *Springer*, 2005.

[21] S.Lacour, *Contribution à l'automatisation du déploiement d'applications sur des grilles de calcul*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, décembre 2005.

[22] C. Kaed, F. Ottogali, and Y.Denneulin, "Cbay : enchères pour le redéploiement de composants sur l internet des machines," *UbiMob*, juillet 2009.

[23] A. Dearle, G. N. C. Kirby, and A. J. McCarthy, "A middleware framework for constraint-based deployment and autonomic management of distributed applications," in *Rapport Technique CS/04/2*, 2004.

[24] M. Belguidoum and F. Dagnat, "Dependency management in software component deployment," *Electron. Notes Theor. Comput. Sci.*, vol. 182, pp. 17–32, June 2007. http://portal.acm.org/citation.cfm?id=1269985.1270076.