Université Paris XIII
Ecole Doctorale Gallilé
Laboratoire d'Informatique de Paris Nord

Université de Sfax
Faculté des Sciences Economiques
et de Gestion de Sfax

Numéro d'ordre : .....

Thèse
Présentée pour l'obtention du titre de
Docteur En Informatique

par Leila ABIDI

REVISITER LES GRILLES DE PCS AVEC DES TECHNOLOGIES DU WEB ET LE CLOUD COMPUTING

Soutenue le 03 mars 2015

Président :	Roberto Wolfler Calvo (Professeur, Université Paris 13)
Directeurs de Thèse :	Christophe Cérin (Professeur, Université Paris 13)
	Mohamed Jemni (Professeur, Université de Tunis)
Rapporteurs :	Eddy Caron (MdC HDR, ENS Lyon, France)
	Ahmed Hadj Kacem (Professeur, Université de Sfax, Tunisie)
Examinateurs :	Hanna Klaudel (Professeur, Université d'Evry, France)
	Kais Klai (MdC HDR, Université Paris 13)
	Pierre Manneback (Professeur Polytech-Mons, Belgique)

Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Préliminaires

Cette thèse rentre dans le cadre d'une coopération qui s'étale sur plusieurs années entre le laboratoire LATICE (LAboratoire de recherche en Technologies de l'Information et de la Communication et génie Electrique) de Tunis (Mohamed Jemni étant le co-directeur de thèse) et Le LIPN (Laboratoire Informatique Paris Nord) de Paris 13 (Christophe Cérin étant le co-directeur de thèse). Ces travaux sont résumés à la Figure 1.1. Sur la période 2006-2010, les travaux de Heithem Abbes (co-tutelle) ont porté sur deux intergiciels de calcul : PastryGrid [ACJ08] et BonjourGrid [ACJ09]. PastryGrid est construit autour d'un mécanisme pair-à-pair de coordination des acteurs du système de calcul alors que BonjourGrid est un méta-intergiciel de grille de PCs dans le sens où il est capable de coordonner les principaux intergiciels de grilles de PCs (BOINC, Condor, XtremWeb). L'objectif commun à ces deux intergiciels est de pouvoir contrôler l'exécution de graphes de tâches de manière beaucoup plus décentralisée ; l'idée étant de se passer de coordination via un unique point de contrôle centralisé.

À partir de 2010-2011, deux extensions du travail de Heithem Abbes sont mises en place. La première est confiée à Walid Saad. Il s'intéresse principalement à la gestion des données dans les grilles de PC, plus particulièrement dans BonjourGrid qu'il a étendu pour pouvoir prendre en compte plusieurs intergiciels de rapatriement/gestion de données de la même façon que l'on peut gérer plusieurs intergiciels de calcul au sein de BonjourGrid.

À la même époque et dans le cadre d'une nouvelle co-tutelle entre le LATICE et le LIPN, nous avons pris le parti, dans mon travail, de revisiter les interactions au sein d'une grille de PCs dans le cadre des technologies du Web. Les outils construits sur la période 1990-2010 ont en effet été conçus avec des technologies et des méthodes ad-hoc et on peut légitimement se poser la question de savoir, à l'heure du Cloud et des services en ligne, comment ces intergiciels de grille pourront continuer à exister. C'est le point

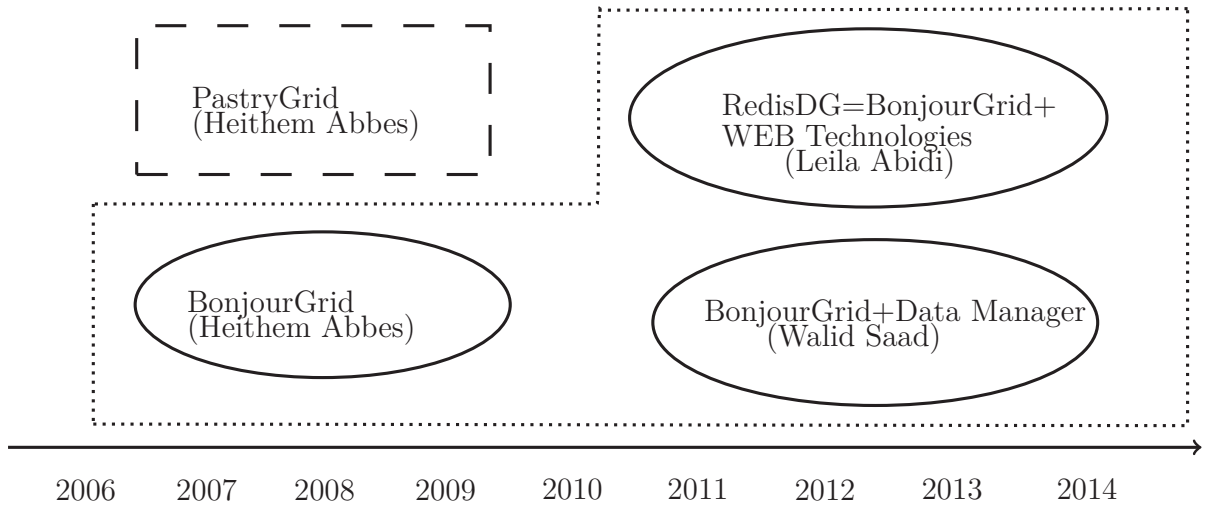


FIGURE *1.1 – Collaboration LATICE - LIPN*

de départ de notre travail.

Nous avons pris l'option de reconsidérer complètement, non pas le concept de grille de PCs, mais les interactions des acteurs des grilles de PC. Notre travail et ses apports vont de la modélisation formelle des interactions et de l'implémentation d'un prototype, à sa validation à grande échelle et son intégration dans un Cloud.

Nous discutons de l'automatisation de l'intégration dans le Cloud SlapOS [CF12] afin de rendre notre service de calcul déployable en un clic et sans intervention d'un administrateur système. Nous montrerons également comment notre travail s'articule avec ceux de Heithem Abbes et de Walid Saad ainsi que son positionnement vis à vis de ce qui se passe dans notre communauté. Nous discutons des limites du Cloud utilisé à travers plusieurs études de cas, ensuite nous présentons l'intégration de notre outil (RedisDG) et nous dégageons des pistes architecturales pour remédier aux problèmes découverts.

Notre thèse défend l'idée que le concept de grille de PCs peut s'interpréter en terme de technologies Web et que cela facilite l'accès au plus grand nombre des grilles de PCs en termes de services déployables et disponibles à la demande, dans le cadre du Cloud Computing par exemple.

1.2 Contexte

Notre travail est à l'intersection des contextes des grilles de calculs, des nouvelles technologies du Web ainsi que des Clouds et des services à la demande.

Depuis leur avènement au cours des années 90, les plateformes distribuées, plus précisément les systèmes de grilles de calcul (Grid Computing), n'ont pas cessé d'évoluer permettant ainsi de susciter des efforts de recherche toujours plus nombreux à travers le monde. Le principe général reste acquis, puisqu'il s'agit d'exploiter la disponibilité de millions d'utilisateurs sur Internet et la présence de nombreuses ressources informatiques largement sous exploitées dans le but de créer une « grande » infrastructure de calcul.

Les grilles de PCs [CF12] ont été proposées comme une alternative aux supercalculateurs par la fédération des milliers d'ordinateurs. Un PC des années 90 est capable d'exécuter la plupart des applications scientifiques, et donc une plateforme de grille de PCs est une plateforme viable pour le calcul scientifique haute performance.

Les détails de la mise en œuvre d'une telle architecture de grille, en termes de mécanismes de mutualisation des ressources, restent beaucoup plus difficile à cerner de par la multitude de définitions qui ont immergé depuis l'article fondateur de Ian Foster [FK99]. Cette puissance de calcul mise à disposition a permis et permet encore à la communauté scientifique de traiter d'innombrables applications gourmandes en calcul telles que les applications de recherche de l'intelligence extraterrestre (SETI@Home [ACK+02]), de prédiction globale du climat (Climaprediction.net [And04]), le repliement et l'agrégation des protéines (Folding@Home [LSS+09]) et l'étude des rayons cosmiques (XtremWeb [CDF+05]). Le succès de ces applications et le nombre des utilisateurs de ces plateformes prouvent le potentiel des grilles de PCs. Plusieurs projets de grilles de calcul ont vu le jour, telles que DataGrid [VBP03], TeraGrid [TER], EuroGrid [LBE03], gLite [GLI], ou Grid'5000 [BCC+06].

Parallèlement, le Web a complètement modifié notre façon d'accéder à l'information. Le Web est maintenant une composante essentielle de notre quotidien : biens dématérialisés, services en ligne, formation et éducation, jeux en ligne, objets connectés,... qui ont conduit à quinze années très riches, et un nombre incalculable d'innovations technologiques. Les équipements ont, à leur tour, évolué d'ordinateurs de bureau ou ordinateurs portables aux tablettes, lecteurs multimédias, consoles de jeux, smartphones, ou NetPCs.

Cette évolution exige d'adapter et de repenser les applications, à savoir, dans notre cas, les intergiciels de grille de PCs qui ont été développés ces dernières années. Ces applications ont été initialement construites au-dessus d'Internet comme couche de connexion afin que l'adaptation ou la remise en question du concept ne viennent pas, de notre point de vue, de l'augmentation de la vitesse d'Internet. Intégrer les technologies du Web dans le domaine scientifique est par ailleurs une piste de recherche active

depuis de nombreuses années et est communément appelée la cyber-infrastructure ou les e-Sciences.

Avant l'apparition des Clouds il y a quelques années, on nommait ASP (*Application Service Provider*) le fait de proposer une application sous forme de service. En remontant un peu plus loin en arrière, dans les années 60, IBM proposait déjà l'informatique «*on-demand*». Les années 80 furent aussi le début des concepts de virtualisation. Tous ces concepts ont amené, petit à petit, à inventer une nouvelle manière de proposer l'informatique «comme un service».

Dans un contexte économique où nous cherchons à rentabiliser au maximum les investissements et limiter l'empreinte écologique (Green IT), le Cloud Computing prétend devenir la solution. Avec une croissance exceptionnelle de 25%, il représentait plus de 56 milliards de dollars pour l'année 2009 d'après le cabinet Gartner et a atteint 150 milliards de dollars en 2013 (soit environ 10% des investissements mondiaux en informatique). Toujours selon Gartner, le Cloud Computing arrive, en 2010, en 1ère position des investissements devant le Green-IT et la virtualisation.

Dans un tel contexte, développeurs, administrateurs et décideurs ne peuvent plus ignorer l'émergence de ce nouveau marché, et doivent de-facto comprendre et analyser ces nouvelles technologies afin de mieux anticiper ce nouveau virage qu'est le Cloud Computing dans notre monde de demain. Pour certains, le Cloud Computing devrait en particulier conduire à l'ère de l'informatique totalement dématérialisée, ne laissant pour seuls dispositifs palpables que les terminaux d'accès (postes de travail, capteurs, etc.), dispositifs dont les capacités, les rôles et les usages futurs seront largement impactés.

1.3 Problématique

Dans le domaine de l'informatique, les grilles de PCs et les Clouds sont deux paradigmes bien connus. D'un coté nous avons les grilles de PCs basées sur le volontariat de ressources, et utilisées pour exécuter des applications nécessitant une grande puissance de calcul. De l'autre coté, le Cloud repose sur le partage des ressources pour assurer la cohérence et des économies d'échelle. Nous nous intéressons au couplage de ces deux concepts. Les grilles de calcul peuvent continuer à exister si nous sommes capables de transformer l'ancienne architecture maître/esclaves en une nouvelle architecture orientés Web afin d'offrir les services à la demande.

Notre objectif est de fournir une grille de PCs en tant que service, c'est-à-dire qui pourrait être utilisée d'une manière transparente pour l'utilisateur, en particulier sans aucune intervention d'un administrateur système. L'objectif est de faire en sorte que cette technologie soit accessible au plus grand nombre de personnes dans la Communauté E-science à travers l'automatisation du déploiement par exemple. Nous espérons que dans l'avenir l'utilisateur d'un intergiciel de grille de PCs n'aura pas besoin d'être

un expert en administration système et qu'une application de grille de PCs pourra être déployée facilement et simplement.

Une des problématiques rencontrées lors de l'intégration d'un système de grille de PCs qui doit être déployé dynamiquement et à grande échelle est comment reconsidérer ces intergiciels de grille de PCs pour les « cloudifier ». L'utilisateur du Cloud est au centre des préoccupations : il doit pouvoir déployer une application sans avoir besoin d'un administrateur système. Ainsi, « cloudifier » les interactions entre les composants « traditionnels » (ordonnancement, coordination, certification des résultats, monitoring,...) et les « nouveaux » (facturation, virtualisation) est un défi en soit. Plus généralement, le défi, au-delà de la mise en œuvre d'un intergiciel de grille de PCs, est de pouvoir mettre à disposition de la communauté scientifique une nouvelle génération de grille de PCs. La question est comment coordonner les nouveaux dispositifs dans un paysage d'outils qui sont des outils du Web 2.0 ?

Avec l'émergence des grands réseaux de communication, les systèmes technologiques et les nouvelles applications deviennent de plus en plus distribuées et à architecture complexe. Cette complexité est due aux exigences de l'évolutivité nécessaire et permanente, à l'hétérogénéité matérielle et logicielle et au fort degré d'interaction entre les différentes entités logicielles constituant les systèmes et les applications. Ainsi, plusieurs problèmes se posent : comment réussir à appréhender le comportement de ces systèmes afin de les concevoir et les réaliser ? Peut-on établir des modélisations fidèles aux systèmes et sur lesquelles on peut réaliser des vérifications formelles ?

Au niveau des infrastructures d'exécution, le défi est plutôt avec les nouveaux appareils (smartphones, tablettes,...), car ils représentent une «puissance de calcul de masse» au lieu des PCs qui seront considérés comme des «nœuds serveurs». Ainsi, il faut savoir intégrer ces dispositifs surtout qu'avant 2020 il est prévu que nous disposions de réseaux 5G à 10Gbp/s (100Mbp/s en 2012 à Tokyo) ; les téléphones auront plus de 64 cœurs et 16 GB de RAM (8 cœurs et 2GB RAM en 2013). De plus, actuellement, on peut utiliser des langages et des systèmes de gestion de base de données basés sur le principe de clef-valeur qui offre une grande scalabilité et de très hautes performances. Les protocoles de communication Web sont aussi un élément clé de l'infrastructure Web 2.0.

Le problème général que nous adressons est donc le suivant : comment faire en sorte qu'un service de Grille de PC soit déployable depuis n'importe où, n'importe quand (à la demande) et sur n'importe quels dispositifs (smartphones, tablettes, PC,...). Une première option est d'intégrer dans le Cloud les intergiciels les plus utilisés à ce jour comme BOINC et Condor. Ceci s'est avéré être une tâche très compliquée, parce que ces systèmes n'étaient pas conçus pour cela dès le début [CT13]. La deuxième option est de repenser les interactions entre les composants usuels d'une grille de PCs en terme de paradigmes issus du Web 2.0. C'est l'objet de ce manuscrit.

Pour résumer, les principales problématiques résident dans :

- L'hétérogénéité et la volatilité des ressources qui sont les principales caractéristiques d'une grille de PCs. Ceci rend la communication, la coordination et l'ordonnancement des tâches très difficiles à accomplir. C'est pourquoi nous avons besoin d'un mécanisme puissant au cœur de notre système afin de garantir un minimum de robustesse et de sécurité.
- Le protocole de communication, pour la coordination des ressources et non pas pour l'échange de données, qui devrait fournir un niveau élevé d'asynchronisme afin de promouvoir la scalabilité. La question à ce niveau est donc : quel est le modèle de communication approprié pour contrôler et coordonner les composants d'une grille de PCs ?
- Les systèmes qui deviennent si complexes que nous devons penser à vérifier formellement. Cela nous permettra d'avoir plus de confiance dans ce que nous implémentons. Ici nous favorisons un va-et-vient entre la modélisation/vérification et le développement. La question à ce niveau est : comment modéliser et vérifier formellement un intergiciel de grille de calcul ? Dans notre domaine, on n'a pas l'habitude de modéliser formellement nos systèmes ; on suit plutôt l'approche traditionnelle de la conception d'un système qui consiste à le concevoir, le réaliser et le tester suivant des scénarios types.
- Le rôle que peuvent jouer les technologies Web 2.0. Elles sont prometteuses et il serait donc avantageux d'en profiter. La question à ce niveau est de savoir comment le Web 2.0 et les grilles de calcul peuvent coexister ?

1.4 Contribution

Notre contribution se résume dans la réalisation d'un intergiciel de grille de PCs que nous avons appelé RedisDG. Dans son fonctionnement, RedisDG reste similaire à la plupart des intergiciels de grilles de calcul, c'est-à-dire qu'il est capable d'exécuter des applications sous forme de «sacs de tâches» dans un environnement distribué, assurer le *monitoring* des nœuds, valider et certifier les résultats,. . . .

L'innovation de RedisDG, réside dans l'intégration de la modélisation et la vérification formelles dans sa phase de conception, ce qui est non conventionnel mais très pertinent dans notre domaine. RedisDG ne s'inspire pas des anciennes architectures maître/esclaves mais plutôt des architectures orientées Web. Il est totalement basé sur la publication-souscription des événements. Il est très léger en terme de codes. C'est un système mono-couche contrairement aux intergiciels de grille de calcul traditionnels. Les étapes de conception et de réalisation de RedisDG sont décrites ci-dessous.

Tout d'abord, nous avons étudié le système BonjourGrid, une approche pour la décentralisation de la gestion et l'organisation autonome des ressources de calcul dans les systèmes de grille de PCs. Nous avons essayé de mieux cerner le comportement

d'un tel système. Pour cela nous avons commencé par une étude des mécanismes de découverte des ressources et des systèmes d'informations, en particulier les protocoles décentralisés. Nous avons étudié les systèmes de publication-souscription. Nous avons réalisé une modélisation formelle du mécanisme de publication-souscription en utilisant les réseaux de Petri colorés [JK09]. Cette modélisation générique nous a conduit à réaliser une modélisation puis une vérification formelle du système BonjourGrid.

Ce travail de modélisation nous a permis de prendre conscience à quel point les systèmes de grille de calcul sont complexes et nous avons alors étudié la possibilité de réaliser un nouveau système de grille de PCs qui rompe avec la complexité traditionnelle.

Notre approche consiste à repenser les grilles de PCs à partir d'une réflexion et d'un cadre formel permettant de les développer, de manière rigoureuse et de mieux maîtriser les évolutions technologiques à venir. Nous avons reconsidéré les interactions entre les composants traditionnels d'une grille de PCs en se basant sur les technologies du Web, et donné naissance à, RedisDG, un nouvel intergiciel de grille de PCs capable de tourner sur les petits dispositifs, i.e. smartphones, tablettes comme sur les dispositifs plus traditionnels (PCs). Notre système est entièrement basé sur le paradigme de publication-souscription, nous entendons la manière dont nous réalisons la coordination des différents composants, la façon dont une machine rejoint le système, la façon dont elle le quitte, la manière d'échanger les données, la manière de contrôler l'exécution,.... RedisDG est développé avec Python et utilise Redis comme système de gestion de base de données clef-valeur scalable.

Afin de valider RedisDG à large échelle, nous avons exécuté une application de la NASA, appelée MONTAGE. La quantité de calcul et de données impliqués dans cette application nécessite un système de gestion de workflow scalable, qui soit en mesure de coordonner et d'automatiser le transfert des données et l'exécution des tâches sur des ressources de calcul distribuées. Avec RedisDG, nous avons réussi à exécuter une instance de cette application avec 1500 tâches. Ce qui fait de RedisDG un système de gestion de workflow scalable. Nous avons aussi validé l'approche avec des expérimentations sur l'infrastructure Grid'5000 en utilisant 340 machines physiques et avec le cloud SlapOS. SlapOS est un système de Cloud Computing décentralisé inventé par l'Université de Paris 13 et par Nexedi. Commercialisé en Europe, au Japon et en Chine, il comporte des fonctions de déploiement et d'orchestration automatiques utilisées en production par SANEF, Mitsubishi, Airbus Defence ou Aide et Action. Les résultats ont prouvé le concept de RedisDG. Avec la modélisation, la vérification, le déploiement et le test de RedisDG, nous avons introduit la notion de «grille de PCs à l'âge du Web» qui correspond à offrir les fonctionnalités d'une grille de PCs sous forme d'un service Web, très facile à utiliser.

1.5 Plan de la thèse

Le rapport de cette thèse est organisé comme suit.

Chapitre 1 : il présente le contexte et la problématique de ce travail, ainsi qu'un résumé de nos principales contributions.

Chapitre 2 : ce chapitre contient les ingrédients nécessaires à la compréhension de la suite de ce manuscrit. Nous y présentons principalement certaines définitions et notations sur les réseaux de Petri place-transition, et sur les réseaux de Petri colorés. Nous y discutons aussi la nécessité de valider les systèmes informatiques.

Chapitre 3 : dans ce chapitre, nous commençons par présenter un état de l'art des systèmes de grilles de calcul. Nous mettons l'accent sur les grilles de PCs, en présentant leurs différentes caractéristiques et propriétés. Ensuite, nous discutons l'impact des technologies du Web sur les applications de grille de calcul. Puis, nous introduisons les systèmes de *Cloud Computing*, leurs caractéristiques ainsi que des exemples de Clouds. Nous détaillons, en particulier, le fonctionnement du Cloud SlapOS. Enfin, nous réalisons une étude comparative entre les deux technologies : Cloud et grille de calcul.

Chapitre 4 : ce chapitre présente nos principales contributions en termes de modélisation et vérification formelles. Nous y détaillons notre démarche scientifique. Nous exposons nos idées et nos réflexions par rapport à la modélisation formelle d'un intergiciel de grille de PCs.

Chapitre 5 : dans ce chapitre, nous détaillons la démarche suivie pour le développement de RedisDG. Nous exposons quelques algorithmes, et nous présentons l'architecture de RedisDG en illustrant ses différentes caractéristiques. L'idée de RedisDG est d'offrir les fonctionnalités ordinaires d'une grille de PCs en tant que service Web. Nous décrivons et nous comparons, aussi, les mécanismes de monitoring. Nous présentons également une évaluation expérimentale à travers une émulation.

Chapitre 6 : ce chapitre présente nos motivations à réaliser des expérimentations à large échelle. Nous introduisons les workflows scientifiques, et nous détaillons Pegasus [Peg] qui est un système de gestion de Workflows. Nous présentons également, MONTAGE, une application de la NASA et son exécution par dessus RedisDG. Finalement, nous présentons les résultats de nos expérimentations réalisées sur l'infrastructure Grid'5000.

Chapitre 7 : ce chapitre est la conclusion de ce manuscrit. Nous proposons aussi des perspectives aux travaux que nous avons réalisés.

Première partie

Eléments de contexte et de vocabulaire

Chapitre 2

Modélisation et vérification Formelle

2.1 Introduction

Dans ce chapitre nous mettons en évidence l'utilité de la modélisation et de le vérification formelle. Nous y discutons la nécessité de valider les systèmes informatiques. Nous présentons principalement certaines définitions et notations sur les réseaux de Petri place-transition, et sur les réseaux de Petri colorés. Nous exposons aussi les principales techniques de vérification formelle, et nous présentons brièvement l'outil CPNTools et ses principales fonctionnalités. Enfin nous présentons le paradigme de publication-souscription.

2.2 Utilité de la modélisation

Avec l'émergence des grands réseaux de communication, les systèmes technologiques et les nouvelles applications deviennent de plus en plus distribuées et à architecture complexe. Cette complexité est due aux exigences d'évolutivité nécessaire et permanente, à l'hétérogénéité matérielle et logicielle et au fort degré d'interaction entre les différentes entités logicielles constituant les systèmes et les applications. Ainsi, plusieurs problèmes se posent : comment réussir à appréhender le comportement de ses systèmes afin de les concevoir et les réaliser ?

Pour mieux maîtriser cette complexité, il est recommandé de disposer de méthodes et d'outils de conception particulièrement efficaces. Au centre de ces méthodes et de ces outils, se trouve la modélisation. D'où la nécessité d'avoir un niveau d'abstraction élevé et de disposer de modèles. Un modèle est une représentation d'un système permettant de le décrire de façon non équivoque.

L'utilisation de modèles prend une part de plus en plus importante dans les systèmes informatiques, que ce soit pour leur définition, leur conception, leur réalisation, leur exploitation, leur maintenance et même leur intégration.

Dans le domaine de Grid Computing on n'a pas l'habitude de modéliser formellement nos systèmes. On suit plutôt l'approche traditionnelle de la conception d'un système qui consiste à le concevoir, le réaliser et le tester suivant des scénarios types afin de vérifier si son comportement est satisfaisant et s'il est nécessaire de l'améliorer voire de le re-concevoir. C'est ce qu'on appelle une approche intuitive.

Cette approche n'est pas des plus efficace : parfois, on doit faire face à des contraintes temporelles, parfois des contraintes économiques ou tout simplement on est face à l'impossibilité de tester le comportement du système. Ainsi, l'approche intuitive peut être longue et coûteuse à mettre en place au final.

Une alternative s'impose. En fait, pour un scénario donné, un modèle permet de calculer numériquement les valeurs des variables de tous les états qu'un système peut avoir, ce qui est plus rapide et plus économique que de les mesurer en faisant des tests. C'est ce qu'on appelle simulation.

La simulation permet ainsi, dans une certaine mesure, de tester si le comportement du système est satisfaisant et de mettre en évidence certains problèmes. Elle ne remplace pas complètement l'expérimentation qui est nécessaire à l'issue d'une simulation satisfaisante.

Les limites de la simulation résident dans le fait qu'on ne peut tester qu'un nombre limité de scénarios, même si le modèle est bon. Si les scénarios choisis ne sont pas pertinents par rapport à l'ensemble des scénarios auxquels le système sera confronté durant son existence, alors il est difficile de prévoir si le comportement du système sera satisfaisant dans tous les cas.

Ainsi, en plus de réaliser un modèle et d'effectuer des simulations sur ce modèle, garantir le comportement d'un système nécessite l'analyse de propriétés spécifiques de ce modèle. L'analyse d'un système repose sur l'étude des propriétés mathématiques de son modèle. C'est la vérification formelle.

Notre travail s'intéresse à la modélisation basée sur les réseaux de Petri, et plus particulièrement, les réseaux de Petri colorés. Ce choix est motivé par un ensemble de facteurs. D'abord, ces modèles sont graphiques et permettent de représenter le système d'une manière intuitive. Ils renseignent, dans un formalisme unique, sur les deux aspects du système représenté : statique (grâce à la structure même du modèle) et dynamique (grâce à l'évolution des jetons dans la structure et éventuellement l'évolution de leurs valeurs). En outre, les réseaux de Petri sont recommandés comme des méthodes formelles notamment dans le domaine de la sûreté de fonctionnement grâce au socle théorique (mathématique) sous-jacent au modèle graphique. En termes de puissance de modélisation, les réseaux de Petri possèdent une très grande expressivité. Ils peuvent

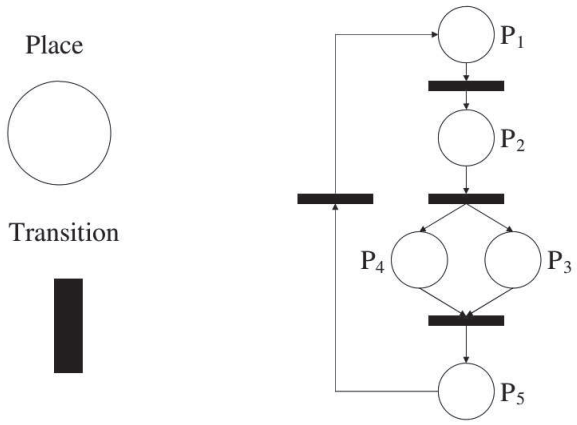


FIGURE *2.1 – Exemple de réseau de Petri*

exprimer des aspects aussi variés que les communications, les contraintes temporelles et les lois de commande. Des outils comme *CPNTools* [CPN] permettent d'offrir une aide à la modélisation, la simulation et l'analyse des réseaux de Petri.

2.3 Les réseaux de Petri colorés

Les réseaux de Petri [CG] sont apparus en 1962, dans la thèse de doctorat du mathématicien et informaticien allemand Carl Adam Petri. Les réseaux de Petri sont des outils graphiques et mathématiques permettant de modéliser le comportement dynamique des systèmes à événements afin de permettre leur conception, leur évaluation et leur amélioration.

Un réseau de Petri (voir figure 2.1) se représente par un graphe biparti (composé de deux types de nœuds) orienté (composé d'arc(s) ayant un sens) reliant des places et des transitions (les nœuds). Deux places (resp. transitions) ne peuvent pas être reliées entre elles. Les places peuvent contenir des jetons, représentant généralement des ressources disponibles. La distribution des jetons dans les places est appelée le marquage du réseau de Petri. Les entrées d'une transition sont les places desquelles part une flèche pointant vers cette transition, et les sorties d'une transition sont les places pointées par une flèche ayant pour origine cette transition.

Un réseau de Petri évolue lorsqu'on exécute (franchit) une transition : des jetons sont retirés des places en entrée de cette transition et des jetons sont déposés dans les places en sortie de cette transition. Ainsi, l'évolution de l'état du réseau de Petri correspond à une évolution du marquage. Le franchissement d'une transition est une opération indivisible qui est déterminée par la présence de jetons dans les places d'en-

trée. L'exécution d'un réseau de Petri n'est pas déterministe, car il peut y avoir plusieurs possibilités d'évolution à un instant donné (exemple : pour une place en amont de deux transitions concurrentes).

On ne distingue pas les différents jetons dans les réseaux de Petri ordinaires. En effet, ils :

- ne capturent pas les symétries d'un problème ;
- ne permettent pas d'associer des informations aux jetons ;
- ne permettent pas de paramétrer la solution d'un problème.

La solution est d'utiliser une notation concise et paramétrée des réseaux de Petri qui sont les réseaux de Petri colorés permettant d'associer une valeur à chaque jeton. Ainsi :

- les jetons sont typés par des couleurs ;
- le nombre de classes (couleurs) de jetons est fini ;
- à chaque transition sont associées différentes couleurs de franchissement (fonctions associées aux arcs, disparition/création de couleurs par le franchissement des transitions, couleurs représentées par des n-uplets).

Les réseaux colorés ont été introduits en 1997 par Kurt Jensen, afin de modéliser des systèmes complexes tout en gardant les possibilités de vérification. Lorsque le nombre d'entités du système à modéliser est important, la taille du réseau de Petri devient rapidement énorme, et si les entités présentent des comportements similaires, l'usage des réseaux colorés permet de condenser le modèle (voir figure 2.2). En effet, une couleur est une information attachée à un jeton. Cette information permet de distinguer des jetons entre eux et peut être de type quelconque. Par conséquent, une place peut contenir des jetons de différentes couleurs et une transition peut être franchie de différentes manières, selon la couleur. Ceci est réalisé en attachant un domaine de couleur à chaque place et à chaque transition. Ainsi, les arcs ne sont pas seulement étiquetés par le nombre de jetons mais aussi par leurs couleurs.

Le franchissement d'une transition est alors conditionné par la présence dans les places en entrée du nombre de jetons nécessaires, qui en plus satisfont les couleurs qui étiquettent les arcs. Après le franchissement d'une transition, les jetons qui étiquettent les arcs d'entrée sont retirés des places en entrée tandis que ceux qui étiquettent les arcs de sortie sont ajoutés aux places en sortie de cette transition.

Ainsi, pour un même système, le nombre de comportements qui peuvent être exprimés par un réseau coloré est nettement plus élevé qu'avec un réseau simple (voir figure 2.2). Ce sont des réseaux très adaptés aux architectures distribuées. D'autant plus qu'à tout réseau coloré correspond un réseau de Petri simple/ordinaire qui lui est isomorphe. Ceci permet donc d'exploiter les mêmes techniques d'analyse que celles développées pour les réseaux ordinaires en plus d'autres qui ont été complétées et adaptées aux réseaux colorés.

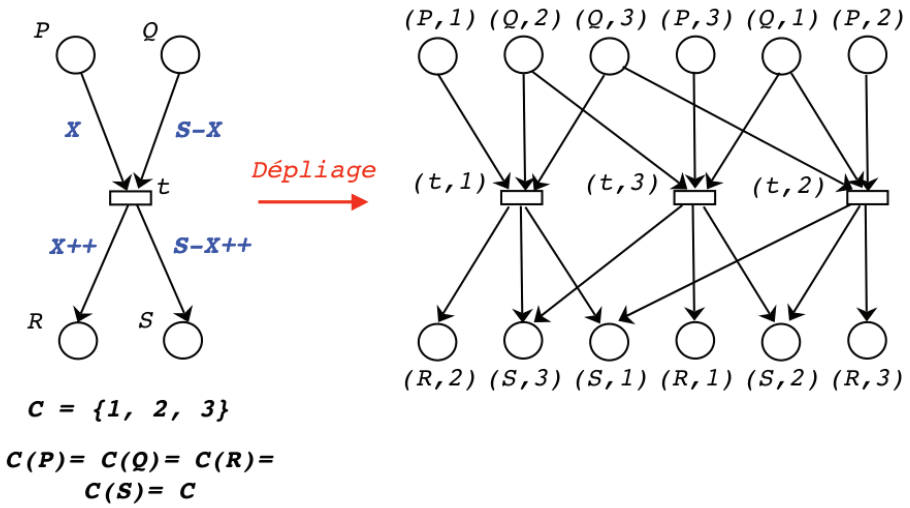


FIGURE *2.2 – Réseau de Petri coloré (gauche) vs réseau de Petri ordinaire (droite)*

Définition formelle : les réseaux de Petri colorés sont standardisés par le IEC (*International Electrotechnical Commission*) comme des réseaux de Petri à haut niveau (ISO Standard 2004). Ils sont définis comme suit : un réseau de Petri coloré est un n-uplet $CPN = (P, T, A, Labels, B, V, C, G, E, I, L)$, où :

1. P est un ensemble fini de places ;
2. T est un ensemble fini de transitions telle que : $P \cap T = \emptyset$;
3. $A \subseteq P \times T \cup T \times P$ est un ensemble d'arcs dirigés ;
4. $Labels$ est un ensemble fini de labels ;
5. B est un ensemble fini d'ensembles non vides de couleur ($types$) ;
6. V est un ensemble fini de variables typées telle que : $\forall v \in V, Types[v] \in B$;
7. $C : P \rightarrow B$ est une fonction d'ensemble de couleur associant un ensemble de couleur à chaque place ;
8. $G : T \rightarrow Expr(V)$ est une fonction de garde affectant une garde à chaque transition telle que : $Type(G(t)) = B$ et $Var[G(t)] \subseteq V$;
9. $E : A \rightarrow Expr(V)$ est une fonction d'expression d'arc affectant une expression à chaque arc telle que : $Type(E(a)) = C(p)_{MS}$, où p est la place connectée à l'arc a et MS désigne le multi-ensemble ;
10. $I : A \rightarrow Expr(V)$ est une fonction d'initialisation affectant un marquage initial à chaque place telle que : $Type(I(p)) = C(p)_{MS}$;

11. $L : P \cup T \rightarrow 2^{Labels}$ est une fonction d'étiquetage attribuant un ensemble de *labels* à chaque *place* et *transition*.

2.4 Vérification formelle

Avant le déploiement d'un modèle de workflow comme dans notre cas, ce dernier doit être vérifié et corrigé. Pour vérifier l'exactitude de la logique du modèle, deux principales approches peuvent être utilisées : la vérification de modèle (*Model-Checking*) [CGP99] et la preuve de théorème (*Theorem Proving*) [Jou04].

2.4.1 La technique de vérification de modèle

La vérification de modèle ou *model-checking* est une approche algorithmique qui consiste à vérifier automatiquement que le modèle d'un système donné satisfait les propriétés attendues. Cette technique demeure la plus utilisée puisqu'elle est totalement automatique et fournit un contre-exemple (feedback) dans le cas de détection d'erreurs. La vérification par *model-checking* consiste à générer tous les états du système qui sont atteignables à partir de l'état initial et à vérifier que la propriété énoncée est toujours satisfaite sur le graphe ainsi obtenu. Il s'agit alors d'effectuer une recherche exhaustive dans l'espace des états. Pour mettre en œuvre cette technique, il faut :

1. modéliser le système par un automate d'états finis ;
2. définir les propriétés à vérifier sur le modèle dans une logique temporelle ;
3. vérifier les propriétés définies sur le modèle.

L'inconvénient de la technique de *model-checking* est qu'elle s'applique uniquement sur des systèmes à états finis et même dans la cas de système à états finis il peut y avoir une explosion combinatoire. Des techniques d'abstractions (éventuellement guidées par l'utilisateur) peuvent être utilisées pour améliorer l'efficacité des algorithmes.

2.4.2 La technique de preuve de théorèmes

Cette technique consiste à réaliser des preuves au sens mathématique du terme, étant donné une description du système, un ensemble d'axiomes et un ensemble de règles d'inférences. La preuve de théorèmes, par rapport à la vérification de modèle, a l'avantage d'être indépendante de la taille de l'espace des états, et peut donc s'appliquer sur des modèles avec un très grand nombre d'états, ou même sur des modèles dont le nombre d'états n'est pas déterminé (modèles génériques). Cependant dans certains cas, le temps et les ressources nécessaires pour que les propriétés soient prouvées peut dépasser des temps acceptables ou les ressources dont dispose l'ordinateur. Cette

technique est très puissante au sens où elle traite des systèmes à nombre d'états infinis, mais elle est indécidable dans le cas général, et par conséquent difficile à automatiser.

2.4.3 Propriétés à vérifier

Vérifier un modèle d'applications revient à contrôler que ce modèle satisfait les besoins de l'utilisateur présentés sous forme de propriétés. Ces propriétés se présentent comme propriétés de sûreté, de vivacité, d'atteignabilité, d'absence de blocage et d'équité [SBB+99].

Une propriété de sûreté énonce que quelque chose de mauvais ne se produit jamais. Les propriétés d'invariance ou d'inatteignabilité font partie des propriétés de sûreté.

Une propriété de vivacité énonce que quelque chose de bien arrivera nécessairement. Ce type de propriétés se vérifie en analysant l'ensemble de toutes les traces possibles du système. Nous avons comme propriétés de vivacité, les propriétés de fatalité. Une propriété de fatalité énonce que sous certaines conditions quelque chose de bien finira par avoir lieu au moins une fois à partir d'un certain état.

Une propriété d'atteignabilité énonce qu'une certaine situation peut être atteinte. Nous pouvons exprimer aussi que quelque chose n'est jamais atteignable et nous parlons alors d'inatteignabilité. Dans ce cas, on se situe dans la classe des propriétés de sûreté. L'atteignabilité peut être simple ou conditionnelle quand on impose une condition sur la forme des chemins qui atteignent l'état concerné.

Les propriétés d'absence de blocage sont des propriétés énonçant que le système ne se trouve jamais dans une situation où il lui est impossible de progresser. C'est une propriété de correction pour les systèmes qui sont supposés ne jamais terminer.

Une propriété d'équité énonce que, sous certaines conditions, quelque chose aura lieu ou n'aura pas lieu un nombre infini de fois. Nous distinguons deux types d'équité : équité faible et équité forte. L'équité faible exprime que si une transition est continuellement activée alors elle est infiniment souvent activée. L'équité forte exprime que si une transition est infiniment souvent franchissable alors elle sera infiniment souvent franchie. Dans notre travail, nous nous intéressons à vérifier l'ensemble de ces propriétés.

2.5 CPNTools : un outil pour spécifier et vérifier des réseaux de Petri colorés

Le développement des réseaux de Petri colorés a été démultiplié par le désir de développer un langage fort de modélisation. La définition des réseaux de Petri est trop formelle : elle convient aux mathématiciens, mais pas aux praticiens. Donc, il faut avoir un outil efficace, qui peut aider à éviter les symboles formels. CPN Tools est conçu pour

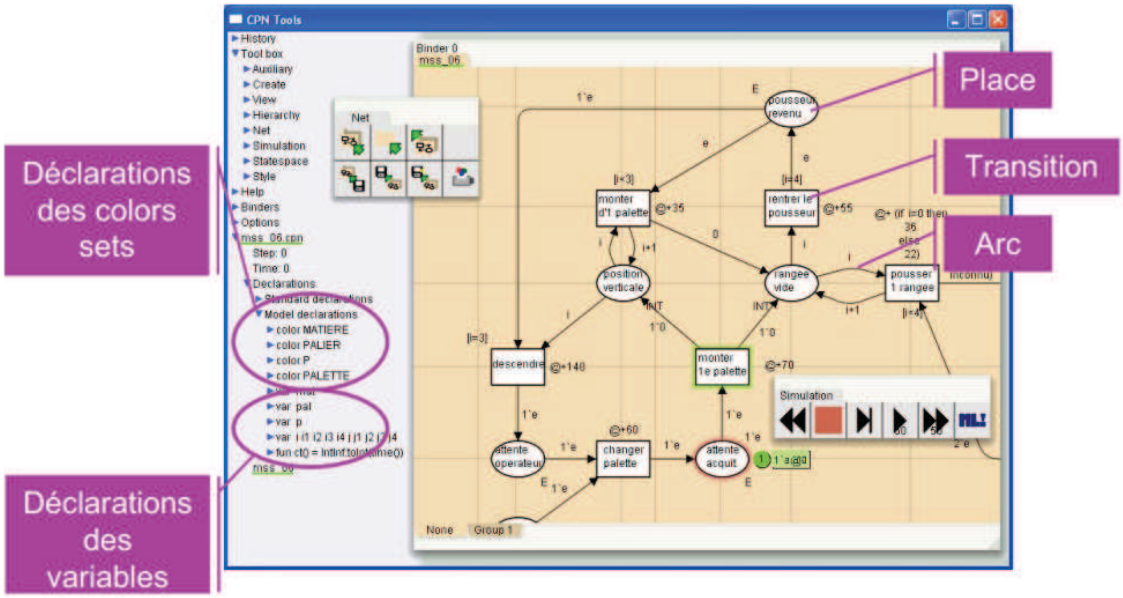


FIGURE *2.3 – Le logiciel CPNTools*

ce but. Il est développé par le CPN Group à l'université d'Aarhus au Danemark. Il est l'un des logiciels les plus utilisés dans son domaine[JKW07], dédié à la simulation de réseaux de Pétri de haut niveau (voir figure 2.3).

Il se compose de :

- CPN Editor : permet d'éditer les réseaux colorés et vérifier leurs syntaxes ;
- CPN Simulator : permet de simuler le comportement de réseau (les règles de franchissement, les transitions franchissables,. . .) ;
- CPN State space Tool : supporte la vérification (L'espace d'état).

CPNTools permet de modéliser graphiquement (dessiner) un réseau coloré dans une page de l'interface utilisateur. Le modèle comprend la structure statique du réseau coloré (places, transitions, arcs) et la partie dynamique (les jetons valués).

Grâce à CPNTools, il est possible d'utiliser des couleurs complexes et des fonctions. CPNTools combine les fonctionnalités des réseaux de Petri colorés et celle des langages. Le langage choisi est le langage fonctionnel *Standard ML* [MHMT97]. Les réseaux de Petri colorés fournissent les primitives pour décrire les processus, tandis que le langage fournit les primitives pour définir des types de données (ensemble de couleurs) et des manipulations des données (expression d'arcs, gardes,. . .)(voir figure 2.4).

Un simulateur intégré peut générer et analyser un espace d'états partiel ou total et un rapport d'espace d'états peut contenir des informations sur les propriétés du réseau

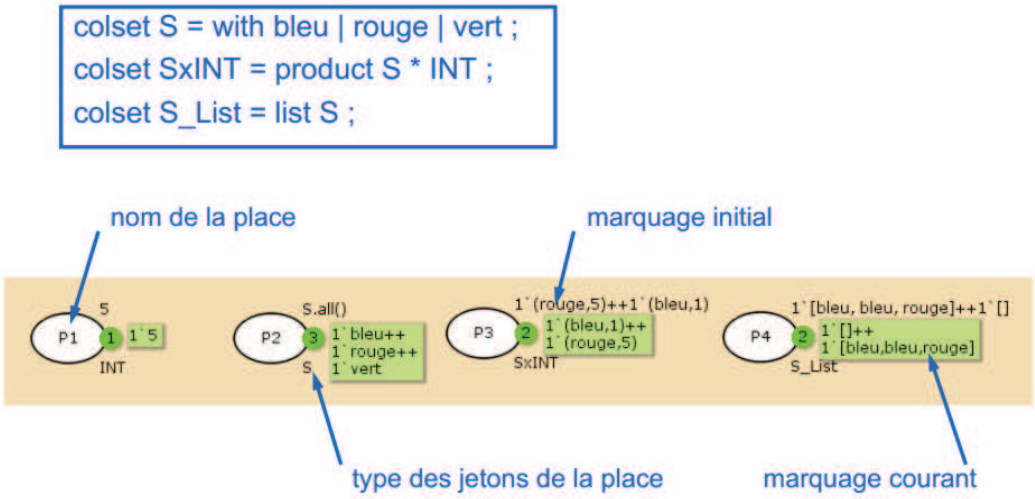


FIGURE *2.4 – Fonctionnalités du logiciel CPNTools*

de Petri coloré telles que la vivacité ou le caractère borné.

Comme le montre la figure 2.4, une place de réseau coloré dans CPN Tools est une ellipse à laquelle sont associés un nom, une couleur et un marquage initial. Les jetons d'une place doivent appartenir à l'ensemble de couleurs de celle-ci. Une transition de réseau coloré dans CPN Tools est un rectangle possédant quatre attributs facultatifs : le nom, l'expression de la garde (placée entre crochets [ ]), le code (programme en langage ML) et la temporisation. Un arc relie toujours une place et une transition. Son seul attribut, l'inscription (ou l'expression), peut contenir soit une ou plusieurs valeurs, soit une variable, soit une fonction.

Les couleurs / types simples existants dans le CPN ML, sont : unité, entier, booléen, string (caractère), énuméré, indice. Basés sur ces types, des couleurs complexes peuvent être construites en faisant par exemple le produit de ces différentes couleurs, ou alors en construisant une liste depuis un ensemble ordonné de variables du même type (même couleur). Le logiciel CPNTools fournit un ensemble de structures et de syntaxes permettant de construire des fonctions. Un tel avantage donne la possibilité de construire un RDP paramétrable dont on peut obtenir une instanciation en changeant seulement les déclarations.

Le réseau de Petri coloré est stocké sur disque sous forme d'un document XML. CPN Tools est utilisé dans différents contextes, académiques comme dans [JKW07] et industriels comme dans [JMFJ07].

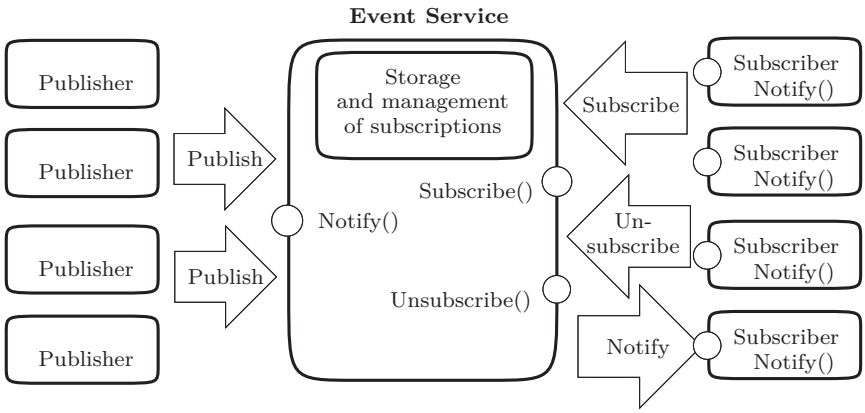


FIGURE *2.5 – Paradigme de publication-souscription*

2.6 Les systèmes de publication-souscription

Une partie de notre travail consiste à définir un protocole d'interaction entre différents composants et ce protocole est entièrement fondé sur la notion de Publication/-Souscription.

Le paradigme de publication-souscription a rapidement émergé comme une solution pertinente afin de satisfaire les besoins des applications conçues pour les environnements hautement dynamiques [EFGK03]. En effet, c'est un modèle de communication asynchrone (voir figure 2.5) qui permet à certains utilisateurs appelés «souscripteurs» (clients ou consommateurs) d'exprimer et enregistrer leurs intérêts, sous la forme de souscriptions, à des classes d'événements (ou de services) produits par d'autres utilisateurs appelés «producteurs» (ou fournisseurs de services).

Le mode de communication utilisé dans les systèmes client/serveur identifie, de façon explicite, le client et le serveur. Il s'agit d'une communication point-à-point. Le mode de diffusion multipoint (ou multicast) occupe une place stratégique au sein de l'Internet. Ce mode de communication généralise l'envoi de message de 1 émetteur à 1 récepteur à la diffusion de message de 1 émetteur à N récepteurs.

Le mode de communication dans les protocoles de type publication-souscription, comporte un nombre non prédéfini d'émetteurs d'informations associé à un nombre également non prédéfini de récepteurs. C'est la nature des informations contenues dans l'événement qui décide de l'acheminement entre les émetteurs et les récepteurs. Généralement, l'information la plus utilisée concerne le type de l'événement, on parle ainsi des protocoles de publication-souscription basés sur les types. Il existe également d'autres protocoles basés sur les thèmes et/ou le contenu [EFGK03, CDKR02].

L'avantage de la communication événementielle asynchrone est le découplage total

entre les clients et les producteurs de services, en termes aussi bien temporels (asynchronisme) que spatiaux (pas de références explicites des émetteurs et des récepteurs). En effet, ce mode de communication est multipoint, anonyme, implicite. Il est multipoint (un-à-plusieurs ou plusieurs-à-plusieurs) car les événements sont envoyés à l'ensemble des clients qui se sont déclarés intéressés. Il est anonyme car le fournisseur ne connaît pas l'identité des clients. Il est implicite puisque les clients sont déterminés par les souscriptions et non pas explicitement par les fournisseurs.

En plus des avantages présentés ci-dessus, les systèmes de publication-souscription simplifient énormément la mise en place d'un intergiciel de grille de calcul. En effet, dans la programmation classique (modèle client/serveur), on doit préciser les cordonnées des clients et du serveur, l'acheminement et les interfaces de communications, ce qui complique l'implémentation du modèle de communication, en particulier si on augmente le nombre de contraintes (disponibilité de serveurs, présence de serveurs de secours, clients volatiles,...). Contrairement à la programmation classique, la programmation mettant en œuvre les protocoles de publication-souscription est bâtie essentiellement sur les deux fonctions citées ci-dessus, ce qui rend l'implémentation d'un système distribué beaucoup plus simple.

Bien que le mode d'interaction publier-souscrire soit un paradigme émergent pour les applications distribuées à grande échelle, nous avons noté des insuffisances quant à la modélisation. Il existe différentes façons permettant de modéliser le paradigme de publication-souscription, aussi bien au niveau statique (style architectural) qu'au niveau dynamique.

Dans [KLK12], les auteurs ont proposé une méthode centrée sur l'architecture pour la conception correcte des systèmes publier-souscrire. Ils ont élaboré une approche de conception qui cherche à promouvoir la réutilisation d'une part, et de concevoir des styles publier-souscrire prouvés corrects par construction, d'autre part. L'approche se base sur un ensemble de schémas de communication prouvés corrects et réutilisables. Ces schémas intègrent des propriétés génériques pour le modèle de communication publier-souscrire. L'approche offre une démarche de construction de styles par la composition de schémas de communication. Ces schémas sont spécifiés formellement en notation Z, ainsi que la démarche de composition. La spécification formelle du style conçu est obtenue par la composition des spécifications des schémas utilisés. Une représentation visuelle en notation UML est également proposée afin de faciliter l'appréhension de la modélisation par des architectes de styles, non spécialistes en Z.

Nous verrons ultérieurement, dans la partie des contributions, que nous nous intéressons plutôt au côté dynamique : la coordination des différents acteurs est basée uniquement sur la publication-souscription des événements tout en accentuant l'aspect du découplage total.

2.7 Conclusion

Dans ce chapitre nous avons présenté quelques notions de base autour de la modélisation et la vérification formelle des systèmes informatique. Ces notions sont nécessaires à la compréhension de la suite de ce manuscrit. Nous avons aussi défini les réseaux de Petri colorés, leur utilité et leur mode de fonctionnement ainsi qu'un outil, CPN-Tools, permettant de les utiliser. Nous avons aussi présenté le principe du paradigme de publication-souscription, autour duquel tourne cette thèse.

Chapitre 3

Architecture des systèmes distribués de notre étude

3.1 Introduction

Dans ce chapitre nous exposons les principaux éléments de vocabulaire relatifs aux systèmes distribués en liaison directe avec notre sujet. Nous commençons par explorer les différents types de systèmes distribués de calcul, notamment les systèmes de grilles de calcul, les systèmes de calcul global et les systèmes pair-à-pair à partir des vues d'esprit des années 90 à 2000.

Nous présentons ensuite les principales fonctionnalités d'une grille de PCs à savoir ; l'ordonnancement, le monitoring et la certification des résultats. Nous présentons aussi le méta-intergiciel BonjourGrid qui est en fait un intergiciel de grille de PCs utilisant le protocole de publication-souscription et qui fera l'objet de notre étude dans la chapitre qui suit. Nous étudions aussi l'impact des technologies du Web sur les applications de grille de calcul, notamment les architectures orientées services, leurs évolutions et leurs limites. Nous définissons par la suite la notion de Cloud Computing, ses principales caractéristiques sur le plan technique comme sur le plan économique, ainsi que des exemples de Cloud. Nous présentons l'architecture et les concepts clés du Cloud SlapOS qui est l'infrastructure que nous utilisons.

SlapOS est un système de Cloud Computing décentralisé inventé par l'Université de Paris 13 et par Nexedi. Commercialisé en Europe, au Japon et en Chine, il comporte des fonctions de déploiement et d'orchestration automatiques utilisées en production par SANEF, Mitsubishi, Airbus Defence ou Aide et Action. Ce Cloud est en rupture par rapport aux définitions classiques de Cloud comme cela sera précisé dans ce chapitre.

Nous terminons par une étude comparative entre les systèmes de grille de calcul et les systèmes de Cloud de calcul.

3.2 Les systèmes distribués de calcul (vues des années 90 à 2000)

L'infrastructure numérique grand public est constituée (1) des appareils de grande consommation (tels que les ordinateurs de bureau, les ordinateurs portables, les tablettes, les consoles de jeux, les lecteurs multimédias, les smartphones) et (2) les réseaux de communication qui les relient entre eux, notamment Internet.

Suite aux exigences du Web en streaming vidéo et jeux graphiques 3D, les équipements sont devenus très performants. Actuellement, un PC typique dispose de 8 Go de RAM, 1 TB de disque et d'une puissance de calcul de 1 teraFLOPS sur la partie GPU. Les connexions Internet à domicile sont de l'ordre de 10 Mbps à 100 Mbps et 1 Gbps, puisque la fibre optique sera déployée dans tous les foyers dans les prochaines années.

L'infrastructure numérique comprend actuellement plus de 1 milliard de PCs privés et jusqu'à 100 millions de GPU capables de réaliser des opérations polyvalentes (pour le graphisme et pour le calcul numérique). Ceux-ci ont une capacité totale de calcul d'environ 100 ExaFLOPS, et de l'ordre de 10 Exabytes d'espace disque libre, accessible via une bande passante réseau de l'odre du petabit/seconde [CF12]. Afin de gérer les pics de charge, l'infrastructure numérique est sur-approvisionnée et donc sous-utilisée. Le CPU moyen et la connexion Internet à domicile ne sont utilisés que pendant un pourcentage minime du temps de la totalité de leur disponibilité qui est quand même significative, même avec des modes d'économie d'énergie.

Un PC moderne est capable d'exécuter la plupart des applications scientifiques actuelles, ainsi l'infrastructure numérique grand publique est une plateforme viable pour effectuer du calcul haute performance. Cette infrastructure a plusieurs avantages sur les Clouds, les supercalculateurs, les clusters et les grilles, à savoir :

- elle a une capacité de traitement et de stockage beaucoup plus grande, et donc permet de faire de « la science » autrement ;
- pour le matériel, la puissance électrique et thermique sont payées par le consommateur, et le matériel est continuellement mis à jour par des composants à la pointe de la technologie ;
- Il est auto-entretenu : les consommateurs résolvent leurs propres problèmes logiciels et matériels.

Différentes approches ont été proposées pour la mutualisation globale et la fédération des ressources à large échelle. Les approches de mutualisation peuvent être classifiées selon trois catégories : les systèmes de grilles de calcul, les systèmes de calcul global et les systèmes pair-à-pair, selon le type de l'infrastructure de connexion (réseaux institutionnels/Internet), le type des ressources (machines dédiées hautes performance/stations de travail) et la nature des utilisateurs (communauté scientifique, industrie, applications publiques/privées).

3.2.1 Les systèmes de grilles de calcul

Les systèmes de grilles de calcul fédèrent les ressources des grappes de calcul qui se trouvent, souvent, dans des institutions et des laboratoires de recherche. L'échelle d'une fédération de grappes peut aller de quelques centaines à quelques milliers de machines.

Les machines d'une fédération de grappes sont plus stables que des stations de travail et fortement connectées grâce à une infrastructure réseau de haute qualité. L'instabilité peut être causée seulement par des pannes ou des opérations de maintenance. En plus, l'environnement des grilles de calcul est fortement sécurisé. En effet, les ressources et les utilisateurs doivent avoir un certificat électronique pour s'authentifier et avoir accès à la grille.

Les applications qui exploitent ce type de plateformes demandent des calculs intensifs et un échange massif des données. D'après Ian Foster et Carl Kesselman [FK99], les grilles de calcul visent à construire une infrastructure matérielle et logicielle qui fournit un accès sûr, consistant, omniprésent et économique à des ressources de calcul haute performance. L'objectif est ainsi de déployer des applications distribuées sur une infrastructure comportant des ressources de calcul, des supports de stockage, des instruments de mesure (capteurs), des outils de visualisation (réalité virtuelle) et des bases de données.

La majorité des grilles de calcul comme EGEE [BHK+14], OSG [OSG], TeraGrid [TER], utilisent Globus [FKNT02] et gLite [GLI] comme intergiciels. Toutefois, la sécurité des ressources et l'authentification des utilisateurs garantissent un environnement sécurisé et fiable, la complexité de la procédure d'installation d'un nouvel élément et l'architecture hiérarchique suivie par Globus, représentent un obstacle qui limite le passage à l'échelle.

3.2.2 Les systèmes de calcul global

L'idée du calcul global (Global Computing) est de regrouper les ressources afin de les exploiter lorsqu'elles sont inutilisées pour effectuer des calculs. Il s'agit d'une généralisation du principe des vols de cycles à l'échelle d'Internet. En effet, la puissance de calcul d'une telle plateforme est fournie par des volontaires en offrant les périodes où leurs ordinateurs ne sont pas utilisés.

En conséquence à ce mode d'exploitation, les ressources sont volatiles et faiblement connectées à cause de l'infrastructure réseau basée sur Internet. Cette idée est exploitée par le projet de recherche d'extraterrestres dans l'univers, Seti@Home [ACK+02], qui cherche à analyser des données astronomiques obtenues grâce à un radiotélescope.

3.2.3 Les systèmes pair-à-pair

Les systèmes pair-à-pair reposent sur un principe d'égalité entre les machines tout en offrant la liberté aux machines de se connecter et de se déconnecter. Il s'agit donc de systèmes très dynamiques qui passent à l'échelle. Ils s'appuient sur une organisation la plus décentralisée possible, sans gestion d'état global et avec des caractéristiques d'auto-organisation.

Les ressources sont majoritairement des ordinateurs personnels connectés à Internet. L'échelle considérée est très grande, de l'ordre de millions de nœuds. Les principales caractéristiques sont une très grande volatilité des nœuds et un nombre important de participants frauduleux.

Les systèmes pair-à-pair sont devenus très populaires à la fin des années 90 et début des années 2000 avec l'avènement des systèmes de partage de fichiers, notamment musicaux, sur Internet. Napster [NAP], Kazaa [KAZ] et eDonkey [DON] en sont des exemples très populaires. Les systèmes pair-à-pair ont connu, aussi, un grand succès dans les domaines du travail collaboratif et des jeux en réseau.

3.3 Les grilles de PCs

Les grilles de PCs sont une variante des grilles de calcul visant à exploiter les ressources, (cycles de calcul, mémoire, espace disque,...) des ordinateurs personnels. La tendance architecturalle pour les grilles de PCs est d'avoir un système de calcul décentralisé suivant le modèle pair-à-pair pour pouvoir passer à l'échelle. Les applications de recherche de l'intelligence extraterrestre (SETI@Home), de prédiction globale du climat (Climaprediction.net) et l'étude des rayons cosmiques (XtremWeb) ont exploité par le passé les ressources des grilles de PCs.

Bien que le succès de ces applications montre le potentiel des grilles de PCs, les systèmes existants, tels que BOINC [And04] et XtremWeb [CDF+05], sont souvent centralisés et souffrent de la dépendance permanente d'une équipe administrative qui garantit le fonctionnement continu du coordinateur.

3.3.1 Caractéristiques des grilles de PCs

3.3.1.1 Participation volontaire

La puissance d'une grille de PCs, en terme d'opérations arithmétiques par secondes par exemple, dépend fortement de la participation de ressources volontaires. Il n'y a pas des contraintes qui obligent un fournisseur de ressources (propriétaire de la machine) d'offrir sa machine à la grille pendant un certain intervalle de temps ou d'attendre la

terminaison de la tâche qui tourne sur sa ressource. Par conséquent, il est intéressant que le système de grille incite les utilisateurs, en particulier ceux qui veulent exploiter la grille sans fournir des ressources, de joindre leurs machines pendant une longue période.

3.3.1.2 Volatilité des ressources

Les grilles de PCs sont construites à partir des machines personnelles des utilisateurs. Il est impérativement nécessaire de ne pas dégrader les performances de la machine, lorsque son propriétaire a besoin de ses ressources (CPU, mémoire volatile,...). Ainsi, les systèmes de grilles de PCs doivent prendre ce comportement en considération et doivent interrompre l'exécution d'une application publique (c'est-à-dire une application soumise par un utilisateur de la grille) lorsqu'ils détectent que le propriétaire de la machine a lancé un programme privé, sachant que le propriétaire peut lancer ses programmes à n'importe quel moment.

Par conséquent, le temps de disponibilité d'une machine volontaire est variable. En plus, il est possible que l'exécution d'une application publique s'arrête d'une manière inattendue à cause d'une déconnexion d'une machine. En conséquence, les cycles des temps disponibles des machines varient selon la fréquence de l'utilisation par leurs propriétaires, ce qui affecte l'exécution des applications publiques. Ainsi, il est nécessaire que les systèmes de grilles de PCs, en particulier les ordonnanceurs, tiennent compte de la volatilité des ressources afin de donner de bonnes performances et des résultats fiables.

3.3.1.3 Environnement dynamique

Les machines des utilisateurs qui sont les ressources des grilles de PCs peuvent joindre ou quitter la grille à n'importe quel moment, sans aucune contrainte. L'état de la grille varie d'une manière continue ; en effet, le nombre de machines connectées, le nombre d'applications soumises, le nombre de tâches en cours d'exécution, la bande passante et l'état des liens entre les machines changent en cours de temps. L'intergiciel de grille doit donc tenir compte du dynamisme de l'environnement.

3.3.1.4 Environnement non fiable

Des machines volontaires anonymes peuvent participer dans la grille de PCs en fournissant leurs ressources. Des machines malicieuses peuvent modifier l'exécution des applications publiques et retourner des résultats erronés. Le système de grille doit garantir la validité des résultats.

3.3.1.5 Panne des ressources

Dans les grilles de PC à large échelle, les machines sont connectées via Internet, ainsi les connexions ne sont pas fiables et peuvent être perdues aléatoirement. De plus, puisque les machines ne sont pas dédiées à la grille, leurs fréquences de disponibilité varient d'une machine à une autre selon l'utilisation de leurs propriétaires. Aussi, la contrainte d'exploiter la machine que lorsque son propriétaire ne l'utilise pas, peut causer l'interruption de l'exécution ainsi qu'un échec de terminaison. L'ordonnanceur doit tenir compte de ce comportement afin que l'exécution des applications se termine correctement et ne soit pas retardée.

3.3.1.6 Hétérogénéité des ressources

Les caractéristiques des machines des grilles de PCs différent considérablement. En effet, les caractéristiques des machines, comme CPU, mémoire, bande passante, disponibilité, comportement de faute et de disponibilité, varient d'une machine à une autre. En conséquence, les décisions d'ordonnancement deviennent de plus en plus difficiles, ce qui peut retarder les temps de terminaison des applications.

3.3.1.7 Passage à l'échelle

Les systèmes centralisés de grilles de PCs présentent un ensemble d'inconvénients comme la difficulté de passage à l'échelle lorsque le nombre de machines et d'utilisateurs devient important et la panne du serveur qui rend tout le système inaccessible. Contrairement aux systèmes centralisés, les systèmes décentralisés se caractérisent par un meilleur passage à l'échelle puisque l'ordonnancement des tâches se fait d'une manière distribuée et de manière locale sur les machines de la grille.

Cependant, la décentralisation comporte des limites par rapport à la centralisation en terme de performance et d'informations sur la grille. En effet, le système décentralisé donne une vue partielle de l'état de la grille, ainsi parfois l'ordonnanceur ne peut pas effectuer la bonne décision puisque il n'a que des informations partielles à sa disposition et ne possède pas une vue globale de la grille.

3.3.2 Principaux constituants d'une grille de PCs

3.3.2.1 Ordonnancement

Le principal défi dans les projets de calcul volontaire est que les machines volontaires, également appelés workers, sont non dédiés, volatiles, source d'erreurs, et peu fiables. En effet, les workers peuvent avoir des cycles inactifs et ne sont donc pas entièrement dédiés à l'exécution d'applications de calcul volontaire. Leur volatilité est due au fait

qu'ils peuvent soudainement quitter le projet sans avoir à retourner des résultats pour le calcul effectué.

Des erreurs liées au réseau et à l'exécution peuvent avoir lieu et ne sont pas prévisibles. Enfin, les résultats renvoyés peuvent être affectés par des attaques malveillantes, des dysfonctionnements du matériel ou du logiciel et ne peuvent donc pas être valides. Lorsque l'une de ces conditions se produit, un système de calcul volontaire devrait être en mesure d'écarter les « mauvais » résultats et redistribuer le calcul aux frais de diminution du débit global (nombre de tâches terminées) et l'augmentation de la latence (durée de vie des tâches). Pour atténuer l'impact de ces conditions, les projets de calcul volontaire se fondent sur un ordonnanceur ou scheduler.

L'ordonnanceur prend des décisions sur le type et la quantité de calcul (tâches) qui devraient être confiées à un worker. En raison de l'hétérogénéité des ressources, en terme de performance et de disponibilité, l'ordonnanceur ne peut pas appliquer les mêmes critères de distribution à chaque machine. En outre, les performances et la disponibilité d'un worker peuvent soudainement changer entre deux demandes de calcul. Idéalement, les ordonnanceurs doivent être en mesure de s'adapter dynamiquement aux caractéristiques de l'environnement de calcul volontaire.

Une telle adaptation doit prendre en compte les deux points de vue divergents des acteurs principaux dans un projet basé sur le volontariat : le point de vue des ressources (workers) et celui des scientifiques (qui consomment les ressources) [CF12]. Les scientifiques vont souvent exiger que leur travail soit retourné le plus rapidement possible et qu'il soit le plus fiable possible. La politique d'ordonnancement entraînée par le point de vue des utilisateurs tend à (1) maximiser la disponibilité des ressources, (2) minimiser la latence totale (3) maximiser le débit et (4) maximiser la fiabilité (aux prix d'une réplication plus élevée et de validations ultérieures).

Les volontaires exigent une utilisation efficace des ressources données. Ils attendent d'être reconnus en terme de crédits ou de points qui sont attribués proportionnellement à la quantité de calcul prévue. La politique d'ordonnancement entraînée par le point de vue des volontaires tend à (1) minimiser la latence, (2) maximiser l'utilisation et (3) réduire au minimum le gaspillage de ressources (par réduction de calculs redondants). Afin d'assurer une certaine qualité de service pour les volontaires, un ordonnanceur doit être capable de générer constamment une charge de travail diversifié, qui est distribuée dans les différents types d'hôtes participants. En prenant en considération ces deux points de vue, la création d'une politique d'ordonnancement efficace devient vite difficile à implémenter.

Même s'il existe une nombreuse littérature sur les algorithmes exacts d'ordonnancement [LGLK79, BLK83, BLR03, PST04, Sga98, Try12, BL99], parfois avec des garanties sur leurs performances, la plupart des politiques d'ordonnancement existantes et appliquées aux projets de calcul volontaires sont basées sur des heuristiques et peuvent être classés en deux catégories : naïves ou basées sur les connaissances. Les politiques d'or-

donnancement naïves attribuent le calcul sans tenir compte de l'historique des workers. À titre d'exemples nous pouvons citer :

1. la politique du premier arrivé premier servi (FCFS-*First Come First Served*), pour laquelle les instances de travail sont envoyées à tout hôte disponible pour faire le calcul ;
2. la politique d'ordonnancement de localité, pour laquelle les instances de travail sont de préférence envoyées à des hôtes qui ont déjà les données nécessaires pour accomplir le travail ;
3. L'assignation aléatoire, pour laquelle les instances de travail sont choisis au hasard.

Les politiques d'ordonnancement basées sur les connaissances prennent en considération l'historique des workers et l'ensemble de la communauté. À titre d'exemples nous pouvons citer ici les politiques à base de :

1. seuils fixes qui vérifient les taux de la disponibilité et de la fiabilité du worker ; si elles dépassent un certain seuil prédéfini, l'ordonnanceur attribue le travail à ce worker [EFT+06] ;
2. seuils variables ; l'ordonnanceur fait varier les seuils lors de l'exécution ; si le nombre de tâches en attente pour la distribution est plus grand que le nombre de workers, alors le seuil diminue, sinon il augmente ;
3. politique d'ordonnancement du *World Community Grid* (WCG), qui affecte le calcul en se basant sur le délai de calcul moyen du worker.

Il existe une autre famille de politique d'ordonnancement, c'est la politique adaptative [HP04, HSLL00, BBR02]. Elle nécessite des techniques intelligentes pour réinitialiser les paramètres de d'ordonnancement. Parmi les techniques possibles, une méthode évolutive distribuée peut efficacement chercher dans un large ensemble de politiques d'ordonnancement possibles, un sous-ensemble de politiques qui minimise les erreurs et les résultats non valides, tout en maximisant le débit du projet.

3.3.2.2 Monitoring

La mise en réseau de milliers d'ordinateurs (grilles de calcul) est de plus en plus adoptée par de nombreux industriels, pour déployer leur calcul sur un système distribué. Ceci d'autant plus que les systèmes propriétaires bien qu'ils répondent la plupart du temps aux besoins, ne sont pas accessibles d'un point de vue financier.

Le déploiement d'applications sur des systèmes distribués nécessite un service de surveillance des nœuds pour détecter les éventuelles défaillances et pannes et mettre en place un système de re-connexions des nœuds défaillants. Ils existent de nombreux outils permettant d'effectuer efficacement ce service. Mais le problème réside dans le fait

qu'aujourd'hui la taille des grilles de calcul devient de plus en plus grande et le système utilisé par les outils de surveillance actuels ne permettent pas d'effectuer efficacement la gestion de ces grilles.

Ces outils sont basés sur un système centralisé : une seule machine lance plusieurs processus et ouvre plusieurs sockets. Il est connu que ce mécanisme ne donne pas de bonnes performances avec un très grand nombre de nœuds. De nombreux articles ont été publiés à ce sujet et de nombreux outils mis en œuvre. Parmi eux on peut citer Chukwa [chu08]. Chukwa est un système de collecte de données pour le suivi et l'analyse des grands systèmes distribués. Il est construit au-dessus de Hadoop [HAD], un système de fichiers open source distribué et la mise en œuvre MapReduce et hérite de l'évolutivité et de la robustesse d'Hadoop. Il comprend également une boîte à outils pour afficher les résultats de suivi et d'analyse, afin de faire le meilleur usage de données collectées.

Les systèmes d'exploitation, les systèmes de fichiers et les réseaux associés présentent des défis uniques pour la surveillance, le suivi des performances et le diagnostic des problèmes. Les outils classiques de monitoring système sont impuissants face au volume de plus en plus important et diversifié des données et à l'état des systèmes de plus en plus gigantesque. En plus du volume de données important, la diversité des systèmes utilisés par les plus grands centres de calcul présente diverses informations provenant de sources multiples, ce qui complique encore les efforts d'analyse.

Le développement de nouveaux outils est nécessaire afin d'offrir un service de monitoring opérant et efficace. Dans l'article [MHD+10], «*Monitoring Tools for Large Scale Systems*», Miller et al. ont détaillé un ensemble d'outils de surveillance du système mis au point par les auteurs et utilisés par les administrateurs système de *Oak Ridge National Laboratory Leadership Computing*. Ces outils comprennent des utilitaires permettant de corréler les performances d'entrée/sortie et les données d'événements avec des systèmes spécifiques, des ressources et des tâches. Lorsque cela est possible, les services existants sont intégrés pour réduire l'effort de développement et accroître la participation communautaire.

À côté de ces techniques, d'autres personnes proposent d'utiliser les appareils électroniques pour effectuer la surveillance et le contrôle des grands systèmes distribués. Ces dispositifs peuvent comprendre des capacités de détection pour la mesure en ligne, des actionneurs pour contrôler certaines variables, les microprocesseurs pour le traitement de l'information et la prise de décisions en temps réel basés sur des algorithmes conçus, et les unités de télécommunication pour l'échange d'informations avec d'autres appareils électroniques ou éventuellement avec des opérateurs humains.

Une collection de ces dispositifs peut être considérée comme un système d'agents intelligents en réseau. Ces systèmes ont la capacité de générer un énorme volume de données spatio-temporelles qui peuvent être utilisées pour des applications de surveillance et de contrôle des grands systèmes distribués.

L'un des défis les plus importants de la recherche dans les années à venir est le

développement de méthodes de traitement des informations qui peuvent être utilisées pour extraire du sens et de nouvelles connaissances des données. L'objectif ultime est de concevoir des systèmes en réseau, d'agents intelligents qui peuvent prendre des décisions en temps réel dans la gestion de grands systèmes distribués, tout en fournissant également un haut niveau d'information aux opérateurs humains.

3.3.2.3 Certification de résultat

Comme dans de nombreux autres domaines de l'informatique, la sécurité des données et la fiabilité des ressources sont d'une importance primordiale dans les systèmes de grilles de PCs. Dans de tels environnements, les menaces existent aussi bien pour les ressources volontaires comme pour les applications.

D'une part, les volontaires ont besoin de protection contre les codes qui, par mégarde ou par malveillance, pourraient nuire à leurs ressources. En effet, un projet basé sur le volontariat peut perturber le fonctionnement régulier des machines bénévoles engendrant ainsi de graves conséquences, ce qui entrave sérieusement la perception qu'a le grand public du calcul bénévole.

D'autre part, le comportement et les résultats produits par des volontaires doivent être correctement examinés pour détecter des actions inappropriées et/ou des résultats erronés, et assurer ainsi la fiabilité des calculs effectués. Par conséquent, la sécurité et la fiabilité sont deux objectifs majeurs qui doivent être examinés avec soin dans le cadre de grilles de PCs. Cependant, comme nous le verrons dans ce qui suit, la sécurité et la fiabilité sont des questions complexes. Ainsi, les volontaires et les utilisateurs des grilles de PCs ont encore besoin d'une bonne quantité de confiance.

Modèle architectural : l'architecture d'une grille PCs comprend trois types d'entités ; les clients, les workers et un côté serveur. Les clients sont ceux qui soumettent des tâches au système afin d'être exécutées correctement et rapidement. Les workers sont les entités qui exécutent les tâches des clients. Ils travaillent en boucle : contacter le serveur pour obtenir les tâches puis calculer les résultats, et ainsi de suite.

Enfin, le serveur fournit l'infrastructure qui centralise l'ensemble de la gestion, la coordination, la création et la distribution des tâches, la réception et la validation des résultats.

Pour rendre disponibles leurs ressources, les volontaires ont besoin d'installer une infrastructure logicielle dans leurs machines afin de joindre le ou les projets dans lesquels ils veulent contribuer. Le logiciel du côté du worker est composé de deux éléments principaux : 1) le module de gestion et 2) le logiciel du worker.

Le module de gestion gère toutes les demandes de l'application et les communications au serveur extérieur. Par exemple, c'est le module de gestion qui demande des tâches

du serveur et c'est aussi lui qui envoie les résultats lorsque les tâches sont terminées. Il télécharge également, pour chaque projet, l'exécutable approprié qui effectue le calcul. Le module de gestion gère également la programmation des tâches, l'application de la configuration définie par le volontaire (par exemple, pas de calcul avant 20 heures ou de 30% de CPU pour le projet A, 45% pour le projet B et 25% pour projet C).

En résumé, le module de gestion fait face à la plupart des fonctions qui s'interfacent soit avec la machine locale (accès aux fichiers, points de contrôle, etc) ou avec le côté serveur de chaque projet. En ce qui concerne le logiciel du worker, il est constitué par le code binaire qui exécute effectivement les tâches du projet. Il effectue réellement le travail de calcul.

Comme l'informatique est une question de matériels, logiciels, et d'erreurs humaines, les grilles de PCs ont besoin de mécanismes de validation puissants pour assurer l'exactitude des résultats. Une approche largement utilisée pour évaluer les résultats est de répliquer le travail entre plusieurs workers et de comparer leurs résultats à la fin, une technique connue sous le nom de vote à la majorité ou concensus. Un résultat est acceptée si la majorité des workers renvoient le même résultat. BOINC utilise le terme *quorum* pour désigner le nombre de résultats qui doivent correspondre afin de valider un résultat. Le *quorum* est fixé par projet, il a une valeur de 2 dans plusieurs projets.

Cependant, la validation par la réplication signifie qu'une unité de travail est exécutée plus d'une fois, et donc la puissance de calcul est effectivement perdue. D'autres techniques existent pour la validation des résultats, bien que le vote à la majorité soit le plus utilisé, principalement en raison de sa simplicité.

En ce qui concerne la sécurité, les grilles de PCs ont besoin de se couvrir contre deux types distincts de menaces : 1) résultats erronés dus à un mauvais fonctionnement du matériel/logiciel ou un comportement malveillant de certains des bénévoles et 2) la vulnérabilité de la grille de PCs en terme de logiciel qui pourrait exposer les machines volontaires à des risques pouvant perturber leurs activités ou leurs données. Le middleware de grille de PCs doit être capable de faire face à ces deux cas. Nous identifions le premier type comme des menaces pour les projets et le deuxième comme des menaces pour les ressources de la grille de PCs.

Menaces en relation avec les projets : bien que les ressources volontaires semblent être des ressources sans coût du point de vue du projet, leur utilisation ne va pas sans problèmes. Un projet de calcul qui recourt à des ressources volontaires doit se protéger correctement contre les effets des ressources et les faux donateurs, les malveillants et les défectueux. Sinon, le projet est perturbé et ses résultats pourraient être sans valeur.

En outre, l'infrastructure Web du projet doit également être protégé contre les menaces courantes qui sévissent dans ces infrastructures. Les principaux types de problèmes que les projets fondés sur le bénévolat ont à traiter sont : 1) des tâches non terminées, 2) des résultats erronés, 3) des résultats altérés et 4) l'attaque sur l'infra-

structure côté serveur.

Plusieurs raisons peuvent contribuer à l'existence de tâches non terminées. Le plus commun est qu'un worker quitte le projet, soit temporellement soit définitivement, sans avoir terminé les tâches qui lui ont été assignées.

Des résultats erronés peuvent être dues à un mauvais fonctionnement matériel, logiciel, utilisateurs, ou une combinaison de tous ces éléments. Les erreurs peuvent être classées en deux types différents : *fail-stop* et *fail-silent* [MS94]. Les premières forcent l'application à s'arrêter de manière inattendue, puisqu'elles sont facilement détectables. Au contraire et comme leur nom l'indique, les *fail-silent* se produisent en silence et pourrait passer inaperçues.

Différentes motivations peuvent conduire un volontaire à tenter d'altérer les résultats. Une de ces motivations est la cupidité de crédit, c'est la tentative d'obtenir le plus grande nombre de crédits en utilisant le minimum de ressources informatique. En effet, de nombreux projets publics récompensent leurs bénévoles avec des crédits virtuels conséquence de l'effort de calcul effectué par la machine.

Ces crédits servent à organiser le classement des workers, une pratique qui est très populaire parmi les bénévoles, en particulier les plus dévoués [And04]. Par conséquent, l'obtention de crédits est une motivation importante pour certains bénévoles, poussant certains d'entre eux à concevoir des stratégies pour déjouer le système de crédit.

Menaces en relation avec les ressources : une question importante en ce qui concerne les grilles de PCs est que le bénévolat d'une ressource ne doit, en aucune cas, exposer la ressource à n'importe quel type de danger en dehors de ceux qui pourraient avoir lieu suite à une connexion à Internet. En effet, l'exposition des ressources à des vulnérabilités pourrait probablement avoir des conséquences graves, et pourrait certainement freiner ou arrêter le concept de volontariat.

Les menaces sur les ressources de la grille peuvent être classés comme 1) accidentelles ou 2) intentionnelles. Les menaces accidentelles peuvent être provoquées par une application ou par le middleware. Elles peuvent perturber l'utilisation régulière de la machine locale, et causer des accidents, une utilisation élevée de ressources, sans oublier les atteintes à la confidentialité et à l'intégrité des données. Comme son nom l'indique, les menaces intentionnelles sont délibérément conçues pour causer un préjudice.

Le *Sandboxing* est une approche possible pour protéger les ressources volontaires du code malfaisant. Le *Sandboxing* est définit comme le phénomène d'exécuter du code arbitraire en toute sécurité en le contenant dans un milieu fermé, c'est-à-dire un code arbitraire peut être exécuté sans compromettre le système sur lequel il est en cours d'exécution [FC08]. Le *Sandboxing* a été utilisé dans l'intergiciel XtremWeb depuis sa création. BOINC met en œuvre une forme bénigne de *Sandboxing*, en s'exécutant dans un compte non privilégié spécialement créé avec un non accès aux données des

utilisateurs. Cela se fait dans les différentes plateformes qu'il prend en charge, à savoir Linux, Windows et Mac OS X.

Les déclinaisons actuelles du confinement sous Linux sont les mécanismes de LXC ou encore le projet Docker qui est très en vogue. LXC, contraction de l'anglais Linux Containers, est un système de virtualisation, utilisant l'isolation comme méthode de cloisonnement au niveau du système d'exploitation. Il est utilisé pour faire fonctionner des environnements Linux isolés les uns des autres dans des conteneurs partageant le même noyau et une plus ou moins grande partie du système hôte. Le conteneur apporte une virtualisation de l'environnement d'exécution (processeur, mémoire vive, réseau, système de fichier,...) et non pas de la machine. C'est pour cette raison que l'on parle de « conteneur » et non de machine virtuelle.

Docker est un projet open source qui automatise le déploiement d'applications dans des conteneurs logiciels. Docker étend le format de conteneur Linux standard, LXC, avec une API de haut niveau fournissant une solution de virtualisation qui exécute les processus de façon isolée. En fait Docker utilise les mécanismes LXC, cgroups, et le noyau Linux lui-même. Contrairement aux machines virtuelles traditionnelles, un conteneur Docker n'inclut pas de système d'exploitation, à la place il s'appuie sur les fonctionnalités du système d'exploitation fourni par l'infrastructure sous-jacente.

Certification des résultats : valider les résultats que les nœuds/workers retournent au coordinateur central est l'un des problèmes fondamentaux des systèmes de calcul basés sur le bénévolat. Dans son ouvrage [Sar02], Sarmenta a formalisé un certain nombre de mécanismes pour certifier les résultats : vote à la majorité, contrôle par sondage et les systèmes basés sur la crédibilité.

Dans le vote à la majorité, les nœuds participent à un groupe de vote, où les votes sont les résultats du calcul. Pour `2m-1` nœuds, le coordinateur accepte le résultat qui apparaît `m` fois ou plus. L'inconvénient de cette méthode est le niveau élevé de redondance. Pour éviter d'exécuter plusieurs fois la même tâche, avec la méthode de contrôle par sondage, le coordinateur central envoie des travaux spéciaux pour tester l'efficacité des workers.

L'hypothèse ici est que les nœuds malveillants ne peuvent pas différencier ces travaux spéciaux des restes des travaux. Les workers qui réussissent ce test de repérage seront sollicités par le coordinateur, et inversement, il écarterait les workers qui échouent à ce test. Malheureusement, la méthode de contrôle par sondage ne résout pas tous les problèmes. En particulier, quand les workers peuvent facilement obtenir de nouveaux identifiants.

Malgré une entité centrale, nous ne pouvons pas compter sur les mécanismes d'authentification complexes, car les volontaires peuvent tout simplement abandonner le projet. Pour surmonter ce problème, les systèmes basés sur la crédibilité utilisent l'approche contraire : le worker n'est pas digne de confiance jusqu'à ce qu'il se révèle être

digne de confiance, et la confiance dans les résultats doit dépasser un certain niveau auprès du coordinateur. Dans le paragraphe qui suit nous passons en revue ces trois mécanismes de base.

BOINC a un démon consacré à vérifier si les résultats sont valables : il s'agit du *validator*. BOINC offre un framework qui permet aux développeurs de projet de créer une fonction de validation. Cette fonction indique à BOINC quand on doit tenir compte de deux résultats identiques. Il est intéressant de noter que le cadre de la validation offre également deux types particuliers de validateurs : *sample bitwise validator()*, et *sample trivial validator()*. Le premier valide les résultats bit par bit, tandis que le second prend en compte deux résultats à correspondre si leur temps de CPU dépasse un certain seuil minimum. Les responsables du projet à exécuter peuvent accepter cette option s'ils ont confiance aux workers.

3.3.3 BonjourGrid : Orchestration de multiples instances d'intergiciels de grille

3.3.3.1 Principe de BonjourGrid

Le principe de BonjourGrid [ACJ09, ACJ10] est de créer, dynamiquement et d'une manière décentralisée, un environnement spécifique d'exécution pour chaque utilisateur sans l'intervention d'un administrateur système. Les environnements sont indépendants. Chaque utilisateur, utilisant sa machine de bureau dans son office, peut soumettre une application. Ici il est important de mentionner que BonjourGrid permet non seulement d'exécuter des applications sans précédences entre les tâches mais il supporte aussi les applications contenant des dépendances entre les tâches. Cette caractéristique est en réalité issue des systèmes de calcul supportés par BonjourGrid comme XtremWeb-CH.

Comme décrit dans la thèse de Abbes.H [Abb09], BonjourGrid déploie un coordinateur, localement sur la machine de l'utilisateur, qui demande des participants (workers). Des négociations pour sélectionner les participants adéquats doivent prendre place ici. En utilisant une infrastructure de Publication-Souscription, chaque machine publie son état (repos, worker ou coordinateur) ainsi que ses caractéristiques : charge locale ou prix d'utilisation, dans le but de fournir des critères de sélection des participants.

Sous ces hypothèses, le coordinateur peut sélectionner un ensemble de machines selon des critères de choix (fréquence de processeur, taille de la mémoire volatile, type de processeur, charge locale de processeur, mémoire libre). Le coordinateur et l'ensemble des esclaves choisi construisent un élément de calcul (EC) qui exécutera, contrôlera et coordonnera les différentes tâches de l'application de l'utilisateur.

Lorsque un EC termine l'exécution d'une application, son coordinateur devient libre, retourne en état de repos et libère tous les esclaves qui lui sont attachés afin qu'ils retournent à l'état de repos. Lorsque aucune application n'est soumise au système,

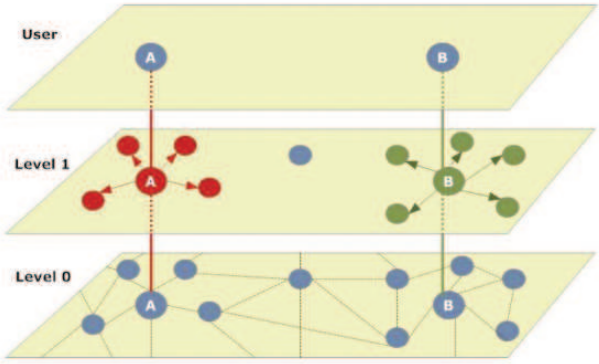


FIGURE *3.1 – BonjourGrid : L'utilisateur A (resp. B) déploie localement un coordinateur sur sa machine avec NA=4 (resp. NB=5) esclaves. Niveau 1 (Level 1) montre l'état de l'infrastructure après la construction de deux éléments de calculs (CEs) pour A et B. Niveau 0 (Level 0) présente la couche physique de l'infrastructure.*

toutes les machines prennent l'état de repos.

L'idée clé de BonjourGrid consiste à utiliser les systèmes de calcul déjà existants mais en orchestrant et en coordonnant plusieurs instances, c'est-à-dire plusieurs éléments de calcul, à travers un système de publication-souscription. Chaque élément de calcul est maintenu par un utilisateur qui a démarré le coordinateur sur sa machine. Par la suite, cet élément de calcul est chargé de l'exécution d'une ou plusieurs applications du même utilisateur.

Comme montré sur la figure 3.1, au niveau utilisateur (user), un utilisateur A (resp. B) soumet son application en utilisant sa machine et ne se rend pas compte de toutes les étapes de construction de l'élément de calcul adéquat pour l'exécution de son application. Le niveau 1 (couche intergiciel) montre que, réellement, un élément de calcul composé de 4 (resp. 5) esclaves est crée dynamiquement, spécifiquement pour l'utilisateur A (resp. B). Le Niveau 0 montre que toutes les machines sont connectées et disponibles pour n'importe quel utilisateur.

Le système BonjourGrid est composé en trois parties fondamentales :

1. un protocole de découverte de ressources complètement décentralisé, basée sur le protocole Bonjour ;
2. un élément de calcul, en utilisant les intergiciels de grilles de PCs existants comme XtremWeb, BOINC et Condor, qui exécute et gère les différentes tâches d'applications ;
3. un protocole de coordination, complètement décentralisé, entre 1) et 2) pour gérer et contrôler les différents éléments de calcul.

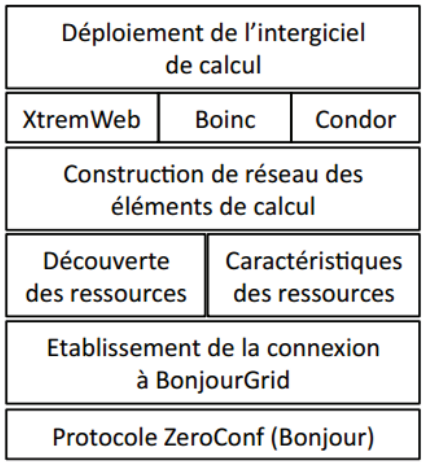


FIGURE *3.2 – Architecture en couches de BonjourGrid*

La figure 3.2 présente une architecture en couches des différentes composantes du système BonjourGrid [Abb09]. Au niveau le plus bas (couche 1), on trouve la couche physique de communication basée sur le protocole ZeroConf. Au dessus (couche 2), la couche connexion de BonjourGrid qui maintient la connexion des différentes machines appartenant au système BonjourGrid. La couche 3 implémente le protocole de découverte de ressources, ainsi que la détection de leurs caractéristiques. La couche 4 prépare l'environnement demandé par l'utilisateur en cherchant et regroupant les machines avec les critères demandés. Une machine connectée à BonjourGrid peut disposer d'un ou plusieurs (selon le choix de l'utilisateur) intergiciels de calcul pré-installés (mais inactifs). La couche 5 présente une machine avec trois intergiciels installés : XtremWeb, Boinc et Condor. Dans la couche 6, BonjourGrid termine la construction de l'élément de calcul en déployant un intergiciel de calcul parmi XtremWeb, Condor et BOINC.

Dans la thèse de Heithem Abbes [Abb09], les trois questions suivantes ont été étudiées : Est ce qu'un système de grille de PCs peut être basé sur un protocole de publication-souscription ? peut-il passer à l'échelle ? Quel est le temps de publication d'un service ? Quel est le temps de découverte d'un service ?

Heithem Abbes a montré, avec des expériences menées sur 300 machines (AMD Opterons connectées via un réseau de 1GO/s) de Grid'5000, que le protocole Bonjour est fiable et adéquat dans la découverte de ressources. En effet, Bonjour découvre plus de 300 services publiés simultanément au même instant, dans moins de 2 secondes avec 0% de perte. Ces expériences supportent et justifient le choix de Bonjour comme protocole sur lequel se base le système de découverte de ressources. En plus, le protocole est utilisé par un industriel majeur (Bonjour est disponible sur toutes les machines

Macintosh d'Apple). Il est d'une grande importance et d'une grande utilité aussi pour la communauté de recherche comme cela a été montré par Heithem Abbes.

3.3.3.2 Changement d'états dans BonjourGrid

Passage de l'état repos à l'état coordinateur : un utilisateur soumet son application via une interface utilisateur locale de BonjourGrid (installée sur sa machine). La machine de l'utilisateur change d'état, elle prend l'état coordinateur et démarre la phase de construction d'un nouvel élément de calcul.

Si cette machine est un esclave pour un autre élément de calcul, BonjourGrid attend jusqu'à la fin de la tâche en cours d'exécution, ensuite il lance le coordinateur afin de ne pas réduire les performances de la machine. C'est un choix préliminaire ; il est possible de migrer les tâches à une autre machine au repos ; s'il n'y a pas de machines au repos, BonjourGrid informe le coordinateur pour mettre la tâche en pause jusqu'à ce qu'il trouve une nouvelle machine au repos ou un esclave qui a terminé l'exécution de sa tâche ; ainsi l'utilisateur ne va pas attendre longtemps, spécialement s'il y a des tâches longues.

Le coordinateur commence par lancer le programme de découverte (ou Browser) sur les machines au repos. Les machines découvertes sont enregistrées dans un dictionnaire. À partir de cette structure de données, le coordinateur sélectionne les machines qui répondent aux caractéristiques demandées par l'utilisateur et crée un nouveau dictionnaire. Le coordinateur continue la recherche des machines au repos, jusqu'à ce qu'il atteint le nombre de machines demandé par l'utilisateur (c'est-à-dire, la taille de l'élément de calcul). Par la suite, le coordinateur arrête le programme de découverte.

Maintenant, le coordinateur doit vérifier si toutes les machines sélectionnées vont accepter de travailler pour lui. En effet, le coordinateur publie un service *WorkerRequest*, pour chaque machine sélectionnée, en utilisant le nom de machine comme type de service. Notons que le programme de découverte effectue une recherche sur les types et non pas sur les noms de services. La confirmation de participation d'une machine «A» au coordinateur «C» signifie que «A» publie un nouveau service *MyConfirmation* en utilisant le nom du coordinateur «C» comme type de service. Uniquement le coordinateur «C» lance un programme de découverte sur les services ayant son nom comme type.

Ainsi, si le browser découvre un nouveau service signifie que la participation d'une nouvelle machine est accordée. Si le nombre de confirmations n'atteint pas la taille de l'élément de calcul, le coordinateur lance de nouveau une découverte sur les machines au repos, mais, maintenant, avec le nombre manquant des machines seulement.

Dans la couche intergiciel, BonjourGrid établie la connexion entre le coordinateur et les esclaves, qui ont confirmé leurs participations, en déployant le système de calcul choisi par l'utilisateur. En effet, lorsque une machine publie le service *CoordinatorSer-*

vice, elle lance en parallèle le programme de coordinateur. Ce coordinateur reste en écoute sur les connexions des esclaves (tournant le programme d'esclave). Lorsque il termine l'exécution de l'application, le coordinateur désactive le service *Coordinator-Service* et arrête le programme Coordinateur. Il retourne à son état initial en publiant le service IdleService.

Passage de l'état repos à l'état esclave : lorsque une machine rejoint le système BonjourGrid, elle prend l'état initial de repos. Cette machine reste en attente des demandes des coordinateurs en lançant un programme de découverte sur les coordinateurs demandeurs. Lorsqu'elle découvre des demandes de participation, la machine ne prend en compte que la première en publiant le service *MyConfirmation* comme décrit ci-dessus. Par la suite, la machine arrête le service de découverte, désactive le service *IdleService* et publie le service *WorkerService* afin d'annoncer son nouvel état : elle n'est plus libre, elle travaille pour un coordinateur.

L'esclave en question tourne le programme esclave avec l'adresse IP du coordinateur. Cet esclave reste en possession du coordinateur tant que ce dernier est en vie. Il exécute un programme de découverte (browser) sur les services de type portant le nom de son coordinateur, pour qu'il soit informé de toutes les notifications issues de ce dernier, en particulier, si le coordinateur n'existe plus (c'est-à-dire la machine a retourné à son état initial de repos). Dans ce cas, l'esclave désactive le service *WorkerService*. Finalement, la machine retourne à son état initial en publiant le service *IdleService*.

3.3.4 L'impact des technologies du Web sur les applications de grille

Intégrer les technologies du Web dans le domaine scientifique est une piste de recherche active depuis de nombreuses années et est communément appelée la cyber-infrastructure ou e-Science. Une grande partie de cette recherche a porté sur les architectures orientées services (SOA-*Service Oriented Architecture*) qui sont basées sur les normes de services Web tels que WSDL (*Web Services Description Language*) et SOAP (*Simple Object Access Protocol*) et leurs nombreuses spécifications associées.

Bien que ces interfaces basées sur XML et les normes de messagerie représentent une simplification majeure dans le domaine de l'informatique distribuée orientée objets, tels que CORBA (*Common Object Request Broker Architecture*), elles ont prouvé dans la pratique être encore trop compliqué. Même si XML est souvent présentée comme un format de balisage « lisible », les normes de base (WSDL et SOAP) restent très complexes et sont habituellement générés par des outils logiciels. Cette situation se complique encore par les nombreuses spécifications qu'il faut ajouter pour la notification, la fiabilité, la sécurité, l'adressage et la coordination. Non seulement tout cela a rajouté de la complexité, mais cela a empêché les services Web d'arriver à maturité

dans le domaine de l'e-Science et ainsi ils n'ont pas vraiment été adoptés comme cela était prévu à l'origine.

À la fin des années 90, il semblait que l'utilisation des services Web dans les grilles était inévitable. La communauté scientifique s'attendait à ce que les services Web dominent la nouvelle génération de logiciels d'entreprise des années 2000. On a également supposé que les grilles serait en mesure de tirer parti de l'important investissement commercial dans ce domaine et être construite en termes de services Web. Cependant, ce n'est pas ce qui s'est passé.

En fait, l'expérience a démontré que les services Web sont souvent compliquées, lents et ayant des fonctionnalités inférieures que les approches traditionnelles. Par exemple, *WS-Security* est assez lent alors que WS messagerie semble avoir une mauvaise presse en plus d'être inadéquat pour les opérations multi-cast. Des normes telles que WSDM (gestion décentralisée) sont inutilement complexes, ce qui entrave sa vaste adoption.

Certes, il existe des bons supports .NET et Java pour les services Web et les spécifications WS. Ces supports fournissent un ensemble de standards à la fois riche, sophistiqué et complexe au niveau de la sécurité, la tolérance aux pannes, les méta-données, la découverte, la notification, etc. Par ailleurs, la montée en puissance du Web 2.0 montre que l'innovation des logiciels commerciaux se passe dans un espace différent de ceux des services Web et au final ceux-ci n'ont pas eu d'adoption significative.

Ainsi, même si l'idée de construire des grilles de PCs en termes de services Web semblait être prometteuse, l'adoption de ces systèmes par l'entreprise ou par la communauté des E-Sciences reste à faire. Les logiciels d'entreprise et les services Web ont évolué, mais chacun dans une direction distincte.

Le Web 2.0 est un terme «fourre-tout» souvent appliqué à un large éventail d'activités sur le Web tels que Google Map et ses applications composites, les blogs, les wikis (avec RSS associé) et les réseaux sociaux tels que Facebook. Il désigne l'ensemble des techniques, des fonctionnalités et des usages du *World Wide Web*, en particulier les interfaces permettant aux internautes ayant peu de connaissances techniques de s'approprier les nouvelles fonctionnalités du Web.

Ces activités mettent l'accent sur la simplicité et la participation des usagers. Ainsi, les internautes contribuent à l'échange d'informations et peuvent interagir (partager, échanger, etc.) de façon simple, à la fois avec le contenu et la structure des pages, mais aussi entre eux.

De nombreux services Web 2.0 comme YouTube, Flickr, del.icio.us, etc, ont des interfaces de programmation ainsi que des interfaces utilisateur. L'approche «do-it-yourself» du Web 2.0 est un modèle très attrayant pour la science ce qui pourrait éventuellement permettre aux scientifiques de l'informatique et aux chercheurs en technologie de l'information de collaborer plus étroitement. Nous pouvons également tirer parti des vastes infrastructures en ligne de Google, Microsoft, Amazon et autres grandes entreprises.

Les fonctionnalités techniques comprennent les interfaces pour l'accès et les applications composites (composées à partir d'autres applications) pour l'intégration de l'information Web. Il y a aussi des fonctionnalités très populaires tels que les blogs et les wikis supportant une communication soit à large échelle soit au sein d'une organisation. Google Maps et les technologies connexes illustrent la puissance des technologies interactives, intégratives et contributives. Elles sont emblématiques du Web 2.0 : elles ont révolutionné le monde extrêmement complexe des systèmes d'information géographique avec des normes XML beaucoup plus simples et des APIs de programmation qui popularisent le processus de développement.

Les technologies du Web 2.0 ont également envahi un territoire plus traditionnel de la cyber-infrastructure. Ainsi, le développement du Web 2.0 inclut les systèmes de Cloud qui prennent en charge le stockage et le calcul distribués, des fonctionnalités qui ont jusqu'à présent été la particularité des grilles.

Du point de vue du développeur, les systèmes de type Clouds offrent beaucoup plus de simplicité au niveau de la programmation, de l'allocation des ressources et des modèles de sécurité que la grille de calcul. Cela peut être attribué à des problèmes de positionnement commercial ; en fait, les grilles sont beaucoup plus orientées recherche, et ont souvent des cas d'utilisation relativement complexes, alors que les systèmes Cloud sont motivés par des considérations économiques et par conséquent doivent faire appel au plus simple pour les cas d'utilisation.

Le Web 2.0 peut jouer en faveur de l'E-Sciences de nombreuses façons. Ses outils peuvent renforcer la collaboration scientifique, à savoir soutenir efficacement les organisations virtuelles, d'une autre manière que les grilles, qui focalisent sur la gestion hautement sécurisée du partage des ressources.

Avec sa notoriété, le Web 2.0 peut offrir des technologies et des logiciels de haute qualité qui (en raison de gros investissements commerciaux) peuvent être très utiles pour l'E-Science et de meilleure qualité que les services offerts par la grille ou les solutions des services Web. En outre, l'ergonomie et le caractère participatif du Web 2.0 peuvent amener les sciences à un public plus large.

On peut facilement combiner les technologies des services Web (SOAP, UDDI, REST,. . .) avec celles du Web 2.0 (HTTP,. . .). Toutefois, dans un tel monde hybride, les systèmes vont naturellement évoluer vers le plus petit dénominateur commun. Ainsi, la complexité de la messagerie SOAP et des spécifications WS ne pourront pas prospérer.

Bien que les technologies du Web 2.0 se soient plus concentrées sur les interactions entre utilisateurs et moins sur le calcul (ordonnancement, exécution de tâches,. . .), le Web 2.0 et les grilles s'attaquent à des classes d'applications similaires. Ainsi, les technologies de composants pour les grilles et le Web 2.0 possèdent des fonctionnalités similaires, et il devrait être fructueux de comparer et combiner les idées réunissant les deux systèmes.

3.4 Le Cloud computing

Depuis 2007, le terme Cloud est devenu l'un des mots les plus à la mode dans l'industrie des Technologies de l'Information (TI). De nombreux chercheurs tentent de définir le Cloud Computing à partir de différents aspects de l'application, mais il n'existe pas de définition consensuelle. Parmi les nombreuses définitions, nous discutons de trois largement citées. D'après Ian Foster et al (2009) [FZRL09], il s'agit de : « Un paradigme informatique distribué à grande échelle, qui est entraîné par les économies d'échelle, et dans lequel un ensemble de plateformes et de services abstraits, virtualisés, dynamiques, scalables, gérés par une puissance de calcul et de stockage, sont fournis à la demande pour des clients externes sur Internet ».

En tant que représentant académique, Foster se focalise sur plusieurs caractéristiques spécifiques qui diffèrent des autres paradigmes de calcul distribué. Pour lui, le Cloud Computing doit être extrêmement scalable, un univers dans lequel les entités informatiques sont virtualisés et fournis comme des services. Ces services sont configurés dynamiquement et entraînés par les économies d'échelle.

Gartner [Gar], qui est une entreprise de conseil en TI, définit le Cloud Computing comme un style de l'informatique dans lequel des fonctionnalités scalables et élastiques sont fournies comme des services à des clients externes en utilisant les technologies de l'Internet. Il s'agit d'une légère révision de la définition originale de Gartner publié en 2008. Gartner a remplacé « massivement scalable » par « scalable et élastique » pour indiquer qu'une caractéristique importante du passage à l'échelle est l'élasticité, et non pas seulement la taille.

D'après le NIST (*National Institute of Standards and Technology* aux États-Unis), le Cloud Computing est un modèle permettant, sur demande, un accès réseau pratique et omniprésent à un ensemble de ressources informatique partagées et configurables (réseaux, serveurs, support de stockage, applications et services) qui peuvent être rapidement approvisionnés et libérés, avec un minimum d'effort de gestion, ou un minimum d'interactions avec le fournisseur.

Par rapport aux deux définitions ci-dessus, le NIST fournit une définition relativement plus objective et spécifique, qui définit non pas uniquement le concept global de Cloud, mais aussi les caractéristiques essentielles des modèles de Cloud Computing, de livraison et de déploiement.

Du point de vue économique, le Cloud Computing est essentiellement une offre commerciale d'abonnement économique à des services externes. Selon le NIST, il existe trois catégories de services qui peuvent être offerts en Cloud Computing (voir Figure 3.3) :

- ***Infrastructure as a Service (IaaS)*** : c'est le service de plus bas niveau. Il consiste à offrir un accès à un parc informatique virtualisé. Des machines virtuelles sur lesquelles le consommateur peut installer un système d'exploitation

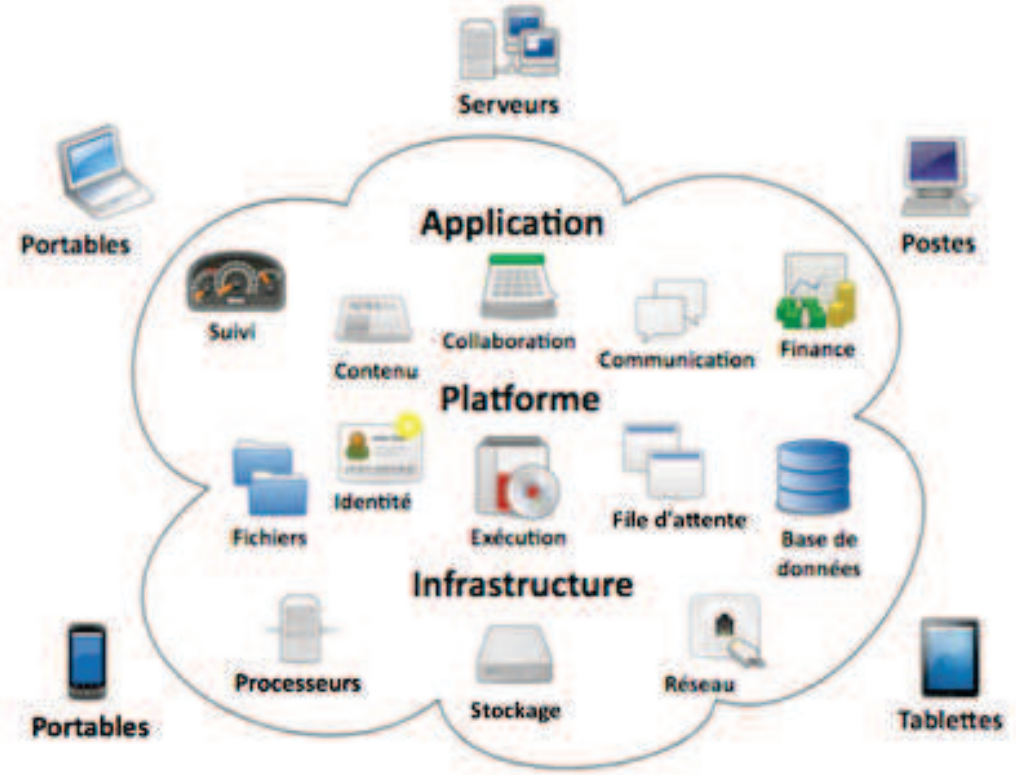


FIGURE *3.3 – Les différents niveaux de services dans le Cloud*

et des applications. Le consommateur est ainsi dispensé de l'achat de matériel informatique. Ce service s'apparente aux services d'hébergement classiques des centres de traitement de données, et la tendance est en faveur de services de plus haut niveau, qui font davantage abstraction des détails techniques.

– ***Platform as a Service (PaaS)*** : Dans ce type de service, le système d'exploitation et les outils d'infrastructure (bases de données, sécurité, stockage) sont sous la responsabilité du fournisseur. Le consommateur a le contrôle des applications et peut ajouter ses propres outils. La situation est analogue à celle de l'hébergement Web où le consommateur loue l'exploitation de serveurs sur lesquels les outils nécessaires sont préalablement placés et contrôlés par le fournisseur. La différence étant que les systèmes sont mutualisés et offrent une grande élasticité - capacité de s'adapter automatiquement à la demande, alors que dans une offre classique d'hébergement Web l'adaptation fait suite à une demande formelle du consommateur.

– ***Software as a Service (SaaS)*** : Dans ce type de service, des applications sont mises à la disposition des consommateurs. Les applications peuvent être manipulées à l'aide d'un navigateur Web, et le consommateur n'a pas à se soucier d'effectuer des mises à jour, d'ajouter des patches de sécurité et d'assurer la disponibilité du service. Gmail est un exemple de tel service. Il offre au consommateur un service de courrier électronique et le consommateur n'a pas à se soucier de la manière dont le service est fourni. D'autres exemples de logiciels mis à disposition en Saas sont Google Apps ou Office Web Apps.

3.4.1 Les modèles de déploiement

Les Clouds peuvent être déployés de différentes manières, selon les domaines d'utilisation. Il existe quatre modèles primaires de déploiement de Cloud.

-**Cloud public** : c'est le paradigme du Cloud Computing standard, dans lequel un fournisseur de services rend les ressources, telles que les applications et le stockage, à la disposition du grand public sur Internet. La facturation est en générale unitaire et détaillée par utilisateur ou par groupe d'utilisateurs. Des exemples de Clouds publics comprennent Amazon Elastic Compute Cloud (EC2), Blue Cloud d'IBM, Sun Cloud, Google AppEngine et Windows Azure Services Platform.

- **Cloud privé** : il ressemble plus à un concept de marketing. Il décrit une architecture de l'informatique propriétaire qui fournit des services à un nombre limité de personnes sur les réseaux internes. Les organisations ayant besoin d'un contrôle précis sur leurs données préfèrent un Cloud privé, afin qu'ils puissent bénéficier des avantages de la scalabilité et d'agilité d'un Cloud public sans céder le contrôle, la sécurité et les coûts récurrents à un fournisseur de service. Ebay et HP CloudStart sont des exemples pour le déploiement de Cloud privé.

- **Cloud hybride** : il utilise une combinaison de Cloud public, Cloud privé et peut même intégrer des infrastructures locales, ce qui est typique pour la plupart des entreprises. La stratégie hybride conduit à la mise en place de stratégies pour la charge du travail à effectuer en fonction de coûts, de facteurs opérationnels et de normes de conformité. Les principaux fournisseurs tels que HP, IBM, Oracle et VMware créent des plans appropriés pour exploiter un environnement mixte, dans le but de fournir des services à l'entreprise. Les utilisateurs peuvent déployer une application hébergée sur une infrastructure hybride, dans laquelle certains nœuds sont en cours d'exécution sur un matériel physique réel et d'autres sur les instances de serveur de Cloud Computing.

- **Cloud communautaire** : ce concept se chevauche avec des grilles dans une certaine mesure. Il se réfère à plusieurs organisations dans une communauté privée partageant l'infrastructure du Cloud. Les organisations ont généralement les mêmes préoccupations par rapport à leurs missions, leurs exigences de sécurité, la politique, et les facteurs de conformité. Un Cloud de communauté peut toujours être agrégé par un

Cloud public pour construire une structure sans barrières.

3.4.2 Caractéristiques du Cloud

En tant que modèle d'approvisionnement des ressources, le Cloud Computing intègre un certain nombre de technologies existantes qui ont été appliquées dans le Grid Computing, l'informatique utilitaire, les architectures orientées services, l'Internet des objets, etc. C'est la raison pour laquelle le Cloud est suspecté de ne pas être une innovation. Dans cette section, nous distinguons entre les aspects techniques, qualitatifs et économiques du Cloud Computing.

3.4.2.1 Aspect technique

Les caractéristiques techniques constituent la base qui assure certaines exigences fonctionnelles et économiques. Toute la technologie n'est pas forcément nouvelle, mais pourrait être améliorée pour réaliser une fonction spécifique.

- **Virtualisation** : c'est une caractéristique essentielle du Cloud Computing. La virtualisation dans le Cloud se réfère à une plateforme matérielle multi-couche, système d'exploitation, dispositif de stockage, les ressources réseau, etc. La première caractéristique importante de la virtualisation est la capacité de masquer la complexité technique aux utilisateurs, et ainsi d'améliorer l'indépendance des services de Cloud. La seconde caractéristique est que les ressources physiques peuvent être configurées et utilisées de manière efficace, étant donné que plusieurs applications sont exécutées sur la même machine. Troisième caractéristique, la récupération rapide et la tolérance aux pannes sont autorisées, parce que l'environnement virtuel peut être facilement sauvegardé et migré sans interruption de service.

- **Multi-location** : c'est un critère hautement requis dans le Cloud, permettant le partage des ressources et des coûts entre plusieurs utilisateurs. La multi-location apporte aux fournisseurs de ressources de nombreuses prestations, par exemple, la centralisation de l'infrastructure dans les régions, avec une réduction des coûts et l'amélioration de l'utilisation et de l'efficacité, avec une capacité de charge élevée.

- **Sécurité** : c'est l'une des plus grandes préoccupations pour l'adoption du Cloud Computing. On ne peut pas douter de l'importance de la sécurité dans tout système traitant des données sensibles et privées. Afin d'acquérir des clients potentiels, les fournisseurs doivent fournir un certificat de sécurité, par exemple, les données doivent être totalement isolées, et un mécanisme efficace de réplication et de récupération doivent être prêts en cas de catastrophe. En terme de complexité, la sécurité est élevée lorsque les données sont distribuées sur une zone plus grande où un plus grand nombre de dispositifs dans des systèmes qui sont partagés par des utilisateurs indépendants. D'autre part la réduction de la complexité est nécessaire, parce que le critère «facilité d'utilisa-

tion» peut attirer plus de clients potentiels.

- **l'environnement de programmation** : cela est essentiel pour exploiter les caractéristiques du Cloud. Celui-ci doit être capable de gérer des enjeux tels que les multiples domaines administratifs, les grandes variations dans l'hétérogénéité des ressources, la stabilité des performances, la gestion des exceptions dans des environnements très dynamiques, etc. L'interface système adopte les API de services Web, qui fournissent un cadre normatif pour l'accès et l'intégration avec et entre les services de Cloud. Grâce aux API pré-définies, les utilisateurs peuvent accéder, configurer et programmer les services de Cloud.

3.4.2.2 Aspect qualitatif

Les caractéristiques qualitatives se réfèrent aux qualités ou propriétés de Cloud Computing, plutôt qu'aux exigences technologiques spécifiques. Une caractéristique qualitative peut être réalisée de plusieurs façons en fonction de différents fournisseurs.

- **Elasticité** : c'est le fait que la prestation de services soit élastique et adaptable, permettant aux utilisateurs de demander le service en temps quasi réel sans ingénierie pour les pics de charge. Les services sont évalués à grains fins, de sorte que le montant de l'offre peut parfaitement correspondre à l'usage du consommateur.

- **Disponibilité** : elle fait référence à une capacité pertinente qui répond aux exigences spécifiques des services externalisés. Dans de nombreux cas d'utilisation, des paramètres de qualité de service tels que le temps de réponse et le débit doivent être garantis.

- **Fiabilité** : elle représente la capacité d'assurer un fonctionnement constant et sans interruption du système. En utilisant les sites redondants, la possibilité de perdre des données et du code diminue considérablement de sorte que le Cloud Computing peut assurer la continuité des activités et de reprise après un sinistre. La fiabilité est une exigence de QoS particulière, mettant l'accent sur la prévention de la perte.

- **Agilité** : c'est une exigence de base pour le Cloud Computing. Les fournisseurs de Cloud doivent être capables de réagir immédiatement à l'évolution de la demande des ressources et des conditions environnementales. En même temps, des efforts sont faits par les clients pour remettre à disposition une application à partir d'une infrastructure en interne pour les fournisseurs de SaaS. L'agilité nécessite les deux côtés pour assurer des fonctionnalités d'auto-gestion.

3.4.2.3 Aspect économique

Les caractéristiques économiques distinguent le Cloud Computing par rapport aux autres paradigmes informatiques. Dans un environnement commercial, les offres de services ne sont pas limités exclusivement à une perspective technologique, mais s'étendent

à une meilleure compréhension des besoins des entreprises. Nous trouvons les différents modes suivants :

- **Paiement à la demande (Pay-as-you-go)** : c'est une approche commune de Cloud Computing, qui signifie que les utilisateurs paient en fonction de leur consommation réelle de ressources. Traditionnellement, les utilisateurs doivent être équipés avec tous les logiciels et l'infrastructure matérielle avant de commencer le calcul, et les maintenir pendant le processus de calcul. Le Cloud Computing réduit le coût de l'entretien des infrastructures et de leur acquisition, afin d'aider les entreprises, en particulier les petites et les moyennes, à réduire les délais de commercialisation et d'obtenir un retour sur l'investissement.

- **Dépenses opérationnelles** : l'infrastructure est généralement fournie par un tiers et n'a pas besoin d'être achetée, de sorte qu'il est plus facile pour les utilisateurs d'entrer dans le monde de l'informatique. La tarification sur une base utilitaire de calcul est à grain fin avec des options basées sur l'utilisation. Il est cependant possible que les fournisseurs de Cloud masquent à l'avenir cette granularité, et mettent en œuvre des accords de prix pour la commodité des clients.

- **Efficacité énergétique** : elle est due à l'aptitude du Cloud à réduire la consommation de ressources non utilisées. Les ressources sont gérées de façon centralisée, de sorte que les coûts supplémentaires de la consommation d'énergie ainsi que les émissions de carbone peuvent être mieux contrôlées que dans les systèmes non-coopératifs.

3.4.3 Quelques exemples de Cloud

Il existe différentes solutions Open Source pour le déploiement de Clouds. Parmi celles-ci, nous avons Eucalyptus [Euc], OpenStack [Opea], OpenNebula [Opeb], CloudStack [Clo] et OpenShift [Opec]. Dans la suite nous présentons brièvement chacune de ces solutions.

3.4.3.1 Eucalyptus

Eucalyptus [Euc] est une solution qui permet l'installation d'une infrastructure de Cloud privé et hybride. Il est écrit en langage Java, C et Python. Il offre un contrôleur de stockage principal et des contrôleurs sur chaque nœud. Le réseau est géré par le composant contrôleur de Cloud, et chaque contrôleur est authentifiée par un système de clés SSH qui sert aussi à l'autorisation pour authentifier les transactions. La disponibilité d'Eucalyptus est limitée, par rapport à l'extensibilité massive, et le code source de certains de ses modules est inaccessible. C'est pourquoi il est abandonné pour d'autres solutions. Eucalyptus est la plateforme utilisée par Amazon EC2.

3.4.3.2 OpenNebula

OpenNebula [Opeb] est une autre solution de Cloud, il est open source sous licence Apache 2. Il est écrit en C++, Ruby et Shell. C'est un projet qui fournit un ensemble de fonctionnalités permettant de gérer complètement un Cloud. Plus exactement, OpenNebula organise le fonctionnement d'un ensemble de serveurs physiques, fournissant des ressources à des machines dites « virtuelles ». C'est l'orchestration et la gestion du cycle de vie de toutes ces machines virtuelles qui est au centre de ce type de solution.

C'est donc au cœur des datacenters que se déploie cette solution. Elle s'adresse en général à des grands comptes dotés d'une infrastructure informatique réseau complexe. OpenNebula est également compatible avec les hyperviseurs classiques. Il peut fonctionner aussi bien avec des outils open source comme KVM, Xen qu'avec des logiciels propriétaires comme ceux de VMware ou encore Hyper-V de Microsoft.

Ce projet n'est pas récent, il a vu le jour en 2005 avec la sortie d'une première version utilisable en 2008. On notera également qu'il s'agit d'un projet européen, développé par une société espagnole.

C'est au niveau du stockage des données que se situent les principales nouveautés. De nombreuses versions ont permis d'obtenir aujourd'hui des évolutions fonctionnelles importantes concernant le support des nœuds de stockage, les fonctions de haute disponibilité et l'ergonomie des interfaces d'administration. Il s'agit d'une solution facile à utiliser pour les centres de données et les Cloud privés.

3.4.3.3 CloudStack

CloudStack [Clo] est une plateforme de Cloud IaaS open source, développé à l'origine par Cloud.com. En Avril 2012, Citrix a fait don de CloudStack à *Apache Software Fondation*, tout en changeant la licence Apache 2.0. CloudStack implémente les API Amazon EC2 et S3, ainsi que l'API vCloud, en plus de sa propre API. Ecrit en Java, CloudStack est conçu pour gérer et déployer les grands réseaux de machines virtuelles. Il prend actuellement en charge les plateforme Cloud VMware, Oracle VM, KVM, Xen et XenServer. CloudStack a une structure hiérarchique, qui permet de gérer plusieurs hôtes physiques à partir d'une interface unique.

3.4.3.4 OpenStack

OpenStack [Opea] est une solution récente en cours de développement. Elle a un grand potentiel en raison de son architecture et de la communauté et par le soutien de ses partenaires du secteur privé. Tout le code est distribué sous licence Apache 2 licence. OpenStack est une plateforme développée par la NASA dédié aux infrastructures massives. Les principales caractéristiques d'OpenStack sont :

– la sclabilité : OpenStack est déjà déployé dans le monde entier dans des entreprises dont les volumes de données est mesuré en pétaoctets, sur des architectures distribuées et massivement extensible. On parle de jusqu'à 1 million de machines physiques, et jusqu'à 60 millions de machines virtuelles et des milliards d'objets stockés.

– compatible et flexible : OpenStack supporte la plupart des solutions de virtualisation du marché comme ESX, Hyper-V, KVM, LXC, QEMU, UML, Xen et XenServer.

– open source : l'intégralité du code peut être modifié et adapté en fonction des besoins. Le projet OpenStack présente également un processus de validation pour l'adoption et le développement de nouvelles normes.

3.4.3.5 OpenShift

OpenShift [Opec] est la solution de RedHat pour faire du PaaS. Jusqu'à présent, Docker dont on a parlé précédemment à propos du confinement et Red Hat utilisaient des versions incompatibles du noyau Linux. Le partenariat qui vient d'être signé au printemps 2014 permet aux développeurs d'utiliser des conteneurs Docker dans OpenShift et de les déplacer facilement. Cela pourrait faire d'OpenShift un acteur majeur à terme, en substituant complètement la notion de machine virtuelle par la notion de conteneurs.

3.5 Le Cloud SlapOS

C'est en 2009-2010 qu'a eu lieu une réelle explosion du Cloud sur le marché avec l'arrivée de Google App Engine, Microsoft Azure, IBM Smart Business Service et bien d'autres. En mars 2010, IELO, Mandriva, Nexedi et Tiolive qui sont quatre éditeurs de logiciels libres en France se réunissent pour fonder la «*Free Cloud Alliance*» (FCA) dans le but de promouvoir des Clouds Open Source, avec une offre globale réunissant IaaS, PaaS et SaaS, constituée de tous les composants libres nécessaires aux applications Progiciel de Gestion Intégrée (ERP-Entreprise Ressource Planning), gestion de la relation client (CRM-Customer Relation Management) ou gestion de la connaissance (KM-Knowledge Management).

En 2011 le projet Resilience, retenu dans le cadre du 12ème appel à projets du fonds unique interministériel (FUI-12) fut lancé. Ce projet regroupe des universitaires (Paris 13, Institut Télécom, INRIA) et des industriels dont Nexedi, Sagem Morpho, Wallix, Vifib. Une synthèse du projet, coté Paris 13 est maintenant en ligne [1]

1. Voir `http://lipn.univ-paris13.fr/~cerin/SyntheseLIPNresilience.pdf`

Le projet Resilience se base sur le Cloud libre SlapOS [SSCC11], dont les principales caractéristiques sont les suivantes et en première approximation :

- il ne repose pas sur de la virtualisation ;
- il ne repose pas sur des centres de données (data-centers) ;
- il reprend en partie une architecture de grille de PCs : des machines à la maison abritent des services et des données ; un «maître» contient un annuaire des services et les services eux mêmes ;
- la propriété d'interopérabilité s'obtient comme dans Grid'5000, c'est-à-dire en déployant le démon SLAPGRID sur des noeuds d'Amazon, d'Azure,... puis en installant sur ces nœuds les bonnes versions logicielles. Cela évite d'utiliser par exemple LibCloud qui est une librairie servant à l'interopérabilité des Clouds, mais rien ne garantit que chacun des Clouds tourne la même version de LibCloud ce qui est généralement une condition nécessaire au bon fonctionnement global.

SlapOS est un système de Cloud Computing distribué dont le but est d'offrir un environnent de Cloud résilient et à moindre coût. SlapOS fournit une plateforme d'automatisation du déploiement des applications et il est basé sur les technologies Buildout (pour le déploiement) et l'ERP Open Source ERP5 (pour la gestion de la relation client et le catalogue des applications déployables).

Un point particulier dans SlapOS est qu'il n'est pas basé sur la virtualisation, toutefois il est possible de déployer des machines virtuelles à travers KVM par exemple mais ceci est une option. Ce choix est motivé d'une part pour éviter l'empilement de couches logicielles qui peuvent nuire in-fine aux performances et d'autre part parce qu'on peut traiter le confinement par des techniques liées au système d'exploitation. SlapOS cherche à bâtir un Cloud en faisant reposer les concepts propre au Cloud sur le système d'exploitation, autant que possible,... ainsi que sur un ERP pour gérer le catalogue des applications et la relation client.

3.5.1 Architecture de la plateforme

SlapOS tourne un démon appelé SLAPGRID qui est construit sur la base des idées de l'intergiciel de grille de PC BonjourGrid qui est ici couplé à ERP5. Cela permet de gérer la plateforme de Cloud avec facturation des services.

La vision SlapOS d'un Cloud est simplement celle d'un ERP couplé à un modèle de déploiement, couplé à des nœuds. Il ne s'agit pas d'une vision pour faire que du HPC dans le Cloud. Cette vision est plus générale puisqu'elle inclue explicitement la facturation (sous-entendu, la mesure des ressources consommées) alors que dans le cas des grilles ou Cloud HPC cette problématique est plutôt relayée au second plan au profit d'une problématique de gestion des machines virtuelles et de leurs migration par

exemple. SlapOS a été conçu dans le cadre d'un projet industriel et non pas dans le cadre d'un projet HPC universitaire ce qui explique sans aucun doute ces choix.

SlapOS est basé sur une architecture (voir la Figure 3.4) dans laquelle les nœuds esclaves sont tous connectés a un nœud maître.

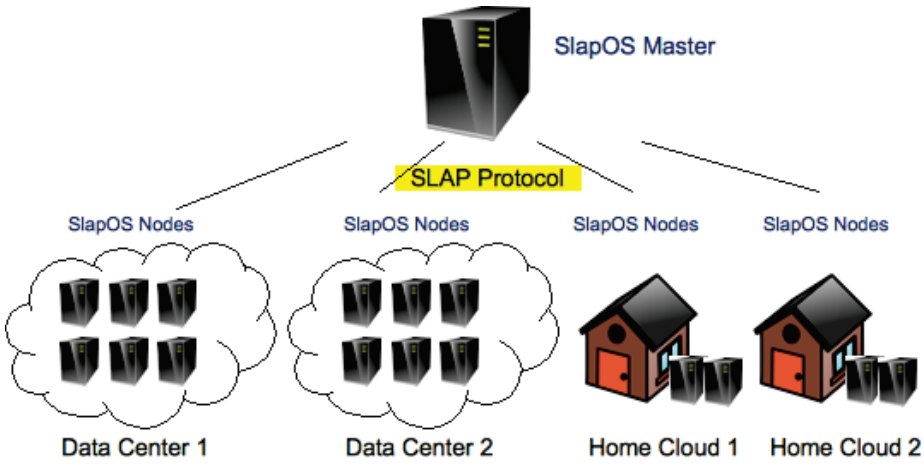


FIGURE *3.4 – Architecture de SlapOS.*

Le nœud maître appelé *SlapOS Master* ou *Master* a pour rôle l'allocation des processus c'est-à-dire. des applications, tandis que les nœuds esclaves appelés *SlapOS Node* ou *Node* ont pour rôle l'installation et l'exécution des services. Le Master constitue un annuaire centralisé de SlapOS Nodes, c'est lui qui possède les informations qui caractérisent chaque Node. Il possède aussi le catalogue des applications qui peuvent être installées dans la plateforme et il dispose de toutes les informations sur l'état des applications et services installés sur tous les nœuds. Ces informations sont utiles en particulier pour établir une facturation en fonction de l'usage d'un service.

Un Node, qui repose généralement sur une distribution Linux minimale, est constitué principalement d'un démon appelé SLAPGRID, d'un environnement de construction et d'amorçage des applications (à base de la technologie Buildout) et d'un autre démon de contrôle des services appelé Supervisord. Il échange des informations au format XML ou Json avec le master grâce au protocole *SLAP Protocol*. Le *SLAP Protocol* fonctionne a travers des connexions en HTTP/HTTPS. Il permet au Master de transmettre la liste des applications à construire sur un Node ainsi que la liste des services à déployer. Il permet aussi au Node de retourner des informations au Master, concernant l'état des ressources ou encore le statut des services.

3.5.2 Concepts clés et caractéristiques de SlapOS

Comme toute technologie avancée, SlapOS définit ses propres concepts qui le différencient d'autres plateformes de Cloud existantes et ces concepts constituent le modèle de fonctionnement du système.

Les computer partitions : la plupart des systèmes de Cloud existant aujourd'hui sont basés sur la virtualisation. Elle permet une bonne isolation des applications vis à vis du système hôte, ce qui donne la possibilité d'exécuter plusieurs applications identiques ou différentes sur une même machine, sans qu'il y ait des interactions (fuite d'information). La virtualisation des ressources physiques de la machine a toujours un coût supplémentaires sur les performances de l'application. Pour éliminer ce coût, SlapOS introduit un concept appelé *computer partitions* (Figure 3.5), qui est un conteneur léger fournissant un environnement d'exécution des applications. Ce conteneur léger n'est pas issue des technologies telles que *chroot* ou encore *LXC*, mais il s'agit tout simplement d'un dossier particulier associé à un utilisateur système qui a des droits particuliers ainsi que d'autres propriétés telles qu'une adresse IPv4, une adresse IPv6 et une interface réseau.

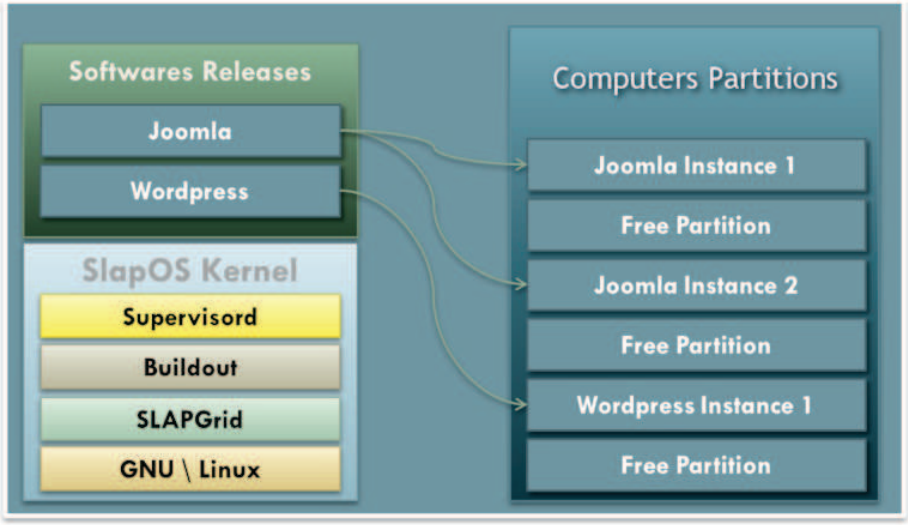


FIGURE *3.5 – Présentation d'un nœud SlapOS.*

Les Stacks et Buildout profile : l'un des modes de fonctionnement de SlapOS consiste à automatiser le déploiement des applications dans la plateforme. Pour cela, SlapOS utilise la technologie Buildout, qui est un système de compilation et d'assemblage d'applications, constitué de plusieurs morceaux. Le *buildout profile* est un fichier

contenant un ensemble de *parts* et de *recettes* permettant de décrire pour SlapOS, et dans la syntaxe de Buildout, comment une application sera assemblée et déployée. La *part*, référencée par son nom, est tout simplement un objet ou une application manipulée par Buildout.

Elle définit une recette qui décrit une logique de gestion et un ensemble de données qui seront utilisées pour la construire. La *recette* est un objet qui sait comment installer, mettre à jour ou désinstaller une part précise. L'ensemble des recettes SlapOS est disponible à l'adresse : http ://git.erp5.org/gitweb/slapos.git. « Cloudifier » une application revient, en grande partie, à écrire des parts et des recettes, une fois pour toute.

Afin de faciliter le déploiement des applications de même type dans plusieurs instances différentes tout en promouvant la réutilisation, SlapOS définit la notion de *stack* qui est un environnement construit par assemblage de composants ou d'applications permettant le déploiement d'une classe d'applications spécifique. Cette technique permettra la généralisation du déploiement des applications de même type, tout en facilitant la construction de leur buildout profile. La complexité de l'intégration est donc masquée dans la stack. Comme exemple, il existe la stack *resilient* qui permet d'ajouter la résilience à une application, ou encore la stack *lamp* qui permet d'intégrer à moindre effort une application Web basé sur Apache, MySQL et PHP.

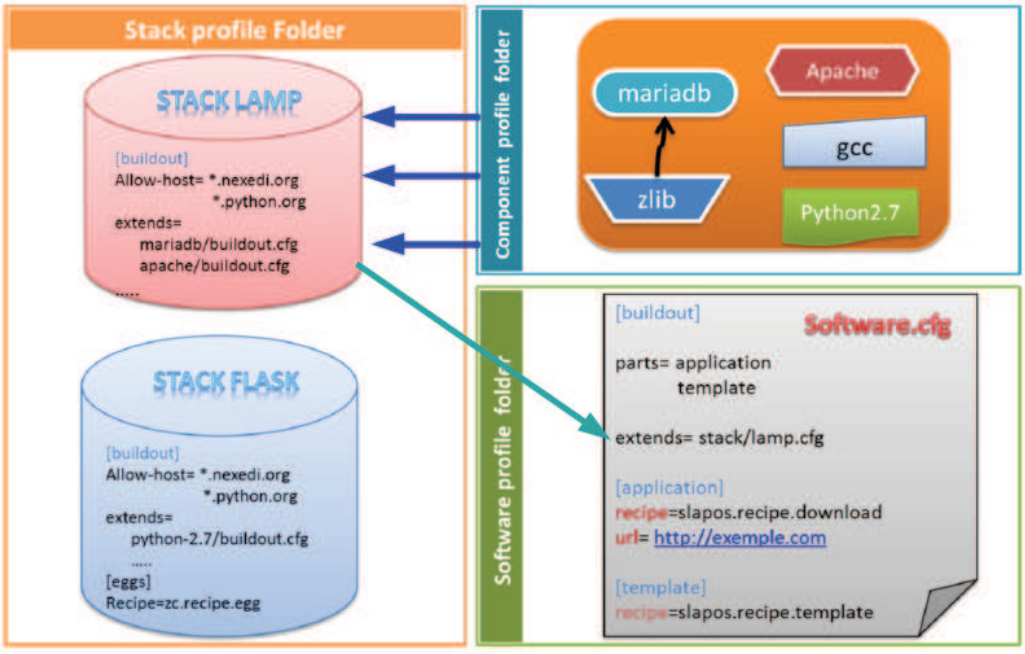


FIGURE *3.6 – Principe d'intégration des applications dans la plateforme SlapOS.*

Les Software Releases : dans SlapOS, un *software release* (voir la Figure 3.5) est construit à base du profile buildout de l'application. Il contient tous les binaires nécessaires pour exécuter l'application ; ces binaires sont obtenus généralement après compilation à partir du profile Buildout. Ainsi, SlapOS est capable de déployer plusieurs instances d'une application à partir d'un seul software release dans plusieurs partitions. Ces instances sont appelées *software instances*. Chaque software instance s'exécutent de manière indépendante vis à vis des autres.

Cette technique de déploiement permet, en théorie, un gain de performance et surtout un usage sobre des ressources d'un nœud. SlapOS peut alors tourner même sur des serveurs de faible performance. N'étant pas conçu sur les base de virtualisation, SlapOS est quand même capable de déployer des machines virtuelles, ceci par exemple grâce à un software release résilient dédié.

Réseaux et adresses IPv6 : SlapOS est conçu pour fonctionner de manière native avec IPv6. L'adresse IPv6 est l'adresse utilisée pour externaliser les services déployés dans les partitions. L'adresse IPv4 est locale à un Node, elle est le plus souvent utilisée lorsque deux services déployés dans des partitions différentes doivent communiquer entre elle. L'IPv6 permet le déploiement aisé de SlapOS, grâce notamment à l'autoconfiguration du réseaux.

L'avantage est de pouvoir disposer facilement d'adresses IP publiques accessibles dans le monde entier et en nombre illimité. Toutefois SlapOS peut également s'installer dans un environnement ne disposant que d'IPv4. Dans ce cas l'administrateur peut choisir d'utiliser un tunnel IPv4-IPv6 déployé en même temps que SlapOS. Une autre solution consiste à utiliser *re6st*, qui est un système de réseau d'overlay résilient conçu pour fournir une connectivité en IPv6 fiable et rapide aux entreprises. Ce réseau d'overlay a été conçu en liaison étroite avec SlapOS et il n'existait pas avant SlapOS.

3.5.3 Confinement d'exécution, sûreté de fonctionnement

La première idée de conception de SlapOS est de considérer que tout est processus ainsi, le système est basé sur la collection d'un ensemble de processus qui communiquent entre eux à base de services internet utilisant des protocoles de communications. Les processus sont surveillés par l'outil Supervisord qui permet de les relancer en cas de problèmes. Nous n'allons pas plus loin ici sur ces questions de tolérance aux pannes car cela est en dehors du cadre mais ces préoccupations sont abordées dans le projet Resilience.

SlapOS combine des hébergements non fiables, des serveurs dédiés à moindre coût et des Clouds à domicile pour atteindre une fiabilité de 99,999%. SlapOS est en réalité beaucoup plus fiable que les approches traditionnelles du Cloud puisque, en sélectionnant des sources indépendantes, il peut survivre en cas de problèmes de force majeure :

tremblement de terre, coupures d'électricité, etc.

3.5.4 Utilisation de SlapOs

SlapOS est un Cloud volontaire et décentralisé ce qui veut dire que tout le monde peut ajouter sa machine dans le Cloud avec SlapOS. Ce modèle qui est tiré des Grilles de PCs permet d'accroître potentiellement les performances du système en agrégeant des serveurs volontaires de petite capacité au moment des pics de charge par exemple. On peut donc traiter l'élasticité de cette manière avec SlapOS. L'utilisation du Cloud SlapOS peut donc se voir à deux niveaux :

- Utilisation des ressources SlapOS disponibles : l'utilisateur peut tout simplement vouloir acheter des ressources afin de déployer son services le plus rapidement possible. Dans ce cas, c'est le Master SlapOS qui choisit les ressources les mieux adaptées pour le service.
- Utilisation de ses propres ressources : dans ce cas, l'utilisateur ajoute des nœuds dans le Cloud qui, une fois connectés avec le Master SlapOS permettront de déployer ses propres services sur ses propres serveurs ou encore permettra à d'autres personnes de déployer leurs services sur ces serveurs.

Le déploiement d'un nouveau service sur SlapOS en utilisant son propre serveur peut se faire de manière simplifiée et seulement en quelques étapes. Pour cela, il convient de :

- Disposer d'un ordinateur avec un système d'exploitation Linux installé.
- Créer un compte sur un Master SlapOS. Le Master SlapOS de l'université Paris 13 est disponible à l'adresse https ://slapos.Cloud.univ-paris13.fr. Ce compte permet d'avoir un espace dans le Cloud et sera utilisé pour enregistrer des serveurs ou déployer des services.
- Créer un nouveau serveur virtuel, il permet de connecter le Master et le serveur physique et l'identifie de façon unique. Ainsi, le Master et le Node pourrons communiquer via le SLAP Protocol. Dans le terminal, il faudra taper la commande slapos node register COMPUTER_NAME [options]. L'enregistrement du nœud peut se faire aussi directement sur le site Web du Master.
- Installer le Node sur le serveur. L'installation peut se faire à l'aide du gestionnaire de paquets sur Linux, par exemple sur Ubuntu, il faudra taper la commande apt-get install slapos-node.
- Enfin, on peut soumettre les applications à compiler sur le serveur via le portail Web du Master par exemple ou encore via un terminal avec la commande slapos supply SOFTWARE_URL. Ici SOFTWARE_URL est l'URL vers le fichier Buildout principal de l'application, généralement c'est un fichier nommé software.cfg (voir la Figure 3.6).

3.6 Différences entre Grilles et Clouds

Les réseaux informatiques locaux et distants permettent de connecter des centaines et des milliers de machines entre elles. Les utilisateurs de ces réseaux ont la possibilité d'utiliser ces connexions pour réaliser des opérations de calcul, des exécutions de programmes et d'applications ou des stockages de données massives qu'une seule machine ne serait en capacité d'effectuer.

L'alternative qui consiste à acheter des supercalculateurs pour réaliser ces opérations n'est ni viable pour les organisations et les entreprises ni pour les particuliers. Cependant de ces besoins sont nées les grilles de calcul puis les Clouds qui, par définition, ont pour premier objectif de réduire le coût de calcul en utilisant des ressources provenant de machines connectées au réseau de l'utilisateur.

Nous croyons que les Clouds, comme Système, ne sont pas orthogonaux aux Grilles, et ils ne sont pas non plus complémentaires des Grilles : en quelque sorte, les Clouds sont l'évolution des Grilles (les deux systèmes à leur tour peuvent être considérés comme l'évolution des Clusters). Les Clouds peuvent être tout simplement constitués de Grilles ordinaires avec des services appropriés. Ainsi, le lien entre Grilles et Clouds provient de l'analogie de leurs structurations. Les Grilles de calcul tournent autour d'infrastructures qui partagent des ressources de calcul et de stockage, alors que les Clouds ciblent surtout l'offre de services et de ressources abstraites qui serviront à réaliser un modèle économique.

Grilles et Clouds doivent offrir des méthodes et des outils grâce auxquels les utilisateurs peuvent découvrir, invoquer et utiliser les ressources mises à leur disposition. Les Clouds ont immergé surtout suite à l'explosion des données massives et hautement distribuées que les utilisateurs et les machines ont besoin de manipuler sur les réseaux sans altérer la qualité des services invoqués.

Les Clouds sont clairement liés aux Grilles, à la fois dans les objectifs et la mise en œuvre. Dans notre cas, par exemple, nous ne voyons pas le Cloud comme une infrastructures pour les calculs hautes performances (HPC-*High Performance Computing*), mais le Cloud comme une infrastructure permettant de déployer des services, dont le service de calcul.

Comparer Grille et Cloud, revient à comparer la Grille de calcul avec un PaaS de Cloud Computing. Une Grille peut être considérée comme un Cloud privé délivrant les services d'un PaaS. Il y a cependant certaines différences entre la Grille et le Cloud Computing. Les grilles sont habituellement sur le site de l'utilisateur, et appartiennent à une organisation, alors que les Clouds sont normalement fournis par les fournisseurs et utilisés en fonction des besoins.

Les Grilles n'offrent pas la possibilité de fournir des serveurs individuellement. Elles ne permettent pas non plus l'auto-administration, y compris l'installation d'une variété de système d'exploitation et d'applications logicielles sur ces serveurs comme le

fait l'IaaS dans le Cloud. Les fournisseurs des Clouds facturent aux utilisateurs leurs consommations et leurs utilisations effectives des ressources et non pas un accès de durée limité ou illimité. Pour les Grilles de calcul, on paie l'accès à une Grille pour en pouvoir profiter. Les utilisateurs de Grilles s'organisent généralement en communautés autour d'un ou plusieurs projets durant lesquels ils se partagent des heures d'utilisation des ressources des grilles.

La nature d'utilisation et de gestion des ressources en Grille de calcul comme en Cloud diffèrent selon les objectifs de chacun. En effet, les grilles, de nature hétérogène et dynamique, doivent gérer différentes ressources ayant divers systèmes d'exploitation et politiques de sécurité. Ainsi, leur architecture se trouve dédiée pour résoudre des opérations de calcul à large échelle en se basant sur un réseau de partage de ressources. Le but est d'obtenir une puissance de calcul suffisante pour les tâches à exécuter. L'exécution des tâches dans une grille dépend d'un gestionnaire local qui coordonne l'affectation des tâches aux ressources (l'ordonnancement).

Cependant, la nature des opérations sur les Clouds dépend plutôt des protocoles et des technologies utilisés (les services Web, les flux, le Web dynamique, etc). Ces protocoles permettent aux Clouds d'utiliser des interfaces abstraites afin d'accéder au plus large spectre possible de ressources de stockage et de calcul existant. Par ailleurs, il reste possible que cela soit réalisé en se basant sur les grilles et leurs technologies qui n'arrêtent pas de progresser sur les domaines de la sécurité, de la virtualisation et de la gestion de ressources en général. En Cloud, on ne parle pas directement d'ordonnancement car les utilisateurs partagent les ressources en même temps et non de façon séquentielle, sans le savoir.

Comme le montre la figure 3.7, Ian Foster et al. ont présenté, dans [FZRL09], la relation entre le Cloud et les autres domaines avec lesquels il chevauche. Ainsi, les technologies du Web 2.0 couvre presque tout le spectre des applications orientées services, où le Cloud Computing se situe. Supercomputing et Cluster sont davantage axés sur les applications « non-services traditionnels ». Les systèmes de Grille chevauchent avec tous ces domaines, leur passage à l'échelle est moins considérable que celui des Supercomputer et des Clouds.

En résumé, Grille et Cloud présentent des similitudes : scalabilité, calcul et stockage à la demande. Ils ont aussi des différences : l'auto-approvisionnement immédiat, paiement à la consommation, et une grande variété d'applications disponibles via le Cloud. Pour synthétiser notre discussion, nous avons dressé le tableau récapitulatif 3.1 afin de lister un maximum de paramètres pouvant être l'objet de comparaisons entre Grille et Cloud Computing.

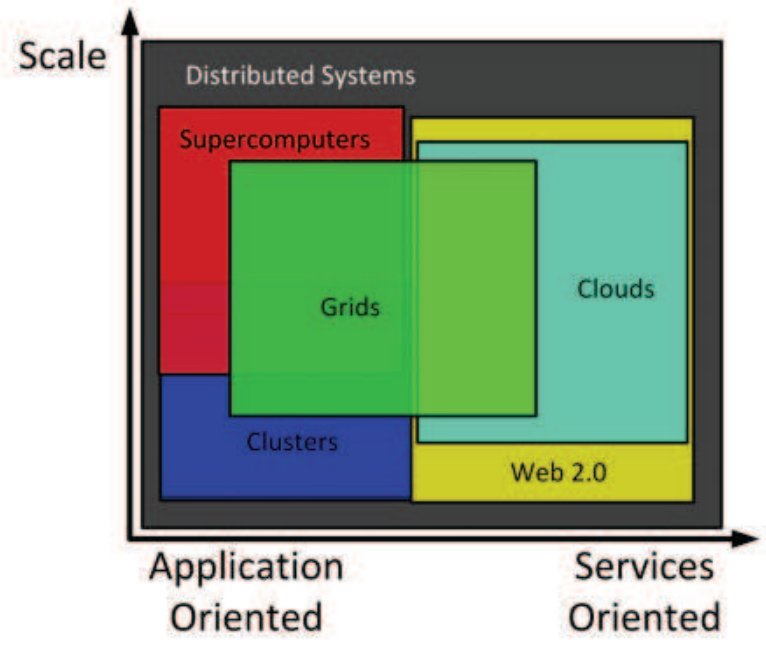


FIGURE *3.7 – Grilles et Cloud : Vue d'ensemble d'après [FZRL09]*

3.7 Conclusion

Dans ce chapitre nous avons présenté tous les éléments en relation avec notre domaine de recherche. Nous avons commencé par présenter globalement les systèmes distribués de calcul, nous nous sommes focalisés sur les Grilles de PCs en décrivant leurs caractéristiques. Nous avons mentionné les différents défis demandés par un intergiciel de Grille de PCs, tels que la volatilité des ressources, un environnement dynamique, palier au manque de fiabilité et à la panne des ressources, prendre en compte l'hétérogénéité, le passage à l'échelle et la participation volontaire. Puis, nous avons présenté le paradigme de publication-souscription, et le système BonjourGrid basé sur l'utilisation de ce paradigme.

Ensuite, nous avons présenté les systèmes de Cloud Computing et plus précisément le Cloud SlapOS qui constitue un élément de base de nos travaux. Dans les chapitres qui viennent, nous montrons comment nous avons contribué à ces concepts.

Critère	**Grid Computing**	**Cloud Computing**
Objectif	Partage collaboratif des ressources	Utilisation des services (niveau d'abstraction élevé)
Axes de calcul	Opérations de calcul intensif	Instances standards et de haut niveau
Gestion du Wokflow	Sur un nœud physique	En instance EC2 (Amazon EC2 + S3)
Réseau d'interconnexion	Partiellement Internet avec une latence et une faible bande passante	Dédié, haut de gamme avec une faible latence et une large bande passante
Découverte	Indexation centralisée et des services d'information décentralisés	Services aux membres
La gestion des ressources	Distribuée	centralisée/distribuée
Allocation/Ordonnancement	Décentralisé	centralisé/décentralisé
Interopérabilité	Normes standards de grille	Web Services (SOAP et REST)
Niveau d'abstraction	Faible(plus de détails)	Haut(éliminer les détails)
Degré de scalabilité	Normale	Haute
Multitâche	Oui	Oui
Transparence	Bas	Haut
Temp d'exécution	N'est pas en temps réel	Services en temps réel
Unité d'allocation	Petite tâche	Toutes formes et tailles
Virtualisation	N'est pas une matière première	Vitale
Portail accessible	Via un système DNS	Via IP uniquement
Transmission	Subi les retards d'Internet	Considérablement rapide
Sécurité	Faible (service de certificat de grille)	Haute (virtualisation)
Infrastructure	Commande de bas niveau	Services de haut niveau (SaaS)
Système d'exploitation	N'importe quel standards	Un hyperviseur (VM) sur lequel peut tourner plusieurs systèmes
Propriétaire	Multiple	Unique
Tolérance aux pannes	Limité (échec de tâche/ Re-éxecution d'applications)	Forte(Les VMs peuvent migrer aisément d'un nœud à l'autre)
Prix des services	Dominé par le bien public ou privé	Prix des services publics, prix réduit pour les gros clients
Facilité d'utilisation	Faible	Elevée
Type de service	CPU, réseau, mémoire, bande passante, dispositif, stockage,...	IaaS, PaaS, SaaS, tout est service
Stockage intensif des données	Convient	Convient
Exemple du monde réel	BOINC, XtremWeb, SETI,...	Amazon Web Services (AWS), Google Apps,...
Temps de réponse	ne peut pas être déterminé	En temps réel
Objet critique	Ressources informatique	Services
Nombre d'utilisateurs	peu	plus
Ressources	Limité (car le matériel est limité)	illimité
Configuration	Difficile car les utilisateurs n'ont pas des privilèges d'administrateur	Très facile à configurer
Future	Cloud Computing	Prochaine génération d'Internet Smart cities peut être,...

TABLE *3.1 – Tableau comparatif*

Deuxième partie

Contributions

Chapitre 4

Modélisation formelle de protocoles utilisant le paradigme Publication-Souscription

4.1 Introduction

Du fait de la complexité de plus en plus forte des systèmes informatiques, il apparaît nécessaire de disposer de méthodes et d'outils de conception et/ou de réalisation qui soient perfectionnés pour capter et traiter les défis actuels, notamment ceux liés à la très grande échelle et l'usage d'Internet comme médium de communication. Au centre de ces méthodes et de ces outils, se trouve la modélisation.

Les réseaux de Petri ont été développés pour permettre la modélisation de classes importantes de systèmes. Ceci recouvre des classes de systèmes de production, de systèmes automatisés, de systèmes informatiques et de systèmes de communication, afin de permettre leur conception, leur évaluation et leur amélioration.

Dans ce chapitre, nous détaillons les étapes suivies pour la modélisation du protocole de publication-souscription ; de la réflexion initiale à la réflexion finale. Nous exposons une étude de cas, qui n'est autre que la modélisation formelle du méta-intergiciel BonjourGrid puisqu'il est construit autour du paradigme de publication-souscription. Nous expliquons comment cette modélisation formelle a joué un rôle très important pour fixer notre choix de protocole de communication et comment on a pu établir une méthodologie de conception autour du paradigme de publication-souscription.

Nous terminons par étudier Redis, un système de gestion de base de données clef-valeur scalable, ses avantages et ses inconvénients, tout en le comparant avec d'autres technologies du Web. Une étude des performances de Redis sera aussi présentée. Suite à l'étude de Redis, nous présentons une discussion autour de la modélisation du paradigme

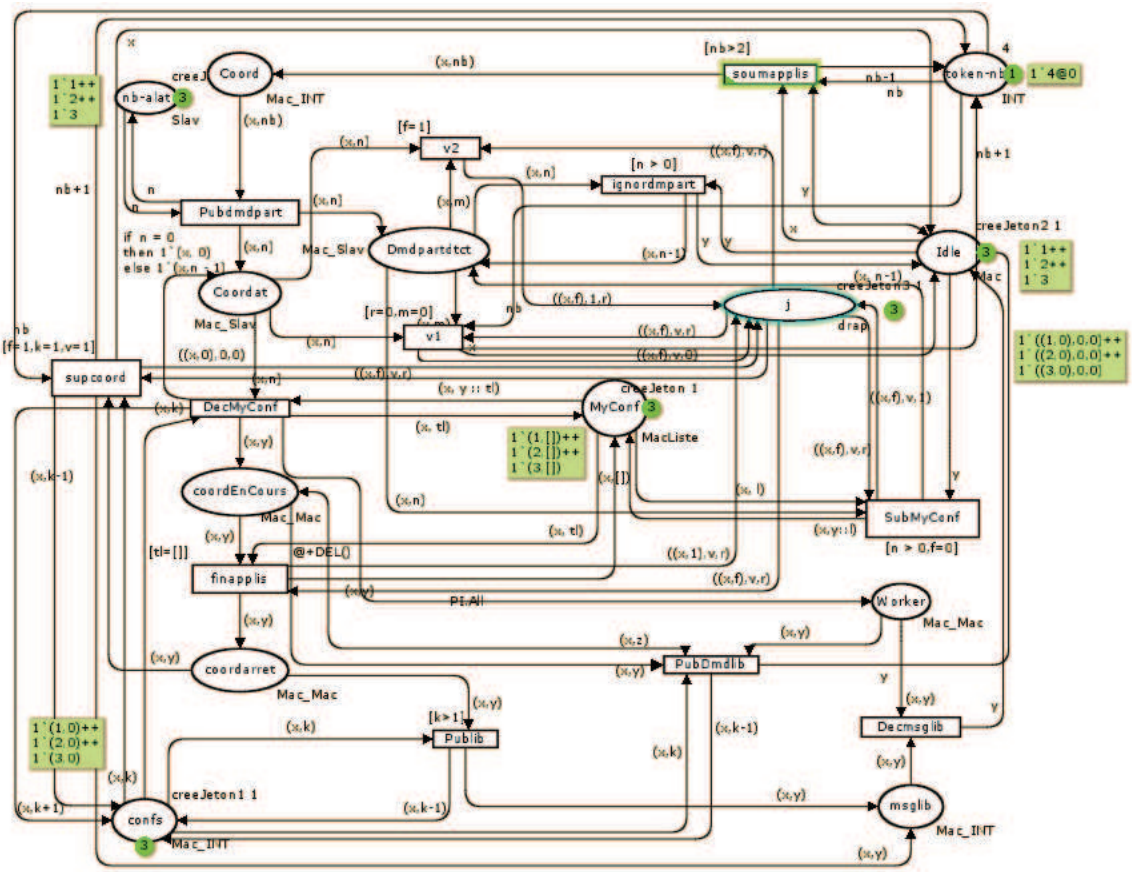


FIGURE *4.1 – Première modélisation de BonjourGrid*

de publication souscription.

4.2 Idées initiales

Le point de départ de notre travail était de vérifier formellement le système BonjourGrid. Rappelons que BonjourGrid a été développé sans l'appui de méthodes formelles ou semi-formelles. Ainsi, le but était de s'assurer du bon fonctionnement du système. Cet objectif nécessitait la validation de certaines propriétés essentielles telles que les propriétés d'accessibilité, sûreté, vivacité, absence de blocage et équité.

Le premier modèle de BonjourGrid (voir figure 4.1) a été conçu en suivant la méthode avec laquelle le système a été développé. Comme cela est en usage, les choix de conception effectués ne s'appuient pas sur une méthode formelle mais restent pour une grande partie liée à l'intuition, à l'idée que l'on se fait.

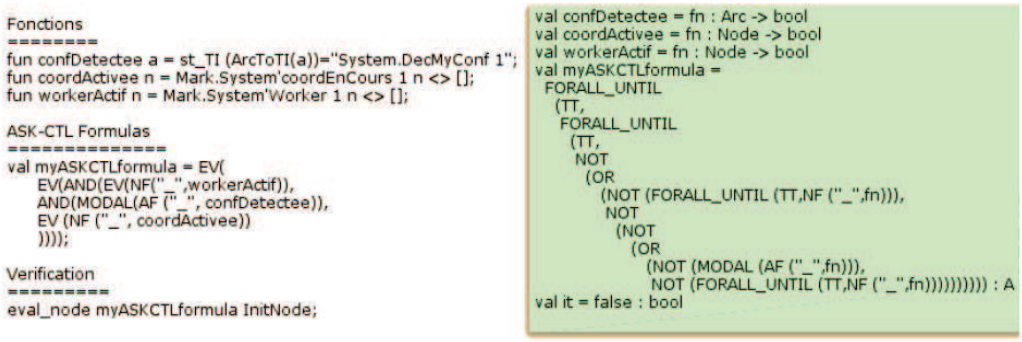


FIGURE *4.2 – Evaluation d'une formule de logique temporelle CTL*

Lors de la validation formelle nous avons rencontré des difficultés pour la vérification de certaines propriétés avec les outils de vérification formelle de CPN Tools. Ces outils visent à évaluer et valider des expressions écrites dans le langage de programmation fonctionnel Standard ML et représentant les propriétés d'accessibilité, de vivacité, etc. La plupart de ces évaluations ont été élaborée en utilisant la librairie formelle ASK-CTL que nous avons installée au sein de CPN Tools spécialement pour cette tâche. Des évaluations ont été faites sur des formules exprimant les propriétés formelles à satisfaire. Certains résultats ne correspondent pas à ce qui était attendu.

La Figure 4.2 illustre l'une des plus importante formule d'évaluation. En effet, dans le système BonjourGrid, si on a un Worker actif alors on a forcément un Coordinateur auquel ce Worker est attaché. Autrement dit, s'il existe un Worker alors il existe au moins un Coordinateur. Cette propriété est écrite en utilisant Standard ML, et on attend que la formule retourne la valeur booléenne `(True)`, mais comme le montre la Figure 4.2, la valeur `(False)` est retournée avec notre modèle.

La non-validation de certaines propriétés nous a amenée à la conclusion suivante :

- soit nous avons réalisé une mauvaise modélisation du système BonjourGrid ;
- soit nous avons bien modélisé notre système mais c'est le protocole de communication qui n'est pas correct ;
- soit le type de modélisation choisi ne correspond pas au système BonjourGrid, qui s'avère très complexe à modéliser.

En fait, la dernière explication est plus apte à être retenue. Nous avons dû, en effet, introduire et faire des choix de modélisation qui ne coïncident pas avec les choix d'implémentation. Rappelons que nous faisons une vérification «à posteriori» du protocole.

Nous aurions pu retravailler la modélisation pour lisser ces problèmes mais nous avons préféré partir sur une autre piste. Comme dans notre domaine des intergiciels de grilles de PCs nous n'avons pas l'habitude de faire de la modélisation formelle, nous

n'avions pas au départ d'idées sur les «bonnes» questions à se poser. Au fil du temps les choses se sont précisées en ces termes :

- Doit-on se focaliser sur les changements d'états du système ou sur les interactions entre les différents acteurs du système de Grille ?
- Faut-il aussi suivre son intuition pour la modélisation, comme on l'a déjà fait par le passé pour le développement ?
- Quelles méthodes adoptées pour garantir un système extensible ?

Il y avait plusieurs pistes possibles pour entamer la modélisation. Nous sommes partis avec les idées d'offrir à la fois un modèle clair, extensible (pour des éventuelles évolutions du système) et surtout bien positionné par rapport au point fort de BonjourGrid qui est la décentralisation de la coordination. Il est évident qu'il faut revoir notre centre d'intérêt initial.

De ce fait, nous nous sommes intéressés essentiellement à la construction d'un élément de calcul de BonjourGrid et plus précisément aux interactions entre les différents acteurs d'un même élément de calcul.

D'un point de vue concret, ces différentes interactions sont gérées par le protocole de coordination qui est en fait le protocole *Bonjour* d'Apple [BON]. Le protocole Bonjour est une implémentation du protocole ZeroConf (*Zero Configuration Network*). Il est essentiellement basé sur le paradigme de publication-souscription. D'où l'idée de commencer par modéliser le protocole de publication-souscription. Ainsi, nous abordons la problématique avec plus d'abstraction afin de pouvoir faciliter le contrôle de la modélisation et de la vérification par la suite.

4.3 Modélisation formelle du paradigme de publication-souscription

La figure 4.3 présente une modélisation formelle du protocole de publication souscription en utilisant les réseaux de Petri colorés par dessus l'outil CPNTools. Il présente un état initial avec 10 composants et 10 événements. Chaque composant peut être fournisseur (*publisher*) d'un événement (EP pour *EventPublished*), un consommateur (*subscriber*) à un événement (ES pour *EventSubscribed*) ou les deux à la fois.

Cette modélisation, qui est le fruit de notre travail, est fidèle au protocole de publication-souscription puisque un composant peut publier un événement autant de fois qu'il le veut, il peut aussi souscrire à un événement autant de fois qu'il le veut. Un événement peut être publié par un ou plusieurs composants. Bien évidemment un ou plusieurs composants peuvent souscrire à cet événement. Les événements publiés sont sauvegardés dans un annuaire qui est modélisé ici par l'état *Registry*. Un composant souscrit à un événement E, il passe à l'état souscripteur en attente (*WaitingSubscriber*).

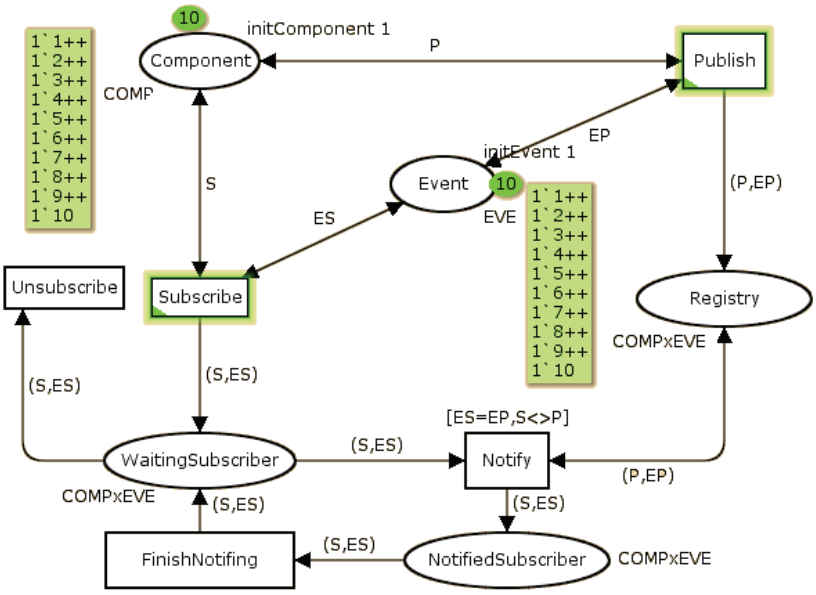


FIGURE *4.3 – Modélisation du protocole de publication-souscription*

Une fois que cet événement est publié, la transition *Notify* sera franchissable. Une condition doit être vérifiée lors du franchissement de la transition *Notify* : «un même composant ne peut pas souscrire à l'événement qu'il a publié», ce qui a été modélisé en utilisant la garde $[S \neq P]$. Bien que cette modélisation montre les différentes facettes du protocole, elle reste tout de même inexploitable puisqu'elle ne présente pas d'état final. Ceci veut dire qu'en cas d'exploration de l'espace d'états, nous aurons une infinité de branchements.

La figure 4.4 présente une partie du rapport généré par CPN Tools et il indique le

```
6  Statistics
7  ------------------------
8
9    State Space
10      Nodes:  27464
11      Arcs:   81911
12      Secs:   300
13      Status: Partial
14
```

FIGURE *4.4 – Extrait du rapport d'analyse de l'espace d'états du modèle du protocole de publication-souscription*

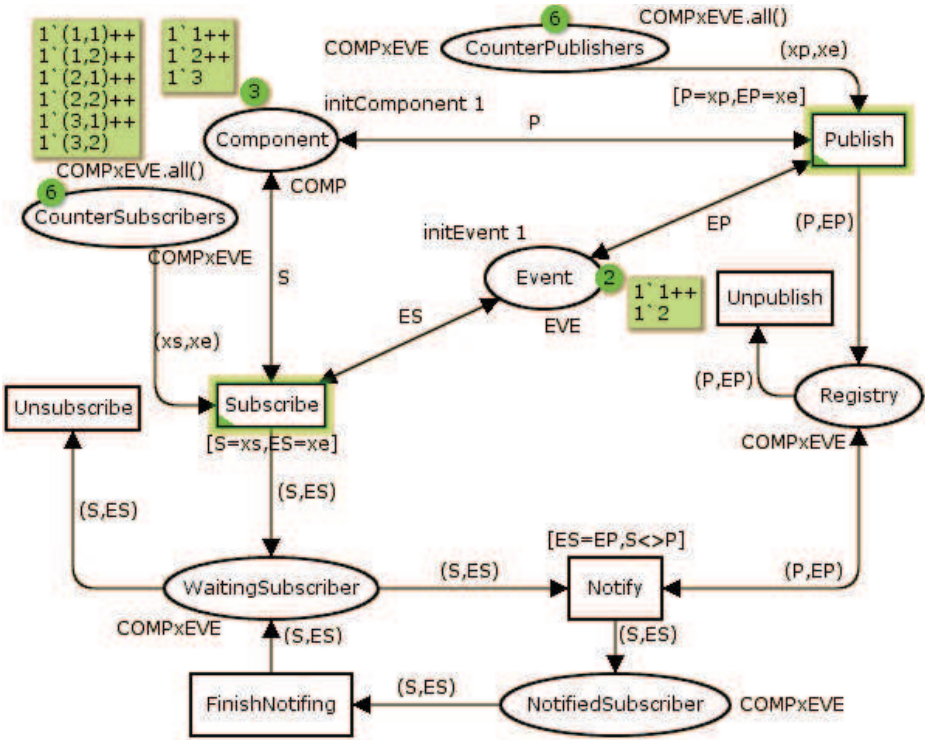


FIGURE *4.5 – Modélisation du protocole de publication-souscription adapté*

nombre de nœuds du réseau, le nombre d'arcs et le temps de calcul. Nous remarquons qu'il y a seulement une partie des arcs qui est générée (et pas tout l'arbre). En effet, les actions *Publish* et *Subscribe* peuvent se produire infiniment. D'où le problème de l'explosion de l'espace d'états. Sur un tel modèle, nous ne pouvons pas faire des analyses ou des déductions. Par conséquent, nous avons fait quelques changements au niveau du réseau de Petri (voir figure 4.5).

Premièrement, nous avons réduit le nombre d'événements et de composants de 10 à 2. Deuxièmement, nous supposons que :

– un composant c ne peut publier un événement e qu'une seule fois ;

– un composant c ne peut souscrire à un événement e qu'une seule fois.

La figure 4.5 prend en considération les changements que nous avons apporté au modèle. On voit bien la différence entre le rapport de la figure 4.4 et celui de la figure 4.6. En fait, sur le modèle adapté, l'espace d'états est fini. des propriétés génériques sont vérifiées, telles que la propriété d'Home et de vivacité. Le rapport nous indique que le système ne présente pas de marquage mortel et que tous les marquages sont des home marquages.

```
 State Space
    Nodes:11296  Arcs:38044
    Secs:22      Status:Full
 Scc Graph
    Nodes:1  Arcs:0  Secs:1
Home Properties
--------------------------------
 Home Markings  All
Liveness Properties
--------------------------------
 Dead Markings  None
 Dead Transition Instances  None
 Live Transition Instances  All
```

FIGURE *4.6 – Extrait du rapport d'analyse de l'espace d'états du modèle du protocole de publication-souscription adapté*

4.4 Modélisation formelle d'une grille de PCs : BonjourGrid

Dans cette section, nous présentons les points clés et les étapes suivies lors de la modélisation de le méta-intergiciel BonjourGrid. L'idée principale est de modéliser BonjourGrid autour de la modélisation formelle du paradigme de publication-souscription. Pour ce faire, nous procédons par étapes.

Nous considérons que la modélisation du protocole de publication-souscription est une boite noire. Nous commençons par y greffer tout ce qui est spécifique à BonjourGrid autour de cette boite comme étant un événement extérieur, de façon à ce que tout le protocole de coordination soit basé sur l'annonce d'événements. Nous commençons par rajouter les fonctionnalités fondamentales de BonjourGrid, telles que l'ajout d'une application à soumettre ou la notion de machine au repos, sans rentrer trop dans les détails. Ensuite, nous utilisons la méthode de modélisation par raffinement.

Le raffinement consiste à définir un système avec un système racine et un ensemble de transformations qui sont soit des enrichissements, soit des remplacements d'une partie par un système. Cette méthode est itérative. Dans notre cas, elle consiste à rajouter des détails fonctionnels du système à chaque itération. Nous nous focalisons dans cette section sur cet aspect itératif afin d'observer l'évolution de notre modélisation formelle.

4.4.1 Modélisation de BonjourGrid : première version

La figure 4.7 montre une modélisation simplifiée du protocole BonjourGrid qui ne couvre pas tous les aspects techniques mais qui se base sur le paradigme de publication-souscription comme noyau central.

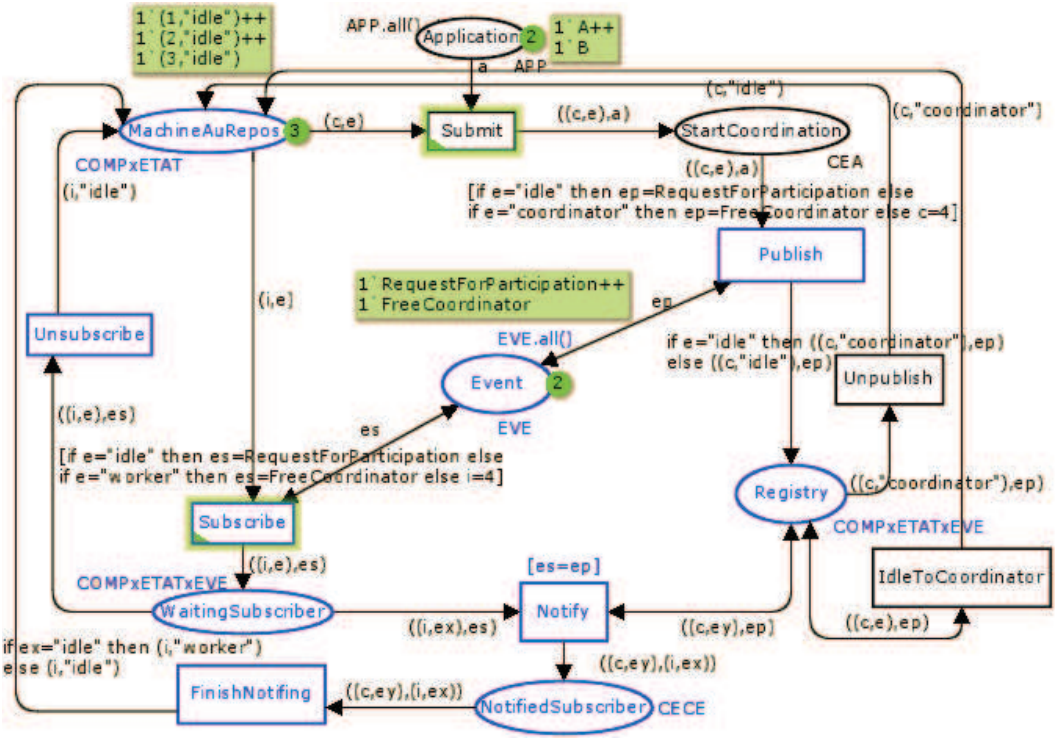


FIGURE *4.7 – Modélisation simplifiée de BonjourGrid*

Les états principaux de BonjourGrid sont : *Idle, Worker*, *RequestForParticipation*, et *FreeCoordinator*. Le réseau de Petri de publication-souscription définit trois connecteurs : *Publish*, *Subscribe* et *Notify*. Le premier connecteur, *Publish*, permet de publier un événement *ep* par une machine *c*. Le deuxième connecteur, *Subscribe*, permet à une machine *c* de souscrire à un événement *es* et le troisième, *Notify*, sert à notifier les souscripteurs par les événements correspondant à leurs souscriptions. Ces trois connecteurs, paramétrés par la paire (*event,machine*), sont représentés par les places *Component* et *Registry* dans la figure 4.7.

À ce stade de modélisation, le défi est de considérer la modélisation du protocole de publication-souscription comme étant une boite noire (ce qui est présenté par la boucle *Publish-Subscribe-Notify* dans la figure 4.7). Nous avons par la suite connecté le protocole BonjourGrid par le protocole de publication-souscription qui a déjà été présenté par le cycle *publication*, *subscription* et *notification*. Les éléments propres à

Participating nodes	States	Arcs	Time (sec.)
2	35	108	0
3	215	1,005	1
4	1,295	8,204	2
5	7,775	62,635	45
6	46,655	458,778	5,956
7	$>$ 300,000	$>$ 3,000,000	$>$ 18,000

TABLE *4.1 – Le rapport de l'espace d'états pour plusieurs configurations*

BonjourGrid sont rattachés à ce cycle sous forme d'entrée/sortie (*inptus/outpus*). En réussissant à appliquer cette idée, nous avons produit une modélisation générique.

Au début du cycle, un composant peut soumettre une application et par la suite publier l'événement *RequestForParticipation* associé à cette application. En même temps, un autre composant peut souscrire à cet événement. Quand les souscripteurs seront notifiés par les événements correspondant, une coordination commence. En ce moment, les machines ayant l'état *idle* passent à l'état *worker*. Les machines ayant l'état *worker* peuvent se souscrire à l'événement *FreeCoordinator* qui permettra de les libérer. En fait, quand l'exécution de l'application se termine, la machine ayant l'état *Coordinator* publie l'événement *FreeCoordinator* par lequel les *Workers* sont notifiés. Ainsi, ils sont relâchés et retournent à l'état *Idle*.

En utilisant l'outil CPNTools nous commençons d'abord par réaliser des simulations manuelles afin d'observer le comportement du système. Durant cette étape nous n'avons découvert aucun problème. Ensuite, nous avons réalisé une simulation plus exhaustive qui permet d'explorer tous les états possibles du système, i.e. tout l'espace d'états. Comme notre modèle est flexible concernant le nombre de participants au système nous avons analysé plusieurs configurations en variant chaque fois le nombre de participants.

Le tableau 4.1 présente une partie de rapport de l'espace d'états retourné par CPN-Tools : le nombre d'états accessibles dans l'espace d'états, le nombre d'arcs dans l'espace d'état et le temps pris par CPNTools pour construire l'espace d'états. Le temps de recherche a été limitée à 18.000 secondes, soit 5 heures. Par conséquent, les données rapportés pour 7 participants sont incomplètes et ne représentent qu'un rapport approché de la réalité. Notre modèle est naturellement soumis au problème de l'explosion de l'espace d'états du fait que le nombre d'états accessibles croît de façon exponentielle par rapport au nombre de participants.

Cependant, l'analyse de ces configurations nous a donné plus de confiance dans le système BonjourGrid compte tenu des faits suivants :

– Nous n'avons pas trouvé d'états de blocage (i.e., des états qui n'admettent pas des transitions exécutables). L'absence de tels états est évidemment nécessaire

dans notre contexte.

- Toutes les transitions possibles sont exécutables. Par conséquent, notre spécification semble être correcte de point de vue événement déclencheur : tous les événements possibles peuvent éventuellement avoir lieu.
- L'espace d'états construit est composé d'une seule composante fortement connexe. Il semble donc impossible d'être pris au piège dans une configuration spécifique ou indésirable du système, autrement dit, tomber dans un état puits. Cette propriété de vivacité est en effet une condition essentielle pour notre système.

4.4.2 Modélisation de BonjourGrid : deuxième version

Nous présentons dans cette partie, une nouvelle version de la modélisation de BonjourGrid. Dans cette version, telle qu'elle est présentée dans la figure 4.8, le principe est le même, garder au centre la brique du protocole de publication souscription et construire tout autour le modèle BonjourGrid. Nous avons pris en considération tous les détails techniques liés au fonctionnement du système, d'une part afin d'avoir une simulation la plus proche possible du comportement actuel du système, et d'autre part afin de pouvoir rechercher des comportements déviants de la spécification.

Nous procédons par étapes de raffinement sur le premier modèle construit (voir figure 4.7). La modalité principale est d'avoir une vision globale du comportement global d'un tel système où l'asynchronisme est le critère principal. Dans l'architecture de publication-souscription, les composants communiquent par échange d'événements. Ainsi, n'importe quel modèle de cette architecture doit prendre en considération deux principaux acteurs : les composants (*components*) et les événements (*events*), et trois principaux services : publication, souscription et notification.

Comme présenté sur le modèle de la figure 4.8, le premier service sert à publier un événement *ep* par une machine ou un composant *c*, le deuxième service sert à se souscrire à un événement *es*, et le troisième pour notifier les souscripteurs par la publications des événements correspondants à leurs souscriptions. La boite noire évoquée précédemment est représentée par la brique bleue sur la figure 4.8. Le changement par rapport à l'ancien modèle se situe au niveau de l'état *Event* qu'on explose en deux états : *EventS* et *EventP* dans le but de différencier les événements publiés et ceux aux quels on peut s'abonner. Cette différenciation nous permet d'éviter l'ambiguïté dans le modèle et aussi d'avoir plus de clarté et de simplicité.

Nous représentons chaque composant dans la place *component* par le triplet *(compId,state,coorId)* où *compId* est l'identifiant d'un composant, *state* peut avoir les valeurs : *"idle"*, *"worker"*, *"coordinator"* et représente les états que peut prendre un composant, *coorId* est l'identifiant d'un coordinateur au quel un composant ayant l'état *"worker"* peut être attaché, *coorId* prend la valeur 0 quand le composant n'est pas un Worker. Un composant/machine peut soumettre une application et publier l'événement

"RequestForParticipation" associé à cette application. La place *Application* est représentée par le couple *(a,nb)* où *a* est l'identifiant de l'application et *nb* est le nombre de machines nécessaires à l'exécution de cette application.

En parallèle, un autre composant ayant l'état *"idle"* peut souscrire à l'événement *"RequestForParticipation"*. Le processus de coordination commence quand toutes les machines souscrites seraient notifiées par l'événement correspondant et acceptent de participer à l'exécution de l'application. Tous les souscripteurs à cet événement sont notifiés par sa publication, mais seulement le nombre *nb* requis par l'application est autorisé à participer à l'exécution. L'exécution peut se faire avec un nombre $n \leq nb$. Ainsi, *nb* représente le nombre maximal de participants à impliquer dans l'exécution de l'application *a*. La place *"counter"* a pour rôle de satisfaire cette contrainte. À ce stade, les états des machines ayant acceptées la participation changent de *"idle"* à *"worker"*, et l'exécution de l'application peut commencer.

Une fois l'exécution terminée, on passe à l'étape de la libération des *"workers"*. En fait, une machine ayant l'état *"worker"* peut souscrire à l'événement *"FreeCoordinator"*. Quand l'exécution d'une application est terminée, le coordinateur publie l'événement *"FreeCoordinator"* par lequel les *"workers"* souscris sont notifiés. Ainsi, les *"workers"* esclaves sont libérés et leurs états changent à *"idle"*. À cet instant, d'autres applications peuvent être soumises, en attente ou en cours d'exécution. Le cycle ne s'arrête pas tant que les conditions de fonctionnement du système sont satisfaites.

Sur ce modèle de BonjourGrid, nous avons réalisé des simulations pertinentes à l'aide des outils fournis par CPNTools. Ces simulations permettent d'explorer tout l'espace d'états du système. Durant cette étape aucun problème n'a été décelé.

Sur ce Réseau de Petri coloré, nous avons pu vérifier certaines propriétés fondamentales en utilisant la logique CTL. On aurait pu s'arrêter à la simulation fournie par CPNTools et aux propriétés générales (vivacité, borné, home, fairness,...) présentées sur le rapport de la figure 4.9, mais nous avons préféré aller plus loin dans la vérification et solliciter le réseau de Petri pour des propriétés spécifiques au fonctionnement de BonjourGrid. Les principales propriétés sont :

- Un événement produit/publié doit être reçu par tous les consommateurs/souscripteurs intéressés par cet événement.
- Un coordinateur C peut commencer l'exécution de son application, si et seulement si il existe au moins une machine M acceptant de participer à l'exécution.
- Si un coordinateur finit l'exécution de son application alors tous les *workers* associés doivent être libérés.
- Un *worker* ne peut être attaché qu'à un seul coordinateur à la fois.

Ainsi, l'analyse du rapport de l'espace d'états retourné par CPNTools et les résultats observés des propriétés CTL, prouvent que notre système est correct. Il ne présente pas

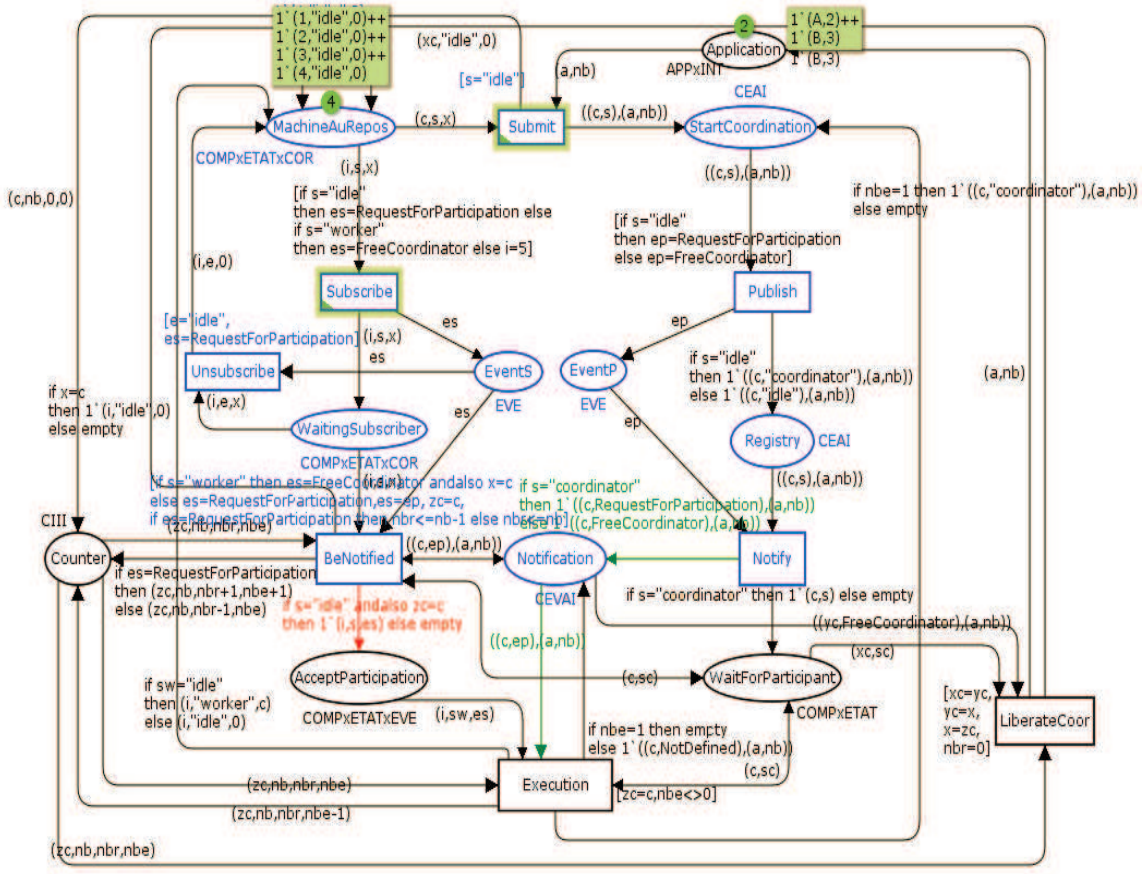


FIGURE *4.8 – Modélisation complète de BonjourGrid*

d'états bloquants. Tous les événements possibles peuvent éventuellement se produire. La propriété de vivacité est bien vérifiée.

La réalisation d'une telle vérification formelle sur un système de grille de calcul est une innovation en elle-même.

4.5 Évolution de BonjourGrid

4.5.1 Motivations pour une évolution

La grille de PCs est l'une des réussites de ces dernières années à cause du bénévolat des nœuds participants dans les projets. Avec l'émergence du Cloud Computing, de nouvelles questions se posent : *où prendre les ressources ?* et *comment les coordonner ?*.

```
Statistics
----------------------------------------------------------------------
  State Space
     Nodes:  11296
     Arcs:   38044
     Secs:   22
     Status: Full
  Scc Graph
     Nodes:  1
     Arcs:   0
     Secs:   1
Home Properties
----------------------------------------------------------------------
  Home Markings
     All
Liveness Properties
----------------------------------------------------------------------
  Dead Markings
     None
  Dead Transition Instances
     None
  Live Transition Instances
     All
```

FIGURE *4.9 – Extrait du rapport d'analyse de l'espace d'états du modèle de BonjourGrid*

Nous croyons que les Desktop Grids continueront de survivre si nous sommes capables de transformer l'ancienne architecture client/serveur en nouvelle architecture orientée Web afin de fournir des services à la demande, donc de les intégrer dans des infrastructures de Cloud.

Il existe également de nombreuses applications scientifiques et économiques qui traitent avec une énorme quantité de données. Afin de traiter de grands jeux de données, ces applications ont généralement besoin d'une infrastructure de calcul haute performance. Toutefois, étant donné que les ressources d'une grille de PCs sont généralement accessibles via les grands réseaux, notamment Internet, le goulot d'étranglement suit la limitation de la bande passante. La question est d'imaginer des architectures capables de masquer (en partie) la limitation de la bande passante. Dans notre cas, nous choisissons un découplage entre la source (les données) et le calcul par la mise en place d'un cache distant et un cache local.

D'après Fedak et al. dans [CF12], dans l'environnement des grilles de PCs, les tâches basiques de gestion des données tel que le stockage fiable de grands volumes de données sont très difficiles à accomplir, premièrement à cause de la volatilité des nœuds. Deuxièmement, la vie privée et la sécurité des données doivent être appliquées sur les grilles de PCs parce que nous traitons avec des ordinateurs non fiables. Le mécanisme de protection des données peut ajouter des frais non négligeable lors du traitement de grands volumes de données.

Troisièmement, étant donné que les ressources sont réparties géographiquement, la conception d'une solution scalable permettant de gérer de grandes masses de données est un problème. La dernière version de BonjourGrid se concentre justement sur ce

dernier problème.

Depuis sa finalisation en 2010, le méta-intergiciel BonjourGrid n'a pas arrêté d'évoluer tout en prenant en considération les évolutions technologiques et de nouveaux besoins. Dans cette section, nous nous focalisons sur la modélisation formelle de l'un des aspect d'évolution de BonjourGrid afin de montrer que notre modèle de départ est flexible et extensible. Mais aussi que notre conception de la modélisation est efficace dans le sens où on peut toujours étendre le modèle en changeant l'angle de vue des abstractions.

En fait, au début notre boite noire était le modèle de la publication-souscription par dessus lequel on a ajouté tous ce qui est spécifique à BonjourGrid, et à ce niveau, l'angle d'abstraction change et notre boite noire est la modélisation de BonjourGrid par dessus laquelle on rajoute tous ce qui est spécifique à son évolution.

Comme expliqué par W.Saad et al. dans [SACJ12], l'idée principale consiste à créer dynamiquement et à la demande, pour chaque application soumise par un utilisateur, un gestionnaire de données *(DataManager)* en plus du système de calcul déjà mis en œuvre. La solution proposée dans [SACJ12] supporte les applications existante sans obligation de modifier leurs implémentations. Le gestionnaire de données est lancé et supervisé par le nœud coordinateur de BonjourGrid en parallèle avec l'intergiciel de calcul choisi (BOINC, XtremWeb ou Condor).

Ainsi, BonjourGrid devient une grille de méta-données qui orchestre simultanément plusieurs instances de gestionnaires de données et de systèmes de calcul. Pour réaliser cette approche, les auteurs composent la plateforme de gestion de données avec deux caches qui interagissent continuellement afin de maintenir la disponibilité des données pendant toute l'exécution d'une application. Chaque cache offre un ensemble de services. Le premier cache est un cache distant pour les activités de placement de données, le but est de réserver l'espace disque pour l'application et de transférer les données nécessaires à partir du site de l'utilisateur à la plateforme BonjourGrid.

Une fois que les données sont placées sur la plateforme, un deuxième cache appelé cache local est automatiquement lancé afin de publier et diffuser les données aux workers. Une autre évolution apporté à BonjourGrid consiste à ne plus utiliser le protocole Bonjour d'Apple mais à utiliser le protocole Redis [RED] pour implémenter le paradigme de publication-souscription.

Notre modélisation formelle initiale de BonjourGrid a été mise au point afin de prendre en considération cette évolution. La figure 4.10 présente le réseau de Petri coloré modélisant cette nouvelle version de BonjourGrid. Comme on utilise Redis [RED] dans la nouvelle version, on a été amené à apporter des modifications sur la brique centrale de publication-souscription. En effet, le paradigme de publication-souscription n'est pas implémenté de la même façon sur Redis et sur le protocole Bonjour. Sur Redis, si un événement est publié et qu'il n'y a pas de souscripteur au préalable pour cet événement, alors l'événement est perdu, ce qui n'est pas le cas pour le protocole Bonjour. On a dû

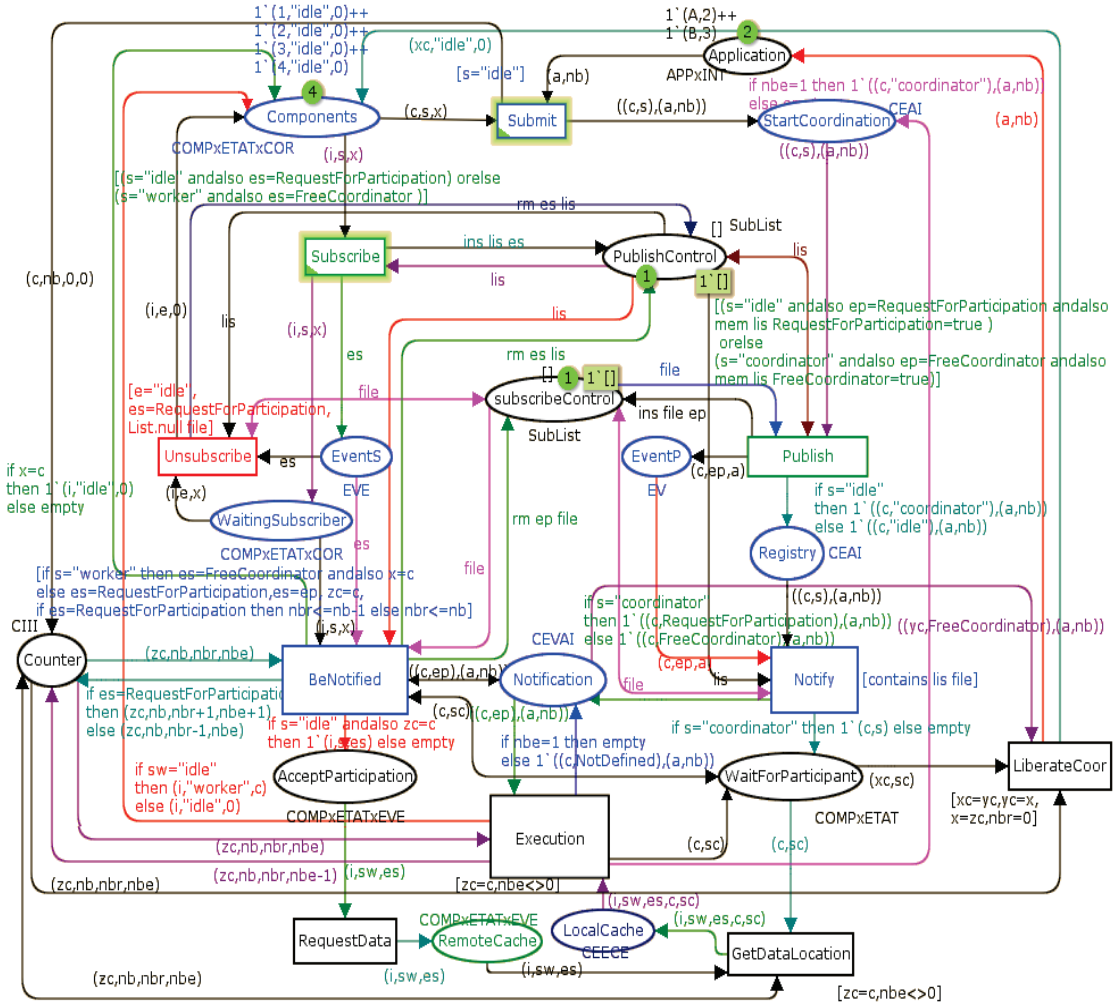


FIGURE *4.10 – Modélisation de la nouvelle version de BonjourGrid basée sur Redis*

adapter le noyau central de notre modélisation pour qu'il prenne en considération cette nouvelle contrainte. Nous avons aussi rajouté les deux composants de «cache local» et «cache distant» qui sont directement liés au processus de l'exécution d'une application.

4.5.2 Discussion

Le système d'interaction Publication-Souscription est un paradigme émergent pour les applications distribuées à grande échelle. Il intéresse autant les industriels que les chercheurs. Certains standards et produits industriels sont directement basés sur ces

systèmes. D'autres systèmes proposés dans la recherche s'en inspirent aussi. La plupart de ces travaux s'intéressent au problème de la construction d'un système le plus idéal possible en termes de scalabilité, efficacité et sécurité. Mais, ils ne se focalisent pas assez sur le problème d'analyse formelle de l'exactitude de tels systèmes. Dans ce sens, certains travaux de recherche ont déjà prêté attention à la vérification formelle des systèmes de Publication-Souscription.

Dans [ZGB03, BGZ05], Baresi L. et al proposent une approche pour la modélisation et la validation des systèmes de Publication-Souscription. Cette approche est basée sur une architecture de composants qui réagissent à des événements. Dans ce travail, les composants sont spécifiés en diagrammes d'états-transitions UML. Les auteurs proposent une approche de validation utilisant la méthode de vérification de modèle (*model-checking*) pour prouver des propriétés sur l'échange de messages qui sont définies par des diagrammes d'état-transition. Les propriétés sont transformées en automates puis traduites en langage PROMELA (*PROtocol MEta LAnguage*) pour pouvoir être vérifiées par le modèle checker SPIN [Hol03]. Ainsi, au lieu d'utiliser les formules de la logique linéaire temporelle (LTL) de SPIN, les auteurs ont interprété les propriétés sous forme d'automates. Selon eux, ceci permettra de représenter des propriétés plus complexes nécessaires pour valider le système modélisé.

Bien que notre approche de modélisation soit aussi basée sur des composants réagissants à des événements, nous préférons jouer sur les atouts des logiques temporelles pour la vérification formelle, en particulier par l'utilisation de la librairie ASK-CTL [CM96].

Dans [GKK03], Garlan D. et al décrivent un framework générique dédié à la modélisation et la vérification formelle du mécanisme Publication-Souscription. Leur système est basé sur un modèle de machine à états assurant la gestion des événements pendant l'exécution du protocole Publication-Souscription. Le framework prend en entrée un ensemble de composants et un ensemble de propriétés du mécanisme Publication-Souscription. L'appariement des deux ensembles est par la suite validé en utilisant des outils de model checking. Ce système reste considéré surtout comme un framework générique dans lequel il existe toujours le risque de ne pas offrir une modélisation et une vérification adaptée à chaque cas spécifique du mécanisme Publication-Souscription. Notre optique est de réussir une modélisation et une vérification formelle la plus adaptée possible à BonjourGrid tout en isolant le mécanisme Publication-Souscription.

Dans [KKJD08, KKD09], Kacem N. et al, motivés par les avantages des analyses formelles, proposent pour leur protocole de coordination un modèle formel en utilisant les réseaux de Pétri colorés. Pour juger l'exactitude de leur modèle et par suite de leur protocole, ils ont vérifié formellement les propriétés comportementales et mis en œuvre un mécanisme de model checking CTL. De notre coté, nous capitalisons aussi sur l'utilisation des réseaux de Pétri colorés et les logiques CTL.

Dans [FL06], les auteurs proposent l'extension de l'ADL *Architecture Description Language* [KC96] afin de supporter la description du style Publier-Souscrire. Il s'agit

d'un langage basé sur la théorie des catégories, laquelle étudie les structures mathématiques et les relations qu'elles entretiennent. Les auteurs spécifient alors les types de composants ainsi que leur comportement. Pour chaque composant, ils spécifient les événements publiés et/ou souscrits. La topologie du service d'événement est non spécifiée. Le langage est assez complexe et nécessite une expertise considérable dans le domaine des catégories.

Bien que les travaux réalisés autour de la modélisation formelle du protocole de publication-souscription restent très intéressants, ils ont chacun abordé le sujet d'un point de vue différent, et avec des niveaux d'abstraction différents.

Notre travail se différencie par rapport aux autres travaux par la démarche utilisée, le niveau d'abstraction considéré, les outils utilisés et surtout le but principal de la modélisation. En fait, dans notre travail, la modélisation du protocole de publication-souscription n'est pas réalisée pour modéliser uniquement le protocole mais surtout pour constituer une brique générique permettant de modéliser n'importe quel système utilisant ce protocole.

Notre démarche est centrée sur la coordination, autrement dit sur l'aspect comportemental en terme d'événements publiés et/ou souscrits. L'aspect structurel et architectural ne sont pas mis en avant. Ainsi, nous nous situons à un niveau d'abstraction très élevé afin de maîtriser la complexité de notre système et ne pas se perdre dans les détails.

Les réseaux de Petri colorés offrent une technique de modélisation formelle qui est bien adaptée pour la modélisation et l'analyse des systèmes complexes. En effet, les modèles hiérarchiques peuvent être construits, l'information complexe peut être représentée par paramétrage des jetons et des outils matures et éprouvées existent pour créer, simuler et analyser des modèles CPN.

D'autre part, notre modélisation formelle présente des atouts majeurs de part la technique et les outils utilisés. En effet, les fonctionnalités intégrées dans l'outil CPN-Tools offrent plusieurs avantages. L'outil inclue un support pour la collecte de données au cours de simulations, et un support pour l'exécution de plusieurs réplications de simulation. Les mécanismes utilisés pour ces fins sont appelés des moniteurs. Ils permettent d'inspecter et contrôler les simulations dans le but de collecter des données, mesurer la performance, définir des points de ruptures de simulation spécifiques au modèle. Ainsi, on peut observer et contrôler complètement le comportement d'un modèle.

Bien que cette analyse méthodologique offre des résultats satisfaisants, elle peut être complétée par une étape beaucoup plus avancée de vérification formelle de notre protocole. Cette étape consiste à analyser des configurations plus importantes afin de contourner le problème de l'explosion d'états. Deux perspectives se présentent :

– Exporter notre modélisation actuelle sur une plateforme dédiée spécialement à la vérification formelle. CPN Tools est en effet très pratique dans une perspective de modélisation et de simulation, mais on peut toujours utiliser un outil de vé-

rification de modèle spécialisé qui permettra de vérifier des configurations plus complexes.Ce travail de modélisation nous a permis de prendre conscience à quel point les systèmes de grille de calcul sont complexes et nous avons alors étudié la possibilité de réaliser un nouveau système de grille de PCs qui rompe avec la complexité traditionnelle.

Notre approche consiste à repenser les grilles de PCs à partir d'une réflexion et d'un cadre formel permettant de les développer, aujourd'hui, de manière rigoureuse et de mieux maîtriser les évolutions technologiques à venir. Nous avons reconsidéré les interactions entre les composants traditionnels d'une grille de PCs en se basant sur les technologies du Web, et donné naissance à un nouvel intergiciel de grille de PCs, RedisDG, capable de tourner sur les petits dispositifs, i.e. smartphones, tablettes comme sur les dispositifs plus traditionnels (PCs). RedisDG est entièrement basé sur le paradigme de publication-souscription, nous entendons la manière dont nous réalisons la coordination des différents composants, la façon dont une machine rejoint le système, la façon dont elle le quitte, la manière d'échanger les données, la manière de contrôler l'exécution,.... RedisDG est développé avec Python et utilise Redis comme système de gestion de base de données clef-valeur scalable.

– Utiliser des techniques de réduction de l'espace d'état afin d'aborder efficacement le problème de l'explosion d'états. La réduction basée sur la symétrie est tout à fait adaptée à notre modèle du fait qu'elle est spécifiquement conçue pour les systèmes composés de processus similaires exécutant le même protocole. Dans le meilleur des cas, l'espace d'états peut être réduit de manière exponentielle [GCGS91].

4.6 Étude des avantages/inconvénients de Redis

4.6.1 Présentation

Redis est principalement un système de gestion de base de données clé-valeur scalable. Il fait partie de la mouvance des langages NoSQL et vise à fournir les performances les plus élevées possibles [RED]. Une des principales caractéristiques de Redis est de conserver l'intégralité des données en RAM. Cela permet d'obtenir d'excellentes performances en évitant les accès disques, particulièrement coûteux. Lorsque la taille des données est trop importante pour tenir en mémoire, Redis peut également utiliser de la mémoire virtuelle.

Redis supporte la réplication via un modèle maître-esclave à des fins de résistance aux pannes et de répartition de la charge. De plus, il implémente une couche supportant le paradigme de publication-souscription.

Ainsi, nous avons à priori un outil «tout-en-un» qui répond à nos besoins initiaux à savoir : le stockage de codes que nous avons à exécuter, le stockage des données d'en-

trée/sortie des programmes qu'on exécute, et le support du mécanisme de publication-souscription. Pour répondre à nos besoins, Redis s'est avéré l'outil le plus adéquat. Cela explique pourquoi nous avons décliné les possibilités de travailler avec XMPP, Nodejs ou Hookbox, que nous introduisons dans la section qui suit afin d'argumenter nos choix.

4.6.2 Comparaison avec XMPP, Nodejs et Hookbox

L'ensemble des articles lus et sites Web consultés pour réaliser cette comparaisoon est donné en Annexe 1.

Les défis principaux sont le support du paradigme de publication-souscription, la scalabilité et un stockage adéquat de données et de codes. La comparaison entre Redis et les différentes technologies citées dans le tableau 4.2 permet de souligner un haut niveau de support des mécanismes de publication-souscription, par Redis et XMPP. Ces approches sont meilleures sur ce critère que Hookbox et Nodejs. Sur le critère de la scalabilité, Nodejs est autant performant que Redis et XMPP, mais Hookbox présente moins de performance. En ce qui concerne le stockage de données et de codes à exécuter, Redis est la meilleure technologie par rapport aux autres, car elle permet le stockage selon l'approche clef-valeur et le support de types abstraits de données (tels que les listes et les tables de hachage).

Les défis secondaires sont la réplication, les performances, la répartition de charge et la persistance. Au niveau de la réplication, Redis reste de loin la meilleure technologie par rapport aux autres. XMPP et Nodejs sont plus performants que Hookbox[Annexe 1], mais Redis se place un peu plus en avant que les autres. La répartition de charge reste un point fort de Redis vis-à-vis de ce que les autres approches peuvent assurer. La persistance des données est évidemment le point faible de Hookbox et un point fort de Redis.

Le point faible de Redis est au niveau sécurité, en fait, il n'offre pas un support solide pour l'authentification, la communication cryptée ou la protection des données. Nous supposons que dans l'avenir, les développeurs de la communauté assureront de telles propriétés. Toutefois, Redis permet une redondance de serveurs afin d'assurer un mécanisme de tolérance aux fautes.

4.6.3 Terminologie de Redis

Redis comprend un objet publish-subscribe qui souscrit à des canaux et écoute des nouveaux messages. Une fois qu'une instance publish-subscribe est créé, les clients peuvent s'abonner à un ou plusieurs canaux à l'aide de la commande
SUBSCRIBE(CHANNEL_NAME). Les clients peuvent publier un message sur un ca-

	HOOKBOX	**NODEJS**	**XMPP**	**REDIS**
Développement actif	2010-2012	2009-2013	1999-2013	2009-2013
Dernière version	0.4	0.10.21	RFC6122	2.6.16
Développé en langage	Python	C++, JavaScript	XML	C ANSI
Plateformes	Windows, MacOS X, Linux	Multi-plateforme	Multi-plateforme	Multi-plateforme
Support du paradigme de Publication-Souscription	++	++	++++	++++
Stockage	++	+++	+++	++++
Scalabilité	++	++++	+++	++++
Performance	++	+++	+++	++++
Réplication	-	+++	+	++++
Sécurité	++++	++	++++	++
Tolérance aux pannes	++	++++	+	++++
Répartition de la charge	+	++	+	++++
Persistance des données (en mémoire)	-	+++	+++	++++
Support de types abstraits de données	-	-	-	++++

TABLE *4.2 – Tableau comparatif*

nal donné via la commande
PUBLISH(CHANNEL_NAME, MESSAGE). Les messages envoyés par les clients sur le canal seront poussés par le serveur Redis à tous les clients abonnés.

Nous considérons la commande SUBSCRIBE comme une opération constante dans le temps, et toute la complexité est située au niveau de la commande PUBLISH, qui effectue une quantité de travail proportionnelle au nombre de clients qui reçoivent le message. Cela signifie que la performance de publication-souscription de Redis peut être comparée avec les cas suivants :

(a) N clients abonnés à M canaux différents, dans ce cas les notifications seront rapides. En effet, Redis utilise plusieurs canaux pour envoyer les messages.

(b) N clients abonnés au même canal, la commande PUBLISH sera lente à réagir sur ce canal. Puisque nous devons envoyer le même message à tout le monde sur le même canal, il faut mesurer le temps global pris au cours de cette étape.

(c) Les événements de publication-souscription peuvent être séquentiels, en cascade (pipeline) ou simultanés.

4.6.4 Évaluation des performances de Redis

Dans cette section, nous évaluons la scalabilité et les performances du protocole Redis pour l'inscription et la découverte des services [LAJ14].

4.6.4.1 Dispositif expérimental

Pour offrir un système de grille de PCs performant sur Internet, il est important de mesurer le temps de réponse de Redis lors de la gestion des ressources provenant de différents réseaux et de multiples domaines d'organisation. Les tests sont réalisés sur Grid'5000 [Gri] en utilisant 300 nœuds entre les sites de Nancy, Grenoble et Toulouse. Nous mettons en place un noyau spécifique contenant le package Redis (les outils client et serveur) afin d'exécuter nos scripts python pour le démarrage du serveur Redis (Start-Redis-Server()), les services de publication (Register-Service()) et les services de souscription (Browse-Service()). Pour simuler ce comportement, nous avons adopté plusieurs scénarios de test (séquentiels ou simultanés) en utilisant un ou plusieurs sites. Ici, nous présentons uniquement les résultats pour les enregistrements simultanés.

4.6.4.2 Expérimentation autour de la publication simultanée dans un seul site

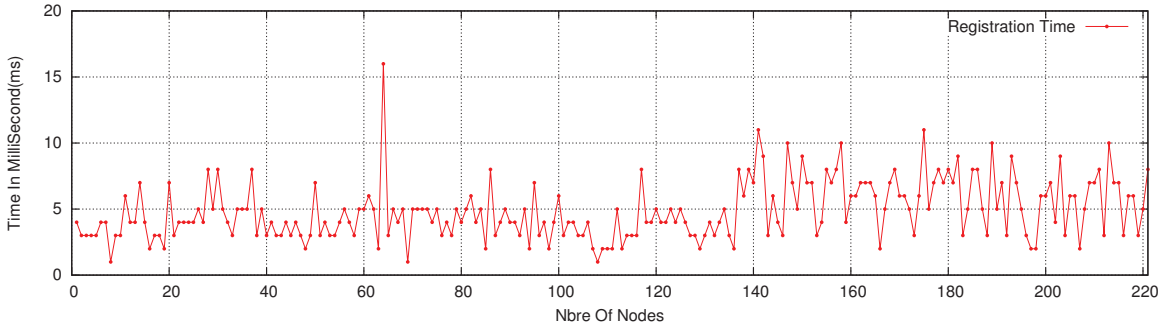
Dans ce test, nous réservons 222 nœuds sur le site de Nancy pour publier un service donné en même temps. Le scénario est divisé en trois étapes :

(a) Exécuter le programme du serveur Redis dans le premier nœud ;

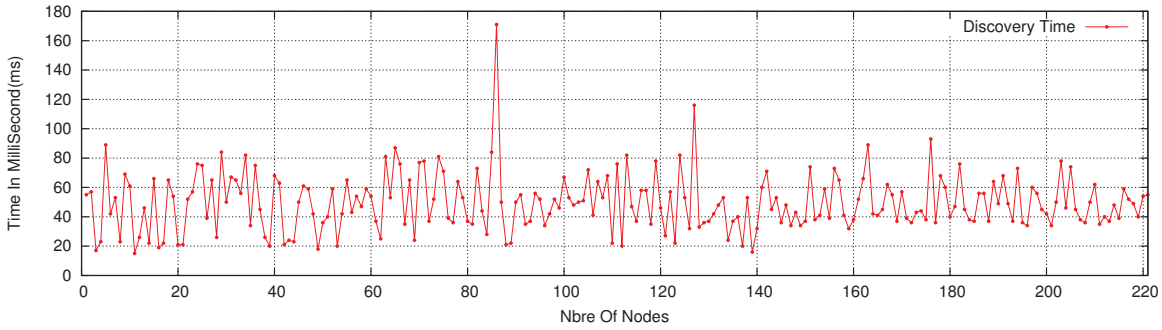
(b) Lancer le script Browse-Service (service_name) dans le second nœud. L'attribut «service_name» est le nom du canal auquel on souscrit.

(c) Exécuter en parallèle sur 220 nœuds restants le script de Register-Service(service_name, service_ID). Nous utilisons le même nom de service pour toutes les publications. La valeur du service_ID est le rang de la machine dans le fichier de la charge de travail. Cet attribut est le message à transmettre par une machine donnée. L'attribut service_ID sera utilisé par le nœud Browse-Service() afin de distinguer le service publié au cours du processus de découverte/souscription, et marquer la fin du temps de publication.

La figure 4.11(a) indique le temps de publication simultanée. Pour chaque nœud, ce temps indique la différence entre la date de soumission et le temps nécessaire pour publier le service sur le canal. Nous lançons la publication d'un service sur chaque nœud. Avec un maximum de 220 machines, le temps de publication écoulé varie entre 2 et 16



(a) Temps de publication



(b) Temps de découverte

FIGURE *4.11 – Publications simultanées utilisant un seul site de Grid'5000.*

ms. Ainsi, la comparaison de Redis avec d'autres analyses précédentes des protocoles Bonjour, Avahi et Pastry [AD08], démontre que Redis présente les meilleurs résultats et n'est pas surchargé.

L'autre valeur importante est le temps nécessaire pour souscrire à un service donné. Pour chaque nœud, les résultats sont présentés dans la figure 4.11(b). Ce temps indique le temps écoulé entre la fin de la publication d'un service unique et l'instant où le nœud souscripteur a bien découvert le service. Les courbes prouvent que le souscripteur peut découvrir tous les services publiés dans un délai moyen égal à 50ms.

4.6.4.3 Expérimentation autour de la publication simultanée dans plusieurs sites

Dans ce test, nous avons utilisé 292 machines réparties sur trois sites de la plateforme Grid'5000. Nous adoptons le même scénario utilisé dans le premier test. La figure 4.12 montre que la publication des services est divisée en trois phases. Une fois que le nœud de souscription souscrit au canal en exécutant le script Browse_Service(), nous activons

en même temps le processus de publication pour un site donné respectivement sur Nancy (115 nœuds), Grenoble (95 nœuds) et Toulouse (82 nœuds).

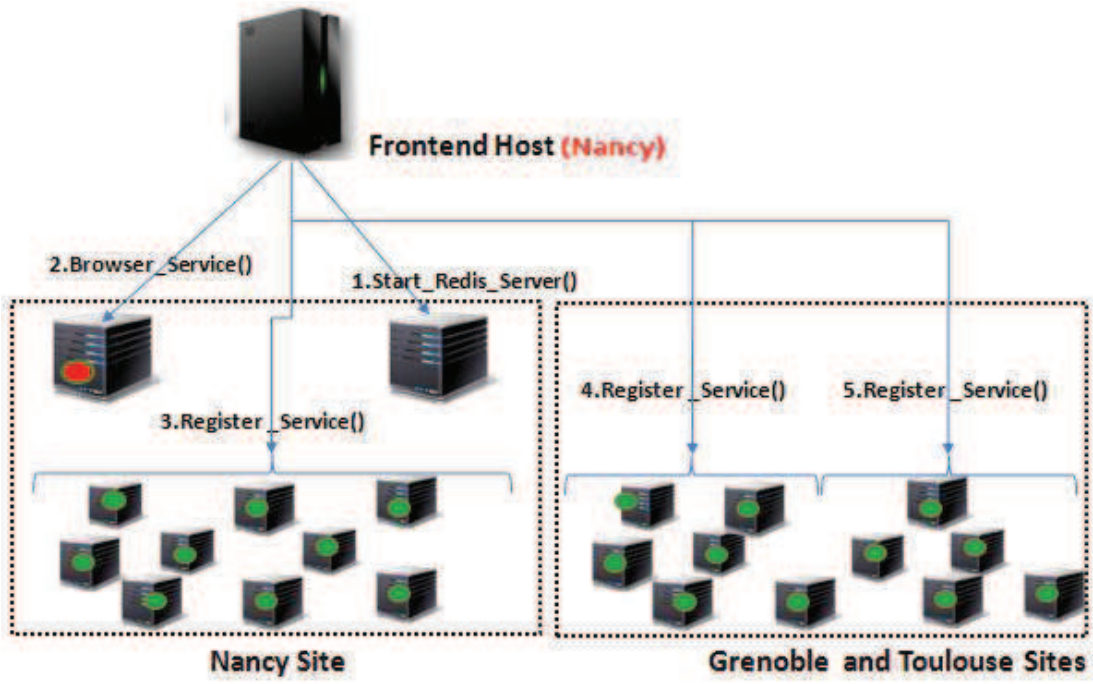
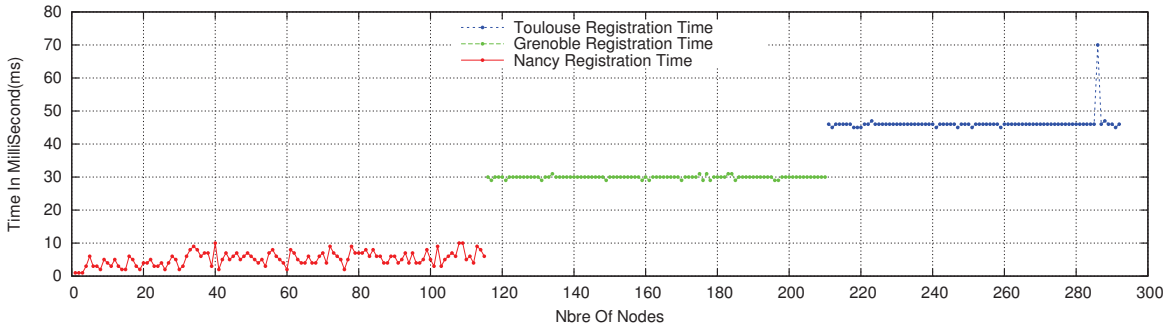


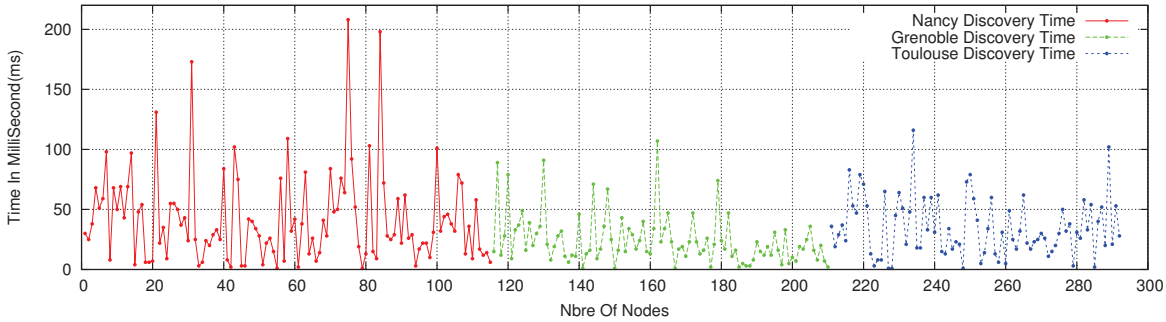
FIGURE *4.12 – Scénario utilisé pour plusieurs sites d'évaluation.*

La figure 4.13(a) indique que le temps de publication augmente d'un site à l'autre. En effet, le temps écoulé varie entre 10ms (Nancy site) à 48ms (Toulouse). Notons que jusqu'à 290 machines Redis n'a pas été saturé et les parcelles de courbes sont presque linéaires. Ce qui met en avance la haute scalabilité du protocole Redis.

Contrairement au temps de publication, le temps de découverte montre une différence entre les sites. Comme on le voit dans la figure 4.13(b), les parcelles ne sont pas linéaires et le temps varie entre 2 à 210 ms. Les sommets des courbes sont expliquées par le fait que le serveur Redis traite le nombre important de connexions clients dans l'ordre (d'une façon séquentielle) en utilisant le même canal.



(a) Temps de publication



(b) Temps de découverte

FIGURE *4.13 – Publications simultanées utilisant trois sites de Grid'5000.*

4.7 Discussion autour de la modélisation du mécanisme de publication-souscription

La façon dont le mécanisme de publication-souscription est implémenté n'est pas la même pour le protocole Bonjour (utilisé dans BonjourGrid) et Redis (choisi pour notre contribution RedisDG). En fait, pour Bonjour, du côté du producteur publiant l'événement, on dispose d'une fonction de la forme suivante :

```
Fonction PublisherService {
Publish(Event e) ;
}
```

Les souscripteurs possèdent une fonction permettant de souscrire à des classes d'événements :

```
Fonction SubscriberService {
```

```
Subscribe(EventType t);
Event Receive();
}
```

Pour Redis, un service est capable d'envoyer un message dans un canal sans connaître les récepteurs à l'avance. Un ou plusieurs services sont abonnés à ce canal et reçoivent les messages correspondants. Avec Redis, on peut résumer cela avec les deux abstractions suivantes :

- SUBSCRIBE pour s'abonner à un canal/sujet/catégories ;
- PUBLISH pour envoyer un message dans un canal.

Au moment où un message est publié, tous les abonnés vont le recevoir en même temps. Ensuite ce message est perdu pour toujours. Si un client n'était pas connecté à ce moment là, il ne recevra jamais le message. Inversement, un client peut attendre infiniment avant qu'un message entre en jeu, parce que il n'y a pas de temporisation de l'attente. C'est ici que réside toute la différence avec le protocole Bonjour.

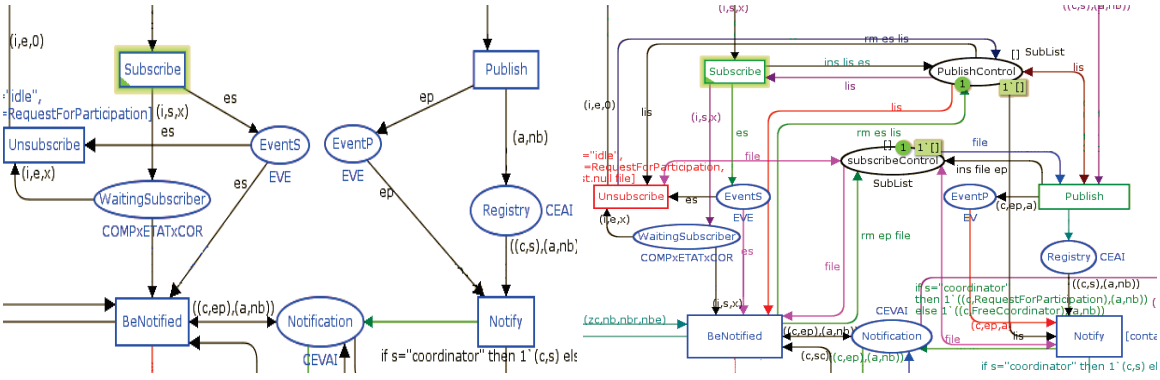


FIGURE *4.14 – Publication-Souscription façon Bonjour vs Publication-Souscription façon Redis*

Cette différence n'a pas été négligeable lors de la modélisation. Comme le montre la figure 4.14, il y a des différences entre la partie gauche concernant la modélisation du protocole Bonjour et la partie droite concernant la modélisation de Redis (ces deux modèles sont une nouvelle fois le fruit de notre travail). En fait, dans Bonjour, les composants peuvent souscrire n'importe quand et publier n'importe quand sans contraintes particulières et sans risque de perdre de l'information et ceci dans la limite du protocole [ACDJ08]. Ce qui reste assez générique. Ainsi, lors de la modélisation, nous n'avons pas besoin de contrôler la publication et la souscription.

Contrairement à Redis, le fait que les messages publiés seront perdus lorsqu'il n'y a pas de souscripteurs potentiels engendrent des contraintes au moment de la modélisation. Ainsi, nous avons dû rajouter des places de contrôle afin de maîtriser les publications et les souscriptions pour qu'il n'y ait pas de pertes de messages et satisfaire la

propriété : «un événement produit/publié doit être reçu par tous les consommateurs/-souscripteurs intéressés par cet événement».

La place *PublishControl* permet de placer les événements auquels on souscrit dans une liste qu'on a appelé *lis*. Avant qu'un composant puisse publier un événement *ev*, on doit d'abord vérifier s'il existe des composants qui ont déjà souscrit à cet événement. La garde *[mem lis ev=true]* placée au niveau de la transition *Publish* permet d'assurer cette contrainte.

D'autre part, la place *SubscribeControl* permet de mettre à jour la liste des événements publiés qu'on a appelé *file*. La notification se fait quand la liste *file* et incluse dans la liste *lis*. Cette contrainte est assurée par la garde *[contains lis file]* au niveau de la transition *Notify*.

Il faut veiller à ce que ces deux listes soient mise à jour à chaque changement du système. C'est ce qui justifie le fait que les deux listes soient reliées à la transition *Unsubscribe* ou aussi à la transition *BeNotified*. Ainsi, actualiser les deux listes au bon moment garantit le respect des différentes contraintes et par conséquent le bon fonctionnement du système.

4.8 Conclusion

Dans ce chapitre nous avons présenté une partie de nos contributions concernant la modélisation formelle. Ces travaux ont fait l'objet de publications internationales dans les conférences : SCC (*IEEE International Conference on Services Computing, 2011*) [ACE11], GPC (*International Conference on Grid and Pervasive Computing, 2012*) [ACK12] et SBAC-PAD (*International Symposium on Computer Architecture and High Performance Computing, 2014*) [LAJ14] pour l'évolution de BonjourGrid et l'étude des performances de Redis.

Chapitre 5

RedisDG : modélisation, prototypage et validation expérimentale

5.1 Introduction

Dans ce chapitre, nous détaillons la démarche suivie afin de développer un nouvel intergiciel de grille de PCs, appelé RedisDG, semblable dans son fonctionnement à BOINC ou Condor mais capable de tourner sur les petits dispositifs, i.e. smartphones, tablettes comme sur les dispositifs plus traditionnels (PCs) (150 lignes de code python dans notre classe Worker). Notre objectif principal est de repenser les interactions entre les composants d'une grille de PCs en termes de technologies WEB actuelles. En effet, il n'est pas garanti que les codes des intergiciels BOINC ou Condor, puissent continuer à tourner sur les infrastructures actuelles. Par exemple, dans le cas du Cloud, Cérin et Takoudjou ont étudié dans [CT13] l'intégration de BOINC puis Condor dans le Cloud SlapOS [SSCC11] et cette intégration ne s'est pas avérée facile et immédiate. Nous commençons par présenter le modèle formel de RedisDG, puis son architecture, ensuite son implémentation ainsi qu'une validation expérimentale et enfin son intégration dans le Cloud SlapOS afin d'offrir les fonctionnalités d'une grille de PCs comme un service Web.

5.2 Modélisation d'un nouvel intergiciel de grille de PCs

Le travail de modélisation que nous avons détaillé dans le chapitre précédent, nous a révélé à quel point les systèmes de grille de calcul sont complexes. Nous avons alors

étudié la possibilité de réaliser un nouveau système de grille de PCs qui rompe avec la complexité traditionnelle. Le plus difficile dans ce domaine est de se fixer sur le protocole de coordination utilisé pour gérer les interactions entres les différents composants d'une grille de PCs. Mais, dans notre cas et comme nous avons déjà réalisé un travail de modélisation formelle autour du paradigme de publication-souscription il était naturel d'exploiter ce travail. Ainsi est venue l'idée de penser un système de grilles de PCs qui soit entièrement basé sur le paradigme de publication-souscription, nous entendons la manière dont nous réalisons la coordination des différents composants, la façon dont une machine rejoint le système, la façon dont elle le quitte, la manière d'échanger les données, la manière de contrôler l'exécution,.... Comme notre idée de base était de fusionner les nouvelles technologies du Web et les composants traditionnels d'une grille de PCs, nous avons opté pour Redis [RED] comme technologie du Web puisqu'il implémente déjà le paradigme de publication-souscription. Nous le justifierons plus tard.

Nous avons commencé par réaliser un modèle formel basé sur notre modélisation initiale du paradigme de publication-souscription comme brique centrale, mais aussi adapté au fonctionnement de Redis (comme expliqué dans la section 4.7).

La figure 5.1 montre le réseau de Petri coloré de notre système RedisDG. Nous distinguons au centre la brique publication-souscription basée sur les trois transitions `Publish, Subscribe, Notify`. Dans Redis, si un événement est publié et qu'il n'y a pas de composants ayant souscrit au préalable pour être notifiés par cet événement, alors il est perdu à jamais. Afin de réaliser un modèle fidèle à ce comportement, nous avons adapté notre brique de publication-souscription de base qui s'est avérée très générique, et nous avons ajouté des places de contrôle afin de bloquer une publication qui n'a pas de souscriptions. Les deux places de contrôle sont : `SubscribeControl` et `ModRedis`. Leurs rôles est de générer, mettre à jour et comparer des listes des événements publiés et des événements auxquels on souscrit.

Au moment de l'implémentation de RedisDG, il fallait prendre en considération ce mécanisme propre à Redis. Ainsi, cette étape de modélisation nous a aidé énormément : l'idée derrière l'ajout des places de contrôle, était d'avoir toujours des souscripteurs en attente d'être notifiés. Au moment de développement de RedisDG, cette idée a été comme une ligne directrice. De ce fait, les workers potentiels dans RedisDG, dès qu'ils rejoignent le système, doivent souscrire à un événement. Un worker dans RedisDG est composé, de manière concrète, de deux *Threads* : un premier qui gère l'exécution d'une tâche et le deuxième qui gère la souscription afin de garantir la continuité du fonctionnement du système.

5.3 L'algorithme de coordination

Nous introduisons dans cette section l'algorithme de coordination de notre système RedisDG. Il s'agit de la vue la plus haute possible. Certains détails techniques seront

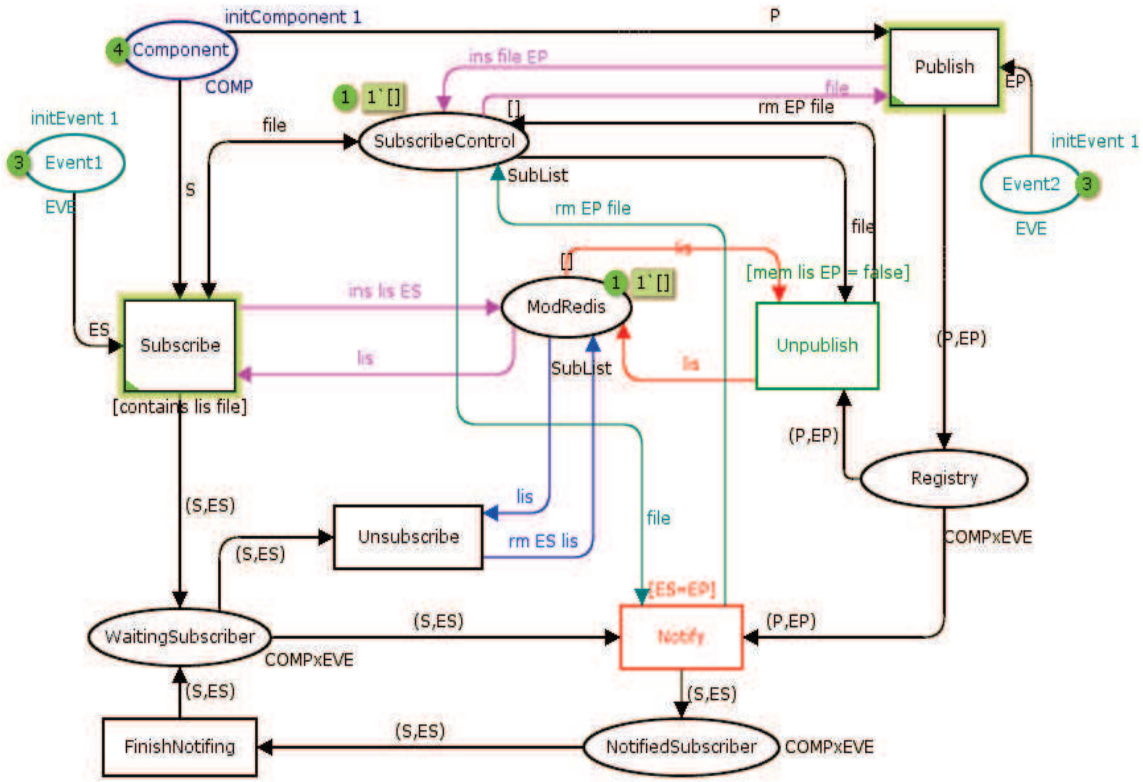


FIGURE *5.1 – Modèle formel de RedisDG*

donnés dans la partie consacrée aux expérimentations. L'algorithme est entièrement basé sur le paradigme de publication-souscription. L'intergiciel obtenu offre les mêmes fonctionnalités que la majorité des intergiciels de grilles de PCs tels que BOINC ou Condor. Il gère les stratégies d'ordonnancement en particulier la détection des dépendances entre les tâches, l'exécution des tâches et la vérification/certification des résultats ; puisque les résultats retournés par les Workers peuvent être manipulés ou altérés par des Workers malveillants. Les objectifs généraux sont :

- Utiliser un paradigme asynchrone (publication-souscription) qui assure autant que possible, un découplage total entre les étapes de coordination (ceci pour des raisons de performance) ;
- Garantir que le système soit résilient par la duplication des tâches et des acteurs. Même si le système est asynchrone et que les tâches sont dupliquées nous devons assurer la progression de l'exécution des tâches. Nous supposons également que les acteurs sont dupliqués pour des raisons de résilience ;

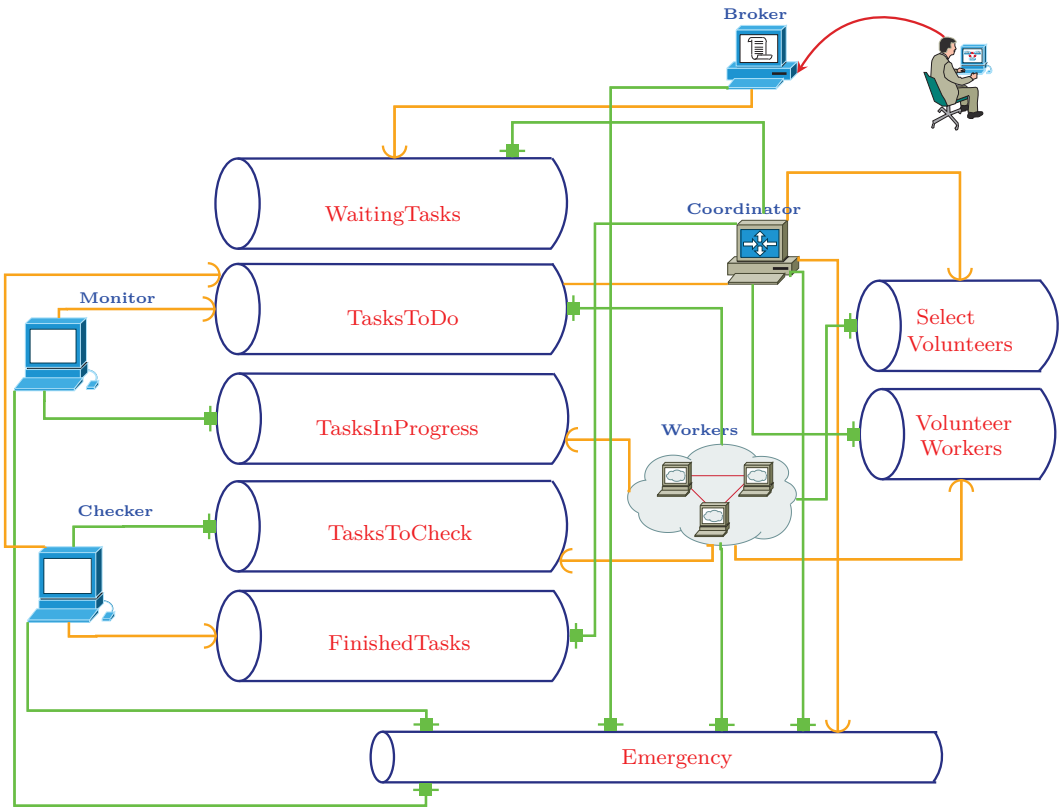


FIGURE *5.2 – Interactions entre les composants*

Dans la figure 5.2, nous présentons le déroulement de l'exécution d'une application dans RedisDG. Dans notre système, une tâche peut avoir cinq états ; *WaitingTasks, TasksToDo, TasksInProgress, TasksToCheck et FinishedTasks*. Ils sont gérés par cinq acteurs ; broker, coordinator, worker, monitor et checker. Pris séparément, le comportement de chaque composant dans le système peut paraître simple, mais nous nous intéressons plutôt à la coordination de ces composants, ce qui rend le problème plus difficile à résoudre.

L'idée clé est de permettre le branchement des composants dédiés (coordinator, checker,...) dans un mécanisme de coordination générale, afin d'éviter de construire un système monolithique. Le comportement de notre système comme présenté dans la figure 5.2 est le suivant :

1. Soumission de lots de tâches. Chaque lot représente un graphe série-parallèle de tâches à exécuter.
2. Le Broker extrait les tâches et les publie sur un canal appelé *WaitingTasks*.
3. Le Coordinator est à l'écoute sur le canal *WaitingTasks*.

4. Le Coordinator commence par publier les tâches indépendantes sur le canal *Tasks-ToDo*.
5. Des Workers annoncent leur volontariat sur le canal *VolunteerWorkers*.
6. Le coordinateur sélectionne les Workers suivant certains critères.
7. Les Workers, à l'écoute au préalable sur le canal *TasksToDo*, commencent à exécuter les tâches publiées. Cet événement d'exécution en cours est publié sur le canal *TasksInProgress*.
8. Durant l'exécution, chaque tâche est sous la supervision du Monitor qui a pour rôle de s'assurer du bon déroulement de l'exécution en vérifiant si le nœud est en vie. Dans le cas contraire le Monitor re-publie les tâches qui ne sont pas arrivées au bout de leur exécution sur le canal *TasksToDo* afin d'être relancées par d'autres Workers.
9. Une fois l'exécution terminée, le Worker publie la tâche sur le canal *TasksToCheck*.
10. Le Checker vérifie le résultat retourné et publie la tâche correspondante sur le canal *FinishedTasks*.
11. Le Coordinator vérifie les dépendances entre les tâches terminées et celles en attente, et recommence à l'étape (4).
12. Une fois l'exécution terminée, le Coordinator publie un message sur le canal *Emergency* afin de notifier tous les composants de la terminaison de l'application.

Algorithm 1 Broker (batch)

```
1: for t ∈ batch do
2:     Broker.publish(t,WaitingTasks)
3: end for
```

L'algorithme 1 présente les actions réalisées par le Broker. Le Broker prend comme paramètre un graphe de tâches qui est représenté par un fichier *batch*. Il construit un dictionnaire de dépendance et publie les tâches sur le canal *WaitingTasks*.

L'algorithme 2 présente le fonctionnement du coordinateur. D'abord, le coordinateur doit s'abonner au canal *FinishedTasks* (ligne 1) afin d'être notifié des tâches terminées. Cela implique qu'il est toujours en attende de tâches terminées. Ensuite, il vérifie le dictionnaire de tâches publié par le Broker sur le canal *WaitingTasks* et il commence par publier les tâches indépendantes sur le canal *TasksToDo* (ligne 5). Enfin, dès qu'il y a un événement publié sur le canal *FinishedTasks*, le coordinateur met à jour son dictionnaire de dépendances et re-publie les nouvelles tâches indépendantes sur le canal *TasksToDo* (ligne 12).

L'algorithme 3 décrit les actions réalisées par un Worker. Un Worker doit, tout d'abord, s'abonner au canal *TasksToDo* (ligne 1) afin d'être notifié par les nouvelles tâches à faire. Ensuite, il procède à l'exécution d'une tâche (ligne 3). Il publie l'état de

Algorithm 2 Coordinator

```
1: Coordinator.subscribe(FinishedTasks)
2: Coordinator.subscribe(WaitingTasks)
3: for t ∈ WaitingTasks do
4:    if t is independent then
5:       Coordinator.publish(t,TasksToDo)
6:    end if
7: end for
8: while True do
9:    for tf ∈ FinishedTasks do
10:      for tw ∈ WaitingTasks do
11:         if tw.depend(tf) then
12:            Coordinator.publish(tw,TasksToDo)
13:         end if
14:      end for
15:   end for
16: end while
```

Algorithm 3 Worker

```
1: Worker.subscribe(TasksToDo)
2: for t ∈ TasksToDo do
3:    Worker.execute(t)
4:    Worker.publish(t,TasksInProgress)
5:    if t == finished then
6:       Worker.publish(t,TasksToCheck)
7:    end if
8: end for
```

la tâche en cours d'exécution sur le canal *TasksInProgress* (ligne 4). Une fois l'exécution terminée, le Worker publie l'événement sur le canal *TasksToCheck* (ligne 5).

Dans l'algorithme 4, nous présentons les actions réalisées par le Monitor. Un Monitor doit s'abonner au canal *TasksInProgress* afin de superviser le déroulement de l'exécution de chaque tâche (ligne 1). Si l'exécution d'une tâche échoue (pour n'importe quelle raison), le Monitor demande la re-exécution en re-publiant la tâche sur le canal *TasksToDo* (ligne 6).

L'algorithme 5 présente le fonctionnement du Checker. Le Checker doit s'abonner à la file *TasksToCheck* afin d'être notifié par les tâches à vérifier. L'approche utilisée pour faire de la certification des résultats est l'approche par duplication de tâches. Si le Checker certifie que le résultat est correct, alors la tâche correspondante sera publiée sur le canal *FinishedTasks* (ligne 4), sinon il demande la re-exécution en publiant la

Algorithm 4 Monitor

```
1: Monitor.subscribe(TasksInProgress)
2: for t ∈ TasksInProgress do
3:    while (t != finished) and (t = OK) do
4:       Monitor.supervise(t)
5:    end while
6:    if t ≠ ok then
7:       Monitor.publish(t,TasksToDo)
8:    end if
9: end for
```

Algorithm 5 Checker

```
1: Checker.subscribe(TasksToCheck)
2: for t ∈ TasksToCheck do
3:    if Checker.check(t) = OK then
4:       Checker.publish(t,FinishedTasks)
5:    else
6:       checker.publish(t,TasksToDo)
7:    end if
8: end for
```

tâche sur le canal *TasksToDo*(ligne 6).

5.4 Outils pour le monitoring d'activité

Nous avons déjà fait observer que l'architecture du Cloud SlapOS reposait sur de nombreux concepts propres aux systèmes d'exploitation. Par exemple le déploiement d'applications se fait dans des partitions au sens Unix, l'isolation des applications repose donc sur des mécanismes de privilèges (au sens Unix). Cette approche Système d'un Cloud est en fait un choix de conception fort. De manière générale, plutôt que de bâtir des systèmes informatiques en construisant des intergiciels qu'il convient de faire interagir, les fonctionnalités qui peuvent être abordées et traitées par le système d'exploitation sont directement exploitées dans le système en construction.

Ce principe est de nouveau utilisé ici pour le monitoring des applications lancées par l'utilisateur. Le scénario et le problème traité est donc le suivant : un utilisateur soumet un workflow à RedisDG ; il fournit donc des codes à exécuter, un graphe de tâches, des données en entrées. Comment relever l'activité des codes utilisateurs pour facturer à l'utilisateur l'usage de ses codes en terme de ressources CPU, disque, réseau,. . . utilisées ?

Deux approches directement liées à ce que fourni un système d'exploitation Linux,

sont maintenant discutées : `SystemTap` et `SysStat`. Il existe de nombreux outils système autres que ceux présentés permettant de tracer l'activité d'un système de type Linux : Kernel marker, DProbes, LTTng, strace, Dtrace, ProbeVue. Le tableau dans l'annexe A donne les principales caractéristiques qui nous intéressent, les possibilités des différents outils,... ce qui justifie nos choix.

5.4.1 Monitoring des applicatons via SystemTap

Les questions qu'un administrateur système se posent sont, entre autre chose, les points suivants :

- Quel processus fait le plus grand nombre d'opérations de lecture/écriture sur mon serveur ?
- Puis-je ajouter des instructions de débogage dans le noyau sans la reconstruction et le redémarrage du système ?

Les options possibles sont alors les mécanismes de :

- `Tracing` : fournit des informations lors de l'exécution du système et donne un aperçu rapide des échanges entre les codes exécutés ; donne beaucoup d'informations. Des outils comme `strace`, `ltrace` et `ftrace` sont utilisés pour le traçage.
- `Profilage` : est-ce que l'échantillonnage a lieu lors de l'exécution, pouvons-nous faire l'analyse après la collecte des événements ? `Oprofile` est par exemple utilisé pour l'échantillonnage.
- `Débogage` : avec cette technique nous pouvons définir des points d'arrêt, examiner les variables, la mémoire, les registres, nous pouvons tracer la pile, etc. Nous pouvons déboguer un seul programme à la fois et le débogueur arrête le programme alors que nous faisons l'inspection. GDB/KDB/LLDB peuvent être utilisés pour cette opération.

SystemTap est un outil permettant de combiner les trois précédents points. SystemTap fournit une interface de ligne de commande et un langage de script pour examiner les activités d'un système fonctionnant sous Linux, en particulier le noyau, dans les moindres détails. Les scripts Systemtap[Sys] sont rédigés dans un langage intermédiaires, ils sont ensuite compilés en des modules noyau et insérés dans le noyau. Les scripts peuvent être conçus pour extraire, filtrer et synthétiser les données, permettant ainsi le diagnostic des problèmes de performance ou des problèmes fonctionnels complexes. SystemTap fournit des informations similaires à la sortie d'outils comme netstat, ps, top, et iostat. Cependant, SystemTap fournit plus d'options de filtrage et d'analyse que ces outils.

Chaque fois que vous exécutez un script SystemTap, une session SystemTap est démarrée. Elle consiste en un certain nombre de passes qui sont faites sur le script avant

qu'il ne soit autorisé à s'exécuter. Une des passe compile le script en un module noyau qui est chargé immédiatement. Dans le cas où le script a déjà été exécuté auparavant et qu'aucun changement en ce qui concerne les composants n'a eu lieu (par exemple, en ce qui concerne la version du compilateur, la version du noyau, le chemin bibliothèque, contenu du script), SystemTap ne compile pas le script à nouveau, mais utilise les fichiers *.c et *.ko du cache de SystemTap (/.systemtap). Le module est déchargé lorsque le script termine.

Exemple de script SystemTap : nous introduisons ici une partie du script que nous avons développé ce qui nous permet d'introduire les principales notions de SystemTap.

L'utilisation de SystemTap se réalise par l'intermédiaire de scripts (fichiers texte d'extension .stp par convention). Les scripts explicitent quels sont les types d'informations à recueillir, et ce qu'il faut faire une fois que l'information est recueillie. Les scripts sont écrits dans le langage de script de SystemTap qui est similaire à AWK et C. Pour la définition de la langue, voir http ://sourceware.org/systemtap/langref/.

L'idée essentielle derrière un script de SystemTap est de nommer des événements, et de leur accoler des gestionnaires d'événements (des `handlers` dans la terminologies Unix). Lorsque SystemTap exécute le script, il surveille certains événements. Lorsqu'un événement se produit, le noyau Linux active le gestionnaire en tant que sous-routine, puis recommence l'attente d'un événement. Ainsi, les événements servent de déclencheurs pour les gestionnaires à exécuter. Les gestionnaires peuvent enregistrer des données spécifiées (pour cumuler le nombre de fois où apparaît un certain événement par exemple) et imprimer à l'écran d'une certaine manière un résultat.

Le langage de SystemTap utilise seulement quelques types de données (entiers, chaînes de caractères et tableaux associatifs de ces précédent types), et des structures de contrôle (blocs, conditionnelles, boucles, fonctions). Les types ne sont pas déclarés et ils sont vérifiés automatiquement à l'exécution.

Les *Tapsets* sont des bibliothèques de sondes et de fonctions pré-écrites qui peuvent être utilisées dans les scripts SystemTap. Lorsqu'un utilisateur exécute un script de SystemTap, SystemTap vérifie les événements et les gestionnaires de la sonde du script à la bibliothèque de Tapsets. SystemTap charge ensuite les sondes et les fonctions correspondantes avant de traduire le script en fichiers .c et .ko.

Cependant, contrairement aux scripts SystemTap, les Tapsets ne sont pas destinés à l'exécution directe, ils constituent la bibliothèque à partir de laquelle d'autres scripts peuvent être définis. Ainsi, la bibliothèque de Tapsets est une couche d'abstraction conçue pour rendre plus facile, pour les utilisateurs, la tâche de définir des événements et des fonctions. Les Tapsets fournissent des alias utiles pour les fonctions que les utilisateurs peuvent vouloir spécifier comme événement. L'alias est la plupart du temps plus facile à se rappeler que les noms des fonctions spécifiques du noyau qui peuvent d'ailleurs varier entre versions du noyau.

Le morceau de script que nous commentons maintenant est le suivant :

```
1  #! /usr/bin/env stap
2  global reads, writes, total_io, total_io_w, total_io_r
3  probe vfs.read.return
4  {
5      if (pid() == $1)
6      {
7         reads[pid()] += bytes_read
8      }
9  }
10 probe vfs.write.return
11 {
12     if(pid() == $1)
13     {
14       writes[pid()] += bytes_written
15     }
16 }
17 /* Main probe and main program
18  *
19  * print IO processes every 5 seconds for pid() == $1
20  */
21 probe timer.s(5) {
22     {
23         w=writes[$1]
24         r=reads[$1]
25         total_io[$1] += w
26         total_io[$1] += r
27         total_io_w[$1] += w
28         total_io_r[$1] += r
29         delete reads
30         delete writes
31 }
32 probe begin {
33   print ("Collecting data... Type Ctrl-C to exit and display results\n")
34 }
35 probe end {
36         printf ("%16ds\t%10s\t%10s\t%10s\n", "Process", "KB-Read",
37                 "KB-Written", "i/o-Total")
38         printf("%16d\t%10d\t%10d\t%10d\n", $1, total_io_r[$1]/1024,
39                 total_io_w[\$1]/1024, total_io[$1]/1024)
40         delete reads
41         delete writes
42         delete total_io
43         delete total_io_w
44         delete total_io_r
45 }
```

Ce script permet de surveiller les activités des entrées/sorties sur le disque toutes les 5 secondes. `probe begin` est la fonction exécutée au début, `probe end` celle exécutée

à la fin. $return est une variable locale qui stocke le nombre réel d'octets lus et écrits par chaque processus à partir du système de fichiers. $return ne peut être utilisé que dans les *Probes* de retour (par exemple, vfs.read.return et vfs.read.return). La fonction `pid()` permet de rapporter l'identifiant du processus à surveiller. Il existe plusieurs autres fonctions, mais les plus utilisées sont : `uid()` quel utilisateur fait tourner ce code ? `execname()` quel est le nom de ce processus ? `probefunc()` dans quelle fonction sommes-nous ?

5.4.2 Monitoring des applications via SysStat

SysStat [1] est une collection d'outils de monitoring pour Linux. Ces outils ont pour noms `sar, sadf, mpstat, iostat, nfsiostat, cifsiostat, pidstat` et `sa`. Nous allons commenter l'outil `pidstat` qui fourni, pour un processus donné et éventuellement ses fils, des informations sur l'activité de ce processus en terme de ressources consommées (CPU, disque, mémoire, réseau).

Les différentes commandes explorent le pseudo système de fichier `/proc` et offrent une interface synthétique des informations disponibles dans ce système de fichier.

Rappelons que sur les systèmes du type Unix, `procfs` (process file system, système de fichiers processus en anglais) est un pseudo-système de fichiers (pseudo car dynamiquement généré au démarrage) utilisé pour accéder aux informations du noyau sur les processus. Le système de fichiers est souvent monté sur le répertoire `/proc`. Puisque `/proc` n'est pas une arborescence réelle, il ne consomme aucun espace disque mais seulement une quantité limitée de mémoire vive. La toute première version de `/proc` était conçue pour remplacer l'appel système ptrace utilisé pour tracer les processus.

La commande `pidstat` qui est lancée pour notre travail est la suivante :

```
pidstat -p "mypid" -h -u -d -T CHILD -r 4
```

L'option `-p` permet de sélectionner le processus pour lequel on applique les statistiques. `mypid` représente l'identificateur de ce processus.

L'option `-h` permet d'afficher toutes les activités à l'horizontale sur une seule ligne. Ceci a pour but de rendre l'analyse plus facile.

L'option `-u` permet de rapporter l'utilisation du processeur. Lors de la déclaration des statistiques pour des tâches individuelles, les valeurs suivantes sont affichées :

L'option `-d` permet de rapporter les statistiques des entrées/sorties. Les valeurs suivantes sont indiquées : kB_rd/s : le nombre de kilobytes lus par le processus à partir du disque par seconde, kB_wrs : le nombre de kilobytes que le processus a écrit ou doit

1. Voir http ://sebastien.godard.pagesperso-orange.fr

écrire sur le disque par seconde, kB_ccwrs : le nombre de kilobytes sur le disque dont l'écriture a été annulé par le processus (exemple : les fichiers cache).

L'option `-T CHILD` permet de préciser que les statistiques doivent être rapportées sur le processus sélectionné et tous ses fils.

L'option `-r 4` permet d'afficher le rapport des statistiques toutes les 4 secondes

Depuis la version 10.1.4 et un travail de Christophe Cérin, `pidstat` possède un mode qui lui permet d'écrire une synthèse des activités du processus inspecté à la réception d'un signal Unix. Cela nous est particulièrement utile dans notre cas comme nous le verrons dans le prochain paragraphe.

5.4.3 Utilisation des outils de monitoring dans RedisDG

Les deux techniques discutées ci-dessus (SystemTap et SysStat) peuvent être utilisées au choix dans RedisDG. En fait, c'est par exemple la classe Python `PidStatClass` qui permet de lancer le monitoring via PidStat comme l'indique le code suivant :

```
1. class PidStatClass:
2.    def __init__(self, pid):
3.      args=["/usr/local/bin/pidstat","-p",pid,"-h","-u","-d","-r","4"]
4.      p1 = subprocess.Popen(args,stdout=subprocess.PIPE,shell=False)
5.      preprocessed1, _ = p1.communicate()
6.      p1.stdout.close()
7.      pool.acquire()
8.      logging.info(preprocessed1)
9.      pool.release()
```

Une autre technique de monitoring consiste à utiliser les fichiers « log » via le module Python `logging`. Ce module définit des fonctions et des classes qui mettent en œuvre un système flexible de journalisation des événements pour les applications et les bibliothèques.

L'avantage principal d'avoir une API de journalisation fournie par un module de bibliothèque standard, c'est que tous les modules Python peuvent participer à la journalisation, de sorte que votre journal d'application peut inclure vos propres messages intégrés avec des messages de modules tiers. Le module `logging` fournit un grand nombre de fonctionnalités et de flexibilité permettant de sauvegarder des messages de type Debug, Info, Warning, Error ou Critical. Pour notre application, on souhaite avoir à chaque message, une indication de l'heure, et éventuellement de la date, afin de pouvoir par la suite utiliser ces fichiers log pour le traçage des courbes précisant les dates de début et de fin de l'exécution d'une tâche. Il est possible de le faire, une fois pour toute, de la façon suivante :

```
1.  #!/usr/bin/python
2.
3.  import logging
4.  import time
5.
6.  logging.basicConfig(
7.     filename='worker.log',
8.     level=logging.INFO,
9.     format='%(asctime)s %(levelname)s - %(message)s',
10.    datefmt='%d/%m/%Y %H:%M:%S',
11.    )
```

Le fichier log obtenu aura la forme suivante :

```
2014-05-02 17:08:03,974 - RedisDG - INFO - Worker publishes: i'm volunteer 365034
2014-05-02 17:08:03,985 - RedisDG - INFO - Worker 365034 receives the task: 3
```

5.5 Implémentation et émulation

RedisDG, est basé sur Redis et développé en Python. il est organisé selon les classes Python suivantes :

- ServerClass : il existe trois possibilités de serveurs ; un serveur principal pour le protocole Redis lui même, un serveur de données pour stocker les données d'entrée/sortie des applications, et un serveur de code pour récupérer le code nécessaire à l'exécution d'une application.
- DataManager : elle définit une instance de la classe ServerClass. Elle définit également les fonctions de chargement des fichiers sur un serveur Redis, pour l'exécution d'un code (binaire ou script).
- MachineClass : permet de définir les propriétés d'une machine (système d'exploitation, mémoire disponible, le type de processeur,...)
- BrokerClass, CoordinatorClass, WorkerClass, MonitorClass et CheckerClass définissent respectivement le comportement d'un *broker, coordinator, worker, monitor et checker*. Elles héritent de la classe MachineClass.

Dans cette section, nous détaillons les étapes de développement de notre système RedisDG. L'utilisateur peut soumettre un fichier XML décrivant l'application à exécuter. L'application est représentée par un graphe de dépendances. Chaque nœud dans le graphe représente une tâche et chaque arc entre deux nœuds représente une dépendance entre deux tâches. Un nœud est décrit en terme d'inputs, outputs et code à exécuter.

Nous fournissons un modèle/template de fichier XML, démontrant comment l'application doit être décrite. Ainsi, toutes les informations relatives à l'application à exécuter

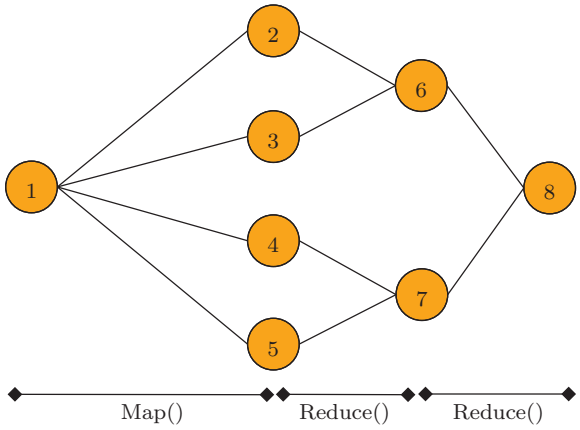


FIGURE *5.3 – Graphe de dépendance des tâches*

sont extraites directement à partir du fichier XML et automatiquement exposées à la structure interne du graphe. Actuellement, nous construisons automatiquement le fichier XML en utilisant la bibliothèque Python `xml.dom.minidom`. Dans le futur, nous envisageons d'avoir une interface graphique dans laquelle l'utilisateur pourra entrer les données pertinentes à l'exécution de son application. Ces données seront utilisées pour construire le fichier XML.

Pour l'émulation de notre système et le test des fonctionnalités, nous utilisons une application «MapReduce» qui compte le nombre d'occurrences de chaque mot dans un texte donné. Nous utilisons le terme d'émulation et pas simulation, car nous exécutons notre code sur une (seule) machine réelle, et non pas à travers un simulateur.

Cette application est représentée par un graphe de 8 nœuds et 10 arêtes. Elle fragmente les données d'entrée en quatre morceaux indépendants qui sont traités par l'opération Map. L'opération Reduce prend les morceaux par paires pour additionner les différentes occurrences de chaque mot. Le code utilisé pour l'opération de la fragmentation est du `Bash` et le code des opérations `Map` et `Reduce` est du `Python`. Notre système démarre en parallèle un Broker, Coordinator, Monitor et Checker ainsi que plusieurs instances de Worker.

Le Broker est géré par deux threads. Le premier prend le graphe en entrée, puis extrait les nœuds. Il associe à chaque nœud une liste de prédécesseurs. Ensuite, il publie tous les nœuds et les prédécesseurs sur le canal *WaitingTasks* auquel le Coordinator est à l'écoute. Ces relations de dépendance sont organisées dans un dictionnaire comme suit :

```
Dictionnaire={
Tache1 : {prédécesseur1,..,prédécesseurN} ;
Tache2 : {prédécesseur1,..,prédécesseurN} ;
```

```
Tache3 : {prédécesseur1,..,prédécesseurN} ;
...}
```

Le deuxième thread est en permanence à l'écoute sur le canal *Emergency* afin de détecter les signaux d'urgence. Si le message `STOP` est annoncé sur le canal alors le Broker est aussitôt notifié, et dans ce cas il tue tous les processus qui lui sont associés.

Le Coordinator est géré par trois threads. Le premier est utilisé pour rester à l'écoute d'un canal appelé *VolunteerWorkers* qui permet d'être notifié par les Workers volontaires et disponibles. Le Coordinator sélectionne les éventuels Workers et publie la sélection sur le canal *SelectVolunteers*. Cette fonctionnalité imite le protocole pour la sélection des Workers que l'on trouve dans n'importe quel système de grille de PCs.

Le second thread est utilisé pour rester à l'écoute du canal *WaitingTasks*, récupérer le dictionnaire des dépendances, l'analyser et publier les tâches indépendantes sur le canal *TasksToDo*. Il permet aussi de traiter les tâches terminées puisque le Coordinator est au préalable à l'écoute du canal *FinishedTasks*. À l'arrivée d'une nouvelle tâche sur ce canal le Coordinator met à jour le dictionnaire de tâches en attente en supprimant la tâche annoncée comme terminée de la liste des prédécesseurs des tâches en attente, dans le but de déceler de nouvelles tâches indépendantes.

Une fois que toutes les tâches sont terminées, le dictionnaire devient vide et le Coordinator publie l'événement `STOP` sur le canal *Emergency* afin d'annoncer la fin de l'application. Le troisième thread est en permanence à l'écoute sur le canal *Emergency* afin de détecter les signaux d'urgence.

En parallèle, des Workers sont lancés. Un Worker est également géré par trois threads. Le premier publie l'identité du Worker sur le canal *VolunteerWorkers* afin d'annoncer son volontariat quand il est libre. Le deuxième est à l'écoute au préalable sur le canal *TasksToDo* afin d'être notifié par de nouvelles tâches à exécuter. Il est aussi à l'écoute sur le canal *SelectVolunteers* dans le but de détecter si le Worker a été sélectionné pour démarrer une exécution. Dans l'affirmative, il lance l'exécution du code respectif qui est téléchargé à partir du serveur de code Redis. Le serveur Redis se trouve quelque part dans le Web, il n'est pas localisé sur la même machine utilisée pour l'émulation. Le troisième thread est réservé à l'écoute sur le canal *Emergency* afin de détecter les signaux d'urgence.

Une fois l'exécution de la tâche commencée, l'événement est publié par le Worker dans le canal *TasksInProgress*. L'identifiant de la tâche, l'identifiant du processus et l'adresse IP du Worker sont publiés sur le canal *TasksInProgress* afin de permettre la surveillance du déroulement de l'exécution de la tâche. Lorsque l'exécution est terminée, le Worker publie la tâche correspondante sur le canal *TasksToCheck* afin de lancer la procédure de certification de résultat pour cette tâche.

Pour le Monitor, nous créons un processus de demande de surveillance et un processus d'arrêt de la surveillance. Le premier est à l'écoute sur le canal *TasksInProgress*.

Quand un événement est publié dans ce canal, le Monitor extrait l'adresse IP du Worker et vérifie si le nœud est en vie par un *ping* toutes les 2 secondes pendant toute la durée de l'exécution de la tâche. Dans le cas où un Worker ne répond pas à une requête *ping*, le Moniteur publie la tâche correspondante dans le canal *TasksToDo* pour qu'elle soit attribuée à un autre Worker. Le second processus décide quand est-ce qu'il doit arrêter la surveillance. Il est à l'écoute sur le canal *TasksToCheck* afin d'être averti par les tâches qui sont déjà terminées, et tuer le processus de surveillance correspondant. Un autre thread est réservé à l'écoute sur le canal *Emergency* afin de détecter les signaux d'urgence.

D'autre part, le comportement et les résultats produits par les volontaires doivent être examinés pour détecter des actions inappropriées et/ou des résultats erronés. D'ailleurs, des résultats incorrects peuvent être dues à un dysfonctionnement matériel ou logiciel, une mauvaise intention des utilisateurs, ou une combinaison de tous ces facteurs. Ainsi, dans un tel environnement hostile, il faut assurer la fiabilité des calculs effectués.

Le Checker a pour rôle de vérifier/certifier les résultats. En fait, plusieurs techniques de certification des résultats existent dans la littérature. Ces techniques ont été classées dans le livre [CF12] au niveau du chapitre 10 intitulé *Security an Result Certification*. Dans RedisDG, nous avons choisi d'implémenter la technique par duplication qui s'inspire de la méthode de « vote à la majorité ». Le principe consiste à dupliquer une tâche un certain nombre de fois afin de l'exécuter sur des Workers différents et parvenir à comparer les résultats retournés.

Dans RedisDG, nous procédons à la duplication du graphe de tâches *k>1 fois*. Comme le montre la figure 5.4, le graphe de tâches présenté précédemment, présentant le job MapReduce, est dupliqué deux fois. Le Checker, quand il reçoit une tâche, s'attend à recevoir les duplicatas de cette tâche. Ainsi, Il construit un dictionnaire pour chaque ensemble de tâches similaires. Nous utilisons la bibliothèque `hashlib` pour calculer la valeur `MD5` du fichier résultat d'une tâche qui est renvoyé par un Worker. Ainsi, le dictionnaire des duplicatas est construit comme suit :

```
Dictionnaire={
FichierResultatTache1  : MD5 ;
FichierResultatTache1' : MD5 ;
FichierResultatTache1" : MD5 ;
...}
```

Une fois l'ensemble des duplicatas d'une même tâche reçu par le Checker, ce dernier peut commencer la comparaison des résultats en procédant à la comparaison des MD5. Si le résultat retourné par tous les Workers est le même, alors on suppose que le résultat est correct et la tâche sera publiée sur le canal *FinishedTasks*. Sinon, le résultat est

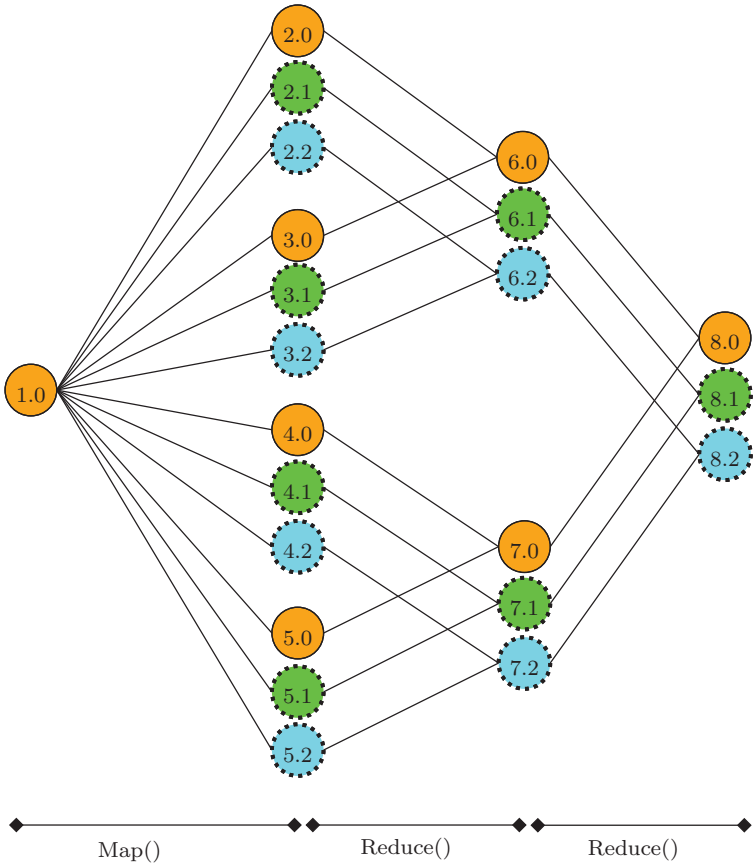


FIGURE *5.4 – Duplication du graphe de dépendance des tâches k fois : k=2*

considéré comme incorrect, et la tâche correspondante est republiée à nouveau sur le canal *TasksToDo*. Le checker réserve un autre thread en permanence à l'écoute sur le canal *Emergency* afin de détecter les signaux d'urgence.

5.6 Intégration de RedisDG dans le Cloud SlapOS

Dans cette partie, nous décrivons l'expérience d'intégration (cloudification) faite avec SlapOS. Cette expérience permet de dégager les aspects avantageux des concepts clés de SlapOS mais aussi les difficultés techniques et conceptuelles rencontrées. Toutefois, le constat est qu'au final SlapOS est un système de Cloud générique permettant l'intégration de nombreux types d'applications [CT13]. Bien entendu ces applications doivent pouvoir être accessibles via le réseaux Internet. L'idée qui nous guide est d'offrir un service de grille de PC déployable à la demande.

Un système de type intergiciel de grille de calcul est basé sur le volontariat. Il permet de lancer des applications scientifiques composées de plusieurs tâches et exécutées en parallèle sur différentes machines. Ces systèmes sont basés sur une architecture client serveur ; les tâches sont gérées par le serveur qui va les répartir sur un ensemble de clients volontaires, Workers, ayant souhaités participer au projet. Chaque client va ensuite remonter les résultats au serveur une fois sa tâche terminée.

La particularité de ces systèmes est d'être dynamique c'est-à-dire qu'ils peuvent modifier leurs comportements au cours de leur fonctionnement. Ces modifications sont le plus souvent liées à la soumission d'une nouvelle tâche, ou l'arrêt d'une autre. La soumission d'une tâche nécessite l'exécution d'une succession de commandes particulières et le téléchargement d'un ou de plusieurs fichiers. Notre application ne fournit pas un portail Web permettant de faire l'administration à distance. L'idéal serait de fournir un terminal avec l'instance, accessible par exemple avec une connexion SSH, mais cela reste un cas moins intéressant car nous ne saurons pas, dans ce cas, bien limiter les accès de l'utilisateur dans la partition. Dans un environnement comme SlapOS, on est donc contraint d'automatiser toutes les actions qui pourront être réalisées pour l'utilisateur (soumission du travail, arrêt de la tâche, etc.). Ainsi on devrait être en mesure de créer une instance RedisDG avec des tâches initiales puis, contrôler et modifier l'instance au cours de son fonctionnement pour que d'autres placements de tâches puissent se faire sur les volontaires, ceci via SlapOS.

Le modèle de déploiement des applications dans SlapOS est basée sur une architecture `Components, Stack, Software` comme le montre la figure 5.5 et comme cela a était présenté dans la figure 3.6 du chapitre 3. Les `Components` sont tous les composants et toutes les dépendances dont on peut avoir besoin pour compiler l'application. La `Stack` permet de faire un module générique pour le déploiement d'un type d'instance précis. Dans le fichier (`Software.cfg`), on indiquera alors comment seront exploités les composants et la stack et les recettes pour le déploiement de application.

Le fichier `Software.cfg` est le profil principal permettant de compiler, télécharger les fichiers nécessaires et installer l'application avec toutes les dépendances, il crée le `Software Release`. Le profil de l'instance appelée `instance-redisdg.cfg` décrit, à l'aide des recettes, le déploiement d'une instance de l'application dans une partition SlapOS. C'est le fichier le plus important car, il permet de définir le type d'instance demandée (coordinateur ou worker), déployer le serveur Redis lorsque cela est nécessaire, et aussi démarrer le démon RedisDG.

Le déploiement de RedisDG dans SlapOS est réalisé grâce à une nouvelle recette appelée `slapos.cookbook:redis.dg` (voir figure 5.6). Elle est écrite en python.

À la fin du déploiement, des scripts exécutables sont générés permettant ainsi de démarrer le démon RedisDG avec des paramètres définis par l'utilisateur. Chaque script appelé `wrapper` est exécuté automatiquement par SlapOS et il est sous le contrôle d'un démon spécifique appelé `Watchdog`. Le rôle du `Watchdog` est de garantir le fonctionne-

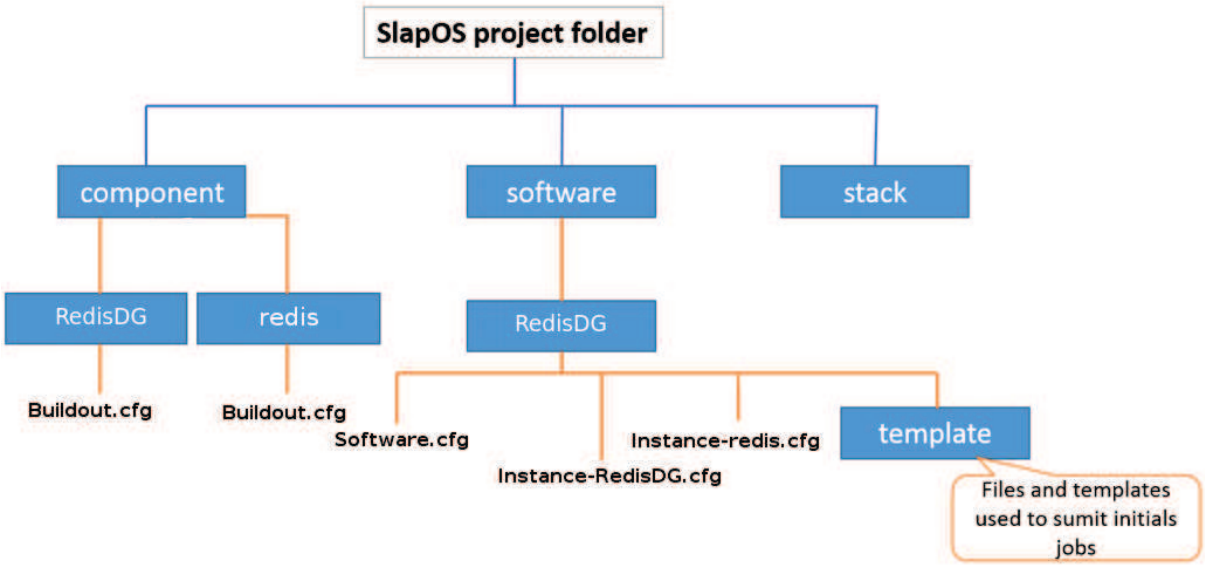


FIGURE *5.5 – Arbre du projet d'intégration de RedisDG dans SlapOS*

ment de l'instance sans aucune interruption involontaire.

La figure 5.6 montre une partie du fichier utilisé pour le déploiement de RedisDG. À la ligne 2 nous définissons le recette utilisée pour le déploiement de l'application, toutes les lignes en dessous présentent les paramètres envoyés à la recette. La ligne 6 spécifie le chemin du `wrapper` à générer. La ligne 14 spécifie le type de l'instance à déployer. La ligne 15 définie l'adresse du serveur Redis. La ligne 13 présente un objet `JSON` utilisé pour décrire le travail à soumettre. La structure de cet objet est comme suit :

```
{"config":PATH_OR_URL,
    "files":{FILE_NAME:PATH_OR_URL,
    FILE_NAME:PATH_OR_URL, ...}
}
```

Dans cet objet, `config` est le chemin ou l'URL du fichier XML utilisé pour décrire notre application. `files` désigne la liste des fichiers d'entrée nécessaires à l'exécution de l'application et décrits dans le fichier XML. Ainsi, du point de vue de l'utilisateur, déployer une instance RedisDG nécessite deux paramètres : un pour le type de l'instance demandée et l'autre pour la description du travail comme un objet `JSON`.

Pour conclure, nous insistons ici que tout le travail présenté ci-dessous est de la responsabilité de l'utilisateur. Ce travail d'intégration est réalisé par l'ingénieur intégrateur, une fois pour toute. Ainsi l'utilisateur n'a qu'à utiliser l'intergiciel RedisDG fournit comme étant un service dans le Cloud SlapOS.

```
1.  [redis-dg]
2.  recipe = slapos.cookbook:redis.dg
3.  python-bin = ${buildout:executable}
4.  #Please provide here the main script of redis-dg
5.  redisdg-script = ${redis-dg:location}/main.py
6.  wrapper = $${basedirectory:services}/redis_DesktopGrid
7.  root-dir = $${buildout:directory}
8.  work-directory = $${rootdirectory:srv}
9.  tmp-dir = $${rootdirectory:tmp}
10. log-file = $${basedirectory:log}/redisDG.log
11. pid-file = $${basedirectory:run}/redisDG.pid
12. #The list of files used in redisDG, one per line
13. job-desc = $${slap-parameter:job}
14. deamon = $${slap-parameter:deamon}
15. redis = $${redis:ipv6}
17.
18. eggs-directory = ${buildout:eggs-directory}
```

FIGURE *5.6 – Le déploiement*

Bien que le Cloud SlapOS facilite le déploiement à large échelle, il peut tout de même exister des risques d'échec du déploiement ou de perte de contrôle dûe à l'automatisation. Pour minimiser les risques, SlapOS propose des mécanismes permettant de contrôler que le service a été correctement déployé. Un script appelé `promesse` sera déposé dans un emplacement spécifique de la partition. Le script est écrit par l'intégrateur et selon une convention bien définit et il a pour rôle de tester que tous les services sont bien fonctionnels (par exemple, une vérification qu'une URL est bien accessible). Il sera ensuite exécuté par SlapOS pour vérifier que le déploiement s'est bien passé. Si ce n'est pas le cas, l'instance sera redéployée à nouveau, dans le but de redémarrer ou de redéployer les services qui n'ont par réussi leur précédente tentative de déploiement.

Il restera donc à être capable de consulter les fichiers log d'une application déployée pour effectuer une vérification du bon fonctionnement ou pour des besoins d'administration. Bien que ceux-ci soient générées lors du fonctionnement des services, un utilisateur externe ne peut consulter les log sans accéder directement à la partition. On a donc une difficulté de suivi pour les applications qui ne sont pas utilisables en ligne de commande comme dans le cas d'un site Web par exemple (pour résoudre cela, on pourra déployer avec notre application une console Web tel que Shelinabox disponible dans le Cloud SlapOS, permettant d'avoir un terminal depuis son navigateur web).

5.7 Conclusion

Alors que la plupart des approches de la littérature [CF12] ont eu besoin par le passé de construire un certain nombre de couches avant la couche propre à l'intergiciel de grille de PCs, notre système est conçu et développé en une seule couche. Ainsi, nous repoussons les pertes en efficacité, vitesse et facilité de mise en œuvre. Notons aussi que nous pouvons avoir plusieurs serveurs Redis ce qui présente un avantage au niveau des propriétés d'équilibrage de charge.

Nous proposons un système avancé de grille de calcul, capable de fonctionner sur les smartphones et les tablettes en plus des machines traditionnelles. En fait, d'après notre expérience dans le domaine, l'intégration des intergiciels de grille de calcul existants sur les petits appareils peut être extrêmement difficile, parce que ce genre de système n'était pas conçu à la base pour intégrer ce type d'appareils présentant une configuration assez spécifique. D'où le besoin de repenser les techniques de coordination, d'interaction, d'exécution et de stockage de données dans les intergiciels de grille de calcul en termes de nouvelles technologies qui soient beaucoup plus cohérentes avec les nouveaux appareils mobiles.

Nous tenons à préciser que notre travail se concentre plutôt sur la coordination des différents composants d'une grille de calcul, et non pas sur l'algorithme d'ordonnancement des tâches. Par conséquent, nous pouvons utiliser n'importe quel algorithme d'ordonnancement comme PAPS (Premier Arrivé Premier Servi) par exemple.

Il est important d'insister sur le fait que le protocole est entièrement basé sur l'échange de messages et que le développement de RedisDG a été guidé par nos différents travaux de modélisation. Notre système sert à la validation des idées et des choix effectués lors de la partie de la conception et de la modélisation.

Enfin, nous adoptons un point de vue centré sur l'utilisateur en considérant que la technologie des grilles de calcul devrait être aussi simple que possible dans son utilisation. D'où le travail de l'intégration de RedisDG dans le cloud SlapOS afin de pouvoir offrir les fonctionnalités d'une grille de PCs comme service Web.

Ce travail a fait l'objet de quelques publications internationales dans les conférences SAC (*28th ACM Symposium on Applied Computing, 2013*) [ADCJ13] et GPC (*International Conference on Grid and Pervasive Computing, 2013*) [ACJ13].

Chapitre 6

Validations expérimentales avec Grid'5000, SlapOS, RedisDG et Pegasus

6.1 Introduction et motivations

L'expérimentation nécessite souvent l'exécution à grande échelle, la simulation en multi-étapes et des pipelines d'analyse de données pour permettre l'étude des systèmes complexes. La quantité de calcul et de données impliqués dans ces pipelines nécessite des systèmes de gestion de workflows (on préférera ici le terme workflow plutôt que flux de travail) scalables qui soient en mesure de coordonner de manière fiable et efficace et d'automatiser le transfert de données et l'exécution des tâches sur des ressources de calcul distribuées [IJT06, JQ12]. Pour cela, on peut combiner des infrastructures d'exécution faites de clusters de campus, des cyber-infrastructures nationales, des Clouds commerciaux et universitaires.

Les technologies de workflows sont responsables de la planification des tâches de calcul sur les ressources distribuées, de la gestion des dépendances entre les tâches, et la mise en scène des données définies dans et hors les sites d'exécution [JQ12]. Comme les workflows scientifiques gagnent en complexité et en importance, les concepteurs de systèmes de gestion de workflows ont besoin d'une compréhension plus profonde et plus large de ce que demande un workflow en termes de ressources et comment il se comporte dans le but d'améliorer les algorithmes pour l'approvisionnement des ressources, l'ordonnancement des tâches, et la gestion des données [DGST09].

Les applications de workflows open-source ne sont pas nombreuses, car, souvent, les utilisateurs de workflows sont réticents à libérer leur codes et leur données [JCD+13]. Une exception est le workflow astronomique «MONTAGE» [Mon], qui a été largement utilisé pour évaluer les algorithmes et les systèmes de workflows. Bien que MONTAGE

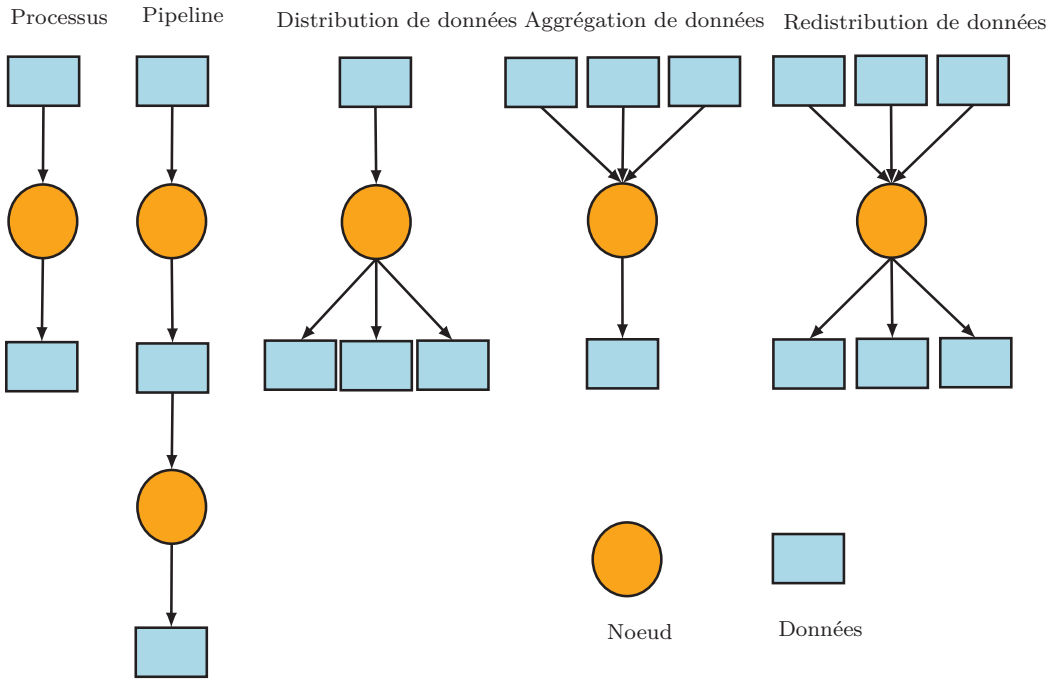


FIGURE *6.1 – Les structures de base d'un workflow*

se soit avéré être une ressource très utile pour la communauté, les systèmes de workflow doivent être destinés à un usage général et ne doivent pas être conçus et évalués sur la base d'un workflow unique.

6.2 Qu'est ce qu'un workflow scientifique ?

Un workflow décrit les dépendances entre les tâches et dans la plupart des cas, il est décrit comme un graphe orienté acyclique (DAG-*Directed Acyclic Graph*), où les nœuds sont les tâches et les arcs indiquent les dépendances entre les tâches. Les workflows scientifiques [IJT06, JQ12] permettent aux utilisateurs d'exprimer facilement les tâches de calcul en plusieurs étapes, par exemple récupérer des données à partir d'un instrument ou une base de données, re-formater les données et exécuter une analyse.

La figure 6.1 montre une partie des structures de base d'un workflow scientifique. Le workflow final est généralement composé de plusieurs de ces composants [BCD+08a].

La structure la plus simple est la structure de *processus* qui fonctionne sur des données d'entrée pour produire des données de sortie. Plusieurs de ces *processus* de données peuvent être combinés de manière séquentielle pour produire la structure de *pipeline*. Dans ce cas, chaque tâche dans le pipeline fonctionne sur la sortie du niveau

précédent et la sortie produite est envoyée en entrée au niveau suivant dans le pipeline.

Des nœuds de *distribution de données* servent à deux fins : ils peuvent soit produire des données de sortie qui sont consommées par plusieurs nœuds ou ils peuvent fonctionner sur de grands ensembles de données et les partitionner en petits sous-ensembles à traiter par d'autres nœuds dans le workflow. La distribution des données appelée aussi le *partitionnement de données*, est un concept assez fréquent dans les workflows. Si le partitionnement des données implique en plus le calcul, les nœuds peuvent consommer beaucoup de temps sur la ressource de calcul. Cependant, le partitionnement entraîne une augmentation du parallélisme dans les niveaux ultérieures du workflow, et c'est en effet la raison principale du partitionnement des données.

Les nœuds *d'agrégation de données* regroupent et traitent les sorties de plusieurs nœuds et génèrent un produit de données combinées. Comme les nœuds d'agrégation de données fonctionnent sur plusieurs entrées de données individuelles, ils peuvent potentiellement consommer beaucoup de temps sur les ressources de calcul. En outre, ces nœuds peuvent représenter une réduction du parallélisme du workflow. Dans certains cas, les données agrégées à partir d'une étape précédente sont *redistribuées* à plusieurs nœuds dans l'étape suivante. Même si le nœud de *redistribution de données* représente un goulot d'étranglement potentiel, le parallélisme est de nouveau augmenté dans les étapes ultérieures. Ces nœuds de redistribution des données existent dans plusieurs workflows scientifiques et représentent un point de synchronisation pour le traitement de données.

6.2.1 Défis de planification de workflows

Les défis en terme de planification de workflow peuvent être divisés en deux parties : la faisabilité et la performance. Dans le premier cas, nous devons trouver une instance de workflow qui soit correctement exécutable, qui identifie les ressources et les données nécessaires et qui gère les données obtenues en les mettant en scène sur les emplacements de stockage appropriés. De toute évidence, des défauts dans l'environnement d'exécution peuvent encore se produire, mais le workflow exécutable généré par le processus de planification doit être correct, afin de minimiser les erreurs lors de l'exécution [BCD+08a].

Les planificateurs de workflow peuvent également optimiser le workflow du point de vue de la performance par ordonnancement des tâches individuelles et l'ensemble du workflow sur les ressources d'une manière qui optimise les performances de workflow global. On peut aussi augmenter la performance de l'ensemble du workflow en ordonnançant soigneusement les parties critiques du workflow. Dans les workflows traitant des données intensives, il est important d'ordonnancer le calcul proche des données.

6.2.2 Défis d'exécution de workflow

L'un des principaux défis dans l'exécution du workflow est la capacité de collecter des informations et faire face aux défaillances. Les échecs peuvent se produire parce que les ressources se volatilisent, les données deviennent indisponibles, les connexions réseaux s'interrompent, des bugs du côté logiciel du système ou dans les composants de l'application apparaissent, ainsi que de nombreuses autres causes [BCD+08a].

Un autre défi dans l'exécution du workflow est la collecte de l'information nécessaire pour identifier les causes des défaillances de sorte qu'elles puissent être évitées dans les exécutions à venir. Il s'agirait notamment de placer des composants de surveillance à tous les niveaux de l'exécution du workflow, à partir du moteur de workflow jusqu'aux composants logiciels du système et l'application. Depuis que les workflows s'exécutent dans des environnements très hétérogènes et reposent sur des piles de logiciels multi-niveaux, la collecte et l'interprétation de cette information est très difficile.

L'instrumentation du processus d'exécution peut également être utile pour fournir l'information sur la provenance et les performances sur les composants du workflow et le workflow entiers. Les informations de provenance, qui spécifient comment les données particulières ont été produites (quel logiciel a été utilisé, les données d'entrée), sont nécessaires pour les scientifiques afin d'être en mesure d'interpréter, de valider et de reproduire les résultats. Les informations sur la performance sont utiles pour les développeurs de composants d'application afin d'identifier les problèmes avec des algorithmes particuliers.

Les systèmes d'exécution de workflow peuvent également améliorer les performances de workflow et soutenir l'ordonnancement effectuée par le planificateur de workflow en prévoyant des fonctionnalités de réservation de ressources. La réservation de ressources de calcul avant l'exécution, élimine certaines des incertitudes associées à l'ordonnancement des composants de workflow sur les ressources. Dans le cas de ressources réservées, les tâches de workflow subissent moins de retard dans les files d'attente des systèmes d'exécution à distance. Ces retards sont également indépendants des tâches soumises par d'autres utilisateurs de la ressource.

6.3 Pegasus : système de gestion de workflows

6.3.1 Introduction

Le projet Pegasus [Peg, DVJ+14] englobe un ensemble de technologies qui contribuent à l'exécution des applications basées sur les workflows dans différents environnements, tels que les ordinateurs de bureau, les clusters de campus, les grilles et les Clouds. Pegasus localise automatiquement les données d'entrée et les ressources de calcul nécessaires à l'exécution du workflow. Il permet aux scientifiques de construire des

workflows de manière abstraite, sans se soucier des détails de l'environnement d'exécution sous-jacent ou des détails des spécifications de bas niveau requis par le middleware (Condor). Pegasus offre également à la communauté actuelle une coordination efficace des multiples ressources distribuées.

Pegasus a été utilisé dans un certain nombre de domaines scientifiques, y compris l'astronomie, la bio-informatique, la sismologie, la physique des ondes gravitationnelles, l'océanologie, et bien d'autres encore. Lorsque des erreurs se produisent, Pegasus tente de récupérer l'erreur si possible en relançant les tâches, en relançant l'ensemble du workflow, en fournissant des points de reprise au niveau du workflow, en essayant des sources de données alternatives, et quand tout cela échoue, en fournissant un workflow de secours contenant une description de seulement le travail qui reste à faire.

Il libère également les ressources de stockage quand l'exécution du workflow est terminée afin que les workflows ayant des grands volumes de données aient suffisamment d'espace pour s'exécuter sur les ressources de stockage limitées. Pegasus assure le suivi de ce qui a été fait, y compris les emplacements des données utilisées et produites, et quel logiciel a été utilisé avec quels paramètres.

Comme cela est présenté à la figure 6.2, les actions précédentes sont réalisées par les cinq sous-systèmes de Pegasus suivants :

- Le **Mapper** : il génère un workflow exécutable basé sur un workflow abstrait fourni par l'utilisateur ou le système de composition des workflows. Il cherche les logiciels, les données et ressources informatiques appropriées nécessaires à l'exécution du workflow. Le Mapper peut aussi restructurer le workflow pour optimiser les performances et apporte des transformations au gestionnaire de données.

- **Local Execution Engine** : il soumet les tâches définies par le workflow dans l'ordre de leurs dépendances. Il gère les tâches par le suivi de leurs états et détermine le démarrage d'exécution de chaque tâches. Il soumet ensuite les tâches à la file d'attente de l'ordonnanceur local.

- **Job Scheduler** : il gère les tâches du workflow, supervise leurs exécutions sur les ressources locales et distantes.

- **Remote Execution Engine** : il gère l'exécution d'une ou plusieurs tâches ; il peut être structuré comme un sous-workflow sur un ou plusieurs nœuds de calcul distant.

- **Monitoring component** : c'est un démon de surveillance de l'exécution lancé lorsque le workflow commence l'exécution. Il surveille le workflow en cours d'exécution, analyse les historiques des tâches et les sauvegarde dans une base de données. Les bases de données contiennent à la fois l'information sur les performances et sur la provenance des données. Il envoie également des notifications à l'utilisateur lui indiquant l'échec, le succès ou l'achèvement d'une tâche.

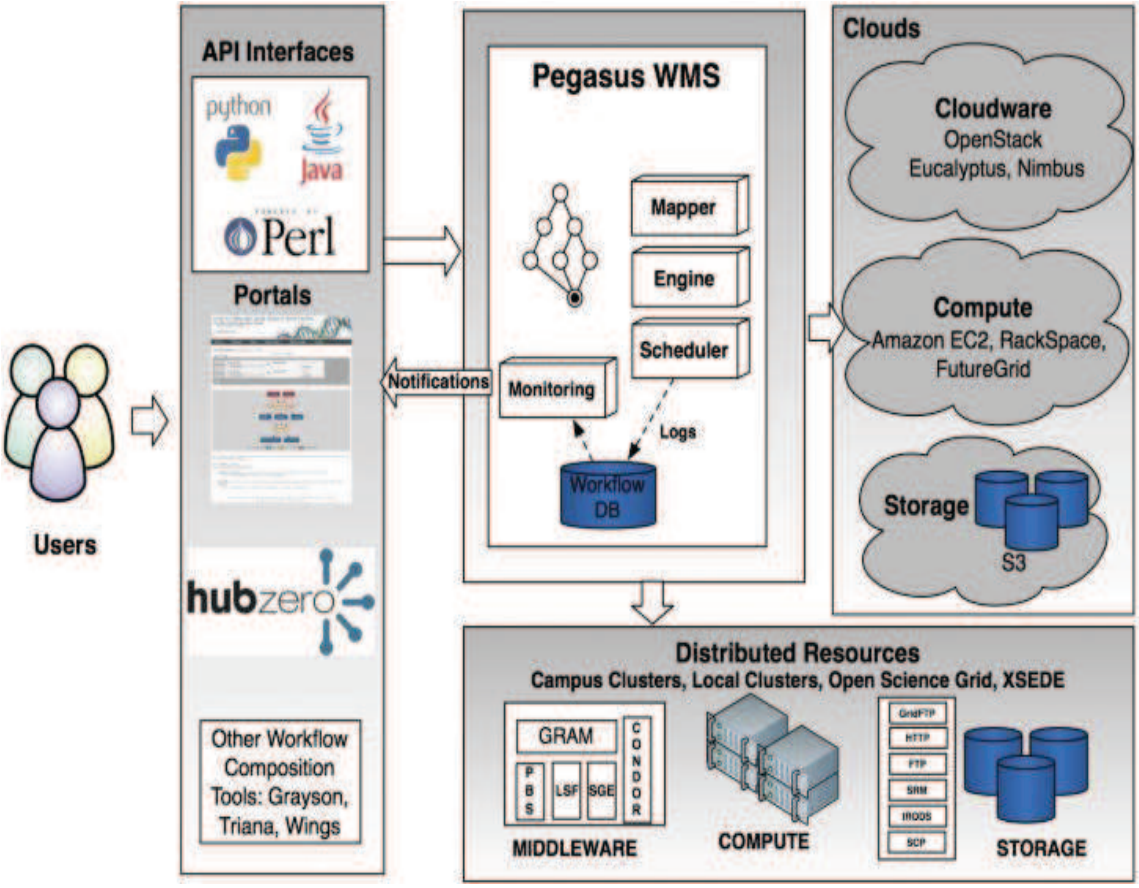


FIGURE *6.2 – Architecture de Pegasus*

6.3.2 Le workflow MONTAGE est utilisé pour le cas test

Le projet MONTAGE [Mon] a été créé par la NASA/IPAC *Infrared Science Archive* comme une boîte à outils open source qui peut être utilisée pour générer des mosaïques personnalisés du ciel à partir d'images d'entrée de format *Flexible Image Transport System* (FITS). Au cours de la production de la mosaïque finale, la géométrie de la sortie est calculée à partir de la géométrie de l'image d'entrée. Les images d'entrées sont ensuite re-projetées pour être dans la même échelle et rotation spatiales. Les émissions d'arrière-plan dans les images sont ensuite corrigées pour être du même niveau dans toutes les images. Les images re-projetées et corrigées sont fusionnées pour former la mosaïque finale [BCD+08a, JCD+13]. L'application MONTAGE a été représentée par un workflow qui peut être exécuté dans des environnements tels que la grille Teragrid [TER].

Dans la figure 6.3, nous montrons un workflow de MONTAGE relativement petit (20 nœuds) généré par le générateur de workflow [BCD+08b]. Le nombre d'entrées traitées par le workflow peut augmenter au fil du temps tant que d'autres images d'une

région particulière du ciel sont disponibles. À ce titre, la structure des changements du workflow pour recevoir l'augmentation du nombre d'entrées implique également une augmentation du nombre de nœuds de calcul.

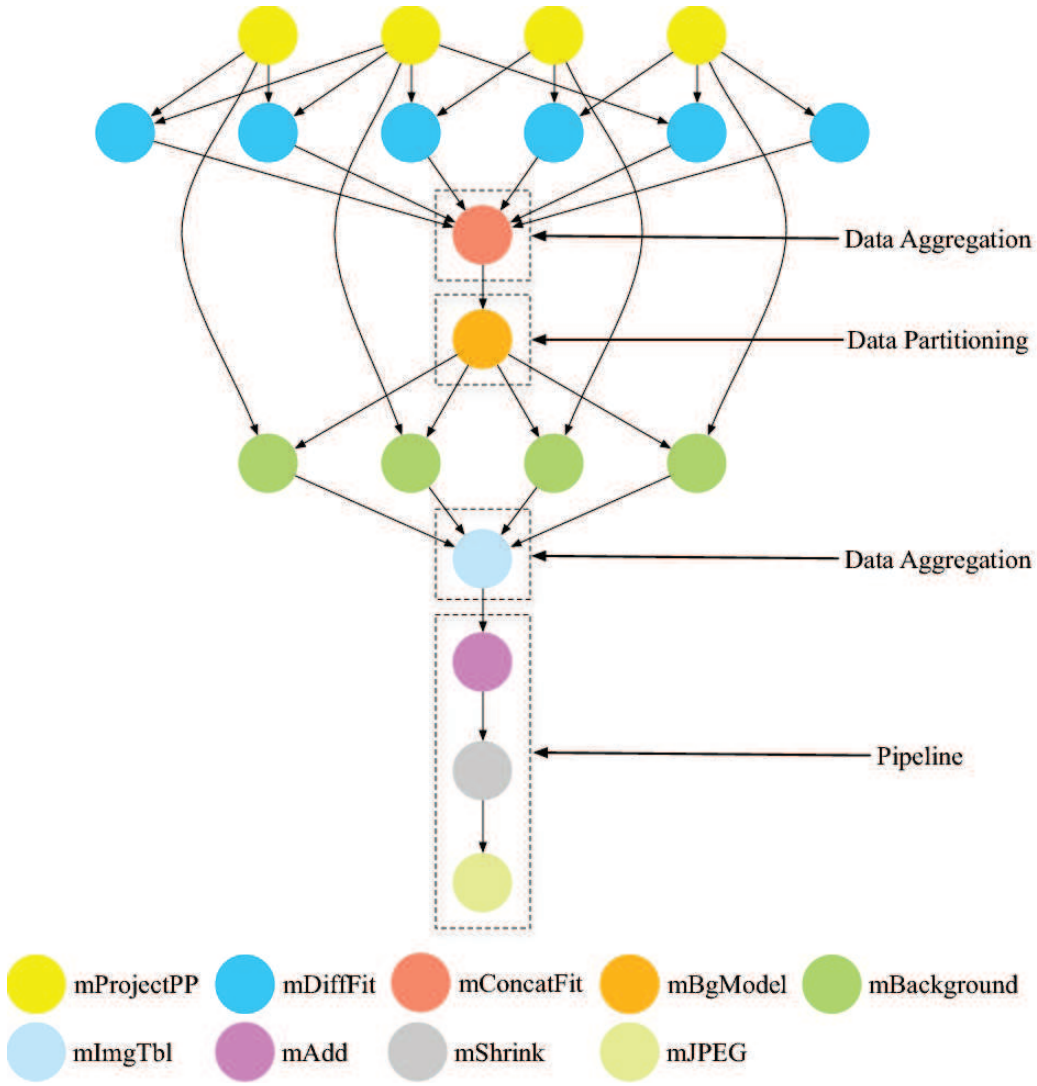


FIGURE *6.3 – Le workflow de MONTAGE*

Le nombre de tâches `mProjectPP` (qui projettent l'image en entrée) est égal au nombre d'images FITS traitées. En sortie, on obtient l'image projetée ainsi qu'une image de zone, dite 'area', celle-ci étant la fraction de l'image qui appartient à la mosaïque finale. Elles seront par la suite traitées ensembles en étapes successives. Une tâche `mDiffFit` calcule une différence pour chaque pair d'images qui se chevauchent. Le nombre de tâches `mDiffFit` dans le workflow dépend ainsi du chevauchement de

l'image.

Les différentes images sont traitées selon un algorithme utilisant la méthode des moindres carrés par tâche `mConcatFit`. La tâche `mConcatFit` correspond à la description d'une tâche d'agrégation de données, tout en étant une tâche de calcul intensif. Ensuite, une tâche `mBgModel` doit déterminer une correction qui est à appliquer à chaque image afin d'obtenir une bonne correspondance globale. Les tâches `mBackground` appliquent la correction en arrière plan à chaque image à part. Les tâches `mConcatFit` et `mBgModel` sont respectivement des tâches d'agrégation et de partition de données. Cependant, elles peuvent être considérés comme un point de redistribution de données. Dans ce cas, il n'y aura pas beaucoup de données partitionnées. Ainsi, la même correction en arrière plan est appliquée à toutes les images.

La tâche `mImgTbl` agrège les données à partir de toutes les photos et créé une table qui peut être utilisée par d'autres tâches dans le workflow. Ainsi, elle représente une simple étape d'agrégation de données. La tâche `mAdd` ajoute toutes les images reprojetées afin de générer la mosaïque finale en format FITS ainsi qu'une image de zone qui peut être utilisée dans un calcul ultérieur. La tâche `mAdd` est la plus intensive en terme de calcul dans le workflow. La taille de l'image FITS est réduite par la tâche `mShrink` en calculant des moyennes de blocks de pixels. L'image réduite est par suite convertie en format JPEG par la tâche `mJPEG`.

6.4 Mise au point et exécution de MONTAGE avec RedisDG

Jusque là, nous avons conçu, implémenté, testé et simulé notre système RedisDG. Pour s'assurer de son fonctionnement, nous avons utilisé soit des applications « jouets » soit des applications simples ne nécessitant pas une grande puissance de calcul. À ce stade, où notre système est fonctionnel et vérifié formellement, faire tourner une application réelle s'impose. Le fait de disposer de données volumineuses, d'un vrai flux de données qui circule et un vrai code à exécuter, nécessitant des données d'entrée et générant des données de sortie, permet de mieux affirmer RedisDG en tant qu'intergiciel de grille de PCs mais aussi en tant que gestionnaire de workflows scientifiques.

Comme mentionné dans le chapitre précédent, l'application à exécuter sur RedisDG doit être décrite sous forme d'un graphe orienté acyclique. C'est pour cela que nous nous sommes orientés vers des applications scientifiques présentées sous forme de DAG. Grâce à Pegasus, l'application MONTAGE est présenté sous forme d'un DAX (*Directed Acyclic Graph in XML*). En fait, Pegasus offre des API en Python, Java et Perl, simples et faciles à utiliser pour la génération du DAX [Peg]. Il fournit une description du workflow, indépendante des ressources. Il saisit toutes les tâches qui font du calcul, l'ordre d'exécution de ces tâches est représenté comme des arcs dans le DAG, et pour chaque

tâche les données d'entrée nécessaires, les données de sortie attendues, et les arguments avec lesquels la tâche doit être invoquée. L'ensemble des données d'entrée/sortie et les exécutables sont désignés par des identificateurs logiques.

La figure 6.4 présente une description Pegasus abstraite d'un workflow MONTAGE à 164 nœuds. On distingue bien 3 parties. La première partie concerne la spécification des fichiers nécessaires à l'exécution de l'application. Il existe 3 types de fichiers : les fichiers *input*, ce sont les fichiers nécessaires au démarrage de l'exécution de l'application, les fichiers *inout* qui présentent les fichiers intermédiaires et enfin les fichiers *output* et qui présentent le résultat final de l'exécution de l'application. La deuxième partie concerne la description des *jobs*. Un *job* est une tâche à exécuter et il présente un nœud dans le workflow. Chaque *job* est spécifié par son identifiant unique, le code à exécuter, les arguments à faire passer pour ce code, la liste des fichiers d'entrée et celle des fichiers de sortie. La troisième partie, quand à elle, s'intéresse à spécifier les parents de chaque nœud fils afin d'établir les liens de dépendances.

Cette description, bien qu'ergonomique, n'est pas adaptée à notre système RedisDG. C'est pour cela que nous avons implémenté une API Python permettant de générer une description RedisDG à partir de la description Pegasus.

Comme présenté dans la figure 6.5 et comme détaillé dans le chapitre précédent, RedisDG nécessite une description abstraite de l'application dans laquelle on spécifie les nœuds, les arcs et les racines. Un nœud désigne une tâche, un arc désigne la dépendance entre deux nœuds, et les racines désigne les tâches indépendantes qui seront lancées en premier. Une tâche est spécifiée par son identifiant, le code qu'elle doit exécuter ainsi que les arguments à passer, ses fichiers d'entrée ainsi que ceux de sortie et aussi les caractéristiques de la machine sur laquelle on souhaite exécuter cette tâche.

Dans le tableau 6.1, nous présentons une instance de l'application MONTAGE paramétrée avec un degré de 2.0 que nous avons exécutée avec RedisDG. Cet exemple présente un workflow à 1446 tâches et 3722 liens de dépendances. L'exécution de l'application nécessite en tout 9423 fichiers d'entrée (y compris les fichiers intermédiaires) et génère 2889 fichiers (y compris les fichiers intermédiaires).

L'exécution de MONTAGE sur RedisDG ne s'est pas déroulée sans problèmes. En fait, lors de la récupération des fichiers sources (input) sur le site de la NASA, on s'est aperçu que les noms de certains fichiers ne correspondent pas à la description XML. Ainsi, il était impossible de lancer l'exécution.

Nous avons également été obligé de vérifier la correspondance entre la description XML et les sources, adapter parfois la description XML, parfois les fichiers sources et s'assurer que les noms de fichiers générés dans le XML étaient les fichiers attendus par l'application. Cette tâche s'est avérée très laborieuse surtout quand il s'est agit d'applications à plus de 1000 tâches (plus de 20000 lignes de code à parcourir et à vérifier).

```
<?xml version="1.0" encoding="UTF-8"?>
<adag xmlns="http://pegasus.isi.edu/schema/DAX">
  <!-- Part 1:  Files Used -->
  <filename file="2mass-atlas.fits" link="input"/>
 ...
  <filename file="p2mass-atlas.fits" link="inout"/>
 ...
  <filename file="shrunken.jpg" link="output"/>
  <!-- Part 2:  Definition of Jobs -->
  <job id="ID000001" name="mProjectPP" version="3.0" level="9"
       dv-name="mProject1" dv-version="1.0">
    <argument>
      -X
      -x 1.03362
      <filename file="2mass-atlas.fits"/>
      <filename file="p2mass-atlas.fits"/>
      <filename file="big_region.hdr"/>
    </argument>=
    <uses file="2mass-atlas.fits" link="input" transfer="true"/>
    <uses file="p2mass-atlas.fits" link="output" register="false"
          transfer="false"/>
    <uses file="p2mass-atlas_area.fits" link="output" register="false"
          transfer="false"/>
    <uses file="big_region.hdr" link="input" transfer="true"/>
  </job>
    ....
  <job id="ID000164" name="mJPEG" version="3.0" level="1"
       dv-name="mJPEG1" dv-version="1.0">
    <argument>
      -ct 1
      -gray <filename file="shrunken.fits"/>
      min max gaussianlog
      -out <filename file="shrunken.jpg"/>
    </argument>
    <uses file="shrunken.fits" link="input" transfer="true"/>
    <uses file="shrunken.jpg" link="output" register="true"
          transfer="true"/>
    <uses file="dag.xml" link="input" transfer="true"/>
    <uses file="dag.xml" link="output" register="false"
          transfer="true"/>
    <uses file="images.tbl" link="input" transfer="true"/>
    <uses file="images.tbl" link="output" register="false"
          transfer="true"/>
  </job>
  <!-- Part 3:  Control-Flow Dependencies -->
  <child ref="ID000037">
    <parent ref="ID000001"/>
    <parent ref="ID000008"/>
  </child>
    ...
</adag>
```

FIGURE 6.4 – *Extrait de la description Pegasus d'un workflow MONTAGE à 164 nœuds*

```
<Application>
  <Description EdgeNumber="258" Name="montage" NodeNumber="164"/>
  <Nodes>
    <Node AVAILABLE_DISK="120Go" AVAILABLE_RAM="20Go"
          CPU_MODEL="Intel Xeon"  Code="mProjectPP" FREE_UNTIL=""
          OS="Linux" Predecessors="[]" Runtime="" nodeID="1" >
      <Argument>-X -x 1.03362 2mass-atlas.fits p2mass-atlas.fits
                big_region.hdr</Argument>
      <Inputs>
        <Input file_name="2mass-atlas.fits" size=""/>
        <Input file_name="big_region.hdr" size=""/></Inputs>
      <Outputs>
        <Output file_name="p2mass-atlas.fits" size=""/>
        <Output file_name="p2mass-atlas_area.fits" size=""/>
      </Outputs>
    </Node>
    ...
    <Node AVAILABLE_DISK="120Go" AVAILABLE_RAM="20Go"
          CPU_MODEL="Intel Xeon"  Code="mJPEG" FREE_UNTIL=""
          OS="Linux" Predecessors="[]" Runtime="" nodeID="164" >
      <Argument>-ct 1 -gray  shrunken.fits 0s 99.999% gaussian
                -out shrunken.jpg</Argument>
      <Inputs>
        <Input file_name="shrunken.fits" size=""/>
        <Input file_name="dag.xml" size=""/>
        <Input file_name="images.tbl" size=""/></Inputs>
      <Outputs>
        <Output file_name="shrunken.jpg" size=""/>
        <Output file_name="dag.xml" size=""/>
        <Output file_name="images.tbl" size=""/>
      </Outputs>
    </Node>
  </Nodes>
  <Edge source="1" target="37"/>
  <Edge source="8" target="37"/>
   ...
  <Racine>1</Racine>
  <Racine>2</Racine>
   ...
</Application>
```

FIGURE 6.5 – *Extrait de la description RedisDG d'un workflow MONTAGE à 164 nœuds*

Niveau	Tâche	Description	Nombre de tâches
1	mProject	Reprojeter une image en paramètres et ses empreintes dans un fichier d'en-tête	301
2	mDiffFit	Trouver la différence entre deux images et établir un plan de correspondance de différence	838
3	mConcatFit	Faire une simple concaténation des paramètres du plan de correspondance à partir de plusieurs tâches mDiffFit dans un seul fichier	1
4	mBgModel	Modéliser l'arrière plan du ciel en utilisant les paramètres de correspondance du plan à partir de plusieurs tâches mDiffFit et calculer des corrections de plan pour les images en entrée qui rectifieront l'arrière plan sur toute la mosaïque d'images	1
5	mBackground	Rectifier l'arrière plan dans une seule image	301
6	mImgtbl	Extraire l'information géométrique de l'en-tête FITS à partir de l'ensemble des fichiers et la stocker dans une table de méta-données des images	1
7	mAdd	Co-ajouter un ensemble d'images re-projetées afin de produire une mosaïque comme spécifié dans un template de fichier d'en-tête	1
8	mShrink	Réduire l'image FITS	1
9	mJPEG	Convertir au format JPEG	1

TABLE *6.1 – Caractéristiques des tâches d'une instance de MONTAGE à 1446 Tâches*

Un autre problème au niveau des arguments est apparu. Dans la description Pegasus, pour certaines tâches, les arguments spécifiés se sont avérés très génériques, ce qui ne permettait pas d'exécuter l'application. Il a été nécessaire de fournir, dans la description RedisDG, des arguments plus spécifiques à l'application afin de pouvoir l'exécuter. Tout ce travail de calibrage a été réalisé en collaboration avec l'équipe responsable du projet MONTAGE [1].

La figure 6.6 présente des images résultats de l'exécution de MONTAGE (instance à 1446 tâches) sur RedisDG.

6.5 Tests à large échelle sur Grid'5000

6.5.1 Utilisation de trois sites géographiques

Pour valider à large échelle notre prototype RedisDG, nous avons réalisé des tests sur Grid'5000 [Gri] en utilisant 170 nœuds pris sur les sites de Nancy, Grenoble et Lyon.

La courbe de la figure 6.7 présente l'état d'exécution des 301 premières tâches du

1. Contact : montage@ipac.caltech.edu

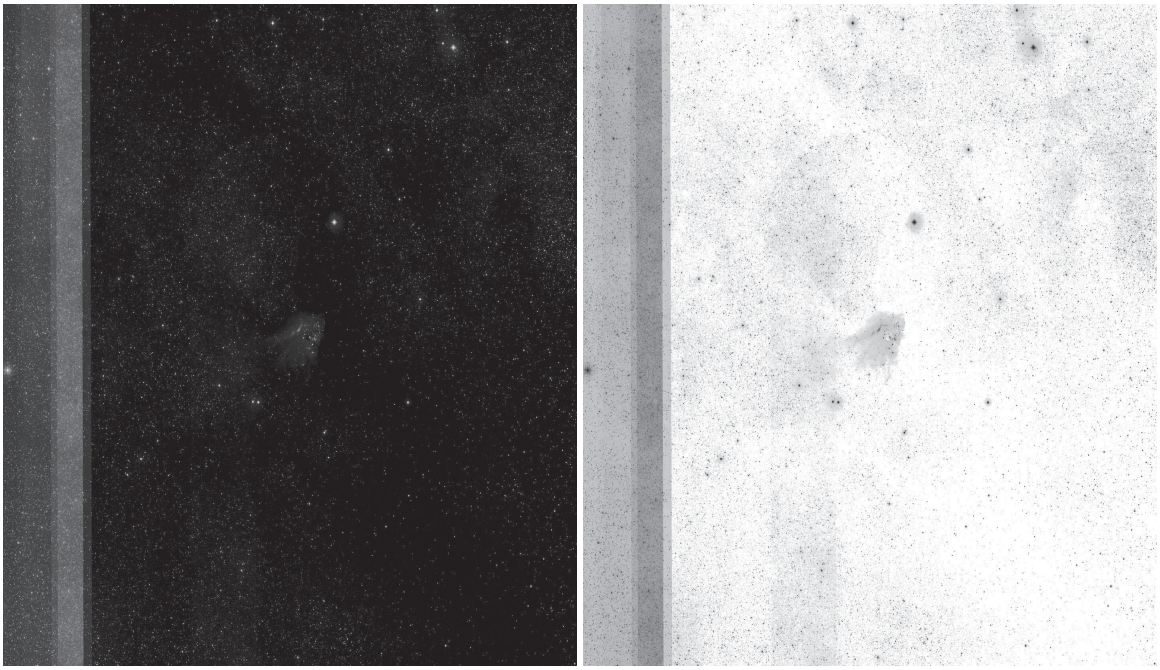


FIGURE *6.6 – Images résultats de l'exécution de MONTAGE*

workflow MONTAGE détaillé précédemment. Ces tâches correspondent au niveau 1 du workflow de la figure 6.3. Ce sont des tâches racines ; elles sont indépendantes. Comme on a à notre disposition 169 workers (170 - 1 qui est le coordinateur) disponibles à la base et qui ont déclaré leur volontariat, la distribution des tâches indépendantes est censée se faire sur tous les workers disponibles, et on est censé observé 169 tâches s'exécuter simultanément sur 169 workers. Mais ce n'est pas ce qui s'est passé en réalité. En fait, comme nous avons implémenté une politique élémentaire d'ordonnancement au niveau de notre coordinateur RedisDG, ce dernier choisit le premier des workers qui se présente.

Dans le tableau 6.2, on présente un extrait du fichier log de l'exécution de MONTAGE contenant les temps de soumission, de début d'exécution et de fin d'exécution de chaque tâche. On remarque que toutes les tâches sont soumises presque en même temps (à quelques millisecondes près) et que le temps de début d'exécution change. On remarque aussi que les tâches 1 et 6 se sont exécutées en parallèle, c'est ce qu'on distingue bien sur la courbe de la figure 6.7. La moyenne des temps des exécutions de chaque tâche est de l'ordre de deux secondes et demi.

Sur la courbe de la figure 6.8, nous présentons l'état de l'exécution de toutes les tâches constituant l'instance de MONTAGE à 1446 tâches. Nous remarquons un niveau de parallélisme des tâches relativement faible par rapport à celui attendu. Dans le tableau 6.3 nous présentons un extrait de l'état de l'exécution des tâches constituant

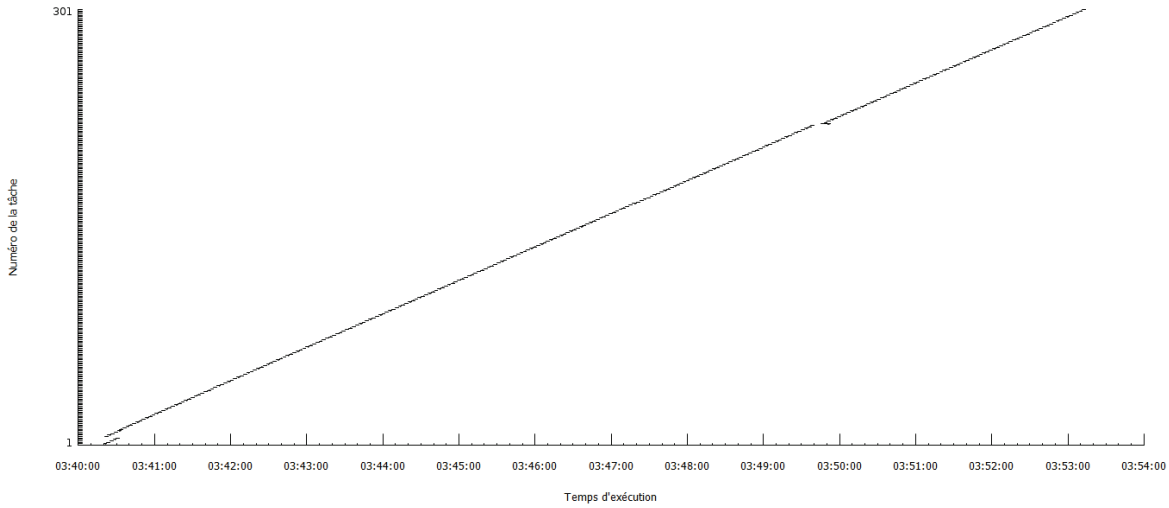


FIGURE *6.7 – Temps d'exécution des tâches 1 à 301*

Task_ID	**Submit_Time**	**Start_Exec_Time**	**End_Exec_Time**
1	03 :40 :15,558	03 :40 :17,424	03 :40 :19,818
2	03 :40 :15,558	03 :40 :19,848	03 :40 :22,287
3	03 :40 :15,559	03 :40 :22,317	03 :40 :24,820
6	03 :40 :15,560	03 :40 :17,536	03 :40 :20,422
12	03 :40 :15,561	03 :40 :32,443	03 :40 :34,966
32	03 :40 :15,564	03 :41 :23,605	03 :41 :26,114

TABLE *6.2 – Extrait du fichier log de l'état de l'exécution de MONTAGE : 1*

le niveau 2 de notre workflow. Nous constatons que la tâche 302 s'exécute en parallèle avec la tâche 12 et de même pour les tâches 303 et 32. Cela correspond au début de notre courbe.

Nous observons aussi le comportement de l'application au niveau des tâches constituant un goulot d'étranglement. Cela correspond aux tâches `mConcatFit` et `mBgModel` de notre workflow.

Dans la figure 6.9 nous avons zoomé sur la dernière partie de la courbe de la figure 6.8. Cela correspond aux niveaux de 5 à 9 de notre workflow. À ce niveau nous avons 301 tâches qui se lancent en parallèle.

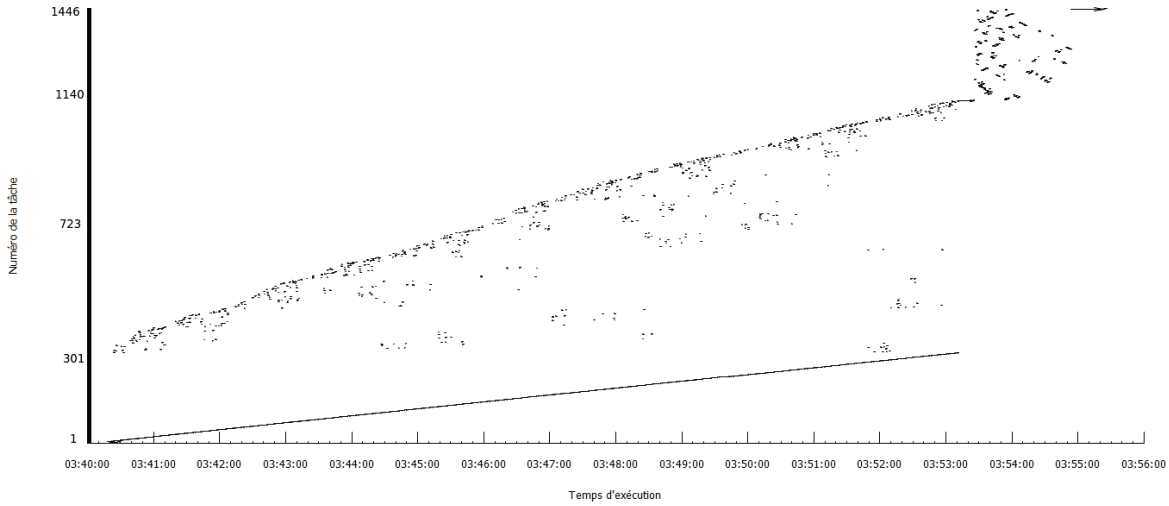


FIGURE *6.8 – Temps d'exécution des tâches 1 à 1446*

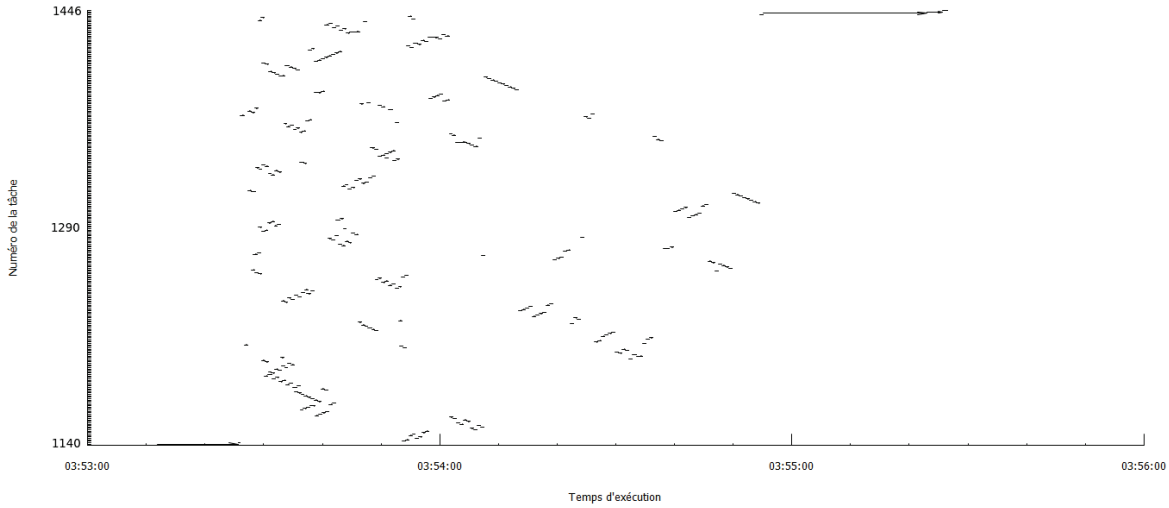


FIGURE *6.9 – Temps d'exécution des tâches 1140 à 1446*

Task_ID	Submit_Time	Start_Exec_Time	End_Exec_Time
302	03 :40 :28,144	03 :40 :32,527	03 :40 :33,356
303	03 :40 :21,281	03 :40 :23,717	03 :40 :24,585

TABLE *6.3 – Extrait du fichier log de l'état de l'exécution de MONTAGE : 2*

6.5.2 Analyse des résultats expérimentaux

Suite à cette série d'expérimentations sur une plateforme réelle, nous concluons que l'ordonnancement des tâches ne s'est pas fait d'une façon équitable. Toutes les machines à disposition de l'application n'ont pas été sollicitées pour participer à l'exécution. Et même celles qui ont participé n'ont pas eu des charges de travail égales. Ainsi, avec notre politique d'ordonnancement actuelle, certaines machines ont plus de travail que d'autres. Ceci est dû au fait que certaines machines publient leurs volontariats plus rapidement que d'autres et notre ordonnanceur fonctionne en mode premier arrivé, premier servi. Avec une telle politique, les workers avec une faible latence avec le coordinateur sont privilégiés. C'est clairement ce que nous observons expérimentalement.

Aussi, dans notre système, un worker chargé d'exécuter une tâche, peut toujours publier son volontariat pour exécuter d'autres tâches et peut être sélectionné plusieurs fois par le coordinateur. Ceci s'explique par le fait que le worker présente deux *threads* ; un thread chargé de l'exécution d'une tâche et l'autre chargé de détecter la publication de nouvelles tâches à faire et aussi d'annoncer le volontariat du worker. Pour l'instant, nous n'avons pas de mécanisme privilégiant un worker libre plutôt qu'un worker occupé.

Pour faire participer toutes les machines volontaires à l'exécution d'une application, nous devons ajuster notre politique d'ordonnancement. Une première solution consiste à ce que le coordinateur sélectionne les workers selon le critère de disponibilité. Une deuxième solution consiste à ce que le coordinateur adopte une politique de tourniquet qui consiste à faire le tour de tous les workers, les uns après les autres pour leur attribuer les tâches.

Pour conclure, tous ces problèmes détectés suite à la réalisation des expérimentations sur Grid'5000 n'ont pas été observé au moment de la simulation/émulation de notre système, d'où l'utilité des expérimentations à large échelle. Ainsi, nous serons amené dans le futur à effectuer un va-et-vient entre l'implémentation et la modélisation formelle de notre système RedisDG afin de reconsidérer ces problèmes dès la phase de modélisation.

6.5.3 Modification de l'algorithme d'ordonnancement

Suite aux expérimentations réalisées précédemment, nous avons décidé de changer notre politique d'ordonnancement afin d'exploiter au mieux les ressources à notre dispo-

sition. En fait, comme précisé dans la section précédente, sur les 170 workers volontaires pour participer à l'exécution de l'application, notre coordinateur a sélectionné les plus rapides à lui répondre. Au final, il n'y a que 30 workers qui ont réellement participé à l'exécution, soit 18%. Le principe même des Grilles de PCs est d'avoir une puissance de calcul à disposition, alors que, d'après ces résultats, cette puissance de calcul est bien là mais elle est clairement sous-exploitée.

Ainsi, ces résultats ont remis en question notre algorithme d'ordonnancement. Nous avons cherché à implémenter une solution qui soit plus efficace au niveau de l'utilisation des ressources. Ainsi, nous avons adopté un algorithme de « tourniquet » permettant de faire le tour de tous les workers, les uns après les autres pour leur attribuer les tâches. Dans ce cas, les workers continuent à publier leurs volontariats, le coordinateur est notifié par toutes les annonces de volontariat, (il y a des notifications qui arrivent plus rapidement que d'autres car cela dépend de la latence avec le coordinateur) mais le coordinateur au lieu de sélectionner le premier arrivé, sélectionne à chaque fois un worker qui n'a pas encore participé, lorsque tous les workers ont participé au moins une fois, on recommence un niveau cycle jusqu'à épuisement des tâches à attribuer.

Comme premiers tests, nous avons relancé notre application MONTAGE (le workflow à 1446 nœuds) sur 200 machines réelles de la plateforme Grid'5000. La figure 6.10 résume le comportement de l'exécution et nous pouvons le comparer avec la figure 6.8. Nous observons que l'allure de la courbe a complètement changé. Ceci est dû au degré de parallélisme plus important. D'après les statistiques réalisées sur les fichiers Log générés par RedisDG, nous constatons que la totalité des Workers, c-à-d les 200, ont participé à l'exécution de l'application (mais pas forcément de façon égale). Comme perspective à court terme, nous envisageons de visualiser le taux de participation de chaque worker. Nous remarquons aussi que le temps total de l'exécution est passé de 16 minutes à 4 minutes, ce qui est flatteur pour notre politique d'ordonnancement.

Nous avons relancé la même application mais sur 340 machines. Nous constatons sur la figure 6.11 que le degré de parallélisme s'est nettement amélioré. Les 340 workers ont tous participé à l'exécution. Mais nous remarquons que le temps total de l'exécution est supérieur à celui avec 200 machines. Cela est probablement dû à l'utilisation d'un nœud peu performant pour les quatre dernières tâches séquentielles. D'un point de vue général, si on veut maîtriser le temps d'exécution, il conviendrait de sélectionner les workers selon des critères de performance et selon la tâche à réaliser. Nous reviendrons sur ces aspects dans la conclusion de cette thèse.

Nous avons aussi observé des pertes de messages en réponse aux sollicitations du coordinateur. Une situation de famine peut arriver dans le cas où il ne reste plus qu'un seul worker pour terminer un cycle de l'algorithme du tourniquet et que les réponses de participation de ce worker se perdent. Techniquement l'algorithme d'ordonnancement devient complexe.

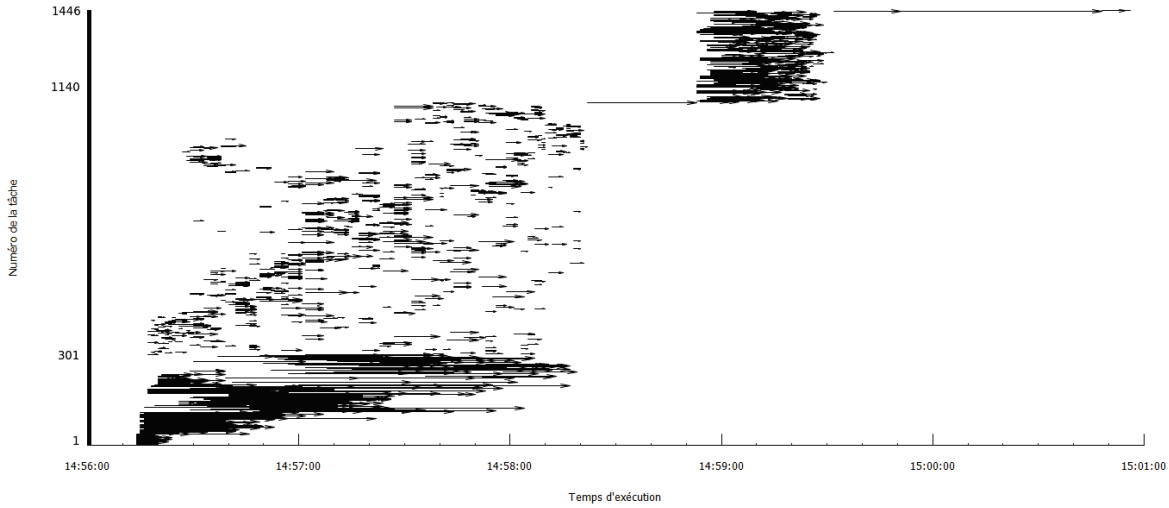


FIGURE *6.10 – Temps d'exécution du Workflow (1446 tâches) sur 200 workers*

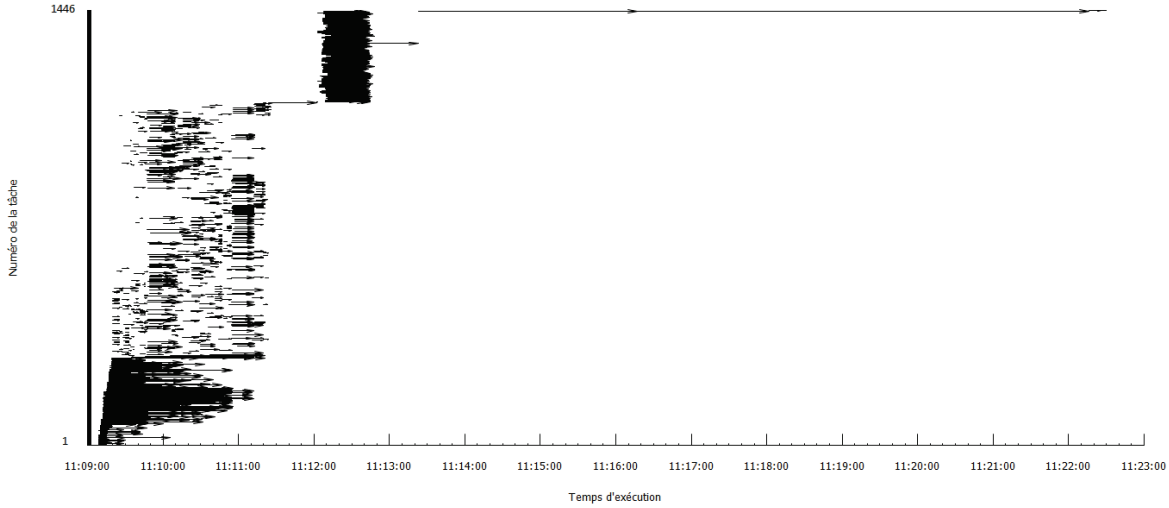


FIGURE *6.11 – Temps d'exécution du Workflow (1446 tâches) sur 340 workers*

6.6 Tests à large échelle sur Grid'5000 en y déployant le Cloud SlapOS

Les expériences qui suivent ont été envisagées pour travailler sur le plan du partage de ressources, ce qui constitue notre premier objectif. Il s'agit avant tout de vérifier que notre système est opérationnel pour ensuite envisager le travail discuté dans la partie des conclusions de thèse. En résumé, ce travail serait le suivant. On considère ici plusieurs utilisateurs HPC (*High Performance Computer*) souhaitant faire des calculs sur un Cloud. Nous souhaitons nous intéresser à des approches d'allocation de machines virtuelles et des applications HPC, comme MONTAGE qui soient efficaces d'un point de vue énergétique par exemple tout en respectant les contraintes SLAs (*Service Level Agreement*).

Dans ce cadre, l'évaluation des politiques d'allocation doit se faire via des comparaisons avec des exécutions sur des plateformes HPC classiques comme des super calculateurs. Un des points essentiel est celui de trouver des métriques d'évaluations qui ne soient pas exclusivement favorables au HPC (c'est-à-dire de prendre un point de vue purement HPC du Cloud). En effet, la qualité de l'exécution d'une application HPC n'est pas la même en fonction des différents niveaux de contrôle suivant que :

a) La personne qui soumet un job a un contrôle total de la machine (souvent le cas dans le HPC classique sur les clusters)

b) Celui qui héberge a le contrôle total (il s'agit d'une vue plus orientée Clouds volontaire)

c) Celui qui fournit le service a le contrôle total (il s'agit d'un vue plus orientée Clouds dans le sens commun)

Notons qu'une plateforme Cloud est avant tout conçue pour optimiser une fonction d'utilité. Juger le calcul HPC à l'aune de cette fonction d'utilité pourrait conduire à des comparaisons plus justes et moins biaisées. Nous pourrions alors considérer deux angles d'études : en premier lieu, il s'agit de trouver les politiques d'allocation efficaces pour répartir les calculs et les communications ; en second, il s'agit de trouver les bonnes formulations du parallélisme pour l'optimisation en liaison avec la granularité des tâches. Notons que des réponses particulières sont attendues de ces deux points car à la différence des plateformes-HPC classiques, les Clouds avec infrastructure décentralisée auront généralement un rapport temps de communication/temps de calcul beaucoup plus important, car basé par exemple sur le réseau Internet. Cependant, le Cloud permet de multiplexer plusieurs calculs sur un même nœud grâce à la virtualisation. Dans notre cas la virtualisation est très économique en terme de ressources consommées car elle se présente essentiellement sous la forme d'une isolation via des conteneurs. Nous allons voir que nous allons pouvoir faire tourner largement plus de 10 « machines virtuelles ». nous préparons ainsi une étude expérimentale plus large que MONTAGE afin de déterminer les compromis à réaliser en terme de gestion des

ressources et des contraintes de SLA pour qu'un Cloud rivalise avec un cluster.

Le second objectif de cette expérimentation est de valider que RedisDG s'intègre dans un Cloud, et peut être offert comme un service. Il nous semble que cela constitue une innovation majeure qui ouvre de nombreuses perspectives comme cela sera souligné dans la conclusion de thèse.

Nous avons vu dans la section précédente comment réaliser l'intégration de RedisDG dans Grid'5000. Pour le déploiement de SlapOS dans Grid'5000 il convient de suivre le rapport technique [AT14] intitulé « Déploiement de la plateforme SlapOS dans l'environnement Grid'5000 ». Nous ne revenons pas sur les aspects très techniques qui sont développés dans ce rapport mais nous voudrions souligner le fait suivant. Comme il s'agit de déployer un système complet (SlapOS) dans un système (Grid'5000), il convient, avant de lancer l'application, de vérifier que SlapOS a bien été déployé et configuré. Plutôt que d'utiliser le post-traitement possible avec Grid'5000 après un déploiement d'image, nous avons décidé pour ce travail de stopper l'automatisation du traitement pour réaliser à la main les dernières vérifications d'usage et le lancement de l'application MONTAGE.

Ainsi, après le déploiement de l'image SlapOS-Grid'5000, nous trouvons les trois traitements les plus importants suivants que nous lançons depuis la machine frontale :

1. le rattachement des volontaires ;
2. création de la recette adaptée à l'application ;
3. le lancement des workers puis du coordinateur.

D'autres traitements sont à réaliser comme par exemple le déploiement du serveur Redis, mais ils présentent moins d'intérêt.

Des exemples de scripts faisant appel aux commandes en ligne SlapOS sont donnés en Annexe avec des commentaires.

6.6.1 Rattachement des volontaires à SlapOS

Cette étape permet de rattacher au Cloud SlapOS chaque nœud volontaire de Grid'5000. Ce rattachement ne peut être réalisé que après le déploiement du *master* SlapOS. En fait, lorsque le *master* et les nœuds SlapOS sont déployés, ils ne communiquent pas encore car les nœuds sont inconnus du *master*. Pour établir la connexion avec le *master*, il faut enregistrer les nœuds. Plus de détails techniques concernant le script sont donnés en annexe B.1. La figure 6.12 résume le déploiement de SlapOS sur Grid'5000 [AT14]. Nous pouvons nous connecter sur le portail du *master* SlapOS pour vérifier que tous les nœuds sont bien actifs et que tous les services sont déployés. Les étapes de configuration ne sont pas automatisables et nécessitent des services externes à Grid'5000.

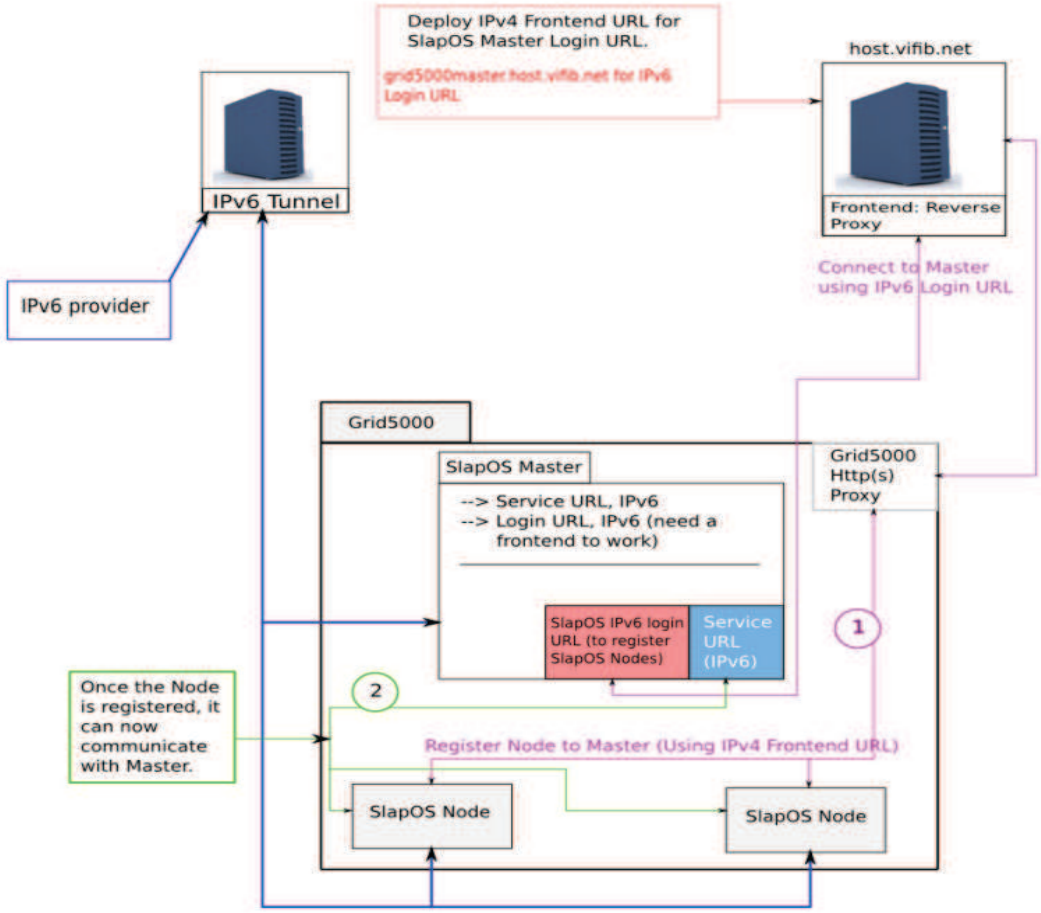


FIGURE *6.12 – Architecture de fonctionnement de SlapOS dans Grid'5000*

Nous pourrions ici envisager d'utiliser un lanceur parallèle comme Taktuk [CHR09] pour réaliser par exemple l'enregistrement des volontaires auprès du *master* SlapOS mais le problème vient que SlapOS n'a pas une interface pour signaler la fin d'une configuration (par des signaux UNIX par exemple). Dans notre cas on pourrait imaginer de tester de manière continue si une variable a bien été positionnée dans une base de données ERP5 de SlapOS. Nous avons préféré programmer un endormissement de valeur déterminée par la pratique et bien supérieur au temps de déploiement. Nous verrons plus tard comment mieux capter ces détails d'implémentation spécifiques aux déploiements de systèmes dans un système.

6.6.2 Création de la recette adaptée à l'application

Dans l'annexe B.1.1, nous présentons la recette RedisDG-SlapOS-Grid'5000 permettant de faire le lien entre les trois systèmes. Dans cette recette nous devons préciser l'adresse IPv6 du serveur Redis que nous avons déployé au préalable sur un nœud de Grid'5000 via SlapOS, fournir le script principal de RedisDG, préciser les fichiers utilisés par RedisDG et enfin envoyer ces informations au *master* SlapOS.

6.6.3 Lancement des workers et du coordinateur

C'est la dernière étape nécessaire au lancement de l'exécution de l'application MONTAGE. Il faut « affecter » un statut à l'ensemble des volontaires attachés à SlapOS. Une machine aura le statut RedisDG-coordinateur et les autres RedisDG-workers. Ainsi, l'exécution de l'application MONTAGE peut démarrer. Plus de détails sur les scripts sont donnés en annexe B.1.2.

6.6.4 Première expérience avec 360 workers : 5 partitions slapOS par nœud et 4 workers par partition

Dans cette première expérience, nous avons réservé 20 machines réelles sur Grid'5000 sur le site de Lyon puisqu'il est le seul à offrir un tunnel IPv4-IPv6. Sur chaque machine réelle nous avons crée 5 partitions SlapOS et sur chaque chaque partition nous avons lancé 4 workers. En écartant les nœuds *master* SlapOS et coordinateur de RedisDG, nous avons au total $18 \times 5 \times 5 = 360$ workers. La courbe à la figure 6.13 présente le comportement de l'exécution de l'application MONTAGE. Dans ce test, nous avons utilisé l'algorithme d'ordonnancement « premier arrivé, premier servi » afin de pouvoir mesurer l'impact de SlapOS sur l'exécution. Nous Constatons que l'allure de la courbe ne diffère pas de celle de la figure 6.8. 45 workers seulement ont participé à l'exécution soit 12.5%.

Pour pouvoir lancer ce test, il nous faut au moins 3 heures afin de vérifier que toute la chaîne de compilation de SlapOs fonctionne sous Grid'5000. La préparation de l'environnement y compris le déploiement du *master* SlapOS, le rattachement des nœuds, le déploiement des workers nécessitent autant de temps afin de configurer les machines et de les redémarrer.

6.6.5 Deuxième expérience avec 400 workers : 20 nœuds et 20 partitions slapOS par nœud

Pour cette expérience, nous avons voulu lancer notre système sur 1000 workers répartis comme suit : $50 machines \times 20 partitions SlapOS$, mais comme nous sommes

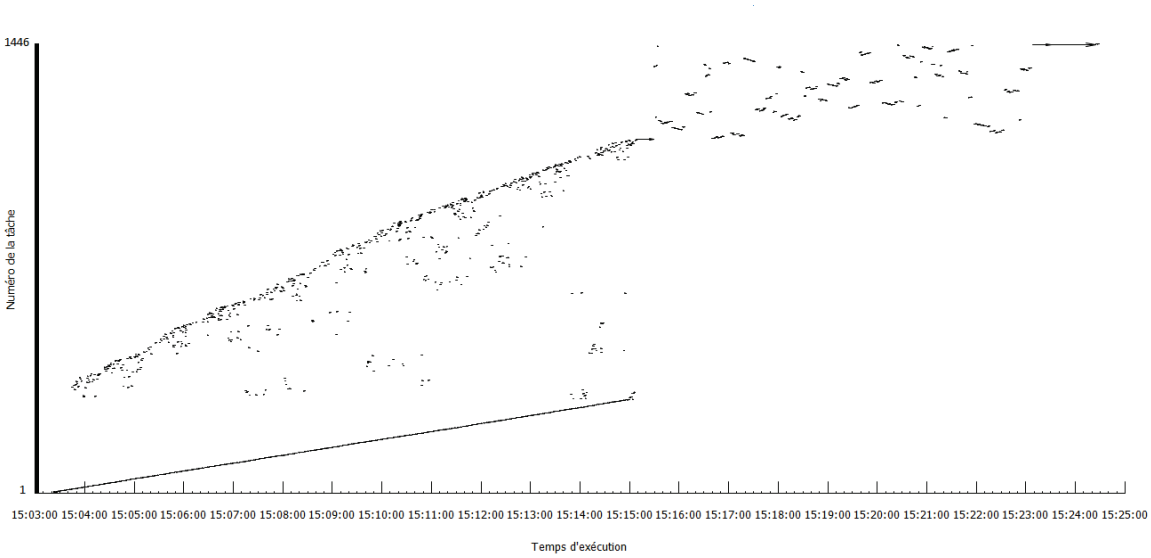


FIGURE *6.13 – Temps d'exécution du Workflow (1446 tâches) sur 360 workers déployés depuis SlapOS*

contraint de travailler sur le site de Lyon parce qu'il est le seul à offrir un tunnel IPv4-IPv6, et aussi travailler avec des machines ayant au moins 8Go de RAM, alors nous nous sommes contentés de 20 machines (le nombre maximal disponibles ayant cette configuration). Ainsi cette expérience, s'est déroulée sur 20 machines $\times$ 20 partitions SlapOS = 400 workers. Comme dans l'expérience de la section 6.5.3, nous avons aussi changé l'algorithme d'ordonnancement « premier arrivé premier servi« pour utiliser celui du « tourniquet » afin de faire participer tous les workers à l'exécution et pas uniquement ceux qui ont le plus faible temps de latence.

La figure 6.14 présente la courbe correspondante à ce test. Nous remarquons que cet algorithme nous fait gagner en temps total d'exécution. Nous passons de 22 minutes pour l'expérience précédente à presque 7 minutes pour cette expérience.

6.7 Synthèse et conclusion

Dans ce chapitre, nous avons tenu à valider expérimentalement notre système RedisDG afin de présenter un système complet. Les résultats obtenus correspondent à nos attentes. Comme objectif à cours termes nous envisageons de déployer plus de workers/-partitions depuis SlapOS afin de réaliser nos expérimentations. Nous envisageons aussi faire exécuter des workflows plus grands (10000 tâches) afin d'étudier les limites de notre système. Le présent travail augure de manipulations qui ne devraient plus poser

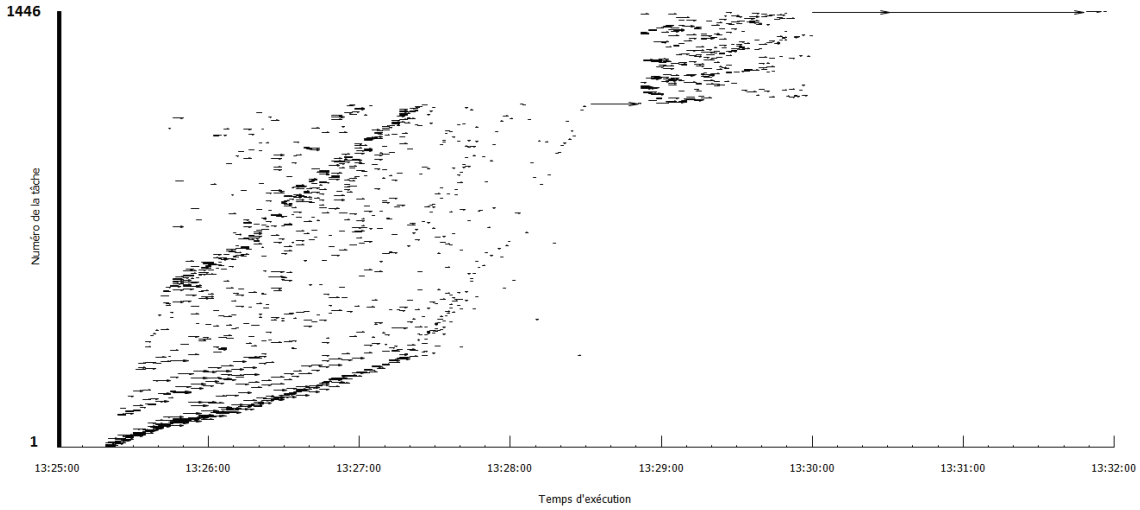


FIGURE *6.14 – Temps d'exécution du Workflow (1446 tâches) sur 400 workers déployés depuis SlapOS*

de problèmes car nous pensons avoir fait le tour des problèmes techniques. Nous voudrions aussi souligner ici combien le travail de calibration et de mise au point devient important lorsque l'on réalise le déploiement de systèmes dans un système. Automatiser cette étape reste un problème difficile pour lequel nous pensons qu'il y a encore de la matière scientifique à creuser.

Chapitre 7

Conclusion générale et perspectives

7.1 Conclusion

Cette thèse est consacrée à revisiter le concept des Grilles de PCs en termes d'interactions entre des composants venant du Web ou apparus récemment comme la notion de Clouds. Il s'agit de pousser un peu plus loin le concept en se demandant par exemple où l'on peut prendre de nos jours des ressources, où l'on peut déployer du calcul. Le système produit, RedisDG, continue de reposer sur les composants historiques comme l'ordonnanceur de tâches, la certification des résultats, la surveillance des nœuds, la redondance des calculs,... et introduit de nouveaux composants comme les serveurs de code ou de données. Le prototype est capable d'exécuter des graphes de tâches issus de moteur de workflows bien connus comme Pegasus. Une partie de la validation expérimentale permet de déployer à la demande le système RedisDG dans un Cloud (SlapOS). C'est une caractéristique importante qui en fait un système unique (on sait que plusieurs autres projets connus essaient aussi d'intégrer des ressources de Cloud à leur système de grille de PCs).

Le protocole de coordination qui a été implémenté repose quant à lui sur une couche de modélisation au moyen de réseaux de Petri qui nous guident pour faire des choix d'implémentation et à avoir une confiance plus grande dans le comportement du système par rapport à un système sans étapes de modélisation et de vérification abstraite. On avait l'habitude, dans la communauté, de construire ces systèmes par des méthodes ad-hoc. Ce travail de modélisation a permis de renforcer l'aspect de robustesse de notre système permettant, ainsi, de mettre un premier pas vers un système de production, libre, pouvant être utilisé comme solution d'expérimentation.

Nous affirmons que les briques de base d'un nouveau type de Grille de PCs sont présentes dans RedisDG. Nous pouvons maintenant envisager un certain nombre de raffinements sur ces briques de base. Ces raffinements constituent les perspectives de notre travail.

7.2 Perspectives

7.2.1 Ordonnancement dans RedisDG

SlapOS est un système de Cloud Computing décentralisé inventé par l'Université de Paris 13 et par Nexedi. Commercialisé en Europe, au Japon et en Chine, il comporte des fonctions de déploiement et d'orchestration automatiques utilisées en production par SANEF, Mitsubishi, Airbus Defence ou Aide et Action. Un des points clé de SlapOS que nous avons utilisé est que nous pouvons déployer des applications dans des partitions (et non à l'intérieur de VM - machines virtuelles). Nous pouvons déployer beaucoup plus d'applications qu'une solution avec des VM sur un nœud physique. Le planificateur SlapOS prend la décision de placement (du maître BOINC par exemple), mais actuellement, la décision n'est pas faite selon une certaine connaissance des applications.

Nous proposons d'étudier le planificateur SlapOS et de le revisiter sur la base de renseignements émis par les applications. La première étape est de montrer que les choix de SlapOS peuvent être «significativement mauvais» malgré le fait qu'il se base actuellement sur des exigences de l'application (RAM,...). Deuxièmement, si nous découvrons un (gros) problème, nous travaillerons sur une adaptation de l'ordonnanceur de SlapOS couplé avec l'infrastructure BOINC par exemple.

Avec SlapOS, deux ou plusieurs applications en cours d'exécution dans différentes partitions accèdent aux bibliothèques partagées comme cela est le cas avec le système Linux ce qui est en faveur de la réutilisation. Intuitivement, il semble préférable d'allouer des sac-de-tâches dans des partitions adjacentes et sur la même machine physique. En cas de graphes avec dépendances avec des codes différents pour chaque nœud du graphe, cela n'a pas d'importance puisque le code ne partage rien et il ne faut pas s'attendre à des performances supplémentaires. Cependant, une tâche qui produirait une quantité importante de données pour la prochaine tâche du graphe des tâches pourrait tirer avantage d'être situées dans des partitions adjacentes. Ces exemples démontrent le couplage possible entre l'ordonnanceur de SlapOS et l'ordonnanceur BOINC dans le contexte de Cloud.

Plus généralement, l'ordonnaceur de Cloud peut prendre une décision de placement (sur la base d'un budget, une contrainte énergétique, SLA,...) qui est orthogonale avec une décision prise par l'ordonnanceur BOINC (dont le but est de minimiser le makespan). Ce problème de placement initial est nouveau pour BOINC et nous avons besoin de prêter attention à la complémentarité entre les ordonnanceurs.

Dans [YN14] les auteurs étudient des plans de déploiement pour minimiser l'énergie dans les Clouds volontaires « à la SlapOS ». La difficulté vient des périodes d'indisponibilité de certains nœuds (les volontaires) et de la modélisation de l'énergie, en particulier lors des migrations nécessaires à la gestion des indisponibilités. Ce travail formel mais

aussi de simulation fournit deux types de résultats. Premièrement il est montré que le problème en question est inaproximable. Deuxièmement, les auteurs se tournent alors vers des solutions à base d'heuristiques et il est montré que dans la pratique, par des simulations exhaustives, que celles-ci se comportent bien sur des jeux d'essai tirés du monde réel. Il s'agit en l'occurrence des traces d'activité des PCs sur Internet et des travaux [IKM12] de Derrik Kondo à Grenoble.

Nous proposons de repartir de la modélisation du problème telle qu'elle est introduite dans [YN14] afin de s'assurer que l'ordonnancement est équitable. Une définition classique nous indique que l'ordonnancement équitable est une méthode d'affectation des ressources à des travaux tels que les travaux obtiennent, en moyenne, une part égale des ressources dans le temps. Dans les expériences à large échelle conduites pour le workflow Montage sur plusieurs sites de Grid'5000 nous avons observé qu'avec l'ordonnanceur actuel de RedisDG que certains nœuds obtiennent beaucoup plus de travail que d'autres et ceci est dû à la latence réseau différentes entre les sites de Grid'5000. En effet certains nœuds publient leurs annonces de participation plus rapidement que d'autres et l'ordonnateur fonctionne en mode premier arrivé, premier servi. L'ordonnancement équitable que nous proposons d'étudier est motivé par cette observation et doit alors s'entendre comme une méthode d'affectation des ressources à des machines telles que les machines obtiennent, en moyenne, une part égale de travaux dans le temps.

Le modèle initial des contraintes posées sur le problème dans [YN14] est donné à la Table 7.1. La contrainte C_1 est évidente : on ne peut déployer que sur des machines disponibles. La contrainte C_2 nous dit que les copies d'applications (pour gérer la résilience) doivent être déployées sur des machines différentes. La contrainte C_3 exprime une contrainte de capacité sur les nœuds.

C_1	*Les applications doivent être déployées sur des machines disponibles* ;
C_2	*Les k_j instances de l'application $j \in N$ doivent être assignées à k_j machines distinctes* ;
C_3	*Sur n'importe quelle machine $i \in M$, il peut pas y avoir plus que q_i applications.*

TABLE *7.1 – Liste des contraintes*

Nous proposons d'ajouter au modèle une contrainte pour signifier qu'à la fin de l'exécution, le plan de déploiement est tel que toutes les machines ont reçu un nombre équivalent de travaux . On peut aussi envisager une version plus contrainte en exprimant qu'à des périodes de temps régulièrement espacées, toutes les machines ont reçu un nombre équivalent de travaux. Ensuite, nous nous proposons d'étudier expérimentalement la solution, à partir du simulateur déjà construit qu'il faudra adapter, et de quantifier l'impact de ces nouvelles contraintes sur la consommation d'énergie.

7.2.2 Intégration de gestionnaires de données

Walid Saad, dans sa thèse, a conçu, implémenté et testé sur Grid'5000 un schéma basé sur des caches distants et locaux pour obtenir / déplacer des données entre le maître et les esclaves.

Il a également mis en place un schéma de pré-chargement fonctionnant au-dessus de Condor et il utilise le langage Condor Dagman pour automatiser le pré-chargement ce qui permet de ne rien modifier à Condor.

Dans le cadre des grilles de PCs en général, comment réutiliser ces idées pour BOINC ? Est-ce que BOINC va pouvoir télécharger / gérer des fichiers volumineux ($>$ 1 Go) comme avec le schéma actuel validé par Walid Saad avec l'application de génomique BLAST ?

Notre objectif est de permettre la gestion de données sur la grille BOINC sans modification de la demande de l'utilisateur. La principale difficulté est l'orchestration / communication entre notre vision du Data Manager et BOINC. Ici, le gestionnaire de données sera utilisé pour planifier des tâches de données, BOINC gère l'ordonnacement des tâches de calcul.

D'autres questions arrivent alors. Comment pouvons-nous remplacer le protocole BOINC Data Transfer par notre gestionnaire de données ?

Pour activer la gestion de données à la demande avec BOINC, nous avons besoin de construire un schéma auto-configurable (transparent aux utilisateurs). Nous devons fournir une API de haut niveau pour gérer automatiquement la file d'attente de gestion des données et la file d'attente gérant les calculs et ceci dans le même cadre. Cette API doit être en mesure d'interagir avec les API BOINC.

Enfin le gestionnaire de données doit prendre en compte la réplication appliquée par BOINC pour valider les unités de travail.

7.2.3 RedisDG et le Cloud SlapOS

7.2.3.1 Le monitoring de ressources

Pendant que nous écrivions cette thèse, SlapOS a évolué sur le plan du monitoring des ressources. Le travail réalisé ne s'appuie pas sur SystemTap que nous avons introduit dans un chapitre précédent mais s'appuie plutôt sur des outils proches de SysStat que nous avons également introduits précédemment.

Dans la nouvelle version de SlapOS, un dossier `var/data-log` est crée, dans lequel sont stockées en temps réel toutes les informations sur la consommation des partitions. Les informations collectées concernent :

– la consommation CPU en temps réel ;

– la consommation mémoire en temps réel ;

– la consommation disque en temps réel ;

– la consommation réseau en temps réel ;

Par contre, dans la version actuelle de SlapOS, les données concernent toute la machine et non pas une partition particulière. Un service appelé monitoring sera déployé sur toutes les machines, ce service retourne une URL permettant d'accéder à distance via le Web (http) aux fichiers contenus dans le dossier `var/data-log`. Ces fichiers pourront servir à différentes analyses pour observer l'évolution de la consommation des ressources sur la machine. Par la suite, nous proposons qu'un nouveau service SlapOS permettant, à partir de toutes les URLs, de faire une analyse de toutes les données collectées sous forme de courbes par exemple. On pourrait également suivre la consommation pour une ou un groupe de machines.

Plus tard, il sera question de remonter les données en fonction des partitions, ce qui est plus difficile : il faut imaginer un service scalable de remontée d'information. Le fait d'avoir ce module de monitoring, plus générique, au niveau du Cloud SlapOS, nous permettra peut être d'alléger le module de monitoring au niveau de RedisDG et faire de lui un service que RedisDG appelle à la demande. Le problème ici est de trouver le bon compromis entre ce qui relève de SlapOS et ce qui est placé sous la responsabilité de RedisDG, en fait de l'application.

SlapOS sera l'orchestrateur pour le projet Wendelin qui démarre fin 2014 entre Gaz de France, Mitsubishi, Woelfel GmbH, Nexedi, l'Université de Paris 13, Telecom ParisTech, INRIA. La proposition Wendelin se situe dans le domaine du Big Data en réponse à l'appel à projet : Informatique en nuage – Cloud Computing et Big Data, dans le cadre des Investissements d'Avenir pour le Développement de l'Economie Numérique. Ce projet concerne un parc d'éoliennes qui envoient régulièrement des informations sur l'état du parc contrôlé par des machines numériques. Il s'agit d'analyser en temps réel ces informations afin d'être réactif en cas de détection de pannes par exemple. La remontée d'information telle qu'elle est prévue par le cahier des charges techniques doit se faire avec l'outil Fluentd.

Fluentd [1] est un collecteur de données qui propose une couche unifiée de gestion d'informations de log. Fluentd vous permet d'unifier la collecte et la redistribution des informations de log pour une meilleure compréhension et utilisation de ces données. Avec plus de 300 plugins il permet d'agréger et de redistribuer des sources de données tout en conservant un cœur de programme qui ne consomme que 30 à 40Mo. La documentation du projet dit qu'il peut traiter 13000 événements par seconde et par cœur. Il pourrait servir comme alternative aux méthodes que nous avons présentées pour la collection de logs. Il n'est pas clair d'imaginer une décentralisation de Fluentd encore plus poussée que ce qui est actuellement disponible. En effet, Fluentd propose de rediriger une source vers une destination et on peut dupliquer ce schéma autant de fois que l'on veut de sorte

1. Voir http ://www.fluentd.org

qu'il n'y a pas vraiment d'organe central dans son architecture. On pourrait à la limite imaginer un réseau d' overlay de démons Fluentd permettant de répartir et de consulter les log à partir d'un point d'entrée. Il semblerait que du coté de la sortie d'information du démon on puisse créer des réseaux en arbre avec des chemins multiples pour permettre la résilience c.à.d pour aller du démon Fluentd à sa destination de manière sécurisée.

Nous proposons de modéliser ces aspects en comparant l'existant avec un protocole qui peut être utilisé en sous main dans SlapOS et qui s'appelle Babel [2] développé par notre collègue Juliusz Chroboczek de Paris 7. De plus nous avons vu qu'avec Redis il y avait des limitations sérieuses de performance lorsque l'on voulait stoker des fichiers volumineux. Il est à craindre qu'il en soit de même avec Fluentd qui, comme Redis, est un outil du monde du Web, prévu pour traiter beaucoup de requêtes mais pas volumineuses et pas un outil pour le traitement intensif de masses de données importantes.

7.2.3.2 La certification des résultats

Puisque SlapOS permet l'agrégation de ressources de calcul sur Internet mais aussi dans des data-centers, on pourrait imaginer de ne déployer la certification que sur les nœuds qui sont exposés sur Internet, ceux dans les data-centers étant considérés comme fiables. Techniquement cela semble réalisable à un coût de développement relativement faible.

Par contre d'un point de vue théorique on peut par exemple se poser la question des critères à mettre en place pour envoyer un code s'exécuter sur une machine externe. Le problème de placement sous-jacent pourrait ensuite s'envisager avec des techniques classiques d'optimisation combinatoire (ILP - Integer Linear Programming, Heuristiques gloutones,...)

On peut aussi s'interroger sur la fraction de machines exposées sur Internet que l'on peut s'autoriser afin d'assurer un « niveau de confiance » donné.

7.2.4 Grille de PCs dans le navigateur

Nous avons ici peut être l'occasion de revenir sur le concept original des grilles de PCs (DG) en considérant que le navigateur et JavaScript sont les deux pivots pour l'avenir. Pour la construction d'une DG nous avons besoin d'une base de données (nous pouvons l'exécuter dans le navigateur selon les technologies SQL Web), nous avons besoin d'un modèle de communication (nous en avons beaucoup dans le Web,... et on peut même se connecter entre navigateurs et échanger des données avec WebRTC par exemple ou les normes HTML5). Nous devons imaginer un réseau overlay entre les navigateurs, et un bac à sable (nous avons déjà des bac-à-sable pour javascript ; nous avons aussi le projet Chromium NaCl qui est un service sécurisé de bac-à-sable).

2. Voir http ://www.rfc-editor.org/rfc/rfc6126.txt

Peut-être même que nous avons déjà les technologies pour exécuter des applications BOINC avec les navigateurs comme nœuds esclaves et sans la nécessité de compter sur un véritable système d'exploitation distribué pour remplacer Linux / Windows. Cette direction serait un moyen de parvenir à l'informatique durable en utilisant n'importe quel dispositif multi-core à faible consommation d'énergie.

L'idée est ici de décentraliser l'architecture DG basée sur la participation de navigateurs Web. Nous aimerions alors travailler pour la mise en œuvre une DG basée sur les navigateurs Web. Qui joue le rôle de l'orchestration de services ? Avons-nous besoin d'un réseau de recouvrement (à la PastryGrid) entre navigateurs ? Quel est le rôle d'un ERP à la SlapOS ? Avons-nous besoin de coder BOINC en Javascript ou utiliser un émulateur de PC écrit en Javascript ?

7.2.5 RedisDG comme nouveau moteur de Workflow à la demande

Pour l'instant nous avons positionné notre travail comme un travail permettant de déployer à la demande un moteur le workflow RedisDG. Cependant, le travail du scientifique expérimentateur, en particulier celui que l'on pourrait appeler le Data-Scientist, ne se résume pas à décrire un fichier XML représentant l'expérience.

En s'intéressant au cycle de vie des données on peut constater que dans beaucoup de disciplines scientifiques, les données doivent non seulement être acquises depuis des centres de production, mais ensuite filtrées, analysées. Elles retournent éventuellement entre temps dans des centres d'archivage. Il faut ensuite (re)formater et nettoyer les données, refaire une analyse (éventuellement par une autre équipe de personnes), il faut qu'ensuite les résultats puissent être commentés et validés par les experts du domaine qui vont procéder à des comparaisons et explorer des alternatives. Enfin, il y a la phase de dissemination des résultats (il faut faire quelques courbes significatives de ce que l'on a trouvé dans les données) et enfin archiver.

Le groupe PREDON (Préservation des données) qui est une émanation de l'IN2P3, donc du CNRS a produit un premier document en 2014 et intitulé *Scientific Data Preservation 2014* où les communautés, principalement de la physique, explique dans cette édition le cycle des données et les enjeux.

Concernant RedisDG cela veut dire qu'il devrait être capable de gérer ces différentes phases. Nous tombons sur un problème de composition de workflow pour lequel une littérature importante existe.

Comme nous venons de le dire, un angle clé pour maîtriser la complexité des systèmes est de gérer les cycles de vie des données, c'est-à-dire les diverses opérations dans lesquelles elles sont impliquées : transfert, archivage, réplication, suppression,... L'étude du cycle de vie est le point de départ du projet BigDattes que LaTICE, le LIPN et

INRIA (Gilles Fedak) ont déposé en 2014 dans le cadre d'un appel d'offre franco-tunisien. Il s'agit alors de modéliser le cycle de vie et à partir du modèle nous allons pouvoir nous intéresser à des problématiques d'élasticité, d'outils facilitant la gestion dans un Cloud.

Pour diminuer la complexité des cycles de vie des données, Active Data, développé par INRIA à l'ENS-Lyon dans l'équipe Avalon, est un modèle de programmation visant à automatiser et améliorer l'expressivité des applications de gestion de données. Une collaboration informelle avec Tunis est en cours.

Active Data est un modèle de programmation et un environnement d'exécution qui permet de programmer des applications en spécifiant le code qui sera exécuté pour chaque étape du cycle de vie des données. Il fonctionne comme suit : les systèmes de gestion des données exposent leur cycle de vie des données intrinsèque selon un formalisme bien spécifié. Nous considérons que chaque création, modification ou suppression d'une donnée par le système de gestion de données comme progression de la donnée pendant son cycle de vie. Nous appelons transition le passage de la donnée d'un état à un autre. Le modèle de programmation proposé par Active Data pourrait être considéré comme à base de transitions.

Le programmeur fournit un code ou un gestionnaire de transition « transition handler » qui est exécuté à chaque fois qu'une transition est déclenchée. En d'autres termes, quand une transition survient à une donnée, un message (ou un événement) est envoyé aux autres nœuds du système pour les notifier de la transition, ce qui provoque l'exécution du code transition handler. Le paradigme utilisé par Active Data pour propager des transitions est basé sur le paradigme de publication-souscription. Les systèmes de gestion de données publient les transitions à un service centralisé appelé Active Data Service.

Le modèle se base sur les Réseaux de Petri, qui est un formalisme et un outil graphique largement utilisé pour l'analyse des systèmes concurrents et pour le partage des ressources comme nous l'avons vu dans cette thèse. Les réseaux de Petri peuvent illustrer le cycle de vie des données d'une façon intuitive : les places, représentées par des cercles sont les états du cycle de vie ; Les transitions, représentées par des rectangles sont les opérations qui se produisent sur les données ; Les jetons dans les places, sont les instances des données dans un état particulier du cycle de vie. Il est fréquent dans les systèmes de traiter des réplicas de données. Chaque réplica de donnée est représenté par un seul jeton du Réseau de Petri.

Active Data donne une vision orientée donnée ; il expose le cycle de vie des données afin de pouvoir faire coopérer plusieurs systèmes hétérogènes. En effet, la mise en relation de différents systèmes est nécessaire pour traiter des volumes importants de données, qui sont généralement acquises depuis différentes sources. Avec ActiveData, il est possible de coordonner des actions entre ces différents systèmes tout en masquant la complexité de cette tâche aux utilisateurs.

L'objectif d'un des axes du projet BigDattes est d'explorer la modélisation et la vérification formelle de propriétés de bon fonctionnement des applications de traitement de données modélisées avec Active Data. Deux types de propriétés seront considérées et pour lesquelles nous pensons pouvoir nous impliquer :

1. Des propriétés liées à l'évolution, dans le temps, des données (dans leur cycle de vie) peuvent être exprimées avec des logiques temporelles et vérifiées avec un model-checker.
2. Des propriétés spécifiques à l'environnement Cloud comme l'élasticité ou la "multitenancy" peuvent aussi être considérées.

Bibliographie

[Abb09] Heithem Abbes. *Approches de décentralisation de la gestion des ressources dans les Grilles de PC*. PhD thesis, Université de Paris XIII (France) and Université de la Manouba (Tunisia), 2009.

[ACDJ08] Heithem Abbes, Christophe Cérin, Jean-Christophe Dubacq, and Mohamed Jemni. Étude de performance des systèmes de découverte de ressources. *CoRR*, abs/0804.4590, 2008.

[ACE11] Leila Abidi, Christophe Cérin, and Sami Evangelista. A petri-net model for the publish-subscribe paradigm and its application for the verification of the bonjourgrid middleware. In Jacobsen et al. [JWH11], pages 496–503.

[ACJ08] Heithem Abbes, Christophe Cérin, and Mohamed Jemni. Pastrygrid : Decentralisation of the execution of distributed applications in desktop grid. In *Proceedings of the 6th International Workshop on Middleware for Grid Computing*, MGC '08, pages 4 :1–4 :6, New York, NY, USA, 2008. ACM.

[ACJ09] Heithem Abbes, Christophe Cérin, and Mohamed Jemni. Bonjourgrid : Orchestration of multi-instances of grid middleware on institutional desktop grids. In *IPDPS*, pages 1–8. IEEE, 2009.

[ACJ10] Heithem Abbes, Christophe Cérin, and Mohamed Jemni. A decentralized and fault-tolerant desktop grid system for distributed applications. *Concurrency and Computation : Practice and Experience*, 22(3) :261–277, 2010.

[ACJ13] Leila Abidi, Christophe Cérin, and Mohamed Jemni. Desktop grid computing at the age of the web. In *Grid and Pervasive Computing - 8th International Conference, GPC 2013 and Colocated Workshops, Seoul, Korea, May 9-11, 2013. Proceedings*, pages 253–261, 2013.

[ACK+02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home : An experiment in public-resource computing. *Commun. ACM*, 45(11) :56–61, November 2002.

[ACK12] Leila Abidi, Christophe Cérin, and Kais Klai. Design, verification and prototyping the next generation of desktop grid middleware. In *Advances in Grid and Pervasive Computing - 7th International Conference, GPC 2012, Hong Kong, China, May 11-13, 2012. Proceedings*, pages 74–88, 2012.

[AD08] Heithem Abbes and Jean-Christophe Dubacq. Analysis of Peer-to-Peer Protocols Performance for Establishing a Decentralized Desktop Grid Middleware. In *Euro-Par Workshops*, page 235–246, 2008.

[ADCJ13] Leila Abidi, Jean-Christophe Dubacq, Christophe Cérin, and Mohamed Jemni. A publication-subscription interaction schema for desktop grid computing. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, pages 771–778, 2013.

[And04] David P. Anderson. Boinc : A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[AT14] Jean-Paul Smets. Alain Takoudjou, Christophe Cérin. Déploiement de la plate-forme slapos dans l'environnement grid'5000. In *HAL Id : hal-00958012*, 11 Mar 2014.

[BBR02] O. Beaumont, V. Boudet, and Y. Robert. A realistic model and an efficient heuristic for scheduling with heterogeneous processors. In *HCW'2002, the 11th Heterogeneous Computing Workshop*. IEEE Computer Society Press, 2002.

[BCC+06] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frederic Desprez, Emmanuel Jeannot, Yvon Jégou, Stéphane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Touché Irena. Grid'5000 : a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4) :481–494, November 2006.

[BCD+08a] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, Mei-Hui Su, and K. Vahi. Characterization of scientific workflows. In *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*, pages 1–10, Nov 2008.

[BCD+08b] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, Mei-Hui Su, and K. Vahi. Characterization of scientific workflows. In *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*, pages 1–10, Nov 2008.

[BGZ05] Luciano Baresi, Carlo Ghezzi, and Luca Zanolin. Modeling and validation of publish/subscribe architectures. In *Testing Commercial-off-the-Shelf Components and Systems*, pages 273–291. Springer Berlin Heidelberg, 2005.

[BHK+14] Rüdiger Berlich, Marcus Hardt, Marcel Kunze, Malcolm P. Atkinson, and David Fergusson. In Rajkumar Buyya, Tianchi Ma, Reihaneh Safavi-Naini, Chris Steketee, and Willy Susilo, editors, *ACSW Frontiers*, 2007-02-14.

[BL99] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5) :720–748, 1999.

[BLK83] J. Blazewicz, J.K. Lenstra, and A.H. Kan. Scheduling subject to resource constraints. *Discrete Applied Mathematics*, 5 :11–23, 1983.

[BLR03] Olivier Beaumont, Arnaud Legrand, and Yves Robert. The master-slave paradigm with heterogeneous processors. *IEEE Trans. Parallel Distributed Systems*, 14(9) :897–908, 2003.

[BON] `http://developer.apple.com/networking/bonjour/`.

[CDF+05] Franck Cappello, Samir Djilali, Gilles Fedak, Thomas Herault, Frédéric Magniette, Vincent Néri, and Oleg Lodygensky. Computing on large-scale distributed systems : Xtrem web architecture, programming models, security, tests and convergence with grid. *Future Gener. Comput. Syst.*, 21(3) :417–437, 2005.

[CDKR02] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe : A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC)*, 20, 2002.

[CF12] Christophe Cerin and Gilles Fedak. *Desktop Grid Computing*. Chapman and Hall-CRC, 1 edition, 2012.

[CG] Annie Choquet-Geniet.

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[CHR09] Benoit Claudel, Guillaume Huard, and Olivier Richard. Taktuk, adaptive deployment of remote executions. In Dieter Kranzlmüller, Arndt Bode, Heinz-Gerd Hegering, Henri Casanova, and Michael Gerndt, editors, *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing, HPDC 2009, Garching, Germany, June 11-13, 2009*, pages 91–100. ACM, 2009.

[chu08] *Chukwa, a large-scale monitoring system*, Chicago, IL, 10/2008 2008.

[Clo] `http://cloudstack.apache.org/`.

[CM96] Soren Christensen and Kjeld H. Mortensen. *Design/CPN ASK-CTL Manual*. University of Aarhus, 1996.

[CPN] `http://cpntools.org/`.

[CT13] Christophe Cerin and Alain Takoudjou. Boinc as a service for the slapos cloud : Tools and methods. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and*

PhD Forum, IPDPSW '13, pages 974–983, Washington, DC, USA, 2013. IEEE Computer Society.

[DGST09] Ewa Deelman, Dennis Gannon, Matthew Shields, and Ian Taylor. Workflows and e-science : An overview of workflow system features and capabilities. *Future Gener. Comput. Syst.*, 25(5) :528–540, may 2009.

[DON] `http://www.edonkey2000.com/`.

[DVJ+14] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J. Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, (0) :–, 2014.

[EFGK03] Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2) :114–131, 2003.

[EFT+06] T. Estrada, D.A. Flores, M. Taufer, P.J. Teller, A. Kerstens, and D.P. Anderson. The effectiveness of threshold-based scheduling policies in boinc projects. In *e-Science and Grid Computing, 2006. e-Science '06. Second IEEE International Conference on*, pages 88–88, Dec 2006.

[Euc] `https://www.eucalyptus.com/`.

[FC08] Bryan Ford and Russ Cox. Vx32 : Lightweight user-level sandboxing on the x86. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 293–306, Berkeley, CA, USA, 2008. USENIX Association.

[FK99] Ian Foster and Carl Kesselman, editors. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[FKNT02] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The physiology of the grid : An open grid services architecture for distributed systems integration. 2002.

[FL06] JoséLuiz Fiadeiro and Antónia Lopes. *A Formal Approach to Event-Based Architectures*, volume 3922 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006.

[FZRL09] Ian T. Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. *CoRR*, abs/0901.0131, 2009.

[Gar] `http://www.gartner.com/technology/home.jsp`.

[GCGS91] Chiola Giovanni, Dutheillet Claude, Franceschinis Giuliana, and Haddad Serge. On well-formed coloured nets and their symbolic reachability graph. In *ATPN'1991*, page 373–396. Springer Verlag, 1991.

[GKK03] David Garlan, Serge Khersonsky, and Jung Soo Kim. Model checking publish-subscribe systems. In Thomas Ball and Sriram K. Rajamani, editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 166–180. Springer, 2003.

[GLI] `http://glite.web.cern.ch/`.

[Gri] `http://www.grid5000.fr`.

[HAD] `http://hadoop.apache.org/`.

[Hol03] Gerard J. Holzmann. *Spin Model Checker, The : Primer and Reference Manual*. Addison-Wesley Professional, September 04, 2003.

[HP04] B. Hong and V.K. Prasanna. Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput. In *International Parallel and Distributed Processing Symposium IPDPS'2004*. IEEE Computer Society Press, 2004.

[HSLL00] E. Heymann, M. A. Senar, E. Luque, and M. Livny. Adaptive scheduling for master-worker applications on the computational grid. In R. Buyya and M. Baker, editors, *Grid Computing - GRID 2000*, pages 214–227. Springer-Verlag LNCS 1971, 2000.

[IJT06] Dennis B. Gannon Matthew Shields Ian J. Taylor, Ewa Deelman, editor. *Workflows for e-Science : Scientific Workflows for Grids*. Springer ; 2007 edition, December 29, 2006.

[IKM12] Daniel Lázaro Iglesias, Derrick Kondo, and Joan Manuel Marquès. Long-term availability prediction for groups of volunteer resources. *J. Parallel Distrib. Comput.*, 72(2) :281–296, 2012.

[JCD+13] Gideon Juve, Ann Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. Characterizing and profiling scientific workflows. *Future Generation Computer Systems*, 29(3) :682–692, 2013. Special Section : Recent Developments in High Performance Computing and Security.

[JK09] Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.

[JKW07] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *STTT*, 9(3-4) :213–254, 2007.

[JMFJ07] Simon Tjell Joao Miguel Fernandes and Jens Baek Jorgensen. Requirements engineering for reactive systems with coloured petri nets : the gas pump controller example. In *In Proceedings of the 8th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 207–222, October 22-24, 2007.

[Jou04] Jean-Pierre Jouannaud. Theorem proving languages for verification. In Farn Wang, editor, *ATVA*, volume 3299 of *Lecture Notes in Computer Science*, pages 11–14. Springer, 2004.

[JQ12] Thomas Fahringer Jun Qin, editor. *Scientific Workflows : Programming, Optimization, and Synthesis With ASKALON and AWDL*. Springer-Verlag Berlin and Heidelberg GmbH and Co. K, 14 août 2012.

[JWH11] Hans-Arno Jacobsen, Yang Wang, and Patrick Hung, editors. *IEEE International Conference on Services Computing, SCC 2011, Washington, DC, USA, 4-9 July, 2011*. IEEE, 2011.

[KAZ] `http://www.kazaa.com/`.

[KC96] Paul Kogut and Paul Clements. A survey of architecture description languages. In *In Proc. of the 8th International Workshop on Software Specification and Design*, page 16. IEEE Computer Society Press, 1996.

[KKD09] Najla Hadj Kacem, Ahmed Hadj Kacem, and Khalil Drira. A formal model of a multi-step coordination protocol for self-adaptive software using coloured petri nets. In *IJCIS 7*, 2009.

[KKJD08] Najla Hadj Kacem, Ahmed Hadj Kacem, Mohamed Jmaiel, and Khalil Drira. Towards modelling and analysis of a coordination protocol for dynamic software adaptation. In Richard Chbeir, Youakim Badr, Ajith Abraham, Dominique Laurent, Mario Köppen, Fernando Ferri, Lotfi A. Zadeh, and Yukio Ohsawa, editors, *CSTST*, pages 499–507. ACM, 2008.

[KLK12] Hatem Hadj Kacem, Imen Loulou, and Ahmed Hadj Kacem. A formal approach to model and verify the behaviour of publish/subscribe architectural style. *IJITCC*, 2(3) :234–252, 2012.

[LAJ14] Christophe Cerin Heithem Abbes Leila Abidi, Walid Saad and Mohamed Jemni. Wide area bonjourgrid as a data desktop grid : modeling and implementation on top of redis . In *26th International Symposium on Computer Architecture and High Performance Computing*, october 2014.

[LBE03] Bogdan Lesyng, Piotr Bala, and Dietmar W. Erwin. Eurogrid–european computational grid testbed. pages 590–596, 2003.

[LGLK79] J.K. Lenstra, R. Graham, E. Lawler, and A.H. Kan. Optimization and approximation in deterministic sequencing and scheduling : a survey. *Annals of Discrete Mathematics*, 5 :287–326, 1979.

[LSS+09] Stefan M. Larson, Christopher D. Snow, Michael Shirts, Vijay S. P, and Vijay S. Pande. Folding@home and genome@home : Using distributed computing to tackle previously intractable problems in computational biology, 2009.

[MHD+10] Ross Miller, Jason Hill, David A Dillow, Raghul Gunasekaran, Galen M Shipman, and Don Maxwell. Monitoring tools for large scale systems. *CUG'10*, 2010.

[MHMT97] Robin Milner, Robert Harper, David MacQueen, and Mads Tofte. *The Definition of Standard ML, revised edition.* 1997.

[Mon] http://montage.ipac.caltech.edu/index.html.

[MS94] H. Madeira and J.G. Silva. Experimental evaluation of the fail-silent behavior in computers without error masking. In *24th Fault Tolerant Computing Symposium FTCS-24*, n/a, 1994.

[NAP] http://opennap.sourceforge.net/napster.txt.

[Opea] http://www.openstack.org/.

[Opeb] http://opennebula.org/.

[Opec] https://www.openshift.com/.

[OSG] http://www.opensciencegrid.org/.

[Peg] http://pegasus.isi.edu/projects/pegasus.

[PST04] Irk Pruhs, Jiri Sgall, and Eric Torng. On-line scheduling. In J. Leung, editor, *Handbook of Scheduling : Algorithms, Models, and Performance Analysis*, pages 15.1–15.43. CRC Press, 2004.

[RED] http://redis.io/.

[SACJ12] Walid Saad, Heithem Abbes, Christophe Cérin, and Mohamed Jemni. A self-configurable desktop grid system on-demand. In Fatos Xhafa, Leonard Barolli, and Kin Fun Li, editors, *3PGCIC*, pages 196–203. IEEE, 2012.

[Sar02] Luis F. G. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. *Future Generation Computer Systems*, 18 :561–572, 2002.

[SBB+99] Philippe Schnoebelen, Béatrice Bérard, Michel Bidoit, François Laroussinie, and Antoine Petit. *Vérification de logiciels : techniques et outils du model-checking*. Vuibert, April 1999.

[Sga98] J. Sgall. On line scheduling-a survey. In *On-Line Algorithms*, Lecture Notes in Computer Science 1442, pages 196–231. Springer-Verlag, Berlin, 1998.

[SSCC11] Jean-Paul Smets-Solanes, Christophe Cérin, and Romain Courteaud. Slapos : A multi-purpose distributed cloud operating system based on an erp billing model. In Jacobsen et al. [JWH11], pages 765–766.

[Sys] https://sourceware.org/systemtap/documentation.html.

[TER] http://www.teragrid.org/.

[Try12] Denis Trystram. Les riches heures de l'ordonnancement. *Technique et Science Informatiques*, 31(8-10) :1021–1047, 2012.

[VBP03] P. Vicat-Blanc Primet. High performance grid networking in the datagrid project. *special issue Future Generation Computer Systems*, January 2003.

[YN14] Paolo Gianessi Congfeng Jiang Yanik Ngoko, Christophe Cérin. Energy-aware service provisioning in volunteers clouds. *International Journal of Big-Data Intelligence (IJBDI)*, 2014.

[ZGB03] Luca Zanolin, Carlo Ghezzi, and Luciano Baresi. An approach to model and validate publish/subscribe architectures, 2003.

Annexe A

Comparaison des outils pour le monitoring d'activité

	Impact sur la mémoire et les ressources	**Détails du monitoring**	**Exécution et codage**
Strace 2001	- Utile pour des programmes avec des **crashs** fréquents et des comportements **inattendus**	- **Opérations** : Diagnostique, débogage, monitoring des interactions entre le noyau Linux et les processus - **Cibles** : appels systèmes et changement d'état des processus. - **Sortie** : Affichage des appels systèmes faits par un programme. - **Avantage** : Plus facile à utiliser qu'un débogueur pour un administrateur système. - **Limites** : Incapacité à détecter autant de problèmes qu'un débogueur de code	- **Basé** sur l'utilisation de l'élément du noyau : **ptrace**.
Dprobes : Dynamic Probes 2004		- **Avantage** : Utilisation d'un mécanisme dynamique d'instrumentation : Kprobe	- **Basé** sur l'insertion dynamique de **probes** dans le code en exécution
LTTng : Linux Trace Toolkit Next Generation 2005	Impact : **Faible** en mode actif et très faible en mode inactif - Monitoring des problèmes de performance dans des systèmes **parallèles** ou de **temps réels**	- **Opérations** : Basé sur un les mécanismes d'instrumentation du noyau Linux : Tracepoint et Kprobes - **Sortie** : Utilise le projet Babeltrace pour traduire les données en un log lisible - **Avantages** : Support basique des architectures basées sur Linux et facilité de personnalisation de l'outil.	- **Basé** sur des **modules** du noyau pour le monitoring et des **librairies** dynamiquement chargées pour le monitoring d'applications - Contrôlé par une session daemon instruite par une interface d'une **ligne de commande**

	Impact sur mémoire et ressources	**Détails du monitoring**	**Exécution et codage**
ProbeVue	- Plateforme légère et dynamique	- **Opérations** : Sondage de processus exécutés, analyse de stats et récupération de données - **Limites** : Langage Vue manque d'agrégations de données et utilise une liste des de types (de données) qui offre fonctionnalités limitées	- Suivi de performance globale Il s'intéresse à des **évènements** spécifiques liés à un ou plusieurs **threads** sans les modifier
Dtrace : Dynamic trace 2005	- Impact : Minimal en mode actif et pas de performance en mode inactif - Utilité : Des dizaines de milliers de probes DTrace probes peuvent être activées en même temps et de nouvelles peuvent être créées dynamiquement	- **Opérations** : Monitoring des problèmes du noyau et des applications de temps réel - **Cibles** : Vue globale de l'état : capacité mémoire, temps CPU, système de fichiers fichiers et ressources et ressources réseau pour les processus actifs - **Sortie** : Une probe génère des informations de faible granularité pour les afficher ou les écrire dans un log, les stocker dans une base de données ou les modifier dans les variables de contexte. - **Inconvénients** : les scripts se comptent plutôt en centaines de lignes.	- Basé sur le **langage D** inspiré du C et qui offre des fonctions et des variables spécifiques. - Structuré en **programmes AWK** : une liste de probes - Probe (associée à une action) analyse l'état d'une exécution en récupérant les variables de **contexte** et évaluant des **expressions** - Suivi des **arguments** utilisés par une certaine fonction en exécution ou la liste des processus accédant à un certain **fichier**. - La modification des variables de contexte permet aux probes d'**échanger** des informations et **coopérer** dans l'analyse de la corrélation des évènements

	Impact sur la mémoire	**Détails du monitoring**	**Exécution et codage**
Kernel marker 2007	-Impact :**Faible** (valeurs immédiates) - Des parties du **code** seraient dynamiquement **désactivées** et **activées** sans utiliser ni références en mémoire ni cache	- **Opérations** : Prédire l'utilisation du Kprobe qui dépend des breakpoints - **Inconvénients** : Les Markers masquent l'instrumentation dans le code source : pénaliser les conventions et la modification sauf si monitoring de tout l'arbre du noyau - Vérification des types limitée aux types scalaires : l'API dépend du format des chaînes de caractères - **Remplacé** par **Tracepoints** : plus simple et sécurisé : probes statiques	- Instrumentation statique - Permet à LTTng ou SystemTap de suivre les informations générées par les points probes. - Modification du code source facile : il contient lui-même les markers. - Branche ajoutée au-dessus d'appel de fonction : ni le stack ni l'appel de fonction ne seront exécutés quand l'instrumentation est désactivée
SystemTap		- **Opérations** : Basé sur les événements (tels que les débuts ou les fins des scripts) et compilé en points probes comme les "tracepoints" Linux. Basé aussi sur l'exécution des fonctions ou les états du noyau ou de l'espace utilisateur. - **Sortie** : Lancé avec la ligne de commande Stap et enregistré sous des fichiers .stp - **Avantages** : attacher des markers DTrace si compilés comme des applications avec des macros du fichier sys/sdt.h	- Basé sur un langage **script** Suivi dynamique :Extraction,filtrage et résumé. Script compilé en **module de noyau** puis **chargé** dans le noyau. Utilise **types** de données et **structures de contrôle**. Types non déclarés : ils sont vérifiés automatiquement à l'exécution - Il utilise des **Tapsets** : bibliothèques de sondes et de fonctions avec des alias pré-écrites
SysStat		- **Opérations** : Surveillance avancée de performance. Créer une base de performance mesurable du serveur - **Sortie** : Monitorer des niveaux du système et générer des information dans des fichiers - **Avantage** : Large couverture des statistiques de performance	- Capacité à formuler, évaluer avec précision et de conclure ce qui a mené à un problème ou un événement inattendu.

Annexe B

Exemples de scripts pour la configuration et le lancement d'une application SlapOS dans Grid5000

B.1 Rattachement des volontaires à SlapOS

Voici ci dessous le script Shell lancé sur chaque nœud volontaire de Grid5000 afin de le rattacher au Cloud SlapOS.

```
echo "BEGIN REGISTER"
http_proxy=http://proxy:3128 https_proxy=http://proxy:3128
slapos node register --interface-name tapVPN
--master-url https://[2001:470:1f14:169:8850:7aff:fe6d:47dc]:10009
--master-url-web https://grid5000master.host.vifib.net/
--login-auth --partition-number 5 $1
--login demo
--password demo
/usr/sbin/slapos-start
slapos node format --now
echo "END REGISTER"
```

Le login et le mot de passe sont demandés, utiliser le login "demo" et le mot de passe "demo" correspondant à un compte utilisateur pré-configuré par défaut dans l'image du master. `slapos node format -now`, permet de vérifier le bon fonctionnement.

B.1.1 Création de la recette adaptée à l'application

Ci-dessous la recette RedisDG-SlapOS-Grid5000 permettant de faire le lien entre les trois systèmes.

```
[buildout]
eggs-directory = /opt/slapgrid/c601293946e22731a4eade123386bd2a/eggs
develop-eggs-directory = /opt/slapgrid/c601293946e22731a4eade123386bd2a/develop-eggs
newest = false
offline = false

parts =
  publish-connection-informations
  redis
  redis-dg
[rootdirectory]
recipe = slapos.cookbook:mkdirectory
etc = ${buildout:directory}/etc/
srv = ${buildout:directory}/srv/
var = ${buildout:directory}/var/
bin = ${buildout:directory}/bin/
tmp = ${buildout:directory}/tmp/

[basedirectory]
recipe = slapos.cookbook:mkdirectory
services = ${rootdirectory:etc}/service/
run = ${rootdirectory:var}/run/
log = ${rootdirectory:var}/log/
[master-passwd]
recipe = slapos.cookbook:generate.password
storage-path = ${rootdirectory:etc}/.passwd
bytes = 4

[redis]
recipe = slapos.cookbook:redis.server
server_bin = /opt/slapgrid/c601293946e22731a4eade123386bd2a \
             /parts/redis/bin/redis-server
ipv6 = ${slap-network-information:global-ipv6}
port = 6379
use_passwd = ${slap-parameter:use_passwd}
pid_file = ${basedirectory:run}/redis.pid
server_dir = ${rootdirectory:srv}
passwd = ${master-passwd:passwd}
config_file = ${rootdirectory:etc}/redis.conf
log_file = ${basedirectory:log}/redis.log
wrapper = ${basedirectory:services}/redis_server

[redis-dg]
<= redis
recipe = slapos.cookbook:redis.dg
python-bin = /opt/slapos/parts/python2.7/bin/python2.7

#Please provide here the main script of redis-dg
redisdg-script = /opt/slapgrid/c601293946e22731a4eade123386bd2a \
                 /parts/redis-dg/main.py
```

```
wrapper = ${basedirectory:services}/redis_DesktopGrid
root-dir = ${buildout:directory}
work-directory = ${rootdirectory:srv}
tmp-dir = ${rootdirectory:tmp}
log-file = ${basedirectory:log}/redisDG.log
pid-file = ${basedirectory:run}/redisDG.pid

#The list of files used in redisDG, one per line
job-desc = ${slap-parameter:job}
deamon = ${slap-parameter:deamon}
#redis = ${redis:ipv6}
redis=2001:470:1f14:169:e0a9:f6ff:fe15:8068
eggs-directory = /opt/slapgrid/c601293946e22731a4eade123386bd2a/eggs

# Send informations to SlapOS Master
[publish-connection-informations]
recipe = slapos.cookbook:publish
server_url = ${redis:ipv6}
port = ${redis:port}
passwd = ${redis:passwd}
use_master_password = ${redis:use_passwd}

[slap-parameter]
use_passwd = false
job = {"config":"/opt/slapgrid/c601293946e22731a4eade123386bd2a \
                /parts/redis-dg/Montage_1446.xml", "files": {}}
#Define type of instance to request. manager | worker | all
deamon = all
```

B.1.2 Lancement des workers et du coordinateur

Ci-dessous le script permettant de démarrer les instances workers. Il doit être lancé par chaque machine worker.

```
REDISDG="https://raw.githubusercontent.com/alaintakoudjou \
                /slapos/master/software/redisdg/software.cfg"
ID_COMP=$(cat /etc/opt/slapos/slapos.cfg | egrep computer_id.*= | awk '{print $3}')
for i in $(seq $1)
 do
  slapos supply $REDISDG $ID_COMP
  slapos request "$2 RedisDGWorker $i" "$REDISDG"
         --node "computer_guid=$ID_COMP"
         --parameters "deamon=worker"
  sleep 20
 done
```

Ci-dessous le script permettant de démarrer le coordinateur.

```
REDISDG="https://raw.githubusercontent.com/alaintakoudjou \
```

```
                /slapos/master/software/redisdg/software.cfg"
ID_COMP=$(cat /etc/opt/slapos/slapos.cfg | egrep computer_id.*= | awk '{print $3}')
slapos supply $REDISDG $ID_COMP
slapos request "$1 RedisDG Master" "$REDISDG"
       --node "computer_guid=$ID_COMP"
       --parameters "deamon=manager"
```