

Numéro National de Thèse : 2017LYSEN089

THESE de DOCTORAT DE L'UNIVERSITE DE LYON
opérée par
l'Ecole Normale Supérieure de Lyon

Ecole Doctorale N° 512
École Doctorale en Informatique et Mathématiques de Lyon

Spécialité de doctorat :
Informatique

Soutenue publiquement le 11/12/2017, par :
Pedro Paulo DE SOUZA BENTO DA SILVA

On the mapping of distributed applications onto multiple Clouds

Contributions au placement d'applications distribuées sur multi-clouds

---

Devant le jury composé de :

JEANNOT, Emmanuel	Directeur de Recherche	Inria Bordeaux	Rapporteur
PASCAL-STOLF, Patricia	Maître de Conférences	IUT de Blagnac	Rapporteure
ELMROTH, Erik	Professeur	University of Umeå, Suède	Examinateur
MORIN, Christine	Directeur de Recherche	Inria Rennes	Examinatrice
PEREZ, Christian	Directeur de Recherche	Inria Grenoble Rhône-Alpes	Directeur de thèse
DESPREZ, Frédéric	Directeur de Recherche	Inria Grenoble Rhône-Alpes	Co-encadrant de thèse

Acknowledgements / Agradecimentos

First of all I would like to thank my advisors Christian Perez and Frédéric Desprez for helping me passing over the many obstacles of this scientific journey.

I also would like to express my deepest gratitude to the members of the jury, Christine Morin, Patricia Pascal-Stolf, Emmanuel Jeannot and Erik Elmroth for their presence at the defense of this thesis. I also would like to thank the revisors of this thesis, Emmanuel Jeannot and Patricia Pascal-Stolf for taking their time to read and evaluate this text and for sharing with me their very important comments.

It was a pleasant experience to work at the LIP and at Inria all those years. I'm very grateful for the support and infrastructure made available which allowed me to develop my research and participate on many conferences and workshops. Clearly, none of that would be possible without LIP's staff composed by assistants, Evelyne Bleslé, Marie Bozo, Chiraz Benamor, Laetitia Lécot, Sylvie Boyer, Catherine Desplanches and Nelly Amsellem, and the team of (ex-)engineers Laurent Pouilloux, Simon Delamare, Mathieu Imbert, Jean Christophe Mignot, Serge Torres and Dominique Ponsard.

I also would like to thank all the PhD students from the LIP. In special, I would like to thank those that became close friends and with whom I shared many happy moments (and eventually some not so happy ones).

I start thanking all the people that survived sharing the same office with me. In special, I would like to thank Violaine VV, Liovaine LV and Radu Carpa. Thanks Violaine VV, my dearest officemate for 2 years long and holder of the "Palme d'or des Co-bureaux", who has always been there for me whenever I needed. Basically you helped me in almost everything since day one: movings (yes, in plural), e-mail and report reviews, administrative stuff, snowboarding, etc. Thanks a lot, VV!!! I must also thank Radu for our long discussions about algorithm ideas, Linux administration (not exactly discussions because the guy is a damn hacker), and existential dilemmas. Also thanks for all the administrative help you gave me during our teachings at Lyon 1. I hope one day we will get over not having bought bit coins when they were cheap. Thanks Liovaine LV, for all the private French classes and sorry for being such a bad student. Hopefully now I have a good enough French "*rhythme*".

Laurent Pouilloux, thanks for teaching me how to develop a decent environment for running my experiences and for introducing me to Grid'5000.

Marcos Assunção, *grazzie* for all the many professional, philosophical and existential discussions. Also thanks a lot for reviewing and correcting all my articles and the text of this thesis.

Aurélien Cavelan and Maroua Maalej, thank you for being part with me of the or-

ganization committee of the "Journée des Doctorants du LIP 2016". It was a really nice experience.

Fabrice Mouhartem, thanks for helping to keep the PhD seminars alive!

Anthony "Ton-ton-thony", thanks for presenting me the "incontournables" of French music. When I'm at a party and "Le Bal Masqué" starts playing, everyone is astonished when I sing along word by word.

Helène Coullon, thanks a lot for the many discussions and advice! Let's embrace the Bretagne!

Thanks Alexandre for all the very interesting related work about stream processing and for taking the time to discuss that with me.

Thank you, Aurélie for all the delicious grandma-made "*confitures*"!

Jad, thank you for accepting to cross the France with me for our movings! Again, let's embrace the Bretagne!

I also would like to thank the many wonderful people I met during my thesis outside the gates of the ENS-Lyon.

Thanks Simon, Cécile and Garance for always being there, for the wonderful trips, barbecues and soirées!

Thanks Ton-ton-thony and Seb, my almost-neighbors, for the many picnics and soirées!

Barbara, it was really cool to share the same city with you again!

Adelha, Mayroquera, M, DJ Rodragson, Renata, Tiago, Rafa and Lina, thank you for the many "voz e violão" parties, discussions and trips. Mainly, thank you for your friendship and for being there!

Christophe, Fanny, Pierre, Antoniette and Mo, the "Le Briey" crew, thanks for all the wine and French culinary classes, and for making our Friday and Saturday nights unique.

Of course I cannot forget to thank the good old friends who stayed in Brazil (but were still very present in my life) and those who I met again in France after four years.

To my high school friends from CEFET-SP, thank you for helping me keeping the moral! Also, thank you for finding the time to go around with me during my flash visits to Brazil! Love you, guys.

The "Caravana da Depressão" will not be forgotten! Thank you Dani Mingatos (have you drank your CBB today?), Leo Takuno, Renato Ramalho, Cris Ikenaga, Fábio Franco, Anderson Borbulhas e Marcelo Hashimoto!

Mylena and D. Montana, thank you for the night long philosophical and existential discussions! Thank you for being there!

Karine, I'm still grateful for the very first ski and snowboard lesson you gave me!

Thank you, Fábio, Sato and Antoine, for our unforgettable adventure in the start-up world! We still have a couple of years to became millionaires before 35, isn't, Fábio?

Thanks a lot Corinne, Arthur, Leonard and the De Decker family, in special Antoine and Agnès, for adopting me so many times during my stay in France, specially during holidays. I will never forget the Christmas of 2014!

Finally, I would like to thank my family, the main supporters of this very special chapter of my academic life. This part of the text will be in Portuguese.

Em primeiro lugar, eu gostaria de agradecer aos meus pais Pedro e Marlene, que, desde minha infância, me ensinaram a importância da busca pela conhecimento. Sou muito grato por vocês terem acreditado no meu potencial mesmo quando falsos educadores lhes diziam que eu seria incapaz de aprender. Obrigado pelos sacrifícios pelos quais vocês passaram para que eu chegasse aqui. Agradeço também às minhas irmãs Luana e Mayra pelo apoio.

Faço também um agradecimento especial à minha avó Natália, que sempre foi um grande exemplo para mim desde criança. Sua vontade de aprender e dedicação aos estudos sempre me inspirou. Estendo (REF) meu agradecimento à todos os membros da família Souza, aos meus queridos tios e tias, primos e primas. Faço um agradecimento especial à minha tia Marli que foi minha primeira professora de francês e que plantou em mim a vontade de conhecer a França.

Não posso deixar também de agradecer aos queridos tios e tias, primos e primas da família Bento. Em especial, agradeço minhas queridas tias Fia, Rosa e Márcia, pelas mensagens virtuais de carinho e preocupação que me enviam frequentemente e por sempre estarem dispostas a me receber com festa durante minhas visitas relâmpago ao Brasil.

Por fim, faço um agradecimento especial à minha companheira Huana que desde meus primeiros passos na vida acadêmica tem me apoiado e motivado. Obrigado por responder com amor e companheirismo os meus frequentes episódios de ausência, mau humor e frustação. Aproveito também para agradecer aos membros da família Ota e em especial à Dona Hatsuyo e à Dona Emiko, por sempre terem me feito sentir parte de suas famílias.

Abstract

The Cloud has become a very popular platform for deploying distributed applications. Today, virtually any credit card holder can have access to Cloud services. There are many different ways of offering Cloud services to customers. In this thesis we specially focus on the Infrastructure as a Service (IaaS), a model that, usually, proposes virtualized computing resources to costumers in the form of virtual machines (VMs). Thanks to its attractive pay-as-you-use cost model, it is easier for customers, specially small and medium companies, to outsource hosting infrastructures and benefit of savings related to upfront investments and maintenance costs. Also, customers can have access to features such as scalability, availability, and reliability, which previously were almost exclusive for large companies.

To place a distributed application on the Cloud, a customer must first consider the mapping between her application (or its parts) to the target infrastructure. She needs to take into consideration cost, resource, and communication constraints to select the most suitable set of VMs, from private and public Cloud providers. However, defining a mapping manually may be a challenge in large-scale or time constrained scenarios since the number of possible configuration explodes. The large offer of Cloud providers in the market and eventual advantages of having an application deployed over different Cloud sites, like redundancy and reachability, for example, make this challenge even more complex. Furthermore, when automating the mapping definition, scalability issues must be taken into account since this problem is a generalization of the graph homomorphism problem, which is NP-complete.

In this thesis we address the problem of calculating initial and reconfiguration placements for distributed applications over possibly multiple Clouds. Our objective is to minimize renting and migration costs while satisfying applications' resource and communication constraints. We concentrate on the mapping between applications and Cloud infrastructures. Using an incremental approach, we split the problem into three different parts and propose efficient heuristics that can compute good quality placements very quickly for small and large scenarios. First we model the problem as a communication oblivious problem and propose vector packing based heuristics able to calculate initial placement solutions. Then, we extend our application and infrastructure models by introducing communication constraints, and propose a graph homomorphism based heuristic to calculate initial placement solutions. In the last part, we introduce application reconfiguration to our models and propose a heuristic able to calculate communication and reconfiguration aware placement solutions.

These heuristics have been extensively evaluated against state of the art approaches such as MIP solvers and meta-heuristics. We show through simulations that the proposed heuristics manage to compute solutions in a few seconds that would take many hours or

days for other approaches to compute.

Résumé

Le Cloud est devenu une plate-forme très répandue pour le déploiement d'applications distribuées. Aujourd'hui, virtuellement tout titulaire d'une carte bancaire peut avoir accès à des services provenant d'un Cloud. De plus en plus d'entreprises peuvent sous-traiter leurs infrastructures d'hébergement et, ainsi, éviter les dépenses d'investissements initiaux en infrastructure et de maintenance. Beaucoup de petites et moyennes entreprises, en particulier, ont désormais accès à des fonctionnalités comme le passage à l'échelle, la disponibilité et la fiabilité, qui avant le Cloud étaient presque réservées à des grandes entreprises. Son modèle attractif de coûts "pay-as-you-go", qui permet aux clients de payer sur demande les services et ressources utilisés, a un rôle important dans la popularisation du Cloud.

Les services du Cloud sont offerts aux utilisateurs de plusieurs façons. Dans cette thèse, nous nous concentrons sur le modèle d'Infrastructure sous Forme de Service. Ce modèle permet aux utilisateurs d'accéder à des ressources de calcul virtualisés sous forme de machine virtuelles (MVs).

Pour placer une application distribuée sur le Cloud, un client doit d'abord définir l'association entre son application (ou ses modules) et l'infrastructure. Il est nécessaire de prendre en considération des contraintes de coût, de ressources et de communication pour pouvoir choisir un ensemble de MVs provenant d'opérateurs de Cloud (Cloud providers) publiques et privés le plus adapté. Cependant, étant donné le nombre exponentiel de configurations possibles, la définition manuelle de l'association entre une application et une infrastructure est un défi dans des scénarios à large échelle ou ayant de fortes contraintes de temps.

La multitude d'opérateurs de Cloud et les avantages d'un placement pertinent d'une application distribuée sur plusieurs sites du Cloud, comme la redondance et l'accessibilité, par exemple, rendent ce challenge encore plus complexe. En outre, pour automatiser ce processus, le passage à l'échelle d'un algorithme d'association ne peut être ignoré. En effet, le problème de calculer une association entre une application et une infrastructure est une généralisation du problème de homomorphisme de graphes, qui est NP-complet.

Dans cette thèse, nous adressons le problème de calculer des placements initiaux et de reconfiguration pour des applications distribuées sur potentiellement de multiples Clouds. L'objectif est de minimiser les coûts de location et de migration en satisfaisant des contraintes de ressources et communications. Pour cela, nous utilisons une approche incrémentale en divisant le problème en trois sous-problèmes et proposons des heuristiques performantes capables de calculer des placements de bonne qualité très rapidement pour des scénarios à petite et large échelles.

Premièrement, nous modelons le problème en tant qu'un problème de placement sans communication et proposons des heuristiques basés sur des algorithmes de vector packing

pour calculer des solutions de placements initiaux. Deuxièmement, nous étendons nos modèles d'applications et d'infrastructures par l'introduction de contraintes de communication et proposons une heuristique basé sur des algorithmes de homomorphisme de graphes pour calculer des solutions de placements initiaux conscients des communications. Troisièmement, nous introduisons le concept de reconfiguration d'applications à nos modèles et proposons une heuristique capable de calculer des solutions de placements conscients des communications et de reconfiguration.

Ces heuristiques ont été évaluées en les comparant avec des approches de l'état de l'art comme des solveurs exactes et des meta-heuristiques. Nous montrons en utilisant des simulations que les heuristiques proposées parviennent à calculer des solutions de bonne qualité en quelques secondes tandis que des autres approches prennent des heures voir des jours pour les calculer.

Contents

1 Introduction **1**
- 1.1 The Challenge of Placing Applications . . . 1
- 1.2 Motivation: Cloud Computing . . . 3
- 1.3 Objective . . . 4
- 1.4 Approach to the Problem . . . 4
- 1.5 Summary of Contributions . . . 5
- 1.6 Publications and Communications . . . 5
  - 1.6.1 Peer reviewed conferences . . . 5
  - 1.6.2 Workshop Presentations (Invited Talks) . . . 5
  - 1.6.3 Poster Presentation . . . 6
- 1.7 Thesis Structure . . . 6

2 Context **7**
- 2.1 Distributed Systems . . . 7
- 2.2 Cloud Computing . . . 8
  - 2.2.1 Defining the Cloud . . . 8
  - 2.2.2 Deployment Models . . . 9
  - 2.2.3 Service Models . . . 9
  - 2.2.4 IaaS Revisited . . . 10
  - 2.2.5 Containers . . . 12
  - 2.2.6 Cloud Infrastructure . . . 13
  - 2.2.7 Discussion . . . 14
- 2.3 Distributed Applications . . . 14
  - 2.3.1 Component-Based Software Paradigm . . . 15
  - 2.3.2 Component-Based Models . . . 16
  - 2.3.3 Discussion . . . 17
- 2.4 Placement on the Cloud . . . 17
  - 2.4.1 Placement in Application Life Cycle . . . 18
  - 2.4.2 Strategies for Performing an Automated Placement . . . 19
- 2.5 Conclusion . . . 21

3 Problem Definition and Methodology **23**
- 3.1 Objective and Problem Definition . . . 23
  - 3.1.1 Distributed Applications . . . 23
  - 3.1.2 Cloud-Based Infrastructure . . . 24
  - 3.1.3 Placement . . . 25

3.2 Approach . . . 28
3.3 Evaluation Methodology . . . 30
3.3.1 Strategy . . . 30
3.3.2 Experiment . . . 30
3.3.3 Baseline Algorithms . . . 32
3.3.4 Evaluation Metrics . . . 32
3.4 Conclusion . . . 32

4 Initial Cost-Aware Placement **33**
4.1 Introduction . . . 33
4.2 Problem Statement . . . 33
4.2.1 Optimization Problem Formulation . . . 34
4.3 Related Work . . . 35
4.3.1 The Multi Dimensional Bin Packing Problem . . . 36
4.3.2 Strategies Based on Exact Algorithms . . . 36
4.3.3 Strategies Based on Meta-Heuristics . . . 37
4.3.4 Strategies Based on Greedy Heuristics . . . 37
4.3.5 Discussion . . . 40
4.4 Improved Greedy Heuristics . . . 41
4.4.1 Choice Of Greedy Heuristics to be Adapted . . . 41
4.4.2 Adding Cost-Awareness . . . 41
4.4.3 Heterogeneous Bins . . . 42
4.4.4 The Greedy Group . . . 43
4.5 Evaluation . . . 46
4.5.1 Methodology . . . 46
4.5.2 MIP Solver and Simulated Annealing Analysis . . . 47
4.5.3 Greedy Heuristics . . . 49
4.6 Conclusion . . . 55

5 Initial Communication and Cost-Aware Placement **57**
5.1 Introduction . . . 57
5.2 Communication-Aware Placement of Distributed Applications on Multiple Clouds . . . 58
5.2.1 Problem Statement . . . 59
5.2.2 CAPDAMP as Graph Homomorphism Problem . . . 60
5.2.3 Optimization Problem Formulation . . . 60
5.3 Related Work . . . 61
5.3.1 Exact Algorithms . . . 61
5.3.2 Meta-heuristics . . . 62
5.3.3 Heuristics . . . 62
5.3.4 Discussion . . . 63
5.4 Modeling the Cloud Infrastructure and Distributed Applications . . . 64
5.4.1 Cloud Network Topology . . . 64
5.4.2 Distributed Application Communication Topology . . . 65
5.5 Two Phase Communication-Aware Heuristic . . . 65
5.5.1 Phase 1: Decomposition . . . 66

5.5.2 Phase 2: Composition . . . 69
5.5.3 2PCAP Algorithm . . . 70
5.5.4 Discussion . . . 72
5.5.5 2PCAP Complexity . . . 72
5.6 Examples . . . 73
5.6.1 Example #1 . . . 73
5.6.2 Example #2 . . . 74
5.7 Evaluation . . . 74
5.7.1 Methodology . . . 77
5.7.2 2PCAP Performance on Small Problems . . . 78
5.7.3 2PCAP Performance on Large Problems . . . 80
5.8 Conclusion . . . 82

6 Communication and Cost-Aware Placement With Reconfiguration 85
6.1 Introduction . . . 85
6.2 Problem Statement . . . 85
6.2.1 Distributed Application and Cloud Computing Models . . . 86
6.2.2 Predicted Duration of Application Execution . . . 86
6.2.3 Reconfiguration Specificities . . . 86
6.2.4 Placement Objective . . . 87
6.2.5 Optimization Problem Formulation . . . 87
6.3 Related Work . . . 89
6.3.1 Migrating Components . . . 90
6.3.2 Modeling Migration Costs . . . 90
6.3.3 Discussion . . . 91
6.4 The 2PCAP-REC Heuristic . . . 91
6.4.1 Application and Cloud Model . . . 92
6.4.2 Reconfiguration Model . . . 93
6.4.3 2PCAP-REC – Two Phase Communication and Reconfiguration Aware Placement Heuristic . . . 93
6.4.4 2PCAP-REC Algorithm . . . 95
6.4.5 2PCAP-REC Complexity . . . 96
6.5 Evaluation . . . 98
6.5.1 Methodology . . . 98
6.5.2 Small Problems (Class A Experiment) . . . 100
6.5.3 Medium and Larger Problems (Class B and C Experiments) . . . 100
6.5.4 Discussion . . . 104
6.6 Conclusion . . . 104

7 Conclusion and Perspectives 105
7.1 Perspectives . . . 106
7.1.1 Different Use Case Perspectives . . . 106
7.1.2 Improvement of Application and Constraint Model Perspectives . . . 107
7.1.3 Multi-objective Optimization . . . 108

List of Figures

1.1 Web application placement example. 2
1.2 Generic view of a placement. 3

2.1 Comparison between VM and container environments. 13
2.2 Application life cycle. 15
2.3 Simple component-based application. 16
2.4 More complex example of component based application. 16
2.5 Topology model from TOSCA as implemented by Winery. 17
2.6 Representation of the "Production/Placement" step. 18

3.1 Distributed application and Cloud infrastructure models. 24
3.2 Example of scenario from Hypothesis 7 26
3.3 Examples of resource constraint problems. 27
3.4 Examples of communication constraint problems. 29

4.1 Initial placement of distributed applications on multiple clouds problem. 34
4.2 MIP solver vs. S.A. – Class A – cost distances. 49
4.3 MIP solver vs. Greedy Group – Class A – cost distances. 50
4.4 S.A. vs. Greedy Group – Class A – cost distances. 50
4.5 S.A. vs. Greedy Group – Class B – cost distances. 51
4.6 Cumulated S.A. vs. Greedy Group – Class B – cost distance. 52
4.7 Executions times from S.A. and Greedy Group. 52
4.8 Execution time ration between S.A. and Greedy Group. 53
4.9 Greedy Group solution ranking. 53
4.10 Distribution of solutions from Greedy Group. 54

5.1 Communication-aware placement example. 59
5.2 Cloud topology and application graph. 64
5.3 Placement example. 67
5.4 Complete Placement Example #1 (cf. Section 5.6.1). 75
5.5 Complete Placement Example #2 (cf. Section 5.6.2). 76
5.6 Application topology schemes. 78
5.7 MIP solver vs. 2PCAP – Class A – cost distances. 79
5.8 Sum of execution times in seconds from 2PCAP and SCIP solver. 79
5.9 Relaxed CAPDAMP vs. 2PCAP – Classes B and C – cost distances. . . 80
5.10 S.A.1 vs. 2PCAP – Classes B and C – cost distances. 81
5.11 S.A.2 vs. 2PCAP – Classes B and C – cost distances. 82

5.12 2PCAP's execution times. 83

6.1 Reconfiguration placement scenario. 88
6.2 Application and Cloud topologies. 92
6.3 Reconfiguration placement example. 94
6.4 S.A.1 vs. 2PCAP-REC – Classes B and C – cost distances. 101
6.5 S.A.2 vs. 2PCAP-REC – Classes B and C – cost distances. 102
6.6 S.A.3 vs. 2PCAP-REC – Classes B and C – cost distances. 103
6.7 Migration cost amortization – Classes B and C – cost distances. 103
6.8 2PCAP-REC execution times for Classes B and C experiments. 104

List of Tables

2.1 Part of the catalog of Rackspace's IaaS services. 10
2.2 Part of the catalog of Amazon EC2's IaaS services. 11
2.3 Part of the catalog of Google Cloud Platform's IaaS services. 11
2.4 Example of Amazon EC2's data transfer costs. 12

3.1 Example of experiment class parameters. 31
3.2 Example of intervals of data generation for dimensions. 31

4.1 Summary of variables used in Equation 4.1. 36
4.2 Rank matching example of First Fit Windowed Multi-Capacity. 40
4.3 Experience classes. 47
4.4 Intervals of data generation. 47
4.5 Execution times from greedy heuristics 54
4.6 Greedy Group potential compositions. 55

5.1 Summary of variables used in this section and in Equation 5.2. 61
5.2 Parameters of experiment classes. 77
5.3 Intervals of dimension data generation. 78

6.1 Summary of variables used in in Equation 6.2. 89
6.2 Problem classes and their parameters. 98
6.3 Intervals of dimension data generation. 99

List of Algorithms

1 Simulated Annealing 21
2 First Fit Decreasing 38
3 Function assign called inside Algorithm 2 38
4 First Fit Decreasing Priority 43
5 Dot Product 44
6 First Fit Windowed Multi-Capacity 45
7 2PCAP 71
8 calculate 96
9 2PCAP-REC (Differences to Algorithm 7 are highlighted) 97

Chapter 1

Introduction

Cloud computing has become a popular platform for deploying applications as it provides an attractive pay-per-use model and enables customers to tap into features that in the past were available only to large corporations, including fast scalability, availability, and reliability. Cloud computing services can be accessed in multiple ways. This includes *Infrastructure as a Service* (IaaS), a very common *service model* that consists of providing computing resources to costumers, usually as a *virtual* environment composed of virtual machines (VMs), onto which they can deploy their applications.

Before deploying applications, users need to define which *Cloud providers* – organizations responsible for providing Cloud based infrastructures to users – to contract and which *Cloud computing resources* to *rent*. Today, this task is challenging. To deal with the increasing demand for Cloud computing services, the number of Cloud providers has grown very quickly. Consequently, the number of possible infrastructure configurations for deploying an application explodes. Thus, in many cases, the process of choosing a set of computers (or VMs) that best suits a distributed application becomes impractical in terms of time complexity.

1.1 The Challenge of Placing Applications

We call *application placement* the two-step process of *installing* an application on a hosting infrastructure (such as the Cloud). The first step involves defining a hosting infrastructure and *logically mapping* the application or its parts onto it. The second step, the actual deployment of the application on the infrastructure, follows the logical map defined in the first step.

Application placement embodies *initial* and *reconfiguration* placements. The former consists in placing an application that has not been previously deployed onto an infrastructure. The latter considers reconfiguring an application, or its parts, that has already been placed.

Usually, a hosting infrastructure must satisfy previously defined application constraints in order to successfully deploy and run an application. Other placement objectives may apply, such as renting cost minimization, data transfer minimization, and energetic consumption minimization.

Figure 1.1, illustrates the placement of a simple Web application. The mapping and deployment of machines $m1$ as host of “Http Server", $m2$ as host of “Application Server”

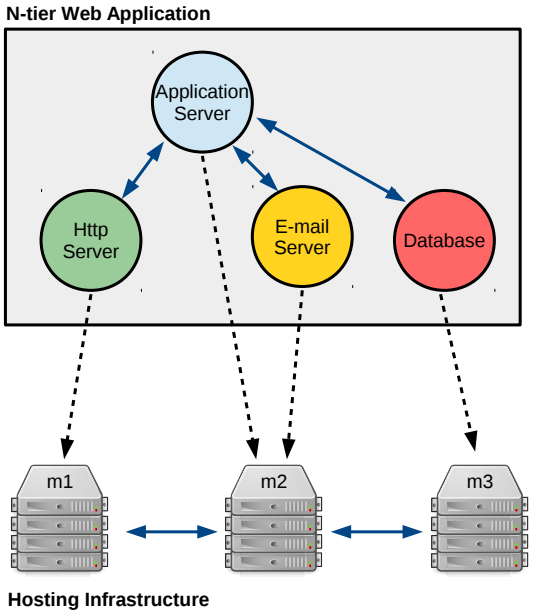


Figure 1.1: Web application placement example. The gray rectangle represents the Web application as a whole, i.e. the way users see it. Inside the rectangle we illustrate as circles the programs necessary to make the application work. Solid arrows indicate communication between programs and hosting infrastructure. Dashed arrows illustrate the hardware onto which those programs were installed.

and "E-mail server", and $m3$ as the "Database" host represent, together, the application placement. Notice that $m1$, $m2$ and $m3$ could be three VMs, for example. It is also important to observe that any of the Web application parts could have been deployed on other machines before being assigned to $m1$, $m2$, or $m3$. In this case, instead of an initial placement, we would have a reconfiguration placement.

Placements can be either *manually* performed or *automated*. In large scenarios, however, as the number of possible configurations *explodes*, it becomes difficult for an application manager to take into account all application constraints manually. In this type of situation, *automation* is an interesting option. Nevertheless, the problem of placing an application onto an infrastructure is *NP-Complete* [54]. This means that there is not a known *polynomial time algorithm* that can solve it optimally. Consequently, scalability issues must be taken into consideration.

We illustrate a more general case of application placement in Figure 1.2. It summarizes the placement process by representing a distributed application, a hosting infrastructure, the mapping between application and infrastructure parts, and the deployment of application parts on the infrastructure. As application parts could have been previously deployed on other infrastructures, this figure is representative of initial and reconfiguration placements.

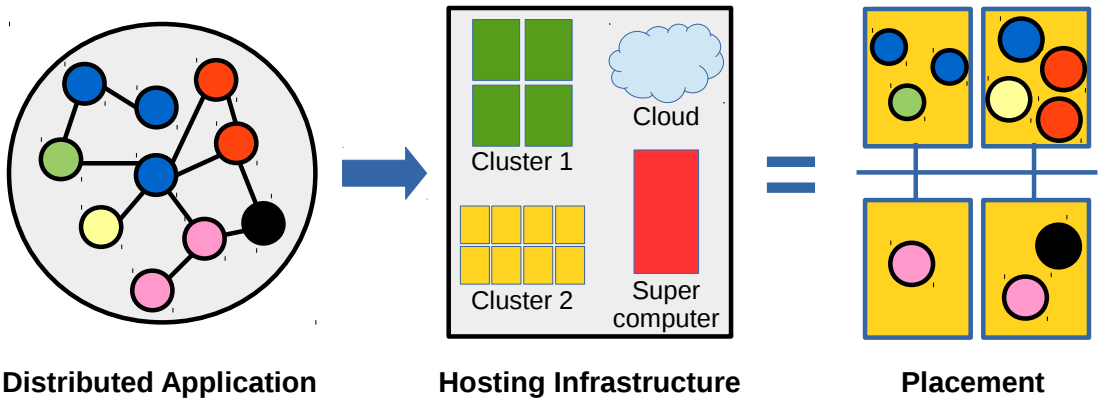


Figure 1.2: Generic view of a placement. The circle to the left represents a distributed application (the subparts of the distributed application are represented in small connected colored circles), the rectangle in the middle represents the available hosting infrastructures and, to the right, the connected rectangles filled with colored circles represent the placement of the distributed application over one of the available hosting infrastructures.

1.2 Motivation: Cloud Computing

When choosing an environment to host a distributed application, a manager must choose among possibly thousands of VM types, from multiple *private* and *public* Cloud providers, those that are capable of hosting each application part considering application requirements, VM prices, and VM resources. If the scenario involves an application that is already running but whose requirements have changed, the application manager also has to consider the trade-off between benefits of moving parts of the application – in terms of performance or renting costs, for example – and *migration costs* – stemming from unavailability of the application during the moving, for example. Considering those issues manually may not be an option since the number of possible configurations is exponential.

Due to the potential size of problems and the time taken by a manager to manually analyze all possible configurations of placement, the challenge is more evident when we consider scenarios where an application must be placed in *narrow time constraints*.

Users may be faced with short-term deadlines to execute their large-scale applications due to economic advantages, position on marketplace, internal strategy, etc.. We list some examples where manually calculating a placement may be a time consuming obstacle:

- A manager wants to quickly deploy an application to benefit of less expensive hosting infrastructure during a resource auction, such as AWS Spot Instances[1] auctions.
- Under a sudden increase in the number of requests to an N-tier Web application, it may be necessary to quickly react and deploy replicas of some of the tiers.
- Large crisis management systems, such as large-scale simulation, data analysis or

[1] https://aws.amazon.com/ec2/spot

information management systems, may have to be deployed immediately after a disaster.

- The launch of a more cost effective hosting infrastructure could trigger the moving/migration of an entire application. To avoid impacting clients, the new placement must be performed quickly.

Hence, automating application placement is therefore crucial. As we discuss in the next sections, there is an extensive literature on this subject but in spite of the important contributions made by previous work, there are still many open issues concerning *scalability* of proposed solutions. Most related work concentrates on solving small to medium sized problems, i.e., few VM types or small applications. They usually propose solutions based on *exact algorithms*, which do not scale, or *meta-heuristics*, which in spite of being able to give solutions for large problems in feasible time, have their solution quality dependent on the amount of time used to compute it.

1.3 Objective

We are interested in the problem of calculating a mapping of parts of a distributed application onto multiple Cloud-based infrastructures with the objective of minimizing costs while meeting resource and communication constraints. Costs stem from VM renting or from migrating previously deployed distributed application parts to other VMs.

We consider placement problems whose settings are beyond the limits of exact algorithms and meta-heuristics. Consequently, we consider that placement problems may involve hundreds of Cloud provider sites and thousands of different VM types. Hence, in this thesis, we concentrate on the study of efficient heuristics, which are scalable and able to compute good quality placement solutions very quickly.

Our interest is in the *mapping step* of the placement, but we will use the terms placement and mapping interchangeably.

1.4 Approach to the Problem

We used an *incremental* approach to model the placement problem and to design and develop scalable heuristics able to deal with large-scale scenarios.

We started with a simplified version of the placement problem which did not take into consideration *communication constraints*. Our solution was based on vector packing heuristics able to compute *initial* placements, i.e. we considered that applications would be deployed for the first time and there *were not previously deployed* parts of applications. Then, we incremented our model by adding communication constraints. For this problem, we proposed a divide and conquer *communication-aware placement heuristic* which uses the previous heuristic. Finally, we added *migration costs* and the ability to represent *reconfiguration scenarios* (i.e. situations where previously deployed applications parts may need to be moved to other hosts) to the placement model, and proposed a reconfiguration-aware extension of the previous communication-aware heuristic.

1.5 Summary of Contributions

In this thesis we proposed three primary contributions and a secondary contribution. The primary contributions are the efficient placement heuristics briefly discussed in Section 1.4 and the secondary contribution is the extensive evaluation process that each of those heuristics had to undergo to be validated. We organize these contributions as follows:

- **Cost-Aware Placement Heuristics:** Set of heuristics based on efficient multi-dimensional bin packing heuristics that can calculate initial solutions for communication oblivious placement problems.
- **Cost and Communication-Aware Heuristics:** Divide and conquer communication-aware heuristic able to calculate initial solutions for placement problems.
- **Cost, Communication and Reconfiguration-Aware Heuristics:** Divide and conquer communication-aware heuristic that can calculate solutions for placement problems. It is able to compute initial and reconfiguration placements.
- **Heuristic Evaluation:** We individually evaluate the proposed heuristics by comparing them to baseline algorithms based on state of the art approaches, namely Mixed Integer Programming solvers and Simulated Annealing meta-heuristics.

1.6 Publications and Communications

The contributions discussed in this thesis were published or presented in several scientific articles and workshops.

1.6.1 Peer reviewed conferences

1. P. Silva, C. Perez, and F. Desprez, Efficient Heuristics for Placing Large-Scale Distributed Applications on Multiple Clouds. *In 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'16)*, May 2016, Cartagena, Colombia, pp. 483–492.
2. P. Silva and C. Perez, An Efficient Communication Aware Heuristic for Multiple Cloud Application Placement. *23rd International European Conference on Parallel and Distributed Computing (Euro-par 2017)*, August 2017, Santiago de Compostela, Spain, pp. 372–384.

1.6.2 Workshop Presentations (Invited Talks)

- **Hot Topics in Distributed Computing**, Flaine, France, March 2015: Presentation of "Efficient heuristics for the placement of component-based applications on the Cloud";
- **New Challenges in Scheduling Theory**, Aussois, France, April 2016: Presentation of "Efficient Heuristics for Placing Large-Scale Distributed Applications on Multiple Clouds";

- **GdR RSD Journées Cloud**, Nice, France, September 2016: Presentation of "Efficient Communication-Aware Heuristics for Placing Large-Scale Distributed Applications on Multiple Clouds";
- **11th Cloud Control Workshop**, Stockholm, Sweden, June 2017: Presentation of "An Efficient Communication Aware Heuristic for Multiple Cloud Application Placement".

1.6.3 Poster Presentation

- **Conférence Parallélisme, Architecture et Système (COMPAS 2015)**, Lille, France, July 2015: Poster "Efficient heuristics for the placement of multi-tier applications on the Cloud".

1.7 Thesis Structure

This thesis is organized as follows. In Chapter 2, we present the necessary context and basic concepts related to distributed systems, distributed applications, and Cloud computing that will be used throughout this thesis. In Chapter 3, we describe our objective, hypothesis, and evaluation methodology.

We describe our contributions in Chapters 4, 5, and 6. In Chapter 4, we propose communication oblivious heuristics able to calculate initial placement solutions. In Chapter 5, we propose communication-aware heuristics that can calculate initial placement solutions. In Chapter 6, we introduce communication-aware solutions to calculate initial and reconfiguration placement solutions.

We conclude this thesis and discuss perspectives in Chapter 7.

Chapter 2

Context

In this chapter we discuss key concepts related to the placement of distributed applications on the Cloud. The objective is to contextualize our contributions (Chapters 4, 5, and 6) and create a common ground for definitions and vocabulary.

We discuss in more detail some of the placement challenges introduced in Chapter 1. First we characterize the main aspects related to hosting infrastructures and distributed systems, focusing on *Cloud Computing*, which is the target platform of this thesis. Then, we briefly describe the main properties of distributed applications, and conclude this chapter with a discussion about the challenges and specificities of placing applications on the Cloud.

2.1 Distributed Systems

In this section we introduce the concept of *distributed systems*, which characterizes the hosting infrastructure of distributed applications.

Definition 1. *"A distributed system is a collection of independent computers that appears to its users as a single coherent system" [106].*

Essentially, distributed systems (cf. Definition 1) are the underlying systems that *host* distributed applications. They are composed of several autonomous computation entities (such as virtual or physical machines) connected over a network. The way in which machines communicate and the fact that processes and resources may be physically distributed across multiple computers should be hidden from users of hosted applications. In the same way, failures in individual computers should be treated by the distributed system without impacting hosted applications.

An infrastructure composed of resources from Cloud providers as well as computer clusters and grids is a good example of distributed system.

For the reader interested in a more extensive discussion on distributed systems, please refer to [106].

2.2 Cloud Computing

Cloud computing is a model that appeared commercially in early 2000s and through which customers could access computing services over the Internet and pay for what they use. Its popularization in late 2000s pushed by very large enterprises, such as Amazon [6], Microsoft [77] and Google [45], changed the way companies managed their computing infrastructures.

In the next sections, we discuss in more detail the main characteristics of Cloud Computing and present its most usual service and deployment models.

2.2.1 Defining the Cloud

There are many Cloud Computing definitions available in the literature [109]. In this thesis we use the definition proposed by the *National Institute of Standards and Technology* of the U.S. Department of Commerce [74] (cf. Definition 2).

Definition 2. *"The Cloud Computing is a model for enabling ubiquitous, convenient,* ***on-demand*** *network access to a* ***shared pool*** *of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be* ***rapidly provisioned and released*** *with minimal management effort or service provider interaction" [74].*

Before expanding Definition 2, we need to present two key actors from Cloud computing: *cloud computing service providers* and *cloud service customers* .

A *Cloud computing service provider* (*Cloud provider* for brevity) is the entity, person or organization responsible for making a service available to *cloud service customers*. It is also responsible for managing the necessary infrastructure to correctly deliver its services to cloud service customers. For example, Google [42], Amazon [6], Microsoft [77], Rackspace [95] and IBM [57] are examples of cloud providers.

A *Cloud service customer* (*customer* for brevity) is "a person or organization that maintains a business relationship with, and uses services from, cloud providers"[18].

From Definition 2 we highlight the words "on-demand", "shared pool" and "rapidly provisioned and released" because they summarize some of the main characteristics of Cloud Computing.

Cloud computing services are served *on-demand* and accessed on the Internet. This means that customers may require computing capabilities from Cloud providers at any time. Furthermore, as computing capabilities must be *rapidly* provisioned, this process must be completely automated. Computing capabilities can also be *elastically* provisioned and released at any time by customers. Elasticity provisioning means a dynamic adaptation of provisioned computing capabilities commonly without service interruption [27].

The contract between a costumer and a Cloud provider is called *Service Level Agreement* (SLA) and establishes prices and quality of services. To satisfy SLAs, Cloud providers resources are *pooled* to serve multiple customers. Those resources, can be physical or virtual and are dynamically assigned and reassigned to customers according to their demand.

Finally, Cloud providers also must dispose ways of measuring, controlling, and reporting precise resource usage from customers. The cost of services are usually based on those metrics, thus, users pay only for what they consume.

2.2.2 Deployment Models

A Cloud Computing Infrastructure may be deployed according to three main types of models: *Private*, *Public*, and *Hybrid*. The difference between them lies in the level of access they provide for customers.

A *private Cloud* (or *corporate Cloud*) infrastructure has its services offered to customers who belong to a single organization. On the other hand, *public Cloud* offers its infrastructure to the general public. *Hybrid Clouds* mix private and public Clouds. In this case, a Hybrid Cloud is composed of two or more Cloud infrastructures, with different deployment models, that bind together logically to form one Cloud.

Most of the main companies offering Cloud Computing services propose Public Cloud services, such as Amazon EC2 [4] or Microsoft Azure [78], and Private Cloud services, such as Amazon Virtual Private Cloud [5] or Microsoft Azure Stack [79].

2.2.3 Service Models

There are three main types of Cloud Computing service models: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS) [74]. The main difference between them is the way computing capabilities are offered to customers, nevertheless all service models share the characteristics previously discussed in Section 2.2.1.

SaaS – Software as a Service

In the Software as a Service (SaaS) model, Cloud providers offer access to applications running on their infrastructure. Usually customers connect to those applications over the Internet, frequently using a web browser, while the provider is responsible for all the underlying infrastructure necessary to run the application with the appropriate service agreement. Online e-mail clients, calendars, file hosting services or text editing tools are examples of SaaS applications. The main advantages of this model is the ease of access specially for data oriented applications and the possibility of paying on-demand (pay as you use) licenses for expensive or sophisticated applications. For example, Evernote [32] and Google Apps [44] allow users to access productivity software over the Internet for free. Autodesk's Fusion 360 [11], offers less expensive on-demand online licences to computer-aided design (CAD) programs.

PaaS – Platform as a Service

In the Platform as a Service (PaaS) model, providers offer instant computing infrastructures as in IaaS and also offer the necessary software layer for a specific objective. For example, a user may want to rent a server ready to run a Map-Reduce application, so it could have Hadoop [9], Cassandra [8] and other related software ready to use. The

Name	RAM	vCPUs	System Disk	Bandwidth	Raw Infra	Managed Infra
General1-1	1GB	1	20GB SSD	200Mb/s	$0.032/hr	$0.005/hr
General1-2	2GB	2	40GB SSD	400Mb/s	$0.064/hr	$0.01/hr
General1-4	4GB	4	80GB SSD	800Mb/s	$0.128/hr	$0.02/hr
General1-8	8GB	8	160GB SSD	1600Mb/s	$0.256/hr	$0.04/hr

Table 2.1: Reproduction of part of the catalog of Rackspace's IaaS services as of available 16/07/2017 at `https://www.rackspace.com/cloud/servers/pricing`

main advantage of this model is that software licensing and resource elasticity are usually managed by the service providers, letting users concentrate only on the application. One example of PaaS is "HDInsight" [80] which provides full configured and maintained machines ready to run *Big Data* related applications. Another one is the Google App Engine [43] which provides the necessary environment to the development of Web applications.

IaaS – Infrastructure as a Service

In the Infrastructure as a Service (IaaS) model, Cloud providers offer instant computing infrastructures to customers. These infrastructures commonly consist of computing servers hosted in *virtual machines* (VMs). While Cloud providers are in charge of the maintenance and management of the physical infrastructure, it is up to the customers to purchase, install, configure, and manage software to be deployed on the provisioned computing infrastructure. Table 2.1 illustrates a list of VMs offered by Rackspace [95]. For example, a customer could rent five "General1-1" servers with 1GB of RAM, 20GB of storage, 1 virtual CPU and network bandwidth of 200Mb/s and pay US$0.032 per hour, per server, totalizing US$0.16 per hour. The main advantage of this model is that it allows users to deploy or *outsource* their applications without having to *invest* on data centers. As this service model is central to the this thesis, it will be presented in more detail in the next section.

2.2.4 IaaS Revisited

As the IaaS model has a central role in this thesis, we discuss in more detail some characteristics of this service model.

Virtual Machines

In Section 2.2.3 we used the term "computing infrastructure" to describe services offered to IaaS customers by Cloud providers. Throughout this thesis, we consider that computing infrastructures are *virtual machines* (cf. Definition 3).

Definition 3. *A virtual machine (VM) is a computer program capable of running an operating system and applications [30]. It must be "an efficient, isolated duplicate of the real machine"[94].*

VMs are hosted in *physical machines*. There can be several VMs running at the same time in the same physical machine, however, each VM is isolated from the others, i.e. a

	vCPU	ECU	RAM (GiB)	Storage (GB)	Linux/Unix Usage
t2.nano	1	Variable	0.5	EBS Only	$0.0059 per Hour
t2.micro	1	Variable	1	EBS Only	$0.012 per Hour
t2.small	1	Variable	2	EBS Only	$0.023 per Hour
t2.medium	2	Variable	4	EBS Only	$0.047 per Hour
t2.large	2	Variable	8	EBS Only	$0.094 per Hour
t2.xlarge	4	Variable	16	EBS Only	$0.188 per Hour
t2.2xlarge	8	Variable	32	EBS Only	$0.376 per Hour
m4.large	2	6.5	8	EBS Only	$0.1 per Hour
m4.xlarge	4	13	16	EBS Only	$0.2 per Hour
m4.2xlarge	8	26	32	EBS Only	$0.4 per Hour
m4.4xlarge	16	53.5	64	EBS Only	$0.8 per Hour
m4.10xlarge	40	124.5	160	EBS Only	$2 per Hour
m4.16xlarge	64	188	256	EBS Only	$3.2 per Hour

Table 2.2: Reproduction of part of the catalog of Amazon EC2's IaaS services as of available 16/07/2017 at `https://aws.amazon.com/ec2/pricing/on-demand/`

Machine type	Virtual CPUs	Memory	Price (USD)	Preemptible price (USD)
n1-standard-1	1	3.75GB	$0.0475	$0.0100
n1-standard-2	2	7.5GB	$0.0950	$0.0200
n1-standard-4	4	15GB	$0.1900	$0.0400
n1-standard-8	8	30GB	$0.3800	$0.0800
n1-standard-16	16	60GB	$0.7600	$0.1600
n1-standard-32	32	120GB	$1.5200	$0.3200
n1-standard-64	64	240GB	$3.0400	$0.6400

Table 2.3: Reproduction of part of the catalog of Google Cloud Platform's IaaS services as of available 16/07/2017 at `https://cloud.google.com/compute/pricing`

VM is not aware from other VMs existence. VMs have access to virtualized resources (e.g. memory, storage, computing) which provide the same functionality as the physical hardware. The piece of software responsible for managing resources and isolation of VMs in the same physical machine is called a *virtual machine monitor* [94] or *hypervisor*.

Usually, Cloud providers dispose of a catalog of multiple *VM types*. Each VM type has distinct resource configurations and price, and it is up to the customers to find the most adapted VM types for their business. We illustrate in Tables 2.1, 2.2, and 2.3 part of the catalog of VM types from Rackspace [95], Amazon EC2 [4], and Google Cloud Platform [45]. Even if we used only small samples from each catalog, it is possible to notice the amount and variety of available VM types.

At customers request, VMs with characteristics of their respective types are started or *instanced* on physical machines from Cloud provider's infrastructure. In general, it is of the interest of Cloud providers to install multiple VMs per physical machine as this allows for fewer hardware, smaller data center foot print, and indirectly reduced power consumption [111]. This process is called *consolidation*.

Data Transfer OUT From Amazon EC2 To Internet	
First 1GB / month	US$ 0.000 per GB
Up to 10 TB / month	US$ 0.090 per GB
Next 40 TB / month	US$ 0.085 per GB
Next 100 TB / month	US$ 0.070 per GB
Next 350 TB / month	US$ 0.050 per GB

Table 2.4: Reproduction of amazon EC2's data transfer costs as of available at 16/07/2017 (`https://aws.amazon.com/ec2/pricing/on-demand/`)

The cost of renting and using VM depends on the contract between customer and Cloud provider. Usually, customers pay for the amount of hours a VM was available and sometimes for some specific resource usage. For example, in Table 2.2 (in the first line), renting an Amazon VM "t2.nano", would cost US$0,0059 per hour. Furthermore, following the prices available at Table 2.4, if the amount of data transfered from the VM to the Internet (outside Amazon) in one month is superior to 1GB the customer would have to pay US$0.09 per transfered GB over the surplus (up to 10TB).

Sometimes it is necessary to move a VM from a physical machine to another. This moving is called *migration* and may be needed because of lack or excess of resources in the host physical machine, or energetic issues, for example. Migrations can be done *live*, i.e. without service interruption, or *offline*, i.e. with service interruption [25].

Live VM migration is usually only possible inside the same Cloud provider. This happens mainly because of *vendor lock-in*, a consequence of hypervisor or VM *image* incompatibility which prevents users from moving their VMs to alternative Cloud providers. A VM image is a file containing a serialized copy of an entire VM with its state that can be used by a hypervisor to instantiate it. Very commonly hypervisors or VM image formats are proprietary to Cloud providers.

2.2.5 Containers

Definition 4. *"Containers are a method of operating system virtualization that allows users to run applications and their dependencies in resource-isolated processes" [100].*

Containers run *container images*, which are executable packages of a piece of software [14, 101] containing necessary system configurations, runtime, and code. The main characteristic of containers is that they run on top of a single kernel instance [101], thus sharing host's operating system and, where appropriate, libraries and binaries [14]. Thanks to that, in general, container images have reduced sizes and containers do not have to deal with the complexity of operating systems. This results in fast instantiation, deployment and boot up [101] of containers.

In this context, containers can be seen as lightweight VMs. As we discussed in Section 2.2.4, VMs run their own operating systems and have hypervisors managing their isolation from host's operating systems and from other deployed VMs. We illustrate that difference in Figure 2.1. Observe that VMs run their different operating systems over a hypervisor while containers share host's operating system.

Sharing host's operating system results in a dependency between containers and host

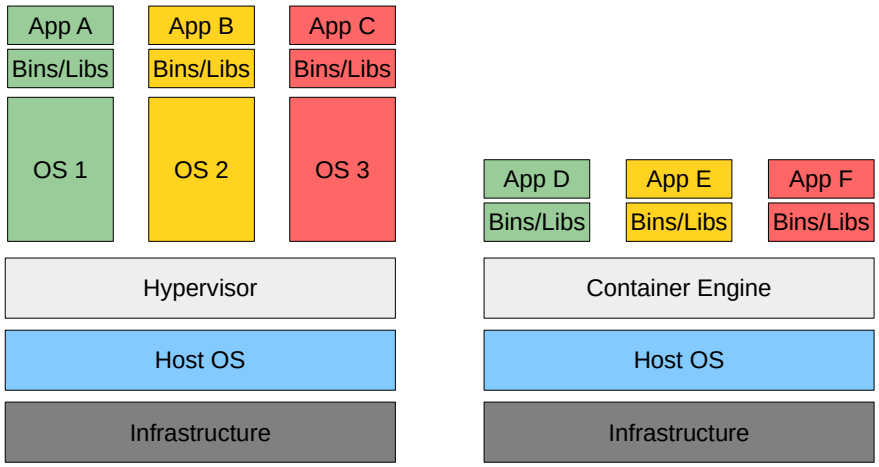


Figure 2.1: Comparison between VM and container environments.

machines. As a consequence, there is a smaller level of *isolation* between containers sharing the same host machine. However, depending on the application, this shortcoming may be acceptable and applications may benefit from container's lightweight features.

Similarly to VMs, container can be moved from a host machine to another through migrations and live migrations. However, as containers share the operating system kernel, the destination host machine must have the same operating system characteristics of the source host machine.

There are many Cloud providers, as Google [45] and Amazon [3], offering *container-based* solutions on the Cloud [1, 89]. In summary, customers provide container images and, Cloud providers the necessary infrastructure to deploy and run those images. As for VMs, Cloud providers also assure fast scalability, availability and reliability and follow a pay-as-you-use fashion.

2.2.6 Cloud Infrastructure

Usually Cloud providers have one or more *data centers*, spread across Cloud provider *sites* or *regions*, where physical machines are concentrated.

Commonly, customers can select the Cloud provider sites where their required resources will be hosted. This is usually done to reduce latency between application users and Cloud infrastructure. Although internal data center topology or details about VMs' hosting physical machines are normally hidden from customers, machines inside a same Cloud provider site are interconnected by fast networks. On the other hand, data centers from different Cloud providers are usually connected over Internet and generally do not have special or dedicated connections.

In spite of that, customers may require resources from different Cloud providers or sites to balance workload among different regions, to reduce the risk of application unavailability due to disasters, or to reduce reliance on a single vendor, for example. The term *multiple* Clouds is used to characterize situations where a customer requires resources from more than one Cloud provider or site.

2.2.7 Discussion

Cloud computing brings interesting service and deployment models which allow costumers to outsource their infrastructures and applications. In this section, we summarized the concept of Cloud computing specially focusing on the IaaS service model which is central for the contributions of this thesis. We presented the main concepts and actors related to VMs, namely Cloud providers, Cloud provider sites, and VM types, and we also discussed some important related issues such as VM type heterogeneity, connection latency and vendor lock-in.

2.3 Distributed Applications

In this section, we introduce the concept of distributed applications and component-based applications. Both are essential for describing our contributions (Chapters 4, 5 and 6) and placement scenarios.

Throughout this thesis we use the definition of *distributed application* described in Definition 5.

Definition 5. *A distributed application is a computer program installed over and capable of being executed on a distributed system. It is composed of one or more sub-programs that cooperate with one another to reach a common goal. In spite of that, it appears to its users as a single coherent application.*

Web applications similar to the one described in Figure 1.1 are examples of distributed applications. Furthermore, we can also cite distributed databases such as Cassandra [8] and MongoDB [82], big data processing frameworks such as Hadoop [9] and Spark [10], and multi-player gaming applications such as Pokemon GO [83] and World of Warcraft [17].

Distributed Application Life Cycle

We summarize in Figure 2.2 a simplified distributed application *life cycle*. In a first interaction, an application is designed based on previously gathered requirements. Once the project is ready, the application is developed and tested. After that, the next step is to put the application in production by placing it on a distributed infrastructure, then get feedback from application users to, finally, based on that, restart the cycle by designing new features.

To materialize this life cycle, specially the development and placement steps (cf. Figure 2.2), it is necessary to describe applications somehow. Very commonly, specific custom-made software is developed to interconnect application parts and to prepare them for the specific hardware onto which they will be deployed. Clearly, the result of this procedure is a coupled distributed application which is strongly dependent on the way its parts are connected and on the deployed infrastructure.

The main objective of this thesis is to understand and propose solutions for the problem of placing distributed applications on the Cloud. We seek a higher level of abstraction and flexibility for our distributed application model so our solutions can be applied to a wider range of situations. We found those two characteristics in the *component-based paradigm* [104].

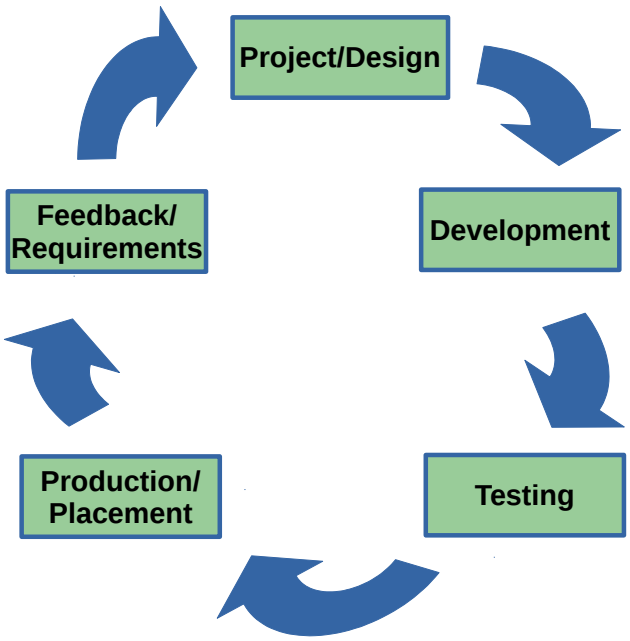


Figure 2.2: Application life cycle. Each rectangle is a step of the application life cycle. Arrows indicate the order of steps.

2.3.1 Component-Based Software Paradigm

In the component-based software paradigm, applications are described as composite systems composed of software components. The main objective of component-based software is to ease reusable software development by reducing application coupling.

Definition 6. *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" [104].*

Components (cf. Definition 6) are the basic building blocks of component-based software. They hide details of their implementation and, through *interfaces*, they expose their dependencies. Commonly, interfaces are *use ports* or *provide ports*, which roughly describe input and output dependencies. By linking components using their interfaces it is possible to build complex composite applications formally called *component assemblies*. Furthermore, as components interact with one another via well-defined interfaces, coupling is drastically reduced. With well defined interfaces and hidden implementation, components may be seen as an independent unit of computing and can be reused in third party projects. Figure 2.3 depicts an example of a very simple component-based application.

In Figure 2.4, we present a more complex example than Figure 2.3 illustrating a holiday reservation system modeled as a component-based application. Notice that components connect among themselves using their require and provide interfaces to form a *graph*.

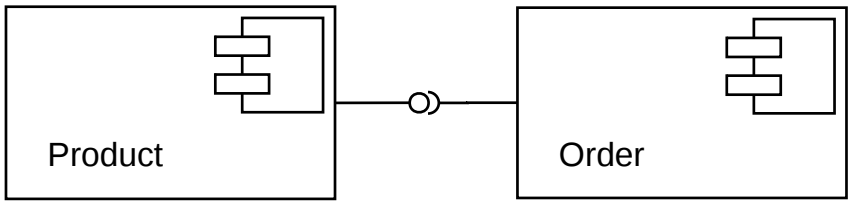


Figure 2.3: Simple component-based application described using Unified Modeling Language 2.0 (UML 2.0) [49]. There are two components with their ports connected. "Order" is consuming the output of "Product".

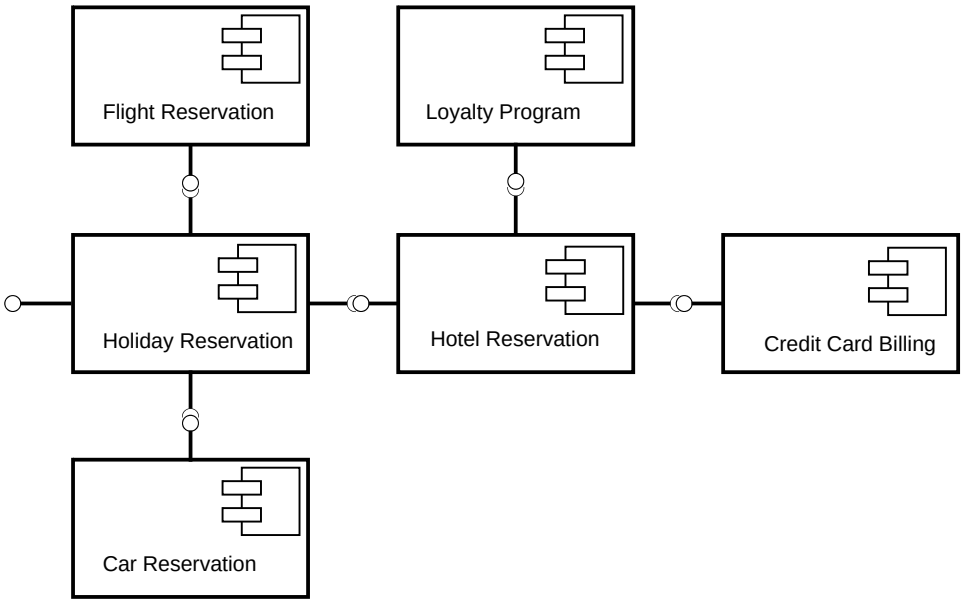


Figure 2.4: Example of a holiday reservation system modeled as a component-based application described using UML 2.0 [49]. This figure was based on the public domain figure available at `https://commons.wikimedia.org/wiki/File:Component-based-Software-Engineering-example2.png` (last accessed 30/09/2017).

2.3.2 Component-Based Models

The model which defines the semantics of interfaces and composition is called *component model*. There are many different component models proposed in the literature [21, 65, 104] such as Corba [48], Fractal [22], and TOSCA [15]. In this thesis we do not adopt any specific component-based model and a more detailed discussion is out of the scope of this work. Nevertheless, we briefly introduce TOSCA [15], a component-based model, to exemplify this concept and its advantages.

TOSCA

TOSCA (Topology and Orchestration Specification for Cloud Applications) [15] is a component model and a standard from the Organization for the Advancement of Structured

Information Standards (OASIS). It was conceived "to describe composite (Cloud) applications and their management" [15]. In summary, when deploying an application on a distributed infrastructure, such as the Cloud, it is necessary to manage the configuration of the infrastructure. This is an error-prone task since, very commonly, it is based on badly documented procedures, manual intervention, and scripts. TOSCAS's addresses these weakness by featuring fully automated deploying related procedures and further management functionalities.

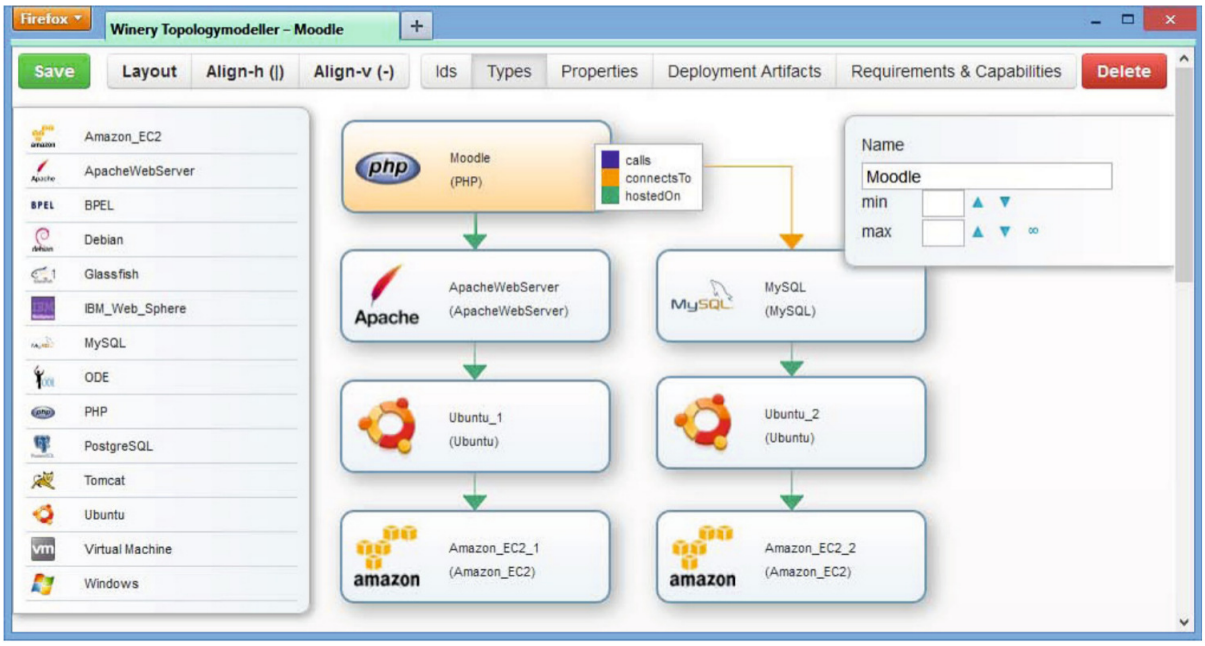


Figure 2.5: Topology model from TOSCA as implemented by Winery [63] a Web based modeling tool. This example was originally published in [63].

To achieve this, TOSCA uses two sub-models: application topology models (Figure 2.5) and manage plans. In the application topology model, hardware, computer programs and their possible relationships are described. In the management plans, the description of how to deploy and manage the application is described in Business Process Management Notation (BPMN) [47] or Business Process Execution Language (BPEL) [103]. Figure 2.5 illustrates an example of TOSCA's topology model.

2.3.3 Discussion

In this section we introduced the concept of distributed applications and discussed the modeling of this type of application using the component-based paradigm. These two points are important throughout the thesis because we need to characterize distributed applications when discussing our placement algorithms in Chapters 4, 5, and 6.

2.4 Placement on the Cloud

In Section 1.1, we concisely introduced the placement of distributed applications on distributed infrastructures and discussed scalability issues and placement automation. In

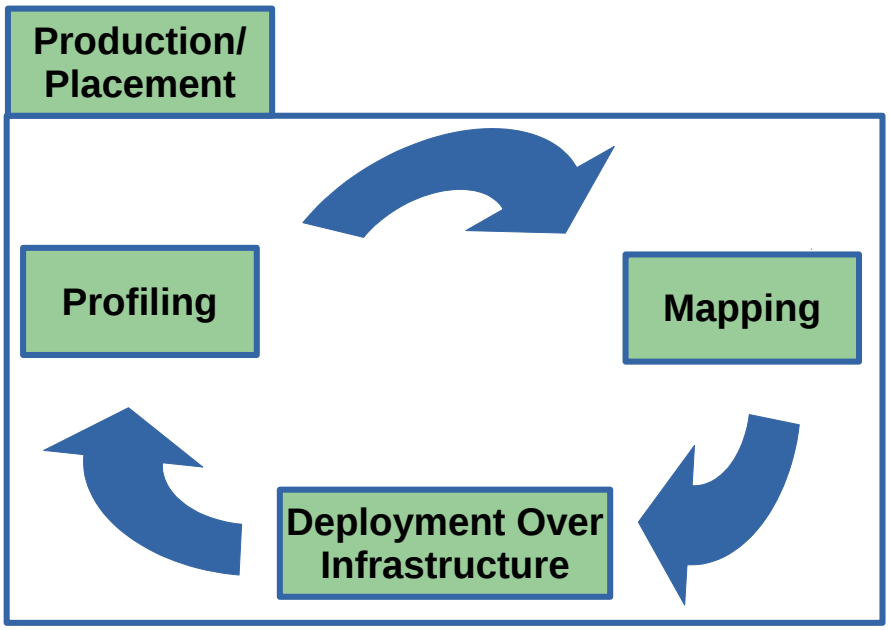


Figure 2.6: Representation of the "Production/Placement" step from Figure 2.2 in detail. Rectangles are sub-steps and arrows indicate the order between them.

this section, we discuss the usage of Cloud based infrastructures – i.e. a distributed systems composed of VMs – as a target for placing distributed applications.

2.4.1 Placement in Application Life Cycle

In this section, we discuss in more detail the step "Production/Placement" from the application life cycle (cf. Figure 2.6).

We decompose this step in Figure 2.2. Before the placement, it is necessary to know what are the *requirements* of the application and what are the *capacities* of the infrastructure. The meaning of requirements and capacities depend on the type and objectives of the application. They can be related, for example, to resource usage (CPU, RAM, disk, etc.), energetic consumption, physical location, software possibilities, etc.

We call *application profiling* the step responsible for gathering requirements from the application and *infrastructure profiling* the step responsible for gathering capacities, aligned to application objectives, from a distributed system. We generically call *profiling* the step where both application and infrastructure profiling are performed.

The profiling step is essential for the placement to be performed, however, it is out of the scope of this thesis to propose or discuss in more detail profiling methods or techniques. There is a wide bibliography about this and we direct the reader interested in more details to [71, 112].

Once requirements and capacities are gathered, the mapping step (cf. Figure 2.6) can be performed. The output of this process is a mapping between parts of the application and machines from the infrastructure. This mapping will be a road map for the actual de-

ployment and configuration of the application (cf. "Deployment Over Infrastructure" step in Figure 2.6). We do not discuss deployment methods in detail. For readers interested in this subject, we recommend [7].

2.4.2 Strategies for Performing an Automated Placement

Calculating a placement can be an issue when application and infrastructure become larger. In this thesis we consider that there is a large offer of VM types from many different Cloud providers. Because of that, the number of possible infrastructure configuration explodes. In those cases, performing placements manually may not be an option and automation emerges as a necessity. However, the placement problem is NP-Complete [54], thus, scalability must be considered.

In the following paragraphs, we briefly analyze some of the main strategies found in the state of the art of placement automation. In this section, we do not discuss it in detail because it largely varies depending on the way placement problems are defined. We prefer to discuss it more extensively in Sections 4.3, 5.3, and 6.3 after presenting and formalizing our placement problem models.

We divide the approaches available in the state of the art on placement automation in three groups: exact algorithms, heuristics and meta-heuristics.

Exact approaches

Exact approaches are those aiming at calculating optimal solutions for a problem. The most common exact approach is the usage of mixed integer problem optimization (MIP) solvers such as SoPlex [113], CPLEX [56] or CBC [87]. For that, it is necessary to model the placement problem using an *optimization problem formulation* which consists of describing the problem as a set of equalities or inequalities called *constraints* and an *objective function*. One example of an optimization problem formulation is represented in Equation 2.1, where x is a vector of variables, c and b vectors of known coefficients and A a matrix of known coefficients.

$$\begin{aligned} \text{Maximize/Minimize} \quad & c^T x \\ \text{subject to:} \quad & \\ & Ax \leqslant b \\ & x \geqslant 0 \end{aligned} \tag{2.1}$$

From that formulation, using linear or nonlinear integer programming techniques, solvers manage to calculate optimal solutions. However, mixed integer programming is NP-Hard [33] and, consequently, solvers are not scalable. To circumvent this issue, many solvers implement heuristics including branch and bound [26] or branch and cut [88] to improve time complexity, however, in the worst case, these heuristics are still bounded by solving complexity.

Hence, using solvers for calculating solutions for placement problems is an option that should be carefully analyzed as execution time can explode as problem sizes grow. We refer to [37, 41] for readers interested in a deeper discussion about mixed integer programming.

Heuristics

A *heuristic* is an approach to solve a problem without guaranteeing optimal solutions as a result. It is commonly used to calculate *approximate* solutions to NP-Hard, NP, or NP-Complete problems which do not have known *polynomial time algorithms* to optimally solve them. It is considered to be a trade-off approach because despite usually being capable of computing solutions faster than exact approaches, heuristic approaches cannot guarantee *optimality* or *completeness* (capacity of calculating all possible solutions).

Heuristics are extensively used for solving placement related problems [38, 59, 34, 107]. Greedy heuristics [38], graph clustering [59], and graph partition algorithms [120] are examples of these approaches. We discuss this related work in more detail in Sections 4.3, 5.3, and 6.3.

Meta-heuristics

Meta-heuristics are problem independent approaches used to calculate approximate solutions to hard (NP-Hard, NP, or NP-Complete) optimization problems. The main characteristics of meta-heuristics are (i) not having to deeply adapt to each problem [20] and (ii) their capacity of effectively reducing large solution *search spaces* by exploring them efficiently [105].

There are many consolidated meta-heuristics in the state of the art [20], including genetic algorithms, simulated annealing and ant colony optimization, for example. However, in this thesis, we are specially interested in the *simulated annealing* meta-heuristic because it is simpler to implement, we were already familiar with it, it does not require lots of parameter tuning to adapt it to our placement scenarios, and it managed to achieve good results in similar contexts [52].

In Chapters 5 and 6 we modify some aspects of the simulated annealing algorithm, namely solution initialization and partial solution generation. Thus, in the next paragraphs, we discuss this algorithm in more detail.

Simulated Annealing

Simulated annealing is based on the annealing process by which the slowly cooling of a hot substance is applied to obtain a strong crystalline structure [76, 105]. This is done through the simulation of the energy changes in a system subjected to a cooling system until convergence to equilibrium, i.e. the steady frozen state.

Algorithm 1 illustrates a basic implementation of simulated annealing. The algorithm receives as input an initial solution, a maximal and minimal temperature and a maximal number of steps. The initial solution is specific to a *concrete* problem. For example, it could be a representation of a valid placement. The maximal and minimal temperatures represent system's temperatures and interfere on the level of search space exploration and, consequently, on the duration of executions. The number of steps represents how many interactions by temperature grade will be done. The output of the algorithm is a valid solution to the problem.

The core part of Algorithm 1 is described between Lines 5 and 11 where for each interaction step in a given temperature *temp*, the energy of newly generated solution *new_solution* is compared to the current energy of *solution*. If *new_solution* has less

Algorithm 1 Simulated Annealing

Input: $initial_solution, temp_{max}, temp_{min}, step_{max}$
Output: $solution$

```
1: temp ← temp_max
2: solution ← initial_solution
3: while temp > temp_min do
4:     step ← 0
5:     while step < step_max do
6:         step ← step + 1
7:         new_solution ←generate_solution()
8:         Δenergy ← energy(new_solution) - energy(solution)
9:         if Δenergy ⩽ 0 or e^(−Δenergy/temp) < random_int() then
10:            solution ← new_solution
11:        end if
12:    end while
13:    temp ←update(temp)
14: end while
15: return solution
```

energy (is colder) than *solution*, then, a better solution is found. Also, to avoid local optima, *solution* may be updated with a probability $e^{\frac{-\Delta energy}{temp}}$ (Boltzmann distribution). After $step_{max}$ interactions, the temperature is updated.

The way designers model functions **generate_solution** and **energy** is the key to adapt simulated annealing to concrete problems. For example, **energy** function must be able to transform a placement representation to energy representation. One possibility is to consider the price of a placement as energy so, a solution that costs less has less energy and, therefore, will be better than a more expensive one.

2.5 Conclusion

In this chapter, our objective was to discuss and define basic concepts concerning distributed application placement approaches which are important for understanding the contributions of this thesis (cf. Chapters 4, 5, and 6).

First, we discussed the Cloud Computing and presented its main characteristics, specially its deployment and service models. Then, we presented in more detail the IaaS service model, which allowed us to discuss Cloud infrastructure and virtualization characteristics.

In a second time, we introduced the concepts of distributed application and distributed systems. We also briefly explained the component-based software paradigm and how its decoupling and reuse properties fit our needs to represent distributed applications.

Finally, we characterized the placement of distributed applications over VMs rented from Cloud providers and described the challenges of automating this procedure. This is the core of this thesis and we will discuss this subject in more detail with an extensive related work in Chapters 4, 5, and 6.

Chapter 3

Problem Definition and Methodology

There are many challenges related to calculating the placement of distributed applications on Cloud-based infrastructure. Just to cite a few, there are issues related to profiling of application and hosting infrastructure, mapping of application parts to hosting infrastructure, deployment, enactment mechanisms, system monitoring, Cloud provider price evolution, etc.. In this chapter, we precisely characterize our problem of interest and the hypotheses taken to solve it. We also describe our approach to the problem and delineate our evaluation methodology.

3.1 Objective and Problem Definition

Objective. *The objective of this thesis is to develop an approach for calculating the placement of distributed applications, modeled as component-based applications, on a Cloud-based infrastructure, composed of virtual machines (VMs) rented from possibly multiple Cloud providers. This placement must satisfy resource and communication constraints defined by the application while minimizing VM renting and reconfiguration costs.*

In this thesis, we concentrate our efforts on the challenge of mapping a distributed application onto a Cloud-based infrastructure. Other placement related issues are out of the scope of this work. In the next sections, we describe our design choices and hypothesis for the modeling of distributed applications, Cloud-based infrastructures and placement.

3.1.1 Distributed Applications

Throughout this work, we model distributed applications using the *component-based paradigm* (cf. Chapter 2). Figure 3.1(a) illustrates an example of a distributed application modeled with a simplified component-based modeling language. We consider that components have requirements which *must* be satisfied by the hosting infrastructure in order to be executed.

Hypothesis 1. *A description of the requirements of each component is available.*

Requirements can be described in terms of resource demands, such as RAM, disk, number of CPU cores or flops. Each resource requirement from a component is called a *dimension*. It is also possible that connections between components establish *connection* or *communication* requirements.

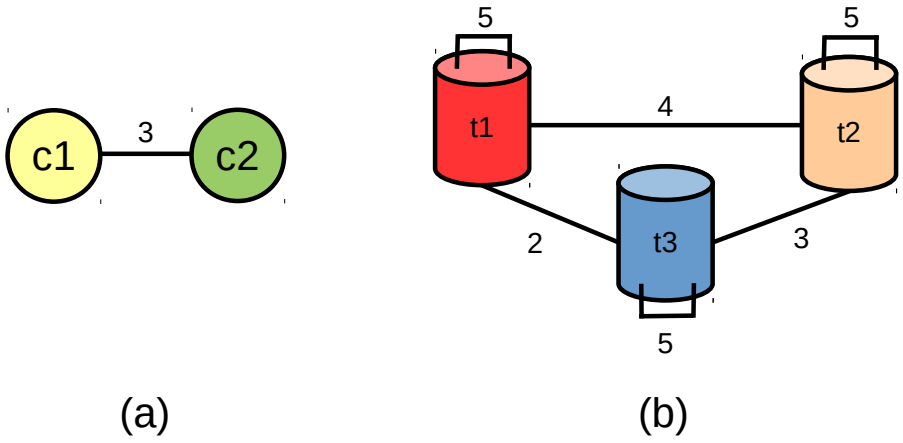


Figure 3.1: In (a), we represent a distributed application as a graph. Circles are application components, edges are connections among components, and edge weights are communication requirements. In (b), we represent a Cloud infrastructure also as a graph. Colored cylinders are VM types, edges are possible connections among instances of VM types, and edge weights are communication capacities.

Hypothesis 2. *Communication requirements are defined in terms of communication quality.*

Communication quality is a metric introduced in Chapter 5 which allows the description of *latency* in an approximative fashion.

Hypothesis 3. *Resource and communication requirements from a component have a numeric representation*[1]*.*

Describing requirements using a numeric representation is a simplification needed by the proposed algorithms presented in Chapters 4, 5 and 6. Figure 3.1(a) illustrates a simple distributed application describing communication requirements.

3.1.2 Cloud-Based Infrastructure

In this thesis, a Cloud-based infrastructure means an infrastructure composed of interconnected virtual machines (VMs) rented from possibly different cloud providers. VMs must satisfy requirements imposed by hosted application components. To do that, VM *capacities* must be larger than application requirements.

Hypothesis 4. *A description of the capacities and renting prices of each VM type is available.*

Capacities, in a similar fashion as component requirements, are described in terms of resource availabilities, including RAM, disk, number of CPU cores or flops. Latencies of connections among VMs and among Cloud provider sites can be described in terms of communication quality.

[1]It suffices that requirements have an ordered representation.

Each resource capacity from a VM type is also called *dimension*. Latency capacities are called *communication capacities*.

The following hypothesis concerning Cloud-based infrastructures are used to simplify issues we do not address in this thesis, allowing us to concentrate on the application mapping problem.

Hypothesis 5. *Resource capacities from VM types and communication capacities from VM connections have a numeric representation.*

Figure 3.1(b) illustrates an example of a Cloud infrastructure model where it is possible to observe resource and communication capacities from VM types and VM connections, respectively.

Hypothesis 6. *Prices practiced by Cloud providers will hold until the end of placement computation.*

We consider that once we started calculating solutions for a given placement problem, Cloud providers' renting prices will not change.

Hypothesis 7. *VMs can be instanced as many times as necessary.*

Let $\mathcal{C}$ be the set of components in the application. Each VM type available can be instantiated up to $|\mathcal{C}|$ times. A VM type can be used to instantiate sufficient VMs to host all application components, in the case where there is only one component per VM. Figure 3.2 illustrates this hypothesis. Observe that VM type t_1 is instantiated exactly $|\mathcal{C}|$ times.

Hypothesis 8. *We do not take into consideration the possibility of* vertically scaling*, i.e. increasing a VM's resource capacities.*

Whenever a component requires more resources than what is available in a VM, it is necessary to instantiate a larger one from a different VM type.

Hypothesis 9. *If part of the application is already deployed in the infrastructure and a reconfiguration is needed, it is possible to interrupt the execution of hosted application components at any time.*

Hypothesis 10. *VM instantiation is instantaneous.*

3.1.3 Placement

In this thesis, the terms application placement and application mapping will be used interchangeably.

Resource constraints are a collection of constraints generated from component requirements. Rented VMs must satisfy resource constraints from the components they host. A resource constraint is satisfied when the dimensions of the hosting VM are larger than or equal to the sum of hosted components dimensions.

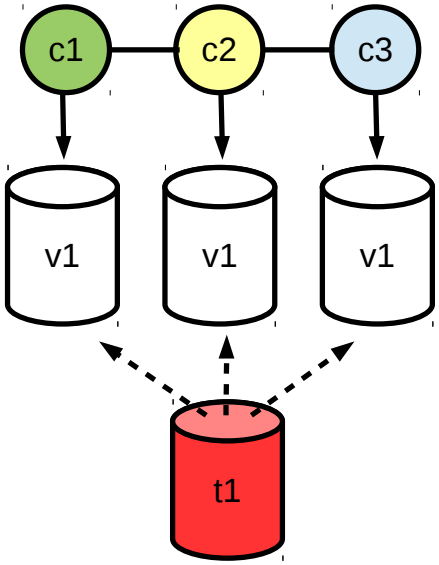


Figure 3.2: Scenario where the maximum number of VMs is instanced from VM type t_1. Notice that the number of VMs is equal to the number of components.

Hypothesis 11. *Resource constraints are hard constraints, i.e. VMs must satisfy resource constraints from hosted components.*

Figure 3.3 illustrates an example of this hypothesis. Components $c1$ and $c2$ have dimensions $(1, 3, 1)$ and $(2, 1, 3)$, respectively. VM types $t1$ and $t2$ have dimensions $(4, 6, 5)$ and $(3, 6, 3)$, respectively. VMs $v1$ and $v2$ are instanced from VM types $t1$ and $t2$, respectively.

Components $c1$ and $c2$ could be assigned individually to $v1$ or $v2$ since both components have requirements smaller than $v1$'s or $v2$'s capacities. They could also be assigned together to $v1$ given that the sum of each dimension of $c1$ and $c2$ is smaller than the capacity of $v1$ $(4 \geqslant 3, 6 \geqslant 4, 5 \geqslant 4)$. However, they cannot be assigned together to $v2$ since the sum of their third dimensions is larger than $v2$'s third capacity $(4 > 3)$.

We call *communication constraints* the set of communication requirements defined by the application that must be satisfied by rented VMs. For any pair of connected components, their hosting VM communication capacities must be smaller than or equal their individual communication requirements.

Hypothesis 12. *Communication constraints are also hard constraints, hence, a valid placement must have every communication requirement satisfied.*

Figure 3.4 illustrates examples of placements regarding communication constraints. Components $c1$ and $c2$ are connected and their connection has a communication requirement $x^{comp}_{1,2} = 3$. Consider that any of the VMs can host both components so we can concentrate on communication constraint issues.

VMs $v1$, $v2$, and $v3$ which can be instanced from types $t1$, $t2$ and $t3$, respectively, describe communication capacities $x^{vm}_{1,1} = 5$; $x^{vm}_{1,2} = 4$; $x^{vm}_{1,3} = 2$; $x^{vm}_{2,2} = 5$; $x^{vm}_{2,3} =$

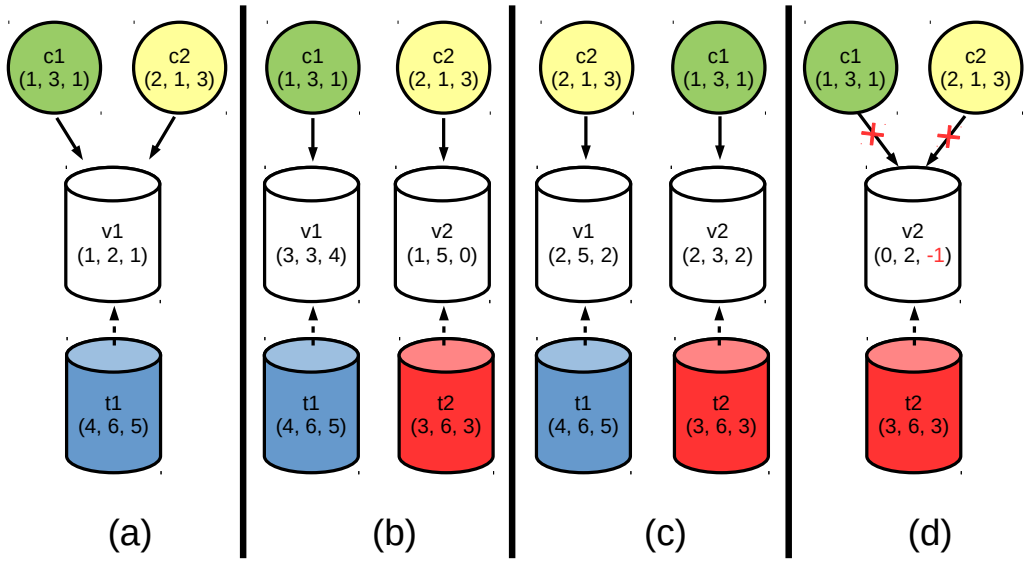


Figure 3.3: Possible placements of application components $c1$ and $c2$ on VMs instanced from VM types $t1$ and $t2$. Components $c1$ and $c2$ are represented as circles, having 3-dimensional resource requirements (1, 3, 1) and (2, 1, 3), respectively. VM types $t1$ and $t2$ are represented as colored cylinders and have resource capacities (4, 6, 5) and (3, 6, 3), respectively. Colorless cylinders $v1$ and $v2$ are VMs instanced from the VM types indicated by dashed arrows. Their available capacities – difference between the total VM capacity and the sum of resources required by components – are also illustrated. Solid arrows indicate components' host VMs. Situations (a), (b) and (c) illustrate valid placements and situation (d) an invalid one – observe that the sum of components requirements is larger than the capacities of the hosting VM.

3 and $x^{vm}_{3,3} = 5$. In this context, the only possible configurations that *do not* satisfy communication requirements defined by the application would be those represented in Figures 3.4(h) and 3.4(i). In these cases, assigning $c1$ to $v1$ and $c2$ to $v3$ or $c2$ to $v1$ and $c1$ to $v3$ is not possible because the component communication requirement between $v1$ and $v2$ ($x^{comp}_{1,2} = 3$) is larger than the communication capacity between $v1$ and $v3$ ($x^{vm}_{1,3} = 2$).

Hypothesis 13, 14 and 15 are necessary to ease the generation of placement problems during the evaluation phase. Nevertheless, it is straightforward to modify the proposed heuristics in order to add or remove costs not currently taken in consideration.

Hypothesis 13. *Placement costs are composed of renting costs and migration costs.*

Hypothesis 14. *The only cost associated to renting a VM is the result of the VM's renting price multiplied by the period, in unit of times, that the VM will be available. We do not take into consideration any other renting related costs (e.g. specific resource usage, data transfer, number of access, etc.).*

Hypothesis 15. *Migration costs are defined by an application manager.*

The notion of migration costs may vary depending on application objectives or customer constraints. For example, transferring data from a Cloud provider to another one may be charged or not, depending on customer's contract with the Cloud provider. As another example, the consequences of application unavailability during a migration may have different financial impacts on different customers. Thus, as those costs are specific to applications or customers, in Chapter 6, we model migration costs as functions defined by an application manager which are called inside our heuristics.

Finally, the following hypotheses are used to simplify the placement problem. Those issues are very interesting research directions to follow from this thesis.

Hypothesis 16. *We do not take into consideration any interference effects that may be consequence of multiple components sharing the same VM.*

Hypothesis 17. *We consider that latency capacities are not affected by multiple components sharing the same connection. This means that latency will be the same no matter how many application components are sharing a connection.*

3.2 Approach

The problem of calculating the placement of a distributed application on a Cloud-based infrastructure, as stated in Section 3.1, is NP-Complete because it can be seen as a generalization of the graph homomorphism problem [54]. To avoid impractical execution times, in this thesis, we propose heuristics able to calculate solutions to that problem.

In summary, we used an *incremental modeling* approach to design and develop those heuristics. While aiming at the placement problem and objective defined in Section 3.1,

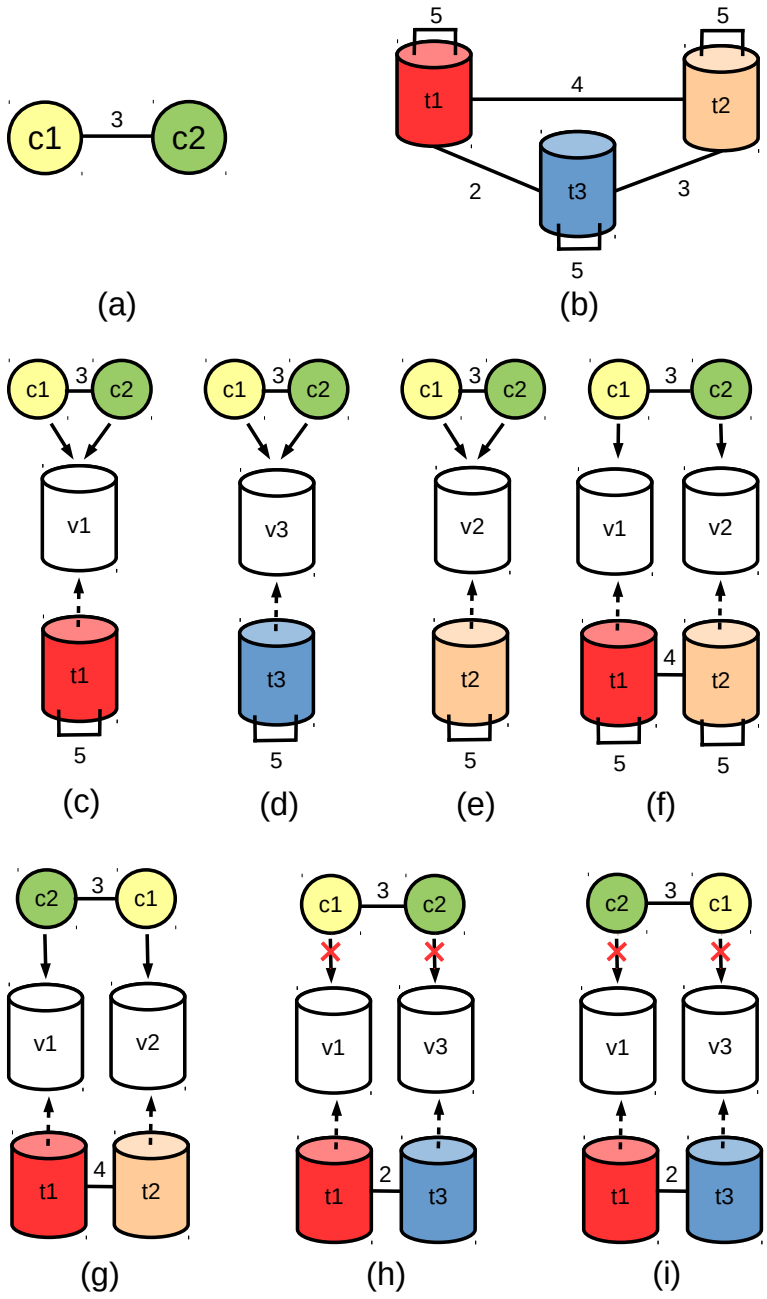


Figure 3.4: Possibilities of placement of a distributed application, described in (a), on a Cloud-based infrastructure described in (b). Circles are application components and edges connecting those circles are communication requirements/capacities. Colored cylinders represent VM types, colorless cylinders, VM instances, and dashed arrows indicate VM types from which VMs were instanced from. Solid arrows indicate the VM to which an application component is assigned. Any of the available VM types could host both c_1 and c_2, thus we omit resource requirements/capacities and concentrate on communication constraints. All placements illustrated in situations (c) to (g) are valid placements. However, situations (h) and (i) are invalid because they do not satisfy communication constraints. Observe that in both cases the communication capacity is smaller than the communication requirement ($2 < 3$).

we started with a simplified model of the placement problem and incremented it as heuristics were being developed.

First we proposed four heuristics capable of computing *initial communication-oblivious* placements of component-based application on Cloud-based infrastructures with the objective of minimizing renting costs and satisfying resource constraints. These heuristics are presented in Chapter 4.

In a second moment, we developed a heuristic capable of computing *initial communication-aware* placements of component-based application on Cloud-based infrastructures with the objective of minimizing *renting costs* and satisfying *resource and communication constraints*. The proposed heuristic makes use of the communication oblivious heuristics and is presented in Chapter 5.

Finally, we proposed a heuristic capable of computing *communication* and *reconfiguration-aware* placements of component-based application on Cloud-based infrastructures with the objective of minimizing renting costs while satisfying resource *and communication constraints*. This heuristic, which is an extended version of the previous communication-aware heuristic, manages to take into consideration previously deployed components and to use *component migration* techniques to reconfigure applications. This heuristic is presented in Chapter 6.

3.3 Evaluation Methodology

In this section we briefly introduce the methods employed to evaluate the performance and accuracy of our contributions.

3.3.1 Strategy

The evaluation of the heuristics proposed in this thesis is done by comparing them to baseline algorithms. The basic idea is to generate a representative set of problems, solve them by the proposed heuristics and baseline algorithms, and then compare their solutions. A *solution* is the composite of a valid mapping between application components and VMs, and the time taken to compute that solution. From a placement solution it is possible to derive its cost, which comprises renting costs and potential migration costs.

In the next sub-sections we discuss our approach to generate the input problems, baseline algorithms, and metrics of comparison used for evaluation.

3.3.2 Experiment

An experiment is the resolution of a set of placement *problem instances* by a *set of algorithms* within a given timeout. A problem instance is a placement problem characterized by *parameters*, application *components*, *VM types*, *Cloud provider sites*, etc., which vary depending on the modeling being used. As discussed in Section 3.2, in this thesis, the placement problem model was designed incrementally.

We organize problem instances in *experiment classes*. Experiment classes are sets of *representative* placement problem instances used during the *evaluation process* as input for proposed heuristics and baseline algorithms. Representative in this case, means that

	A	B	C
# dimensions	4	5	6
# components	3,5,7,10	10,20,30,40,50	60,80,100,120,140
# vm types	100,250,500,700	500,1000,1500,2000	2500,5000,7500,10000
# sites	25,50,100	100,300,500	500,750,1000
# tree height	3,5	5	7
application topology	l,s,f,r	l,s,f,r	l,s,f,r
connection schema	u	d,a,u	d,a,u
# problem instances	384	720	720

Table 3.1: Example of experiment class parameters. This table is identical to Table 5.2.

Dimension	**Requirements**	**Capacities**
(i)	800 to 3000	1000 to 3500
(ii)	1 to 16	2 to 32
(iii)	1 to 32	2 to 40
(iv)	50 to 3500	150 to 4000
(v)	5 to 30	10 to 80
(vi)	1 to 8	1 to 16

Table 3.2: Example of intervals of data generation for dimensions. This table is identical to Table 5.6.

different types and sizes of problems are generated resulting in a better observation of the performance and general behavior of evaluated algorithms.

Problem instances are generated following values previously defined by the experiment classes to which they belong. Application component and VM type dimensions are randomly generated following previously defined intervals of data generation for dimensions. Communication requirements and capacities are generated similarly.

Tables 3.1 and 3.2 illustrate examples of experiment class parameters and intervals of dimension's values generation, respectively. Both tables are used in the evaluation of communication-aware heuristics presented in Chapter 5.

Each line of Table 3.1 represents an experiment parameter and each column, the value of a parameter for a given experiment class, which may be A, B or C, in this example. The last line indicates the number of problem instances in each class. In Table 3.2, dimension values for component requirements are picked uniformly from the range defined in the second column (Requirements) and values for VM type capacities are picked uniformly from the range defined in the third column (Capacities).

Having different classes helps us organize and group problem instances with similar characteristics. In this thesis, experiment classes are specially used to classify problem instances by *size*. The size of a problem instance relates to the magnitude of its experiment class's problem parameters.

The heuristics proposed in this thesis are designed for large-scale scenarios. However running them also on smaller problems allows us to understand their general behavior in different situations and to examine them with respect to less scalable baseline algorithms.

For example, in Chapter 5, we used three different experiment classes A, B, and C (cf. Table 3.1). Small problems from Class A were used to compare the proposed heuristics to exact baseline algorithms. Since this type of exact algorithm does not scale well, it would be impractical to do that analysis using larger problems. Class B problems were used to compare the proposed heuristics to meta-heuristic based algorithms, which scale way better than exact algorithms but still would take too much time to calculate solutions for very large problems. This type of problem is present in Class C and was used to compare the proposed heuristics to scalable baseline heuristics.

3.3.3 Baseline Algorithms

As we discussed throughout this chapter, to evaluate the heuristics proposed in this thesis, we compare their solutions to baseline algorithms. In summary, in this thesis we use two different categories of baseline algorithms: a MIP solver, capable of computing optimal solutions, but not scalable, and a simulated annealing meta-heuristic, a more scalable algorithm which manages to calculate good quality solutions if it is given enough time.

Depending on the modeling of the placement problem (cf. Section 3.2), baseline algorithms parameters may vary (e.g. solver timeout or strategy of computing an initial solution for simulated annealing).

3.3.4 Evaluation Metrics

A solution calculated by a proposed heuristic or baseline algorithm is composed of three elements: the mapping between components and VMs, the cost of this mapping, and the execution time taken to compute the solution. In summary, the evaluation objective is to indicate that the proposed heuristics manage to compute good quality solutions very quickly. To do so, we observe two metrics: cost distances and cumulated execution time.

Cost distances are the difference between the cost of the solution calculated by a proposed heuristic and that calculated by a baseline algorithm. For example, if the heuristic solution has cost h and the baseline algorithm's solution has cost b, the cost distance will be $h - b$. We prefer to format this value as a percentage of baseline solution, thus $\frac{h-b}{b}$. This indicates that the heuristic solution is $100.\frac{h-b}{b}\%$ worst than the baseline. For example, if $h = 10$ and $b = 5$, thus h is $100.\frac{10-5}{5} = 100\%$ worse than b.

Cumulated execution times describe the time taken by a given algorithm to solve a set of problem instances.

3.4 Conclusion

In this chapter we presented the objective of this thesis and the hypothesis used during the design, development, and evaluation of the proposed heuristics. We also have briefly discussed our evaluation methodology, by delineating the process of evaluation, and have presented how experiments are organized, the employed baseline algorithms, the main metrics and the vocabulary used in the next chapters.

Chapter 4

Initial Cost-Aware Placement

In this chapter, we describe a first approach for the problem of placing a distributed application over multiple Clouds. As we discussed in Section 3, at this moment we concentrate on issues associated to mapping application components to virtual machines without considering communication or reconfiguration constraints. The resulting heuristics are used as building blocks for more sophisticated approaches able to take into consideration communication or reconfiguration constraints. Those approaches are described in Chapters 5 and 6.

4.1 Introduction

This chapter investigates efficient algorithms to compute an initial placement for distributed applications on multiple clouds. In this context, automating the application placement is crucial and has been vastly explored in the literature [61], specially in previous works about cloud brokering [50]. We consider that we are dealing with thousands of VM types from tens or hundreds of different cloud providers and tens or hundreds of application components. We consider that components do not communicate with each other. Due to the potential size of placement scenarios and to the NP-hardness [54, 90] of the problem, scalability is a central concern.

In this chapter we present heuristics capable of calculating good quality placements of distributed applications on multiple clouds. Those heuristics, based on *First Fit decreasing* greedy heuristic, manage to calculate placements equivalent to state of the art solutions using considerable less time.

The characterization of the placement problem is presented in Section 4.2 and an extensive related work on the domain is discussed in Section 4.3. The proposed heuristics are detailed in Section 4.4 and their evaluation in Section 4.5.

4.2 Problem Statement

An instance of the distributed application placement on multiple clouds comprises a set of application components, or just *components*, for short, that must be placed on *virtual machines* (VM) rented from possibly multiple *cloud providers* which offer a set of *VM types*. There is a *cost* associated to renting a VM.

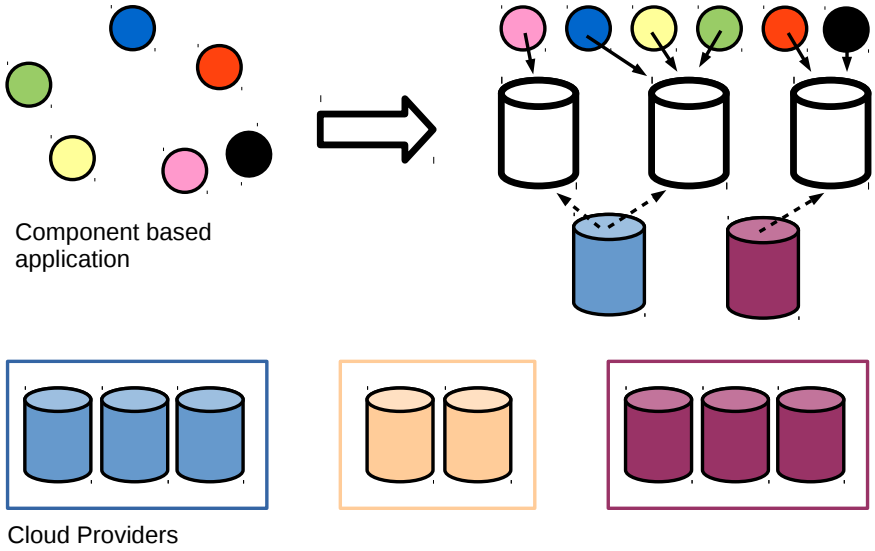


Figure 4.1: Initial Placement of Distributed Applications on Multiple Clouds Problem: mapping components from a distributed component based application to virtual machines instanced from virtual machine types originating from possibly different cloud providers. The objective is to minimize renting costs and to satisfy application's performance constraints.

Each component has *requirements* that must be satisfied by the *capacities* of a rented virtual machine on which it will be placed. To satisfy a placement constraint, a capacity must be larger than or equal to a requirement. Examples of requirements or capacities are: 100 MB of RAM, 10 GB of disk storage, 200 flops of processing, etc. Requirements and capacities are also called *dimensions* of a component or VM, respectively.

We consider that there is no limit on the number of VM instances for any VM type. No component can be assigned to more than one VM, but each VM may hold various components. The objective is to assign all components to VMs so that requirements of each component are met, the capacities of each VM are respected, and renting costs are minimized. This is illustrated in Figure 4.1.

In this chapter, we do not take into consideration *network/communication* constraints neither *a priori* information concerning expected workload, dynamic actors that would allow online modifications of the placement, and renting times.

Throughout this chapter we refer to the described problem as the *Initial Placement of Distributed Applications on Multiple Clouds Problem* or *IPDAMP*.

4.2.1 Optimization Problem Formulation

In this section we formalize the IPDAMP using an optimization problem formulation. The result is described in Equation 4.1 and will be used as input to a MILP solver in

Section 5.7.

Consider $\mathcal{I}$ a set of components, $\mathcal{T}$ a set of virtual machine types and D dimensions of interest. Let $r_{i,d}$ be the value of requirement (or dimension) $d \leqslant D$ of component $i \in \mathcal{I}$, $c_{t,d}$ the value of capacity (or dimension) $d \leqslant D$ of VM type $t \in \mathcal{T}$ and p_t the price of renting an instance of t per unit of time.

Let $v_{k,t}$ be the k-th rented VM instanced from type t. Notice that $1 \leqslant k \leqslant |\mathcal{I}|$. If we consider that only VMs of type t are rented, then at most $|\mathcal{I}|$ of them will be needed. This is the case where there is only one component per VM. Hence, the set $\mathcal{V} = \{v_{k,t} \mid 1 \leqslant k \leqslant |\mathcal{I}|, t \in \mathcal{T}\}$ containing all VMs from type t that could be rented has size $|\mathcal{V}| = |\mathcal{I}| \times |\mathcal{T}|$.

Let $v \in \mathcal{V}$ and $c_{v,d}$ be the capacity of dimension $d \leqslant D$ of rented VM v, i.e., $c_{v,d} = c_{t,d}$ and p_v is the price paid for renting VM v, i.e., $p_v = p_t$.

Let $m_{i,v} = 1$ if a component i is assigned to a rented VM v, and 0 on the contrary. Let $a_v = 1$ if v were assigned to at least one component and 0 on the contrary.

We formalize this placement problem using an optimization problem formulation in Equation 4.1. It will be used as input for a MIP (Mixed Integer Programming) solver in the evaluation section (cf. Section 4.5) for calculating optimal solutions. A summary of variables defined in this section and used in Equation 4.1 is available in Table 4.1.

$$
\begin{aligned}
\text{Minimize } & \sum_{v \in \mathcal{V}} p_v . a_v \\
\text{s.t.} & \\
& \sum_{v \in \mathcal{V}} m_{i,v} = 1 && \forall i \in \mathcal{I} && (i) \\
& \sum_{i \in \mathcal{I}} m_{i,v} . r_{i,d} \leqslant c_{v,d} && \forall v \in \mathcal{V} && (ii) \\
& && 1 \leqslant d \leqslant D \\
& a_v = \begin{cases} 1 & \text{if } \sum_{i \in \mathcal{I}} m_{i,v} > 0 \\ 0 & \text{otherwise} \end{cases} && \forall v \in \mathcal{V} && (iii) \\
& m_{i,v} \in \{0,1\} \\
& a_v \in \{0,1\}
\end{aligned}
\tag{4.1}
$$

Constraint (i) guarantees that each component is assigned to at most one VM, (ii) ensures that no instantiated VM has more components than it can host, (iii) guarantees that $a_v = 1$ when there is at least one component assigned to v.

4.3 Related Work

In this section, the IPDAMP (cf. Section 4.2) is described as a generalization of the classic multi-dimensional bin packing problem (also known as vector packing problem). This way a much wider range of related work, beyond Cloud related bibliography, is reached allowing for a richer state of the art study.

$\mathcal{I}$	Set of components.
$\mathcal{T}$	Set of VM types.
$\mathcal{V}$	Set containing all potential VMs.
D	Number of resources/dimensions.
$v_{k,t}$	k-th rented VM instanced from type $t \in \mathcal{T}$
p_t	Price of VM type $t \in \mathcal{T}$.
p_v	Price of VM $v \in \mathcal{V}$.
a_v	$a_v = 1$ if $v \in \mathcal{V}$ is being used, 0 otherwise.
$m_{i,v}$	$m_{i,v} = 1$ if component $i \in \mathcal{I}$ is assigned to VM $v \in \mathcal{V}$, 0 otherwise.
$r_{i,d}$	Requirement of component $i \in \mathcal{I}$ on dimension d.
$c_{v,d}$	Capacity of VM $v \in \mathcal{V}$ on dimension d.

Table 4.1: Summary of variables used in Equation 4.1.

4.3.1 The Multi Dimensional Bin Packing Problem

The IPDAMP is an instance of the *cost-aware multi-dimensional bin packing problem with heterogeneous bins*, which is a generalization of traditional *multi-dimensional bin packing problem* (MDBPP).

In the classic MDBPP, given a set of n-dimensional *items* and a set of identical n-dimensional *bins*, it is necessary to assign all items to bins using the least number of bins possible. In its cost-aware version, each bin has an *opening price*, and the objective becomes spending the least possible.

The mapping between the cost-aware MDBPP with heterogeneous bins and the IPDAMP is direct. Items are components, bins are VM types, item dimensions are component requirements and bin dimensions are VM capacities. The opening price of a bin is the price of renting a VM.

To the best of our knowledge, no work discusses the IPDAMP as presented in Section 4.2, but since bin packing and more specifically the MDBPP and their applications have been vastly explored, there is interesting related work that can be used as starting point to design a solution to our problem. We divided the related work into three groups based on their solution strategies: exact algorithms, meta-heuristics and greedy heuristics.

4.3.2 Strategies Based on Exact Algorithms

Strategies based on exact algorithms (cf. Chapter 2) are able to calculate optimal placement solutions. However, as the MDBPP is NP-Hard [90], impractical execution times become an issue to be taken into account.

In [52], a solver which uses column generation and branch and bound algorithms to solve multiple type two dimensional bin packing problems is presented. [98] uses a mixed integer programming (MIP) solver to calculate application placements on VMs, VM resource allocation and consolidation, meeting *SLA* constraints. In the latter, the number of dimensions is raised to four (CPU, memory, I/O, and bandwidth) in comparison to the former, however only experiences with at most 20 VM types are performed during the evaluation. On the same subject, authors of [108] utilize a MIP solver to the problem

of VM consolidation aiming at satisfying application *SLAs* and limiting the number of VM migrations. Also, they allow for a large number of dimensions, approximating their problematic to ours. In [118], a MIP solver is used on a control theory based approach to dynamically calculate the resource allocation for adaptive applications.

Despite optimal solutions, all approaches described in this section suffer from scalability issues. Depending on the size of the problem, the execution time from an exact algorithm can easily be in the scale of days, as discussed in Section 4.5.2. Also, except for [98], the cited work is not cost-aware, i.e. none of the solutions considers a price associated to opening a bin. We address this limitation in our approach and use a MIP solver to generate optimal solutions to evaluate the proposed heuristics.

4.3.3 Strategies Based on Meta-Heuristics

A common approach to address bin packing, and consequently placement related problems, is the usage of meta-heuristic strategies (cf. Chapter 2), including genetic algorithms, simulated annealing, particle swarm optimization, ant colony optimization, etc.

The work in [52], [23], and [72] discuss genetic and simulated annealing based heuristics. They describe their placement problems as linear programming problems and use their objective functions as fitness or energy functions and their constraints as selection criterion or cooling strategy.

In [46], an approach to do the placement of workflow tasks on the Cloud using a genetic algorithm is presented. However, in spite of considering the problem of data locality, it models only two resources and it is implicit that workflow tasks and virtual machine types must be homogeneous. On the same subject, but addressing the task and virtual machine homogeneity issues, authors in [96] propose a particle swarm optimization based strategy. However, the very high computation complexity of the algorithm is not adequate to our objectives. The same issue characterizes [35], which uses an ant colony optimization approach to calculate workload placement on the cloud.

In [36] another ant colony based approach for VM consolidation with the objective of minimizing resource wastage or energy consumption is presented and compared to other state of the art algorithms. The time consuming of the proposed algorithm is relatively small, however at the price of using homogeneous physical machines (the VM hosts).

Meta-heuristic based strategies have their solution quality constrained to the available execution time, meaning that, for large problems, the necessary time to calculate a near optimal or good quality solution may be impractical. Also, this type of algorithm usually heavily depends on several application specific tuning parameters to work well. We address large problem instances that must be solved in feasible time, consequently, this solution may not be adequate. Furthermore, despite meta-heuristics being possibly orders of magnitude slower (seconds versus hours) than greedy heuristics, often the quality of solutions does not follow this proportion, as discussed in [52] and [35].

4.3.4 Strategies Based on Greedy Heuristics

The usage of greedy heuristics and more specifically First Fit decreasing based approaches are known to be good options to calculate good solutions to the bin packing problem in feasible time [90, 116].

An implementation of First Fit decreasing is illustrated in Algorithm 2. First, items are sorted in decreasing order of size (cf. Line 2) and bins are shuffled (cf. Line 3). Then, items are assigned to the *first* bin they fit into (cf. Lines 4 to 11). In this manner, it tries to assign largest items first. Function "assign"(cf. Line 7) is described in Algorithm 3. Notice that it tries to assign items to open bins before opening new ones.

Algorithm 2 First Fit Decreasing

Input: $items, bins$
Output: $open_bins$
1: $open_bins \leftarrow [\ ]$
2: **sort**($items$, $decreasing$)
3: **shuffle**($bins$)
4: **for** i **in** $items$ **do**
5: | **for** b **in** $[open_bins, bins]$ **do**
6: | | **if** fits(i, b) **then**
7: | | | **assign**$(i, b, open_bins)$
8: | | | **break**
9: | | **end if**
10: | **end for**
11: **end for**
12: **return** $open_bins$

Algorithm 3 Function assign called inside Algorithm 2

Input: $item, bin, open_bins$
1: **if** bin is open **then**
2: | **add**$(item, bin)$
3: **else**
4: | $b \leftarrow$ **open**(bin)
5: | **add**$(item, b)$
6: | **open**$_bins \leftarrow open_bins + b$
7: **end if**

First Fit has a approximation ratio of $\frac{11}{9}$ [116] for one-dimensional bin packing problems. However, this result does not apply in the multi-dimensional case. When the number of dimensions is greater than two, it is known that there is no asymptotic polynomial time approximation scheme for the problem, unless $P = NP$ [13]. Other work [62] shows that the best possible approximation for this problem would be a $(1 + \ln D + \epsilon)$-approximation for $D \geqslant 2$, where D is the number of dimensions, and any $\epsilon > 0$. Finally, [114] does an analysis on execution times and concludes that an algorithm with time complexity $O(n \log n)$ cannot do better than a D-approximation. In spite of that, there are many approaches which does not have formal guarantees but perform well in practice. We concentrate our state of the art analysis on this type of work.

In the next sections, we present approaches from the state of the art that propose greedy heuristics able to deal with MDBPP with heterogeneous bins.

Measuring Multi-Dimensional Items

First Fit decreasing heuristics depend on sorting one-dimensional items. When dealing with a multi-dimensional problem, sorting those elements is not straightforward. One approach to tackle this problem, presented in [90], describes different procedures for *measuring* or giving a score to multi-dimensional items. In summary, the authors proposed functions that receive a multi-dimensional element as input and returns a scalar. This type of function would later be named *measure* in [38].

We illustrate this concept with very simple measure functions. Let $i_1, i_2 \in \mathcal{I}$ be two items having dimensions $d_1 = (3, 5, 2)$ and $d_2 = (4, 3, 3)$. The function "size" of those elements could be simply the sum of their dimensions: $size(i_1) = 3 + 5 + 2 = 10$ and $size(i_2) = 4 + 3 + 3 = 10$. It could also be the euclidean norm of their dimensions, thus $size(i_1) = \sqrt{3^2 + 5^2 + 2^2} = \sqrt{38}$ and $size(i_2) = \sqrt{4^2 + 3^2 + 3^2} = \sqrt{34}$, for example.

[38], [39], and [90] present *Weighted Sum*, *Average Sum*, *Exponential Sum*, and *Priority* measures. They are described in detail bellow.

Let i_d and b_d be the values of dimension $1 \leqslant d \leqslant D$ of item $i \in \mathcal{I}$ and bin $b \in \mathcal{B}$ respectively. We deliberately use the same nomenclature of sets used in Section 4.2 because, at the end of the day, components can be described as items, VM types as bins, and VMs as open bins.

- **Measure Average Sum** [90]: This measure uses the average sum of dimension values.

$$\mathcal{M}_{as}(i) = \sum_{d=1}^{\mathcal{D}} (\frac{1}{|\mathcal{I}|} . \sum^{j \in \mathcal{I}} j_d) . i_d \tag{4.2}$$

- **Measure Exponential Sum** [90]: This measure uses the exponential of average sum.

$$\mathcal{M}_{es}(i) = \exp(\epsilon . \mathcal{M}_{as}(i)) \tag{4.3}$$

  For some constant ϵ.

- **Measure Weighted Sum** [38, 90]: This measure uses the weighted sum of dimension values.

$$\mathcal{M}_{\text{ws}}(i) = \sum_{d=1}^{D} \alpha_d . i_d,\ 1 \leqslant d \leqslant D \tag{4.4}$$

  α_d is a scaling vector that can assume the following values: 1, $\frac{1}{C_d}$, $\frac{1}{R_d}$, and $\frac{R_d}{C_d}$ where $C_d = \sum_{b \in \mathcal{B}} b_d$ and $R_d = \sum_{i \in \mathcal{I}} i_d$.

- **Measure Priority** [38]: This measure uses the maximal normalized value of dimensions.

$$\mathcal{M}_{\text{p}}(i) = max(\frac{i_d}{\sum_{b \in \mathcal{B}} b_d}),\ 1 \leqslant d \leqslant D \tag{4.5}$$

The heuristics visited in this section describe very interesting ways of solving multi-dimensional bin packing problems. Furthermore, evaluation of the discussed *measures* available in the related work indicate that they are an effective way to deal with multi-dimensionality and heterogeneity. However, before applying those heuristics to IPDAMP it is necessary to address *bin opening costs* issues.

Heuristics that do not use measures

Dot Product [90, 38] is a greedy heuristic which assigns items to bins that maximize a dot product between an item and all bins, as described in Equation 4.6

$$DP(i) = \sum_{d=1}^{D} i_d.b_d, \forall b \in \mathcal{B},\ 1 \leqslant d \leqslant D \tag{4.6}$$

The Dot Product heuristic tackles multi-dimensionality and heterogeneity issues, and has promising results, however, it is not cost-aware, preventing its usage on IPDAMP.

A First Fit based algorithm called *First Fit Ordered Deviation* is discussed in [52]. Its strategy manages to efficiently solve the MDBPP, however it is limited to two-dimensional bin packing problems and only homogeneous bins are considered.

First Fit Windowed Multi-Capacity [66] is a First Fit based algorithm which aims at balancing residue capacities of open bins. As it has an important role in this chapter, we explain it in more detail. This heuristic assigns items to bins with the objective of balancing the usage of dimensions through a rank matching mechanism. It considers that *bins are homogeneous* and item requirements are described in terms of percentage of bin's capacities. For example, dimensions 1, 2, and 3 of a three-dimensional item i could be described as $i_1 = 0.67b_1$, $i_2 = 0.42b_2$, and $i_3 = 0.8b_3$, indicating that the values of item's dimensions are 67%, 42% and 80% of corresponding dimensions of bin b. For each item, a *rank*, based on the magnitude of each dimension, is constructed. The rank of item i would be [2,1,3], meaning that the largest dimension is in the third position. Open bin ranks are also constructed as items are assigned to them. The objective, then, is to find open bins that match item's ranks. In the *Choose Pack* [66] variation of this heuristic, in which we are interested, matching a bin rank means that, given a window of size $w \leqslant D$, *the smallest dimension of a bin must be one of the w largest dimensions of an item.* If no open bin can hold the item, then, it is assigned to a new bin. Table 4.2 illustrate an example of the ranking system.

item / bins	[1,2,3]	[1,3,2]	[2,1,3]	[2,3,1]	[3,1,2]	[3,2,1]
[2,1,3]	match	match	–	match	–	match

Table 4.2: Rank matching example of First Fit Windowed Multi-Capacity (Choose Pack) for $w = 2$. Item ranking is illustrated in the left side and bin rankings in the upper part. As the two largest dimensions of the item are in positions one and three, the heuristic will match every bin having a "1" in position one or three.

Preliminary results from First Fit Windowed Multi-Capacity are promising [66], however, before applying it to the IPDAMP, it would be necessary to address its heterogeneity issues and cost obliviousness.

4.3.5 Discussion

The discussed literature has shown that the cost-aware MDBPP with heterogeneous bins and its applications have important open challenges. We discussed an extensive bibliography about the MDBPP, a subproblem of IPDAMP, and despite the many contributions

from related work, we identified a range of issues that limit the usage out of the box of the proposed algorithms, namely, the *cost-obliviousness* of all discussed work and *homogeneity of bins* of some heuristics.

4.4 Improved Greedy Heuristics

In Section 4.3.4, we discussed a set of greedy heuristics created to solve the MDBPP. Due to their limitations, namely *cost-obliviousness* and *homogeneity of bins*, however, it is not possible to use those heuristics directly for solving the IPDAMP.

Our strategy for this issue is, instead of developing new algorithms from scratch, to adapt existing MDBPP heuristics to the IPDAMP. In the next sections we describe the heuristics chosen for adaptation and the implemented modifications. In Section 4.4.1 we list the algorithms chosen for being improved to support IPDAMP placement scenarios, in Section 4.4.2 we explain the adaptations applied to the heuristics to add cost-awareness to them, and in Section 4.4.3, we present the necessary modifications to add bin heterogeneity to the problem. Finally, in Section 4.4.4, we present the resulting heuristics and discuss their algorithms.

4.4.1 Choice Of Greedy Heuristics to be Adapted

The algorithms chosen for adaptation with their respective measures (if applicable), are *First Fit Decreasing Priority* [38] (FFDP), *First Fit Windowed Multi-Capacity* [90] (FFWMC), and *Dot Product* [90] (DP). The measure *Weighted Sum* [90, 38] (WS) will be employed inside FFD-WMC. The criteria for implementation was reproducibility of the algorithm, promising results concerning execution time and solution quality in the source article, and consistent performance in our preliminary tests (the latter were based on the methodology described in Section 4.5.1).

4.4.2 Adding Cost-Awareness

In the traditional MDBPP, the objective is to minimize the number of bins used to allocate items. Hence, the cost of a solution is associated to the number of open bins. However, in the cost-aware MDBPP, there is a price for opening bins and they may vary.

To adapt First Fit based greedy heuristics to the IPDAMP, one option is to change the way bins are sorted. In classic First Fit decreasing (cf. Algorithm 2), the list of bins is usually shuffled and items are assigned to the first bin they fit into. In a cost-aware environment with the objective of minimizing bin opening costs, one approach would be to sort the bin list by decreasing *profitability*. In this way, the algorithm would try to assign items to the *most profitable* bins first.

One way to quantify profitability is to use the ratio $\frac{capacity}{price}$. For example, in the cost-aware MDBPP, a solution with ten \$1 bins of size $s > 0$ is better than a solution with one \$20 bin of size $10s$. Both solutions successfully assign all items to bins, but the former costs \$10 and the latter \$20. The ratio $\frac{capacity}{price}$ of the first bin ($\frac{s}{1}$) is larger than that of the second one ($\frac{10s}{20}$).

Another important point to minimize costs is to open bins the least often possible, hence, using the capacities of already open bins is imperative. To do so, before looking for new bins, we verify if items can be assigned to any of already open bins.

For the heuristics using measures, namely FFDP, DP, and WS, first, it is necessary to adapt their functions so they can be used to measure bins, too. For that, it is enough to replace i_d by b_d, the capacity of bin $b \in \mathcal{B}$ over dimension $1 \leqslant d \leqslant D$. To include the profitability ratio, we multiply the metric by $\frac{1}{p_b}$ where p_b is the cost of opening bin b. This process is described bellow for Weighted Sum and Priority measures.

- Weighted Sum

$$\mathcal{M}_{ws}^{\text{bin}}(t) = \frac{1}{p_b} \sum_{d=1}^{D} \alpha_d . b_d \tag{4.7}$$

  Coefficient $\alpha_d = \frac{R_d}{C_d} = \frac{\sum_{i \in \mathcal{I}} i_d}{\sum_{b \in \mathcal{B}} b_d}$ is a scaling vector [39].

- Priority

$$\mathcal{M}_{\text{p}}^{\text{bin}}(t) = max \left(\frac{1}{p_b} . \frac{b_d}{\sum_{b \in \mathcal{B}} b_d} \right) \tag{4.8}$$

For the Dot Product (DP) heuristic, it is also necessary to consider bin prices when calculating the dot product. As described in Equation 4.9, we introduce the profitability ratio by multiplying each dot product by $\frac{1}{p_t}$. In this way, dot products of cheaper bins will be larger than those of more expensive ones. To be able to explore open bins before opening new bins, a dot product between unassigned items and open bins is calculated before opening a bin.

$$DP^{bin}(i) = \frac{1}{p_b} \sum_{d=1}^{D} i_d . b_d, \ \forall b \in \mathcal{B} \tag{4.9}$$

To make First Fit Decreasing Windowed Multi-Capacity (FFD-WMC) cost-aware, bins are sorted using the cost-aware version of measure *Weighted Sum* (Equation 4.7). Hence, more profitable bins would be scanned first. At this point, however, FFD-WMC does not have heterogeneous bins, thus all bins have the same opening price. This issue will be addressed in the next section.

4.4.3 Heterogeneous Bins

Among the heuristics in which we are interested, First Fit Decreasing Windowed Multi-Capacity (FFD-WMC), considers that bins are homogeneous. As discussed in Section 4.3.4, items requirements are described in terms of percentage of bin's capacities and it is essential that item's dimensions share the same base so they can be compared. Our strategy to allow for heterogeneous bins, is to construct a maximal bin, which is composed of the largest dimension capacities from all bins combined and use it as basis of comparison. Thus, ranks dimensions become percentages of this maximal bin dimensions.

For example, consider a scenario with three different bins e, f, g having dimensions $[2, 4, 5]$, $[2, 2, 3]$, and $[6, 3, 4]$, respectively. The reference maximal bin b^{max} would have dimensions [6,4,5] containing the largest dimensions from e, f, g : first dimension from

g, and second and third dimensions from e. In this manner, dimensions $[3, 1, 4]$ of an item i would be described as $[3, 1, 4] = [0.5 \times b_1^{max}, 0.66 \times b_2^{max}, 0.8 \times b_3^{max}]$ and i's rank would be [1,2,3].

4.4.4 The Greedy Group

In Sections 4.4.2 and 4.4.3 we described procedures to add cost awareness and bin heterogeneity to four different heuristics/measures: *First Fit Decreasing Priority* (FFDP), *First Fit Windowed Multi-Capacity* (FFWMC), *Dot Product* (DP), and *Weighted Sum* (WS), which is used by FFWMC.

In this section we present the pseudo-code of the three resulting heuristics FFDP, FFWMC, and DP in Algorithms 4, .5, and 6, respectively.

First Fit Decreasing Priority (FFDP)

Algorithm 4 illustrates the heuristic FFDP. It makes use of function **sort_using_measure** (Lines 2 and 3) which takes as input a set of multi-dimensional items or bins, a measure function, and a sorting parameter. The sorting function uses the measure function to calculate the size of the set of items or bins and them sort it using the sorting parameter (increasing or decreasing order). Notice, that the way items and bins are sorted is the only difference between FFDP and the classical First Fit Decreasing (cf. Section 2).

In Algorithm 4, items are sorted decreasingly using Priority measure $\mathcal{M}_p$ (cf. Equation 4.5) in Line 2. Similarly, in Line 3, bins are sorted decreasingly using the cost-aware Priority measure (cf. Equation 4.8).

Algorithm 4 First Fit Decreasing Priority

Input: $items, bins$
Output: $open_bins$

```
1:  open_bins ← [ ]
2:  sort_using_measure(items, M_p, decreasing)
3:  sort_using_measure(bins, M_p^bin, decreasing)
4:  for item ∈ items do
5:      for bin ∈ [open_bins, bins] do
6:          if fits(item, bin) then
7:              assign(item, bin)
8:              break
9:          end if
10:     end for
11: end for
12: return open_bins
```

Dot Product (DP)

Algorithm 5 illustrates the heuristic DP. In summary, the objective of this heuristic is to assign items to bins or open bins that maximize dot product function DP^{bin}. It uses two

important functions **compute_dot_prods** (cf. Lines 2 and 4) and **get_bin_with_max_dot_prod** (Lines 9 and 5).

Function **compute_dot_prods** receives a set of items, a set of bins, and a "dot product" function as parameters. It outputs a matrix containing the result of the application of the "dot product" function between each item and bin. Function **get_bin_with_max_dot_prod** receives a "dot product" matrix and an item as input, and outputs the bin with the largest "dot product" value which has sufficient space to host the input item.

The algorithm works as follows. First, **compute_dot_prods** is called (cf. Line 2), with all items, all bins and the cost-aware "dot product" function DP^{bin} (described in Equation 4.9) as parameters. The matrix with all results is stored in variable *dot_products*. Then, the heuristic will go through the set of items (between Lines 3 and 12). For each *item*, it will try to assign it to open bins first and, if it does not fit to any of them, to a new bin.

When checking open bins, the heuristic first needs to calculate the "dot product" function DP^{bin} between the item being assigned and all open bins. The results are stored in vector *open_bin_dot_prods* (cf. Line 4). Then, this vector is searched by function **get_bin_with_max_dot_prod** (cf. Line 5) in order to find an open bin that can host the item being assigned and which maximizes the "dot product" function DP^{bin}. If an open bin with these characteristics is found, the item is assigned to it. Otherwise, the heuristic will have to open a new bin.

A new bin that can host the item being assigned and whose "dot product" function DP^{bin} is maximized is the result of a call to function **get_bin_with_max_dot_prod** (cf. Line 9). The item can finally be assigned to that bin (cf. Line 10).

This process is repeated until all items are assigned to bins.

Algorithm 5 Dot Product

Input: *items*, *bins*
Output: *open_bins*

```
1:  open_bins ← [ ]
2:  dot_prods ← compute_dot_prods(items, bins, DP^bin)
3:  for item in items do
4:  |   open_bin_dot_prods ← compute_dot_prods(item, open_bins, DP^bin)
5:  |   open_bin ← get_bin_with_max_dot_prod(open_bin_dot_prods, item)
6:  |   if ∃ open_bin then
7:  |   |   assign(item, open_bin)
8:  |   else
9:  |   |   new_bin ← get_bin_with_max_dot_prod(dot_prods, item)
10: |   |   assign(item, new_bin)
11: |   end if
12: end for
13: return open_bins
```

First Fit Windowed Multi-Capacity (FFDWMC)

Algorithm 6 illustrates an implementation of the FFDWMC heuristic. There are four functions which are called inside FFDWMC: **create_max_bin**, **calc_rank**, **sort_using_measure**, and **match**.

Functions **sort_using_measure** and **assign** are explained in Sections 4.4.4 and 4.3.4, respectively.

Algorithm 6 First Fit Windowed Multi-Capacity

Input: $items, bins$
Output: $open_bins$

```
1: open_bins, item_ranks, bin_ranks ← [ ]
2: max_bin ← create_max_bin(bins)
3: for item ∈ items do
4: |   item_ranks[item] ← calc_rank(item, max_bin)
5: end for
6: for bin ∈ bins do
7: |   bin_ranks[bin] ← calc_rank(bin, max_bin)
8: end for
9: sort_using_measure(bins, M_ws^bin, decreasing)
10: for item ∈ items do
11: |   stop ← False
12: |   for open_bin ∈ open_bins do
13: |   |   if fits(item, open_bin) then
14: |   |   |   if match(item_ranks[item], open_bin_ranks[open_bin]) then
15: |   |   |   |   assign(item, open_bin)
16: |   |   |   |   open_bin_ranks[open_bin] ← calc_rank(open_bin, max_bin)
17: |   |   |   |   stop ← True
18: |   |   |   |   break
19: |   |   |   end if
20: |   |   end if
21: |   end for
22: |   if stop then
23: |   |   break
24: |   end if
25: |   for new_bin ∈ bins do
26: |   |   if fits(item, bin) then
27: |   |   |   assign(item, bin)
28: |   |   |   open_bin_ranks[bin] ← calc_rank(bin, max_bin)
29: |   |   |   break
30: |   |   end if
31: |   end for
32: end for
33: return open_bins
```

Function **create_max_bin** takes as input all bins being considered and returns a

maximal bin which will be used to build item and bin rankings, as described in Section 4.4.3.

Function **calc_rank** takes as parameter one item/bin and the maximal bin and returns the ranking of its dimensions in relation to the maximal bin (cf. Section 4.3.4).

Function **match** receives the ranks of items and a set of bins/open bins, as input. It outputs "True" if there is a *match* between item and bins/open bins rankings and "False" otherwise. The procedure behind rank matching is explained in Section 4.3.4.

The algorithm is straightforward. First, in Line 2, a maximal bin is created and used to calculate rankings for each item (cf. Line 4) and bin (cf. Line 7). Then, bins are sorted decreasingly using measure $\mathcal{M}_{ws}^{\text{bin}}$. The core of the heuristic is represented between Lines 10 and 32. In summary, the heuristic will try to first assign items to open bins. The latter are chosen following the dimension ranking matching of each item with each potential open bin. If an open bin matches the ranking of an item, it will be assigned to that bin. Otherwise, that item will be assign to the first bin it fits into as in traditional First Fit Decreasing heuristics (cf. Section 4.3.4).

4.5 Evaluation

This section evaluates the performance of the greedy heuristics described in Section 4.4 which are able to calculate solutions for the IPDAMP. This is done through a comparative analysis between the proposed heuristics to two state of the art solutions, namely, a MIP solver and a simulated annealing meta-heuristic.

Before that, it is important to present how the evaluation experiments were performed and how their input data were generated.

4.5.1 Methodology

In this section we discuss the methodology used to evaluate the proposed greedy heuristics. In Chapter 3 we have discussed in detail the methodology, notation, and metrics used for evaluating our contributions. Hence, in this section, we summarize concepts previously presented and focus on particularities of proposed heuristics' evaluation, such as experiment format, problem classes parameters or test platform characteristics.

As we discussed in Section 3.3, an *experiment* is the resolution of a set of placement *problem instances* by a set of algorithms within a given *timeout*. A problem instance is composed of a group of components and a group of virtual machine types, both describing dimensions requirements and capacities, respectively.

We define two classes of experiments, *A* and *B*, distinguished by problem instances sizes as described on Table 4.3. The small problem instances from Class A are used to evaluate the performance of greedy and meta heuristics against an exact algorithm, as the latter presents scalability issues. Class B is composed of large problem instances which are used to evaluate the greedy heuristics against meta-heuristics.

To construct problem instances it is necessary to generate the values of VM capacities, prices, and component requirements. The procedure we use is the generation of pseudo-random values picked uniformly inside an interval using the method *randint* from Python's module *random*. Table 4.4 presents those intervals in detail.

	Class A	**Class B**
# dimensions	1,2,3,...,8	1,2,3,...,8
# components	1,2,3,...,19	10,20,30,...,100
# vm types	100,200,300,...,1000	1000,2000,3000,...,10000
# problem instances	1520	800

Table 4.3: Experience classes.

Dimension	**Requirements**	**Capacities**
(i)	800 to 3000	1000 to 3500
(ii)	1 to 16	2 to 32
(iii)	1 to 32	2 to 40
(iv)	50 to 3500	150 to 4000
(v)	5 to 30	10 to 80
(vi)	1 to 8	1 to 16
(vii)	1 to 10	5 to 40
(viii)	10 to 80	10 to 80

Table 4.4: Intervals of data generation.

To generate the VM renting prices, we use the capacities from the first 4 VM type dimensions, in a way that the larger they are, the more expensive is the renting price. We simulate different prices from different cloud providers through the generation of pseudo-random coefficients, as before, using the method *randint*, from predefined intervals. The price of a VM type $p_t = \alpha+\beta+\gamma+\delta$ where $\alpha = c_{i,1} \times randint(1,3)$; $\beta = c_{i,2} \times randint(8,20)$; $\gamma = c_{i,3} \times randint(5,8)$; $\delta = c_{i,4} \times randint(10,15)$, if $c_{i,4} \leqslant 500$, otherwise $\delta = c_{i,4} \times randint(20,25)$.

Our test platform and greedy heuristics were developed in Python and experiments were conducted on Dell PowerEdge R720 (2 CPUs, 6 cores) and AMD Opteron 6164 HE 1.7GHz (2 CPUs, 12 cores) from *Taurus* and *Stremi* clusters from *Grid'5000* [12].

4.5.2 MIP Solver and Simulated Annealing Analysis

We are interested in evaluating the performance of our greedy heuristics with the very large problem instances from Class B (cf. Table 4.3). It would be interesting to compare the solutions from greedy heuristics to the optimal of each problem instance, however, as we are dealing with a NP-Hard problem, this is impractical.

Our strategy, then, is try to calculate the optimal solution of each problem instance from a Class A experiment (cf. Table 4.3) giving the MIP solver 30 hours per instance to do this task. Using these data, we validate the performance of our simulated annealing implementation and use it as a baseline algorithm to analyze the performance of the greedy heuristics on the large Class B problem instances.

MIP Solver

To evaluate the performance of MIP solvers, we integrated to our test platform the *SCIP* solver [2], a framework for constraint integer programming and branch-cut-and-price (the formulation of the placement problem is described in Section 4.2.1).

We conducted one Class A experiment using SCIP with a timeout of 30 hours. The framework managed to calculate the optimal for around 34% of all Class A problem instances. These solved problem instances are mainly characterized by having a small number of application components, virtual machine types, and dimensions. This performance is expected as we are dealing with a NP-Hard problem. Even if we consider that there is room for improvement of the solver performance by optimizing the modeling or by using a faster solver, we would still expect a low rate of solved instances due to the nature of the problem we are dealing with.

Simulated Annealing

As discussed in Section 4.3, using meta-heuristics to solve problems similar to the bin packing problem is a very common approach. Among all algorithms of this type, including genetic algorithms, particle swarm optimization, ant colony optimization and others, we choose the simulated annealing because, in addition to successful experiences in other similar contexts [52], it has less configuration parameters, thus it is easier to apply it to our problem.

We used the *Simanneal* [91] module, which is written in Python and was easily integrated to our test platform. Also, we conducted Class A and Class B experiments using a 10 minutes timeout per problem resolution. We use 10 minutes instead of the 30 hours given to SCIP solver because 10 minutes is more realistic and also because during our tests we noticed that, in most of cases, after this time, the solution improvements became scarcer.

We observed that simulated annealing managed to output solutions for all problem instances in less than ten minutes. Furthermore, it managed to compute optimal solutions for around 97% of the problems where the optimal was known. As SCIP managed to calculate the optimal for around 34% of Class A problems, this totalizes around 33% of all problems.

We illustrate in Figure 4.2 the distances between simulated annealing solutions and optimal solutions. To construct this graph, we grouped all Class A problem instances by number of components and plotted, in form of box plot, the distances (or differences) between solution costs from MIP solver and simulated annealing. Throughout this chapter we group solutions by number of components because this parameter showed to be the one that most interferes on solution cost in a consistent way. We can notice that in spite of the small variation observed as the number of components grows, the distance is always smaller than 3% and the median is always zero, except when the number of components is 17.

Even if it was only possible to compare the solutions from simulate annealing to the optimal in a reduced portion of the problems, we have a promising indication of the capabilities of this meta-heuristic. This justifies the usage of simulated annealing as baseline in our further analysis.

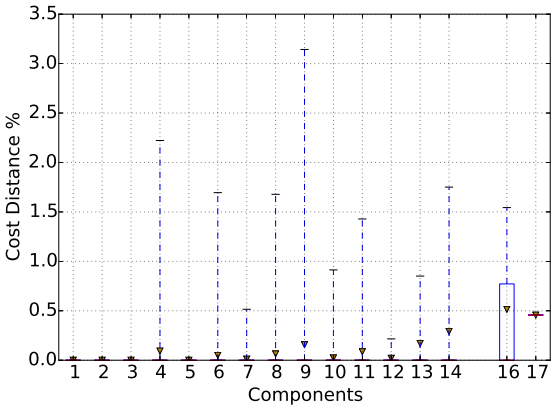


Figure 4.2: Distances between solution costs from MIP solver and simulated annealing aggregated by number of components for Class A experiment.

4.5.3 Greedy Heuristics

In this section, our objective is to evaluate the performance of proposed greedy heuristics. The main goal is evaluating them using the large Class B experiences, however, we also investigate the performance of the greedy heuristics using the 34% of Class A problems whose optimal solution is known.

The evaluation strategy is, at first, analyzing the performance of the greedy heuristics *together*, i.e., comparing them in group to other algorithms and then, in a second moment, evaluating them individually. We consider that the greedy heuristics are executed sequentially, thus, when *group comparing*, the execution time is the sum of the execution times from all involved greedy heuristics. Then, we keep only the best placement – less expensive – among all solutions calculated by greedy heuristics.

We use data gathered from Class A and B experiments with a timeout of 10 minutes to solve each problem instance. Class A and B experiments are used to evaluate greedy heuristics against the MIP solver and simulated annealing respectively.

Greedy Group Analysis – Class A Experiment

At first we compare solutions from the group of heuristics to the available 34% of optimal solutions from Class A problem instances. Figure 4.3 illustrates *cost distances* (cf. 3.3.4) between the group of greedy heuristics solutions and optimal values.

The first thing to notice is that despite giving very few optimal solutions (about 3.4%), the greedy heuristics group managed to output solutions at most 30% more expensive than the optimal with medians varying between 5.52% and 22.25%.

As simulated annealing managed to calculate optimal solutions in 97% of cases, we do not plot the distances between the greedy group and simulated annealing as the results will be essentially the same as illustrated in Figure 4.3. Nevertheless, we plot the distances between the greedy group and simulated annealing considering *100% of Class A problems*. This is illustrated in Figure 4.4. In this case, medians vary between 5.56% and 24.88%, and distances between -8.22% and 45.86%. Notice that this is consistent with medians interval found when comparing the greedy group against optimal solutions

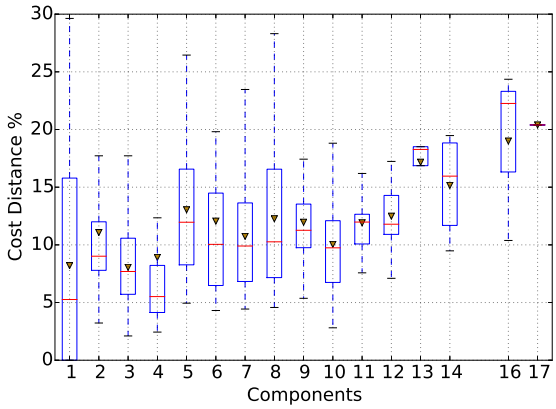


Figure 4.3: Distances between optimal solution costs and greedy heuristic group's aggregated by number of components for Class A experiment.

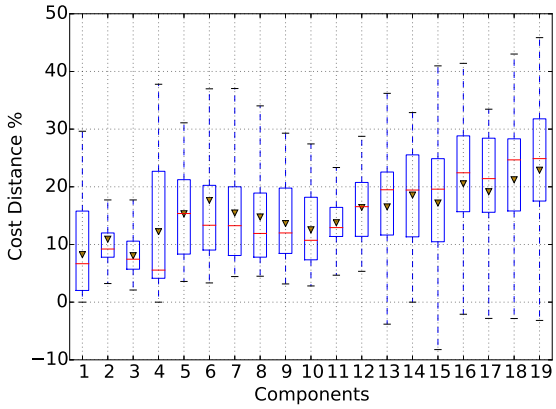


Figure 4.4: Distances between solution costs from simulated annealing and greedy heuristic group aggregated by number of components for Class A experiment.

(medians varying between 5.52% and 22.25%). Negative distances (around 3.18%) refer to situations where greedy heuristics group managed to output a better solution than simulated annealing.

Finally, in both graphs it is possible to identify a degradation of greedy heuristics group solutions as the number of components raises, specially when the number of components is greater than 12.

To complete this first analysis, it is important to analyze all algorithms execution times. The solver was given a timeout of 30 hours per problem instance to solve all Class A problems and was able to solve only about 34% of them. Simulated annealing was given a 10 minutes timeout per problem instance but managed to calculate solutions for all Class A problems in around 2.47 hours (average of about 7 seconds per problem). The greedy block, was given a 10 minutes timeout, however, they took only 23.21 seconds to calculate solutions for all Class A problems (average of 20 milliseconds per problem instance).

There are some preliminary conclusions taken from this first analysis. The quality of solutions is at most 30% worst than the optimal but they are calculated much faster,

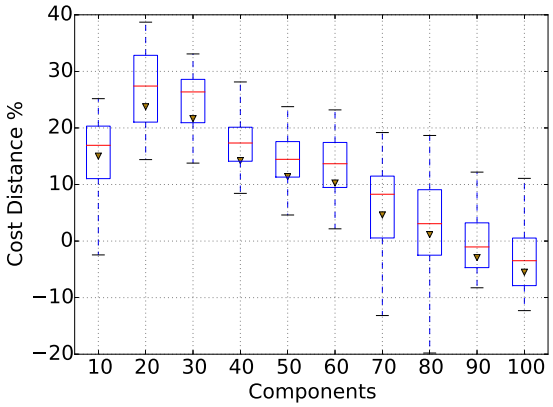


Figure 4.5: Distance between simulated annealing solution costs and greedy heuristic group aggregated by number of components.

indicating a reasonable solution quality. Also, when comparing Figures 4.3 and 4.4 it is possible to identify that cost distance medians follow a similar pattern, which indicates that simulate annealing would be an interesting choice as a baseline algorithm.

Greedy Group Analysis – Class B Experiment

Figure 4.5 illustrates the cost distance between the group of greedy heuristics and simulated annealing. Solutions are aggregated by the number of components from solved problem instances. It is possible to observe that the greedy heuristic group managed to output a better solution than simulated annealing to around 15% of problem instances. One can notice that it happens more frequently when the number of components is bigger than 70. This is observed because the timeout of 10 minutes is not sufficient for simulated annealing to calculate a better solution as the size of problem instances grows.

In the remainder 85% of problem instances, where simulated annealing outputs better solutions, we can also observe that although the distances are always smaller than 40%, the median never exceeds 30%.

Figure 4.6 helps us to have a better understanding on how worse was the solution cost of greedy heuristics on these 85% of problem instances where simulated annealing calculated better solutions. The Y-axis is the percentage of solved problem instances and the X-axis is the cost distance between the greedy heuristic group and simulated annealing. The solid curve is the aggregated percentage of problem instances. We can observe that around 40% of solutions are between 11% and 20% worse than simulated annealing and, most importantly, that around 95% of solutions are at most 30% worse than simulated annealing's ones. This clearly indicates that, depending on the application requirements, the degradation of solution quality may not be very significant, especially when taking into account the difference between execution times from the group of greedy heuristics and the simulated annealing which will be discussed in the following lines.

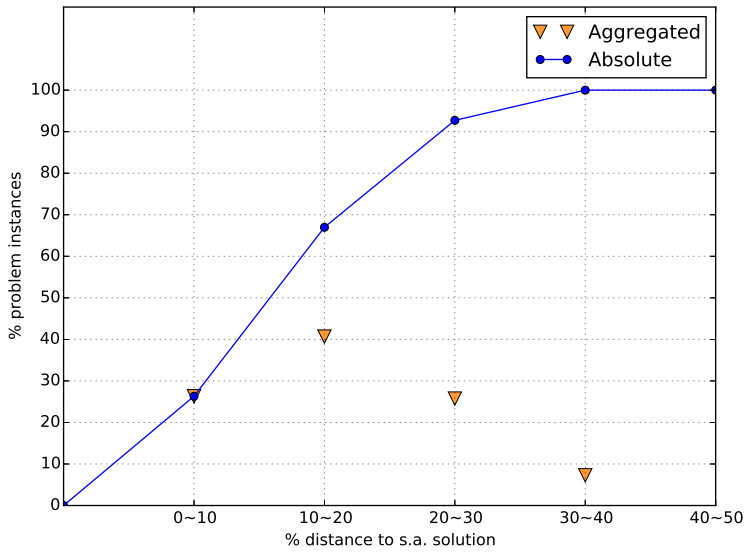


Figure 4.6: Distance between solution costs from simulated annealing solutions and greedy heuristics group on problem instances where simulated annealing solutions were better.

In Figure 4.7 the execution times to solve problems with the same number of components are summed up. While the sum of execution times from greedy heuristics vary from 25 to 210 seconds, simulated annealing's ranges from 26200 to 42185 seconds, i.e. from 7.3 to 11 hours.

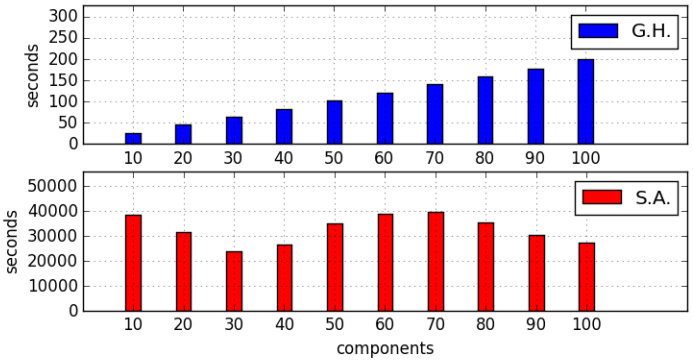


Figure 4.7: Sum of execution times from greedy heuristics group (above) and simulated annealing (bellow) to solve all problem instances. Results are aggregated by number of components.

This can be better seen in Figure 4.8, which aggregates the ratio between greedy heuristics group and simulated annealing execution times by number of components and plot this data as a box plot. We are using ratios instead of distances percentages because of the huge gap between values. The sum of greedy heuristic group execution times is at least 96 times and at most around 4046 times faster than simulated annealing's.

It is well known that the usage of heuristics involves a trade-off between solution

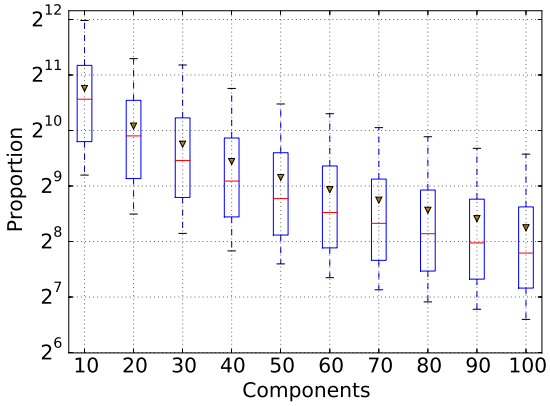


Figure 4.8: Ratio between simulated annealing and greedy heuristics execution times aggregated by number of components.

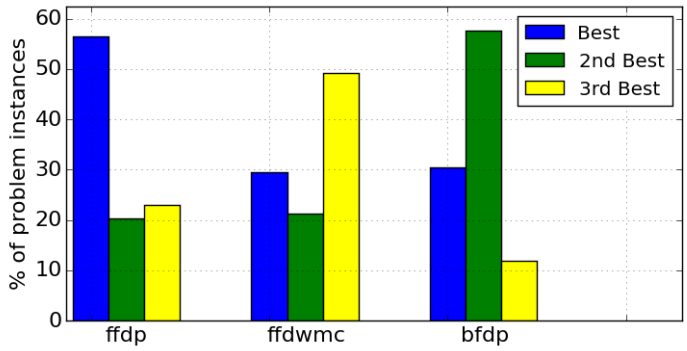


Figure 4.9: Percentage of best, second best and third best solutions by greedy heuristic.

quality and execution time. The analysis of the performance of our greedy heuristics as a block indicated that even with an execution time between 96 to 4046 times smaller, the greedy heuristics managed to output comparable and sometimes better solutions than simulated annealing's for large problems.

Individual Analysis

In this section, we evaluate the greedy heuristics individually using a Class B experiment. Our objective is to understand their behavior and also to investigate how the reduction of the group of greedy heuristics would affect solution quality.

Figure 4.9 illustrates the percentage of best, second best, and third best solutions per greedy heuristic. We notice that First Fit Decreasing Priority (FFD-P), First Fit Decreasing Windowed Multi-Capacity (FFD-WMC), and Dot Product (DP) have the best solutions for around 56%, 29%, and 30% of problem instances respectively. Even if DP has a relatively small number of best solutions, it manages to output second best solutions to almost 57% of problems. Thus, DP gives the best or second best solution to around 74% of all problems. FFD-P and FFD-WMC manage to do the same to 76% and 50% of problem instances, respectively.

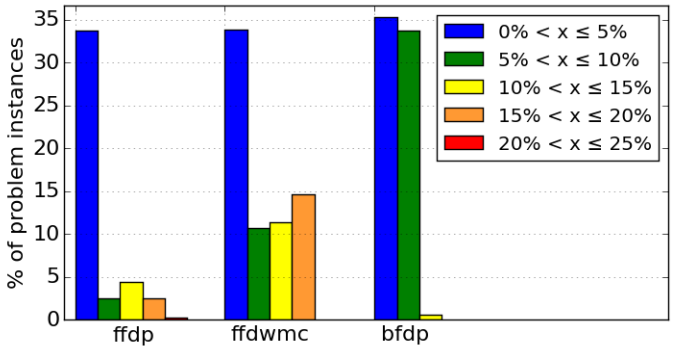


Figure 4.10: Distance in % from best greedy heuristic solution per algorithm and number of occurrences in % of problem instances.

Figure 4.10, which illustrates the percentage of problems where a greedy heuristic did not have the best solution and the distance to it, helps us understand the behavior of solutions that were not the best ones. The most important aspect to notice in this graph is that, for all considered algorithms, the maximum distance to the best solution is bellow 25% and that, in most of the cases, it is bellow 10%. Also, one can notice that DP manages to give a solution at most 10% worst than the best greedy solution for 99.20% of problem instances where it does not output the best solution.

This first analysis based only on solution cost quality indicates the superiority of FFD-P and DP against FFD-WMC, however, to have a better understanding, it is necessary to verify their individual execution times too.

Table 4.5 summarizes the individual execution times from the considered greedy heuristics. Those values are the sum of the execution times used to compute solutions to all Class B problem instances. We can observe that despite giving good solutions, DP responds for around 90.78% of the sum of greedy heuristics execution times, followed by FFD-WMC and FFD-P, responsible for around 6% and 2.44% respectively. This indicates that it may be interesting to reduce the size of the greedy heuristics group to have a smaller execution time. However, it is also important to verify the impact of this reduction on the solution quality.

Algorithm	**Time (s)**	**Participation**
B.F. Dot Product	1012.535	90.78%
F.F.D. Priority	27.25	2.44%
F.F.D. Windowed Multi-Capacity	75.57	6%

Table 4.5: Execution times from greedy heuristics

Table 4.6 presents metrics related to possible combinations of greedy heuristics groups. It is possible to verify that using only FFD-P improves the execution time in 97.55%. However, doing so also degrades 43.47% of solutions in about 17.38%. Also, we can verify that it is possible to improve the execution time in 90% with a smaller impact over solution quality when using FFD-P and FFD-WMC together. In this case, we observe that around 19.02% of solution costs would suffer a degradation between 0.87% and 4%,

	MAX	AVG	MIN	MED	DEG	IMP
DP	9.93	4.46	0.02	4.64	41.08	9.21
FFD-P	17.38	3.73	0.01	1.97	43.47	97.55
FFD-WMC	16.34	7.07	0.01	5.56	69.58	93.22
DP & FFD-P	6.20	2.07	0.02	1.38	18.61	6.77
DP & FFD-WMC	9.93	4.72	0.02	4.73	50.13	2.44
FFD-P & FFD-WMC	4.00	0.87	0.01	0.58	19.02	90.78
DP & FFD-P & FFD-WMC	0.00	0.00	0.00	0.00	0.00	0.00

Table 4.6: MAX, AVG, MIN and MED are maximum, average, minimum, and median of solution costs distances, in percentage of solutions from the original group of heuristics. DEG is the percentage of solutions that were degraded and IMP is the execution time improvement in percentage of original execution time.

in average. Thus, clearly, if it is necessary to improve execution time, the best option is to remove DP from the group of heuristics.

Finally, we identify 3 possible configurations for the greedy heuristics group: (i) the fastest one, which would be composed uniquely of FFD-P, with an improvement of 97.55% of execution time but with around 43.47% of its solutions degraded, (ii) the medium term, which would be composed of FFD-P and FFD-WMC, with an improvement of 90.78% of execution time but having 19.02% of solutions degraded and, (iii) the slowest configuration, composed of FFD-P, DP, and FFD-WMC which give the best solutions. It is important to notice, however, that "slowest" here means a configuration capable of solving all Class B problem instances in less than 19 minutes with per problem execution time averages of 1.4 s, 0.03 s and 0.1 s for DP, FFD-P and FFD-WMC, respectively. We also observed maximum execution times of 5.7 s, 0.23 s and 0.35 s for DP, FFD-P, and FFD-WMC, respectively, i.e., very short execution times.

4.6 Conclusion

In this chapter we discussed the problem of calculating initial placements of distributed applications over Cloud based infrastructures. Applications built upon several components have to be deployed over multiple Clouds to benefit from many different VMs with different renting costs. This is indeed a complicated problem, specially as the complexity of these applications and the number of parameters and features grows.

Our objective was to describe fast algorithms to solve the problem of calculating an initial placement for component-based applications over multiple clouds. In addition, we considered that the parameters of this problem (number of VM types, multiple cloud providers, number of components, number of dimensions, and objective functions) could be large, which might prohibit the usage of solutions such as MIP solvers and meta-heuristics.

To achieve that objective, we adapted efficient greedy heuristics originally conceived to solve the multi-dimensional bin packing problem to our problem. After a detailed evaluation, we indicated that our greedy heuristics were capable of calculating solutions compatible with meta-heuristics solutions but calculated at least 100 times faster. Cer-

tainly, our approach is better suited for situations where there is space for a light degradation of solution quality in exchange of a reduced execution time. It is also possible to use our greedy heuristics solutions as a first solution input for meta-heuristics or exact algorithms. Virtually any application of the cost-aware multi-dimensional bin packing problem with heterogeneous bins may take advantage of our results and algorithms.

Finally, the heuristics presented in this chapter will be part of the cost and communication-aware heuristics described in Chapters 5 and 6.

Chapter 5

Initial Communication and Cost-Aware Placement

In this chapter, we improve the application and infrastructure models discussed in Chapter 4 to be able to describe communication constraints. Using the greedy heuristics discussed in that chapter as building blocks, we introduce communication and cost-aware heuristics that can quickly calculate initial placements of distributed applications over Cloud-based infrastructures.

5.1 Introduction

In Chapter 4, we presented greedy algorithms that could compute *communication-oblivious* initial placements of distributed applications on multiple Clouds while minimizing renting costs and satisfying resource constraints. In this chapter, we improve the application and Cloud models previously introduced to make it possible to describe *communication constraints*. We also propose a heuristic that takes these new constraints into consideration and can calculate initial placements of distributed applications on multiple Clouds while minimizing renting costs and *satisfying resource and communication* constraints.

As we discussed in Section 2.4, placement scenarios can be very large. We consider we are dealing with thousands of VM types from hundreds of different Cloud providers and hundreds of application components. Furthermore, we consider that, besides of resource requirements (cf. Chapter 4), distributed applications may also describe communication requirements. This results in a NP-complete [54] problem and, consequently, scalability issues must be taken into consideration.

In this chapter, we introduce 2PCAP, an efficient and scalable heuristic that mixes graph-based algorithm concepts that can compute good quality initial cost and communication-aware placements very quickly for small and large scenarios.

In the next section we present the problem in more detail and model it as a *mixed integer programming* problem. In Section 5.3 we discuss the state of the art. Section 5.4 presents our application and Cloud models. Section 5.5 details the proposed heuristic which is evaluated in Section 5.7.

5.2 Communication-Aware Placement of Distributed Applications on Multiple Clouds

In this section we characterize the *communication-aware placement of distributed application on multiple Clouds problem* (CAPDAMP). We present distributed application and Cloud-based infrastructure models and formalize CAPDAMP. For a more extensive discussion about our hypotheses and application and Cloud models, we refer the interested reader to Chapter 3.

We model distributed applications as component-based applications (cf. Section 2.3.2). Each application *component* has *resource requirements* and connections between components have *communication requirements*. We consider that there is a description of *virtual machine (VM) types capacities* and *renting prices*. Furthermore, we consider that *communication* capacities of VMs potentially instanced from these VM types is available. Resource requirements may be number of CPU cores, RAM or disk usage, for example. Our hypothesis is that communication requirements are described to satisfy *latency* needs. Hence, in this thesis, we consider that communication requirements are described in terms of latency.

Our objective is to map each application component to a VM instance in order to *minimize renting costs* and satisfy *resource and communication constraints*. Resource capacities of a VM instance *must be larger than or equal to* the sum of resource requirements of components it hosts (cf. Section 3.1.3). Similarly, communication requirements between each pair of components must be less than or equal to the connection capacities between VMs hosting them (cf. Section 3.1.3). We suppose that connection requirements and capacities are used to characterize communication latencies and that they can be expressed in a numerical way[1].

In this chapter we focus on *initial placements*, i.e., we consider that the application or part of it is not previously deployed. Reconfiguration scenarios are discussed in Chapter 6. Figure 5.1 illustrates an example of a CAPDAMP scenario.

In the next sections we formalize the CAPDAMP (cf. Section 5.2.1) and present two ways of modeling it: as a graph homomorphism problem and as an *optimization problem*. Both models will be important for the evaluation of our contribution.

[1]It is sufficient that requirements have an ordered representation.

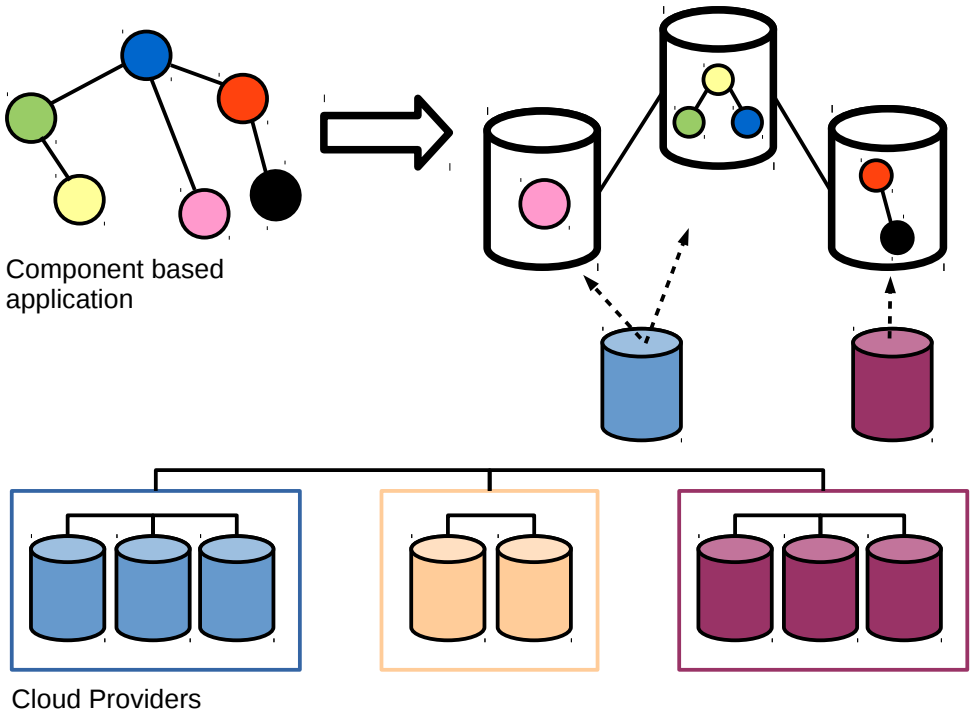


Figure 5.1: Communication-aware placement example. A component-based application is represented as a graph where nodes represent components and edges represent connections between components. Colored boxes represent Cloud providers and colored cylinders inside of them are VM types. Transparent cylinders are VM instances and dashed arrows indicate the VM type from which a VM was instanced. The color of a VM type indicate its origin Cloud provider and the black connectors between VM types and Cloud providers indicate connections. Our placement objective is to map application components to VMs while minimizing costs and satisfying resource and communication constraints.

5.2.1 Problem Statement

Let $\mathcal{I}$ be a set of components, $\mathcal{T}$ a set of VM types with D resources (or dimensions) of interest. Let $r_{i,d}$ be the requirements of component i on dimension d, $c_{t,d}$ the capacity of VM type t on dimension d and p_t the price of renting a VM of type t per unit of time.

Each VM type $t \in \mathcal{T}$ is hosted in a *machine group* $s \in \mathcal{S}$ where $\mathcal{S}$ is a set of machine groups, a set of machines indistinguishable from the connection constraint point of view. This means that VM types belonging to the same machine group share the same *connection capacities*. Machines group can be clusters or *Cloud provider sites*, for example. Each VM type can be instanced $|\mathcal{I}|$ times (cf. Section 3.1.3). A component must be assigned to exactly one VM but each VM may host several components.

The requirement of a connection between components $i \in \mathcal{I}$ and $j \in \mathcal{I}$ is represented as $x_{i,j}^{comp}$. Similarly, the capacity of a connection between two machine groups $s \in \mathcal{S}$ and $u \in \mathcal{S}$ is represented as $x_{s,u}^{mg}$. As we consider that VMs belonging to the same machine group share the same connection properties, the connection capacity between a VM $v \in \mathcal{V}$,

that belongs to machine group s, and a VM $n \in \mathcal{V}$, that belongs to machine group u, is represented by $x^{vm}_{v,n}$ and, furthermore, $x^{vm}_{v,n} = x^{mg}_{s,u}$.

The objective is to assign all components to VMs in a way that resource and communication requirements of components are satisfied, VMs' capacities are not exceeded, and renting costs are minimized.

5.2.2 CAPDAMP as Graph Homomorphism Problem

Graph are recurrently used as a representation of application and infrastructure topologies. In manner of fact, the CAPDAMP can be modeled as a generalization of the NP-complete graph homomorphism problem [54, 53]. Formally, let G and H be two graphs, $V(G)$ and $V(H)$ be the vertices of G and H, and $E(G)$ and $E(H)$ be the edges of G and H. A homomorphism from a graph G to a graph H is a mapping $\alpha : V(G) \rightarrow V(H)$ such that:

$$(x,y) \in E(G) \Rightarrow (\alpha(x),\alpha(y)) \in E(H) \tag{5.1}$$

In the context of the CAPDAMP, G would be the application graph, and H a graph describing the topology of the Cloud. A placement would be the function α.

CAPDAMP is a generalization of the graph homomorphism problem because in the latter there is no assumption concerning edge weights, vertex weights, or mapping costs.

5.2.3 Optimization Problem Formulation

We model the CAPDAMP as an *optimization problem formulation*. In this section, we present a general modeling of that problem as a *mixed integer programming* (MIP) problem which will be used as input for a *solver* in Section 5.7, where we evaluate our contributions.

Let $\mathcal{V} = \{v_{k,t} \mid 1 \leqslant k \leqslant |\mathcal{I}|, t \in \mathcal{T}\}$ be the set containing all VMs from type t. The price of v is $p_v = p_t$ and its initial capacity $c_{v,d} = c_{t,d}$.

Let $m_{i,v} = 1$ if a component i is assigned to a rented VM v, and 0 otherwise. Let $a_v = 1$ if v hosts at least one component and 0 on the contrary.

The optimization problem is described in Equation 5.2 and variables and constants are summarized in Table 5.1.

$$\begin{aligned}
&\text{Minimize} \sum_{v \in \mathcal{V}} p_v.a_v \\
&\text{s.t.} \\
&\quad \sum_{v \in \mathcal{V}} m_{i,v} = 1 && \forall i \in \mathcal{I} && (i) \\
&\quad \sum_{i \in \mathcal{I}} m_{i,v}.r_{i,d} \leqslant c_{v,d} && \forall v \in \mathcal{V}, 1 \leqslant d \leqslant D && (ii) \\
&\quad a_v = \begin{cases} 1 & \text{if } \sum_{i \in \mathcal{I}} m_{i,v} > 0 \\ 0 & \text{otherwise} \end{cases} && \forall v \in \mathcal{V} && (iii) \\
&\quad m_{i,n}.m_{j,v}.(x^{vm}_{n,v} - x^{comp}_{i,j}) \geqslant 0 && \forall v,n \in \mathcal{V}, \forall i,j \in \mathcal{I} && (iv) \\
&\quad m_{i,n}, m_{j,v} \in \{0,1\}
\end{aligned} \tag{5.2}$$

In Equation 5.2, (i) guarantees that each component is assigned to at most one VM; (ii) ensures that no instantiated VM has more components than it can host; (iii) guarantees that $a_v = 1$ when there is at least one component assigned to v, and (iv) guarantees that network requirements are satisfied.

$\mathcal{I}$	Set of components.
$\mathcal{T}$	Set of VM types.
$\mathcal{V}$	Set containing all potential VMs.
D	Number of resources/dimensions.
p_t	Price of VM type $t \in \mathcal{T}$.
p_v	Price of VM $v \in \mathcal{V}$.
a_v	$a_v = 1$ if $v \in \mathcal{V}$ is being used, 0 otherwise.
$v_{k,t}$	k-th VM of type $t \in \mathcal{T}$.
$m_{i,v}$	$m_{i,v} = 1$ if component $i \in \mathcal{I}$ is assigned to VM $v \in \mathcal{V}$, 0 otherwise.
$r_{i,d}$	Requirement of component $i \in \mathcal{I}$ on dimension d.
$c_{v,d}$	Capacity of VM $v \in \mathcal{V}$ on dimension d.
$x^{comp}_{i,j}$	Requirement of connection between component $i \in \mathcal{I}$ and $j \in \mathcal{I}$.
$x^{mg}_{s,u}$	Capacity of connection between machine group $s \in \mathcal{S}$ and $u \in \mathcal{S}$.
$x^{vm}_{v,n}$	Capacity of connection between VM $v \in \mathcal{V}$ and $n \in \mathcal{V}$.

Table 5.1: Summary of variables used in this section and in Equation 5.2.

5.3 Related Work

We address the CAPDAMP (cf. Section 4.2), which stands for communication-aware placement of distributed applications on multiple Clouds problem. The CAPDAMP is a generalization of the *graph homomorphism problem*. This problem is NP-complete [54] and, consequently, approaches to this problem must take scalability issues into account.

In the next sections, we discuss state of the art approaches to the CAPDAMP and related communication-aware problems and position 2PCAP in relation to them. We divide this literature into three groups based on their approach and ordered by scalability: exact approaches, meta-heuristic approaches, and heuristic based approaches.

5.3.1 Exact Algorithms

There are many approaches for the CAPDAMP and related problems based on exact algorithms, but in this section we focus mainly on those based on *Mixed Integer Programming* (MIP) problems and *solvers*. An important characteristic is the expressiveness of MIP modeling, which allows for the description of problem details that would not be possible or would be more difficult to express using other strategies.

In [102], a Mixed Integer Programming (MIP) is proposed for the placement of distributed applications on the Cloud. The objective is to maximize availability by modeling fault-tolerance measures. Similarly, [58] proposes a MIP to minimize application downtime. Both approaches neither consider renting cost minimizations nor allow for more

than two dimensions of interest. In [68], a very expressive MIP to compute the placement of services on multiple Clouds is presented. Despite allowing for cost optimization, heterogeneous VM types, and resource constraints, it does not allow for an explicit description of communication constraints. Finally, in [60], a hierarchical approach to the process placement in multi-core clusters is presented. However, only the *communication problem* is considered and both processes and hosting machines are homogeneous, contrary to our work.

In the context of CAPDAMP, despite their ability to calculate optimal solutions, exact algorithms are not scalable. Consequently, they are an attractive approach only when placement scenarios are small.

5.3.2 Meta-heuristics

Meta-heuristics such as genetic algorithms, simulated annealing, and ant colony optimization, are very powerful tools for calculating solutions for some variations of the CAPDAMP thanks to their ability to handle very complex models [70].

In [24] and [117], the authors propose two very similar approaches based on genetic algorithms to calculate the placement of services on the Cloud targeting cost minimization while satisfying CPU, memory, disk and latency constraints. In [55], a simulated annealing based approach to the VM consolidation problem is presented. In the same topic, an ant colony algorithm for a multi-objective VM consolidation problem aiming at minimizing energy consumption and resource waste is described in [40]. In [36] another ant colony based approach for the VM consolidation problem is proposed.

The main issue of using meta-heuristics for communication-aware problems is that, in summary, there is a correlation between the time given to a meta-heuristic to calculate a solution and its quality. Also as they are generic tools, i.e., tools that are not created for specific problems, their solutions are very sensitive to parameter tuning and other configurations. Finally, it is important to highlight that depending on the CAPDAMP characteristics, using meta-heuristics may not be recommended. Specially when search space is rugged and/or plain [105] (it contains too many *local optimal solutions* and low correlation between them).

5.3.3 Heuristics

The CAPDAMP is NP-complete [54] and heuristics may be good options to quickly compute solutions. In this section we describe heuristics based mainly on graph algorithms and discuss them in regard to CAPDAMP and 2PCAP.

In Section 5.2.1 we discussed that using graphs is a common way of modeling application and infrastructure topologies. We also briefly presented the CAPDAMP as a generalization of a graph homomorphism problem.

In spite of the many existent approaches to problems related to CAPDAMP, to the best of our knowledge, there are no state of the art heuristics that take into consideration all CAPDAMP's constraints and objectives, such as renting costs minimization, resource and communication constraint satisfaction, or component and VM heterogeneity. Many

approaches ignore part of problem parameters and model the placement problem as a graph partition problem [19] (NP-Hard).

A graph based greedy heuristic for calculating the task mapping on supercomputer clusters is presented in [28]. Also using a greedy heuristic, an approach to place tasks on the Cloud is proposed in [64]. Using a max-clique based approach, [16] and [69] describe algorithms for the consolidation of VM types. A similar problem is addressed in [84], which adds the challenge of having to place a virtual network aiming at satisfying resource and network constraints. Using a min cut approach, a hierarchical representation of the network and a graph modeling of the application, [75] tackles the traffic-aware virtual machine placement on data centers. A hierarchical approach for the deployment of distributed scientific applications on the Cloud is presented in [34]. In [120], a graph matching algorithm based on a *graph query* approach for the service placement on the Cloud is proposed. In [119], a game theory approach is proposed to the allocation of virtual network for VMs from multiple data centers. Finally, [51] presents a heuristic based on a relaxed MILP to compute a solution for a VM consolidation problem.

The aforementioned articles aim primarily at solving the communication constraint problem letting the packing problem (resource constraint problem) in second place. Graph-based modeling can efficiently describe communication constraints. However describing at the same time resource constraints and renting costs tend to be more difficult. As a result, issues such as VM/machine heterogeneity, renting costs, and multi-dimensionality of resources are not addressed.

The authors of [59] propose an approach for placing services onto the Cloud with the objective of minimizing costs. They employ a hierarchical Cloud topology description similar to ours, they use clustering heuristics to solve the communication problem and bin packing heuristics to the packing problem. However, the bin packing algorithms only consider memory and CPU constraints, the Cloud topology levels are predefined and they consider communication constraints as soft constraints.

5.3.4 Discussion

As the CAPDAMP is NP-hard, using *exact algorithms* to calculate optimal placements is feasible only for very small problem instances. To overcome this limitation there is a plethora of more scalable approaches based mainly on meta-heuristics and heuristics. *Meta-heuristics* have their solution qualities correlated to the time given to process a problem. Hence, depending on problem size, using a meta-heuristic may still be unfeasible. Furthermore, as they are generic tools, meta-heuristics tend to be very sensitive to parameter tuning specific for each scenario.

Other *heuristics* usually aim primarily at solving the graph partitioning (or communication constraint) problem letting the packing (or resource constraint problem) in second place. Graph-based modeling can efficiently describe communication constraints, however, describing at the same time resource constraints and renting costs tends to be more difficult. Thus, issues like VM heterogeneity, renting costs and multi-dimensionality are not addressed at the same time.

In this section, we propose an efficient and scalable heuristic which addresses the aforementioned problems. Using graph clustering and multidimensional bin packing strategies, it manages to calculate good quality solutions very quickly, as evaluated in Section 5.7.

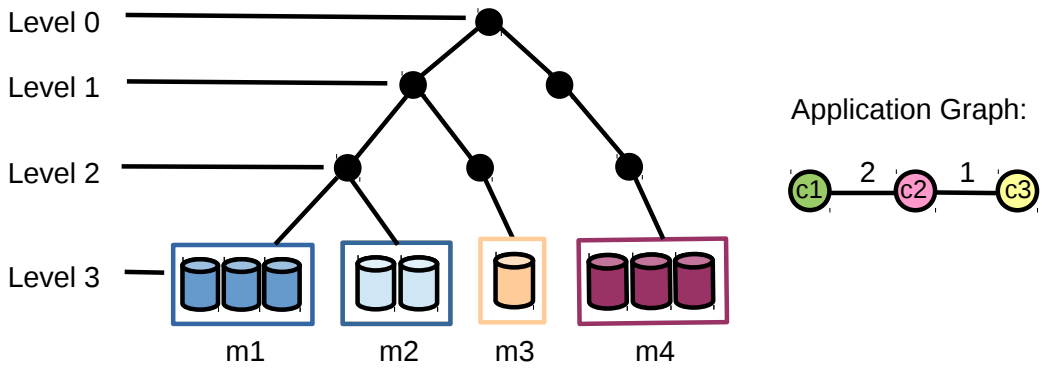


Figure 5.2: Cloud topology (left side) and application graph (right side).

5.4 Modeling the Cloud Infrastructure and Distributed Applications

In Section 4.2, we discussed application and Cloud infrastructure models including ways of representing communication constraints. We explained that, in this thesis, communication requirements and capacities are described in terms of latencies. However, it is still necessary to address the challenge of how to measure or model latencies in a virtualized environment such as the Cloud.

In the next sections we discuss that problem in more detail and introduce an approach of describing communication constraints that will be crucial for the heuristic proposed in Section 5.5.3.

5.4.1 Cloud Network Topology

An important challenge related to modeling communication constraints for the CAP-DAMP is gathering precise information from Cloud provider's networks. Due, particularly, to virtualization, it is difficult to precisely predict latencies or network bandwidth between machines. This task is even more difficult when machines are hosted in different Cloud providers.

As discussed in Section 2.1, this issue happens because, commonly, Cloud providers do not make available details concerning their internal network infrastructure and, usually, Cloud providers are connected among themselves over Internet and not over any dedicated network. In spite of that, it is possible to have a more coarse-grained latency prediction. In general, network latencies are better between machines hosted in the same Cloud provider than between machines from distinct Cloud providers. The same logic applies for Cloud provider sites (cf. Section 2.1).

Thus, we are interested in a solution that is able to make use of *uncertain* or *less precise* information from Cloud infrastructure latencies. Our approach to this issue is to model Cloud based infrastructure as a tree. An example of this modeling is illustrated in Figure 5.2.

In the Cloud *topology* tree, *leaves* are sets of VM types which we call *machine groups*. Each *inner node* represents a *connection* to all machine groups available in the sub-tree having the inner node as root. Finally, the level of each inner node represents the *quality* of the machine group connection. Notice that we describe quality in terms of latency, hence the better the quality, the smaller the latency.

Given two machine groups s_1 and s_2, we call *maximum connection quality* the highest internal node which interconnects s_1 and s_2. For example, in Figure 5.2, machine groups $m1$ and $m2$ are connected at levels 0, 1, and 2. Thus, their maximum connection quality is 2. On the other hand, $m1$ and $m3$ (also $m2$ and $m3$) are connected only at level 0, and consequently, they have a maximum connection quality of 0.

VM types sharing a same machine group are interconnected with connection quality equals to the level of the leaves. In the example illustrated in Figure 5.2, VM types' connections inside the same machine group have quality 3. Thus, the closest an internal node is to the leaves, the smallest will be the latency between machine groups connected by it.

5.4.2 Distributed Application Communication Topology

To describe distributed applications we extend the component-based model presented in Chapter 4 by including an approach to modeling communication requirements in addition to the previously discussed resource requirement model.

Distributed applications are modeled using the component-based paradigm and represented as *graphs*. Components are *nodes* and connections between components are *weighted edges*. Weights are *connection requirements*, defined in terms of connection quality as discussed in Section 5.4.1, matching the Cloud topology.

In the example illustrated in Figure 5.2, components $c1$ and $c2$ communicate and require a connection quality of *at least* 2, while $c2$ and $c3$ require at least 1.

5.5 Two Phase Communication-Aware Heuristic

The proposed heuristic named *Two Phase Communication-Aware Heuristic* (2PCAP) takes advantage of the tree structure of the multi-cloud topology (cf. Section 5.4.1) to calculate solutions for initial placements of distributed applications on multiple Clouds.

2PCAP is a divide-and-conquer algorithm which looks for *subsets* of machine groups that satisfy connection requirements from *subsets* of components. Once they are found, placements of component subsets on machine group subsets, called *sub-placements*, are calculated using fast *communication-oblivious* multi-dimensional bin packing heuristics (cf. Chapter 4). Finally, the placement of the entire application is obtained by composing sub-placements.

2PCAP has two phases:

(i) First, it recursively *decomposes* components and machine groups into subsets, creating communication-aware sub-placements. This procedure is repeated until sub-placements that can be calculated by communication-oblivious heuristics are generated (the bottom of the recursion).

(ii) Secondly, the computed sub-placements are recursively compared from the leaves to the root of the tree. Those with the best cost *compose* the solution for other sub-placements until the final placement is computed.

Figure 5.3 illustrates an execution of 2PCAP and will used throughout this section. We use tables to represent the heuristic's *decomposition* steps. The name of the tables refer to the level of decomposition (in Figure 5.3 they are L0, L1 and L2). *Component subsets* are represented in the left part of the tables and *machine group subsets*, in the upper part.

In the next subsections we explain decomposition and composition phases in detail.

5.5.1 Phase 1: Decomposition

The decomposition phase has fundamentally 2 steps: the decomposition of component and machine group subsets and the computing of sub-placements. We discuss this phase in more detail in the next paragraphs.

Component Subset

A *component subset* is a set of nodes from a connected subgraph from the application graph. Component subsets have the property that every connection between its components has a communication quality requirement superior or equal to ℓ, where $0 \leqslant \ell < H$ and H is the height of the multi-cloud tree. $\mathcal{I}_\ell$ is the set of all component subsets on level ℓ.

In the example illustrated in Figures 5.3(b) and 5.3(c), there is only one component subset composed of components $c1$, $c2$ and $c3$, i.e. $\mathcal{I}_0 = \mathcal{I}_1 = \{\{c1,c2,c3\}\}$. In Figure 5.3(d), however, there are two component subsets: one composed of c_1 and $c2$ and another one composed of $c3$, i.e. $\mathcal{I}_2 = \{\{c1,c2\},\{c3\}\}$.

Observe that the union of all component subset elements forms the set of components, i.e. $\cup_{i_\ell \in \mathcal{I}_\ell} = \mathcal{I}$ and $0 \leqslant \ell < H$.

Machine Group Subset

A *machine group subset* contains machine groups from sub-trees of the multi-cloud tree topology. All machine groups contained in the same subset are connected with connection quality superior or equal to ℓ, where $0 \leqslant \ell < H$ and H is the height of the multi-cloud tree. $\mathcal{S}_\ell$ contains all subsets built on level ℓ.

In Figure 5.3(b), there is only one machine group subset composed of all machine groups, namely $m1, m2, m3$ and $m4$, i.e. $\mathcal{S}_0 = \{\{m1,m2,m3,m4\}\}$. In Figure 5.3(c) (level 1), there are two machine group subsets, one composed of $m1, m2$ and $m3$ and another composed of $m4$, i.e. $\mathcal{S}_1 = \{\{m1,m2,m3\},\{m4\}\}$. Finally, in Figure 5.3(d) (level 2), there are three machine groups, i.e. $\mathcal{S}_2 = \{\{m1,m2\},\{m3\},\{m4\}\}$.

Notice that the union of all machine group subset elements forms the set of machine groups, i.e. $\cup_{i_\ell \in \mathcal{S}_\ell} = \mathcal{S}$ and $0 \leqslant \ell < H$.

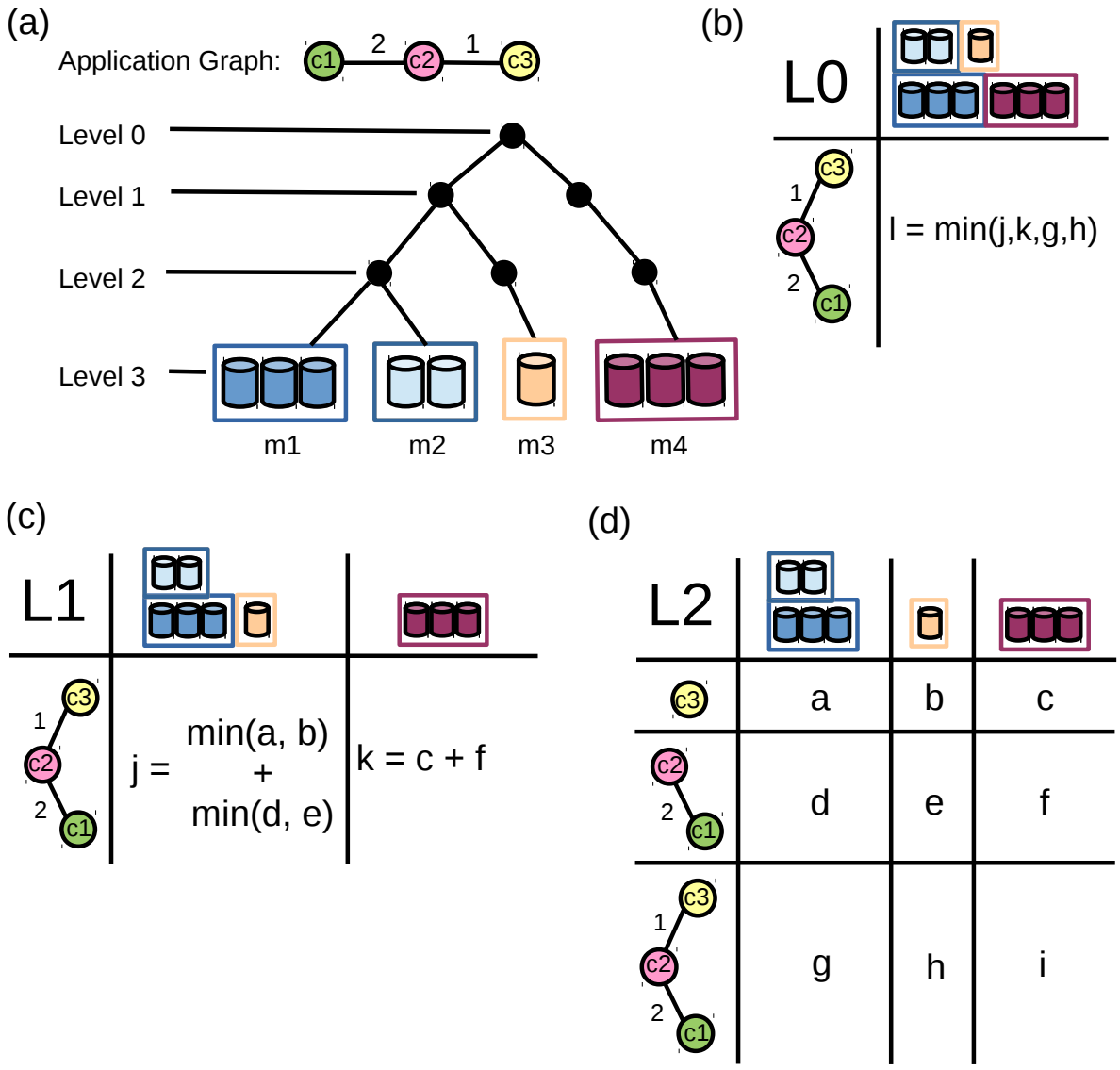


Figure 5.3: Placement example. Tables represent heuristics' *decomposition* steps. The name of the tables refer to the level of decomposition (L0, L1 and L2). *Component subsets* are represented in the left part of the tables and *machine group subsets*, in the upper part.

Component and Machine Group Decomposition

The process that *generates* component and machine group subsets is called *decomposition*.

Machine group subsets are decomposed through gathering all inner nodes from the Cloud topology tree at a given level ℓ. These nodes actually form a forest of subtrees and each set of leaves from each subtree is a valid machine group subset. At level 0, there will always be only one internal node, which happens to be the root of the Cloud topology.

In the example presented in Figure 5.3(b) (level 0), there is only one machine group subset $\mathcal{S}_0 = \{\{m1,m2,m3,m4\}\}$. The decomposition of this subset on level 1, represented in Figure 5.3(c), has two machine group subsets, i.e. $\mathcal{S}_1 = \{\{m1,m2,m3\},\{m4\}\}$, given that there are two subtree roots at that level.

To decompose component subsets, all connections of the original application graph that require a connection quality smaller than a given level ℓ are removed. The nodes of each resulting connected sub-graph is a valid component subset. Notice that, at level 0, there will always be only one component subset which is equal to the application.

In the example depicted in Figure 5.3(b) (level 0), there is only one component subset $\mathcal{I}_0 = \{\{c1,c2,c3\}\}$. Given that in the application there is not a connection quality requirement smaller than one, at level 1 (L1), represented in Figure 5.3(c), there is also only one component subset $\mathcal{I}_1 = \{\{c1,c2,c3\}\}$. However, at level 2 (cf. Figure 5.3(d)), the decomposition result in two subgraphs and consequently, two component subsets, i.e. $\mathcal{I}_2 = \{\{c1,c2\},\{c3\}\}$.

Sub-placement

A *sub-placement* is the placement of a subset of components on a subset of machine groups. Similarly to an "usual" placement (cf. Section 5.2) a sub-placement aims at minimizing VM renting costs while satisfying resource and communication constraints established by the application. Sub-placements are calculated during decomposition and are selected during composition (cf. Section 5.5.2) phases.

Let $i_\ell \in \mathcal{I}_\ell$ and $s_\ell \in \mathcal{S}_\ell$ be the sets of all component and machine group subsets, respectively, constructed on level $\ell, 0 \leqslant \ell < H$. The sub-placement of i_ℓ on s_ℓ can only be calculated if it is a *bottom sub-placement*.

A sub-placement is a *bottom sub-placement* if all sub-placements at level ℓ can be calculated by communication oblivious heuristics while still satisfying communication quality requirements. Formally, let $x^{comp}_{i,j}$ be the communication requirement between components i and j and let $x^{mg}_{m,n}$ be the communication capacity between machine groups m and n. Formally, a bottom sub-placement has the following property:

$$x^{comp}_{i,j} \leqslant x^{mg}_{m,n},\ \forall i,j \in i_\ell, \forall m,n \in s_\ell. \tag{5.3}$$

This means that any pair of VM types from machine groups contained in s_ℓ will satisfy the communication requirements from any pair of components contained in i_ℓ.

In the example illustrated in Figures 5.3(b), 5.3(c), and 5.3(d), each intersection between machine groups and component subsets is a sub-placement. The sub-placement at level 0 (cf. Figure 5.3(b)), is the sub-placement of $\{c1,c2,c3\}$ on $\{m1,m2,m3,m4\}$. One can notice that it is not a bottom sub-placement since the constraint described in Equation 5.3 does not holds. For instance, if $c1$ and $c2$ were placed on $m4$ and $c3$ on

$m1$, the communication constraints between $c2$ and $c3$ would not be satisfied. Likewise, sub-placements in Figure 5.3(c) are not bottom sub-placements. If $c1$ were placed on $m1$ and $c2$ and $c3$ on $m3$, the connection constraint between $c1$ and $c2$ would not be satisfied. Nevertheless, in Figure 5.3(d) (level 2), Equation 5.3 is satisfied by all sub-placement.

Let ℓ_{max} be the highest connection quality requirement present in the component graph. We can observe that the sub-placement of every $i_\ell \in \mathcal{I}_\ell$ on $s_{\ell_{max}} \in \mathcal{S}_{\ell_{max}}$ is a valid bottom sub-placement. Decomposing component and machine group subsets at levels superior to ℓ_{max} would result in an unnecessary fragmentation. As a consequence, there would be less components and VM types available per sub-placement which leads to a reduction of the *packing level* of sub-placements and, potentially, to higher renting costs. Hence, there is no reason to continue the decomposition process beyond ℓ_{max}.

In the example illustrated in Figure 5.3(a), the highest communication quality requirement is 2. Hence, $\ell_{max} = 2$ and the decomposition stops at level 2 (cf. Figure 5.3(d)).

Packing Improvement

2PCAP heuristic does not compute all possible sub-placements during the decomposition phase. For example, in Figure 5.3(d), there are no bottom sub-placements computed using subsets composed of nodes $\{c_1c_2,c_3\}$, $\{c_1,c_3\}$, or $\{c_2,c_3\}$. Doing so would result in an exponential complexity, and would lead to excessively long execution times for large problems.

To further explore the solution space without increasing too much the algorithm's time complexity, 2PCAP calculates sub-placements of every generated component subset which is not part of a bottom sub-placement on machine group subsets generated at level ℓ_{max}. This procedure is called *packing improvement* because, as components subsets are larger, 2PCAP will try to assign more components to a same VM and this improves solution packing. Packing improvement is depicted in the last line of Figure 5.3(d). Sub-placements g, h, and i of $\{c1,c2,c3\}$ were generated at $\ell = 0$ (cf. Figure 5.3(b)).

Summary

During the decomposition phase, component and machine group subsets are decomposed until bottom sub-placements are generated. This happens at level ℓ_{max}, which is the highest communication quality requirement from the application. At that step, efficient communication-oblivious heuristics calculate each bottom sub-placement.

5.5.2 Phase 2: Composition

Once all necessary bottom sub-placements are calculated, 2PCAP starts the process of *composition of sub-placements*. The objective is to choose, at each composition step, the less expensive solutions among all available sub-placements. Given the set $\mathcal{I}'_{\ell+1}$ containing all component groups decomposed from $i_\ell \in \mathcal{I}_\ell$ and the set $\mathcal{S}'_{\ell+1}$ decomposed from the machine group $s_\ell \in \mathcal{S}_\ell$, let $u^{is}_{\ell+1}$ be the sub-placement of $i_{\ell+1}$ on $s_{\ell+1}$. Thus, the solution for the sub-placement of i_ℓ on s_ℓ is one of the following:

Case 1: $u^{is}_{\ell+1}$, if $\mathcal{S}_{\ell+1} = \mathcal{S}_\ell$ and $\mathcal{I}_{\ell+1} = \mathcal{I}_\ell$.

Case 2: $\sum_{i \in \mathcal{I}_{\ell+1}} u^{is}_{\ell+1}$ for $s \in \mathcal{S}_{\ell+1}$, if $|\mathcal{S}_{\ell+1}| = |\mathcal{S}_\ell|$ and $|\mathcal{I}_{\ell+1}| > |\mathcal{I}_\ell|$;

Case 3: $\sum_{i \in \mathcal{I}_{\ell+1}} min(u^{is}_{\ell+1}, \forall s \in \mathcal{S}_{\ell+1})$, if $|\mathcal{S}_{\ell+1}| > |\mathcal{S}_\ell|$;

The decomposed subset of components and machine groups of Case 1 are identical to the original subsets. In this case, $u^{is}_\ell = u^{is}_{\ell+1}$.

In Case 2, the decomposed subset of machine groups is identical to the original, but this is not true for the decomposed component subset. In this case, 2PCAP composes the $|\mathcal{I}_{\ell+1}|$ sub-placements on $s_{\ell+1}$. This case is illustrated in Figures 5.3(c) and Figures 5.3(d). Sub-placements c and f (cf. Figure 5.3(d)) compose the sub-placement k (cf. Figure 5.3(c)).

In Case 3, when $|\mathcal{S}_{\ell+1}| > |\mathcal{S}_\ell|$ machine groups are decomposed in more than one subset. Thus, for each decomposed component subset there are $|\mathcal{S}_{\ell+1}|$ possible sub-placements, from which, only the less expensive one is used in the composition process.

For example, level 1 sub-placement j illustrated in Figure 5.3(c) is composed of sub-placements a, b, d and e calculated at level 2 as depicted in Figure 5.3(d). Level 0 sub-placement l (cf. Figure 5.3(b)) is composed of sub-placements j and k from Figure 5.3(c).

5.5.3 2PCAP Algorithm

A summarized pseudo-code of 2PCAP is presented in Algorithm 7. It receives as parameters a component subset cs, a machine group subset mgs and a level – whose value is zero in the beginning of execution – and returns a placement of cs over mgs.

The functions used inside Algorithm 7 are **calculate**, **is_calculated**, **plac**, **decompose**, **compose** and **size**.

- Function **calculate** takes as arguments a component subset cs and a machine group subset mgs and calculates the sub-placement of cs over mgs. For that, **calculate** calls the multi-dimensional bin packing-based heuristics proposed in Chapter 4.

- Function **is_calculated** checks if the sub-placement of a component subset cs over a machine group subset mgs, both subsets passed as arguments to that function, was previously calculated. This is a very simple optimization which avoids calculating sub-placements more than once.

- Function **plac** takes as arguments a component subset and a machine group subset and returns a placement previously calculated by *calculate*.

- Function **decompose** gets a level and subset of components or machine groups and returns the decomposition of that subset at that level.

- Function **compose** builds a new sub-placement by combining two sub-placement passed as argument.

- Function **size** receives an array of component or machine group subsets as argument and returns its size.

Algorithm 7 2PCAP

Input: $level, comp_subset, mg_subset$
Output: min_cost_plac

```
1: min_cost_plac ← ∞
2: if is_calculated(comp_subset, mg_subset) then
3: |   return plac(comp_subset, mg_subset)
4: else if level = level_max then
5: |   calculate(comp_subset, mg_subset)
6: |   return plac(comp_subset, mg_subset)
7: else if level < level_max then
8: |   comp_decomposition ← decompose(comp_subset, level)
9: |   mg_decomposition ← decompose(mg_subset, level)
10: |   if size(mg_decomposition) = 1 then
11: |   |   plac ← null
12: |   |   for cs in comp_decomposition do
13: |   |   |   temp_plac ← 2PCAP(level + 1, cs, mg_subset)
14: |   |   |   plac ← compose(plac + temp_plac)
15: |   |   end for
16: |   |   min_cost_plac ← plac
17: |   else if size(mg_decomposition) > 1 then
18: |   |   plac ← null
19: |   |   for cs in comp_decomposition do
20: |   |   |   min_plac ← null
21: |   |   |   for ms in mg_decomposition do
22: |   |   |   |   temp_plac ← 2PCAP(level + 1, cs, ms)
23: |   |   |   |   if cost(temp_plac) < min_plac then
24: |   |   |   |   |   min_plac ← temp_plac
25: |   |   |   |   end if
26: |   |   |   end for
27: |   |   |   plac ← compose(plac + min_plac)
28: |   |   end for
29: |   |   min_cost_plac ← plac
30: |   end if
31: |   for ms in mg_decomposition do
32: |   |   temp_plac ← 2PCAP(level_max, comp_subset, ms)
33: |   |   if cost(temp_plac) < cost(min_cost_plac) then
34: |   |   |   min_cost_plac ← plac
35: |   |   end if
36: |   end for
37: end if
38: return min_cost_plac
```

2PCAP (cf. Algorithm 7) is recursive. It decomposes component and machine group subsets, i.e. break the problem in smaller sub-problems between Lines 7 and 30 until step ℓ_{max}. Then, bottom sub-placements are calculated between Lines 4 and 6 and their solutions are used to compose other sub-placement solutions, following the cases discussed in Section 5.5.2. Case 1 is described between lines 2 and 3, Case 2 between lines 10 and 16, and Case 3 between lines 17 and 30. A bottom sub-placement may not have a solution. In this case, 2PCAP marks that bottom sub-placement as invalid and gives it an infinity cost.

The packing improvement procedure (cf. Section 5.5.1) is described between lines 30 and 36 of Algorithm 7.

5.5.4 Discussion

An important point is that we consider that the application graph is *connected*. We do this without loss of generality because it would suffice to add edges – more precisely *bridges* – with weight zero connecting disconnected parts of the application graph to build a communication *equivalent* connected graph.

We also consider that the Cloud topology is a tree, hence, it has only one root. We also do this without loss of generality because it is always possible to add a new level to the Cloud tree by creating a new root, connecting it to the old roots, and incrementing by one all application's communication requirements.

5.5.5 2PCAP Complexity

The complexity of 2PCAP is dominated by decomposition operations (**decompose** function) and the computation of sub-placements (**calculate** function).

Function decompose

Let $\mathcal{I}$ and $\mathcal{S}$ bet the sets of components and machine groups, respectively. The decomposition of component subset cs at level ℓ breaks the connected subgraph formed by components of cs, by removing edges having weight larger than ℓ. In the worst case, **decompose** will have to verify each edge of this subgraph, hence, it will have to check $|\mathcal{I}|.(|\mathcal{I}|-1)$ edges. The complexity of **decompose** for component subsets is, thus, $O(|\mathcal{I}|^2)$.

The decomposition of a machine group subset mg at level ℓ consists of gathering the leaves, i.e. machine groups, from subtrees with roots at level ℓ. Its complexity is associated to the number of internal or leaf nodes it has to visit. Hence, the maximum number of nodes would be $O(|\mathcal{S}|.(\ell_{max}))$, the case where the only point of connection between machine groups in the subtree is its root.

The **decompose** function is called whenever machine group and component decompositions are generated. In the worst case, a component subset could generate at most $|\mathcal{I}|$ component subsets in level ℓ_{max}, $|\mathcal{I}|-1$ in level ℓ_{max-1}, $|\mathcal{I}|-2$ in level ℓ_{max-2}, and so on. Hence, $O(|\mathcal{I}|.\ell_{max})$ component subsets could be generated. The same logic applies to machine group subset decomposition. In the worst case, a machine group subset would generate $|\mathcal{S}|$ machine group subsets in level ℓ_{max}, $|\mathcal{S}-1|$ in level ℓ_{max-1}, $|\mathcal{S}-2|$ in

level ℓ_{max-2} and so on. Consequently, $O(|\mathcal{S}|.\ell_{max})$ machine group subsets could be generated. Putting together component and machine group subset decomposition, **decompose** would be called $O(|\mathcal{S}|.\ell_{max}.|\mathcal{I}|.\ell_{max}) = O(|\mathcal{S}|.|\mathcal{I}|.\ell^2_{max})$ times.

The total complexity of function **decompose** is the number of times it is called times its complexity. Hence, it is $O(|\mathcal{S}|.|\mathcal{I}|.\ell^2_{max}).(O(|\mathcal{I}|^2) + O(|S|.(\ell_{max}))) =$
$O(|\mathcal{I}|^3.|S|.\ell^2_{max} + |\mathcal{S}|^2.|\mathcal{I}|.\ell^3_{max})$.

Function calculate

The **calculate** function computes a sub-placement and is performed only at level ℓ_{max}. To do this, **calculate** makes use of multi-dimensional bin packing based heuristics (discussed in Chapter 4). Those heuristics have a complexity of $O(c.t.\log t)$, where t is the number of VM types and c the number of components. Hence, **calculate** has a complexity of $O(|\mathcal{I}|^2.|\mathcal{S}|.|\mathcal{T}|.\log|\mathcal{T}|)$, where $\mathcal{T}$ is the set containing all VM types from all Cloud providers.

Discussion

Putting together the complexities of functions **decompose** and **calculate** we find that the complexity of 2PCAP is $O(|\mathcal{I}|^3.|S|.\ell^2_{max} + |\mathcal{S}|^2.|\mathcal{I}|.\ell^3_{max} + |\mathcal{I}|^2.|\mathcal{S}|.|\mathcal{T}|.\log|\mathcal{T}|)$.

5.6 Examples

In this section, we present complete examples of simulation of 2PCAP for two placement scenarios. The main difference between them is application and Cloud topology.

5.6.1 Example #1

In Figure 5.4, we illustrate a complete execution of 2PCAP. It is out of the scope of this section to explain in detail procedures and concepts related to that algorithm. We invite the interested reader to refer to Section 5.5.

The algorithm starts its execution at level 0 (Figure 5.4(b)), with one component subset composed of the entire application and a machine group subset composed of the entire Cloud infrastructure. 2PCAP's objective is to recursively *decompose* component and machine group subset aiming at using them to build sub-placements (cf. Section 5.5.1). Then, it uses the results of these sub-placements to *compose* a final placement (cf. Section 5.5.2). Notice that sub-placement x is not a *bottom sub-placement* (cf. Section 5.5.1), thus 2PCAP must start the decomposition process of component and machine group subsets at level 1. The result of this procedure is illustrated in Figure 5.4(c).

At level 1 (cf. Figure 5.4(c)), the result of component subset decomposition is the removal of the connection between $c1$ and $c4$. The resulting component subset has the same components as the subset generated at level 0 (cf. Figure 5.4(b)), thus they are equivalent. With respect to machine group subsets, however, as there are two subtrees which have root at level 1, two machine group subsets are generated. As sub-placements u and v are not bottom sub-placements, two calls to component and machine group decomposition at level 2 are performed. The results are illustrated in Figure 5.4(d).

The outcome of component subset decomposition at level 2 (cf. Figure 5.4(d)) is the removal of connections between components $c2$ and $c4$, and $c3$ and $c4$ from the component subset illustrated in Figure 5.4(c). The resulting component subset is equivalent to that generated at level 1. As there are three roots of machine group subtrees at level 2, there will be three machine group subsets. Sub-placements r, s and t are not bottom sub-placements, thus three calls to component and machine group decompositions at level 3 are performed. The results are illustrated in Figure 5.4(e).

At level 3 (cf. Figure 5.4(e)), connections between components $c1$ and $c3$, and $c1$ and $c4$ from the component subset illustrated at Figure 5.4(d) are removed generating three different component subsets. As there are four roots of machine group subtrees in the infrastructure at level 3, there will be four machine group subsets. At this level, all sub-placement are bottom sub-placements and thus can be calculated using communication oblivious placement heuristics. Observe that the component subset illustrated in the fourth line of Figure 5.4(e) was generated at level 0 due to the *packing improvement* process described in Section 5.5.1.

At this point, 2PCAP starts the composition phase. In summary, the results of bottom sub-placements are used to compose intermediary sub-placements, following the composition phases described in Section 5.5.2, until level 0 is reached and the final placement is composed.

5.6.2 Example #2

In Figure 5.5, we illustrate the placement of a star topology application. As this example is similar to Example #1 (cf. Section 5.6.1), we do not discuss it step by step. Notice, however, that in Example #2 all cases of composition discussed in Section 5.5.2 are present. Case 1 is illustrated in the compositions of sub-placements o, p, q, and r, Case 2 is illustrated in the composition of sub-placements m and n, and Case 3 in the composition of sub-placement s.

5.7 Evaluation

It would be interesting to compare the solutions computed by 2PCAP to optimal ones. However, as we previously discussed in Section 5.3, the communication-aware placement of distributed applications on the multiple Clouds problem (CAPDAMP) is NP-Complete. This means that a polynomial time algorithm to solve it is unknown. Consequently, depending on the size of the problem instance it may not be possible to find an optimal solution in practicable time.

Our strategy to circumvent this problem is to divide the evaluation in two steps. First, using *small problem instances* and a MIP solver, we compare 2PCAP solutions to optimal. In a second step, using meta-heuristics and a relaxed version of the CAPDAMP as a baseline algorithm to compute lower bounds, we discuss the performance of 2PCAP on medium and large problem instances. Before discussing the results, we present how the experiments were performed and how input data were generated.

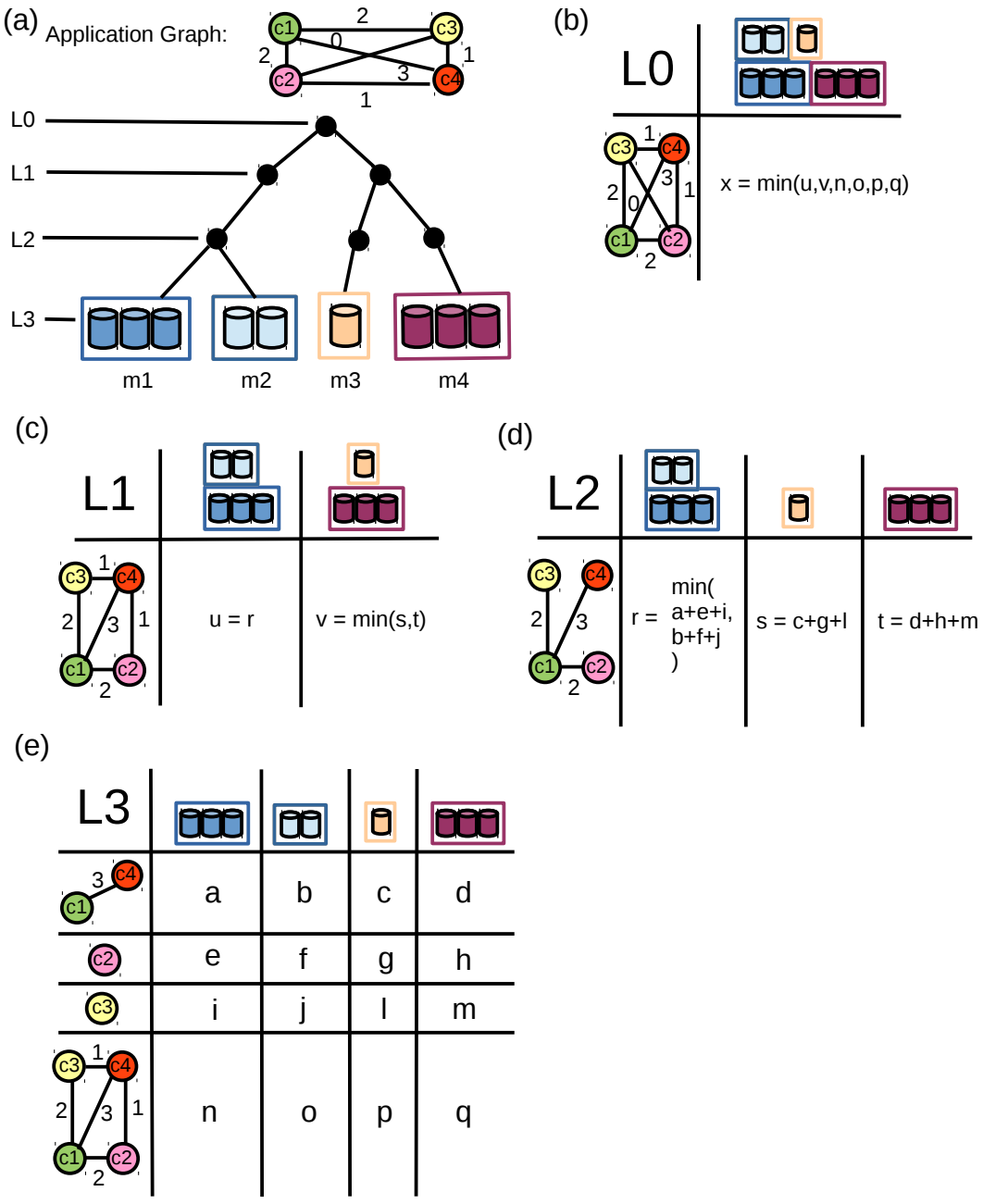


Figure 5.4: Complete Placement Example #1 (cf. Section 5.6.1).

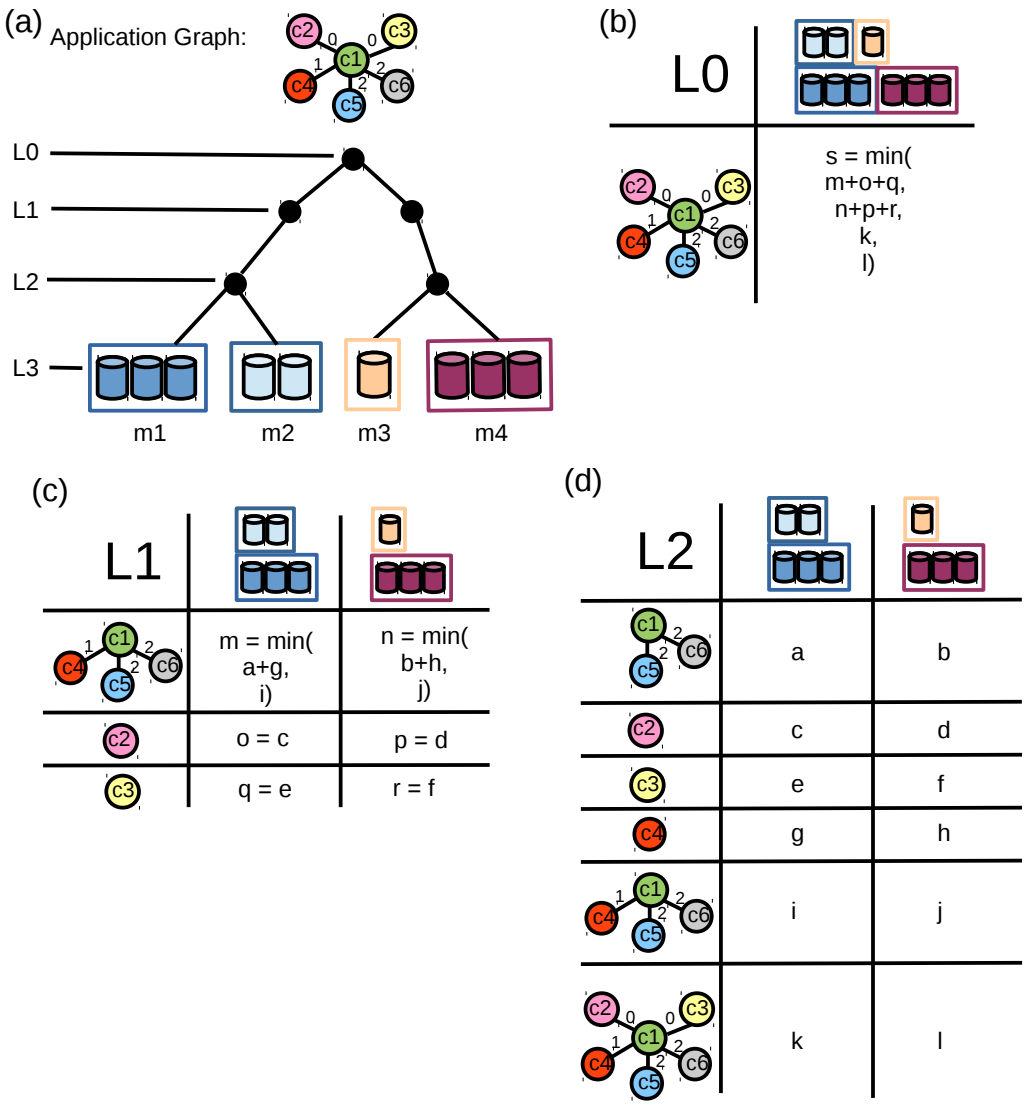


Figure 5.5: Complete Placement Example #2 (cf. Section 5.6.2).

5.7.1 Methodology

In Chapter 3 we have discussed in detail the methodology, notation, and metrics used for evaluating our contributions. Hence, in this section, we summarize concepts previously presented and focus on particularities of proposed heuristics' evaluation, such as experiment format, problem classes parameters or test platform characteristics.

As discussed in Section 3.3, an *experiment* is the resolution of a set of placement *problem instances* by a set of algorithms within a given *time*. Each problem instance has seven parameters: the number of considered resources or *dimensions*, the number of components, the number of VM types, the number of sites, the height of the Cloud tree, the topology of the component-based application and the multi-cloud tree connection schema.

Experiments are organized in three *experiment classes*, namely *A*, *B*, and *C*. Small, and thus easier to solve, problem instances compose Class A; medium-sized problem instances are present in Class B, and, finally, large problems form Class C. Table 5.2 details the range of problem instance parameters that define each class and the total of generated problem instances per class.

	A	B	C
# dimensions	4	5	6
# components	3,5,7,10	10,20,30,40,50	60,80,100,120,140
# vm types	100,250,500,700	500,1000,1500,2000	2500,5000,7500,10000
# sites	25,50,100	100,300,500	500,750,1000
# tree height	3,5	5	7
application topology	l,s,f,r	l,s,f,r	l,s,f,r
connection schema	u	d,a,u	d,a,u
# problem instances	384	720	720

Table 5.2: Parameters of experiment classes. Application topologies are: line (l), star (s), full connected (f), or random (r). Tree connection schemas are: distant(d), agglomerate (a), or uniform (u).

Component requirements and VM capacities are pseudo-random values, picked uniformly from pre-defined intervals (Table 5.3). We consider that VM types are distributed equally among the sites. We generate three different component communication patterns: *distant*, *agglomerated*, and *uniform*. The difference between them is the probability of connecting two or more subtrees. The *distant* pattern has higher probability to connect subtrees near the root; *agglomerated* gives higher connection probabilities to subtrees near the leaves and the uniform schema gives the same connection probability $\left(\frac{1}{h_t}\right)$ to every subtree, where h_t is the height of the Cloud tree.

The four component-based application topologies we consider are *line*, *star*, *full connected* (cf. Figure 5.6), and *random*. In the random schema, a pair of components is connected with a probability of 50%. Communication requirements from component connections are pseudo-random integers picked uniformly between 0 and $h_t - 1$.

Renting prices depend on the resource dimensions of each VM. Let $c^*_{t,d}$ be the ratio $\frac{c_{t,d}}{max_d}$ between the capacity $c_{t,d}$ of dimension d from VM type t and max_d the maximum

Dimension	Requirements	Capacities
(i)	800 to 3000	1000 to 3500
(ii)	1 to 16	2 to 32
(iii)	1 to 32	2 to 40
(iv)	50 to 3500	150 to 4000
(v)	5 to 30	10 to 80
(vi)	1 to 8	1 to 16

Table 5.3: Intervals of dimension data generation.

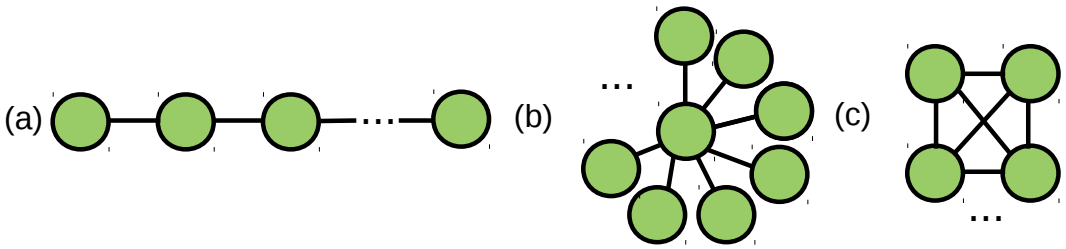


Figure 5.6: Schemes of part of the generated application topologies. (a) *line*, (b) *star* and (c) *full connected*.

value for dimension d (cf. Table 5.3). Each dimension is multiplied by a coefficient to create scenarios where some dimensions are more expensive than others. The values of those coefficients were chosen in such a way that α,β,γ,δ,ϵ,ζ would roughly reflect prices practiced by real Cloud providers. α,β,γ and δ could be translated as CPU performance, number of cores, memory and disk, respectively. ϵ and ζ are randomly chosen.

The price of a VM type p_t is $\alpha + \beta + \gamma + \delta + \epsilon + \zeta$, where $\alpha = c^*_{t,1} \times random(1,3)$, $\beta = c^*_{t,2} \times random(8,20)$, $\gamma = c^*_{t,3} \times random(5,8)$, $\delta = c^*_{t,4} \times random(10,15)$, if $c^*_{t,4} \leqslant 500$, otherwise $\delta = c^*_{t,4} \times random(20,25)$, $\epsilon = c^*_{t,5} \times random(5,10)$, and $\zeta = c^*_{t,6} \times random(2,5)$.

The 2PCAP algorithm is implemented in Python. Experiments were conducted on Dell PowerEdge R630 2.4GHz (2 CPUs, 8 cores) nodes from the *Parasilo* and *Paravance* clusters of the *Grid'5000* experimental platform[2].

5.7.2 2PCAP Performance on Small Problems

This section makes use of Class A problems instances (cf. Table 5.2) to calculate a set of optimal points and to compare them to solutions computed by 2PCAP.

We integrated to our test platform the *SCIP* solver [2], a framework for constraint integer programming and branch-cut-and-price (the formulation is described in Section 5.2.3). We conducted a Class A experiment using SCIP with a timeout of 24 hours. This means that SCIP had 24 hours to solve each problem instance from Class A.

SCIP solver was able to solve only around 48% of Class A problem instances in time, i.e., around 180 problem instances. Figure 5.7 illustrates the *cost distance* from solutions computed by 2PCAP to the optimal ones as a percentage of the latter for

[2]cf. `https://www.grid5000.fr`

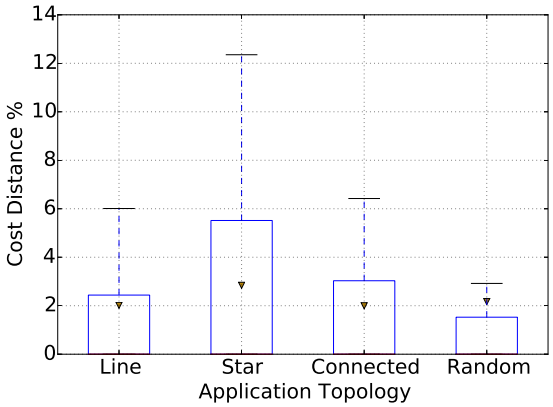


Figure 5.7: Cost distances as percentage of optimal between 2PCAP solutions and optimal solutions aggregated by application topology type. Cost distance averages are illustrated by brown triangles.

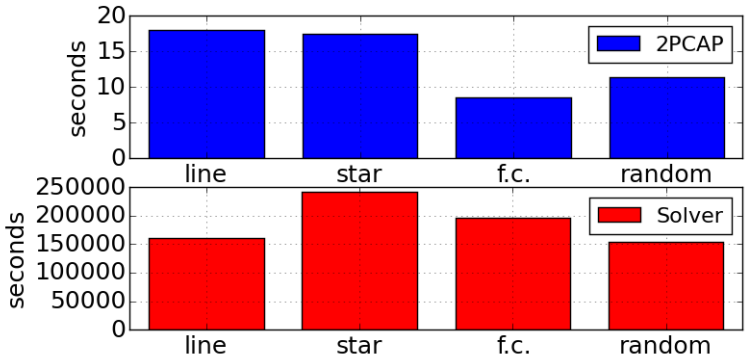


Figure 5.8: Sum of execution times in seconds from 2PCAP and SCIP solver executions aggregated by application topology. Only the execution time for problem instances successfully solved by SCIP are computed. Notice that 100000 seconds are equivalent to around 28 hours.

problem instances successfully solved by SCIP. Cost distances are grouped by application topology.

In Figure 5.7 we can see that cost distances vary between 0% and at most 12.3%. The median is always 0% and the average between 2% and 3%.

Figure 5.8 complements this data. It depicts the sum of the execution times in seconds that each approach used to calculate the 48% of Class A problem instances solved by SCIP, grouped by application topology schema. While 2PCAP takes some seconds to solve all problems, the solver's execution time is in the scale of days. Hence, in spite of being much faster than the solver, 2PCAP manages to produce a solution at most around 12% worse than the optimal and, solutions medians are optimal.

5.7.3 2PCAP Performance on Large Problems

It is impractical to generate optimal solutions for large scenarios. Hence, in this section, we evaluate 2PCAP by comparing it to more scalable baseline algorithms. We do this in three steps: first we compare 2PCAP to Simulated Annealing (S.A.) meta-heuristics, then we compare 2PCAP to heuristics running a relaxed version of the CAPDAMP, and finally we analyze the improvement of 2PCAP solutions by SA.

We implemented a S.A. meta-heuristic (cf. Section 2.4.2) using the Python module *Simanneal* [91] and computed placements for the medium sized Class B problem instances.

Despite giving a timeout of 1 hour per problem instance to SA, the meta-heuristic managed to solve only around 12% of all Class B problem instances. This happens mainly because of the size of CAPDAMP's search space that has a large amount of invalid solutions and many isolated local optimum solutions.

2PCAP vs. Relaxed CAPDAMP

In Section 5.3, we discussed some heuristics from the state of the art which used *relaxed* versions of NP-hard problems to quickly compute solutions. In a similar way, we used a relaxed version of CAPDAMP as *lower bounds* to evaluate 2PCAP. The relaxed version has been obtained by removing all communication constraints from the CAPDAMP, reducing it to a cost aware multi-dimensional bin packing problem. Then, we use a S.A. algorithm initialized with the best solution among those calculated by 2PCAP and other multi dimensional bin packing based heuristics, discussed in Chapter 4, with one hour timeout per problem instance.

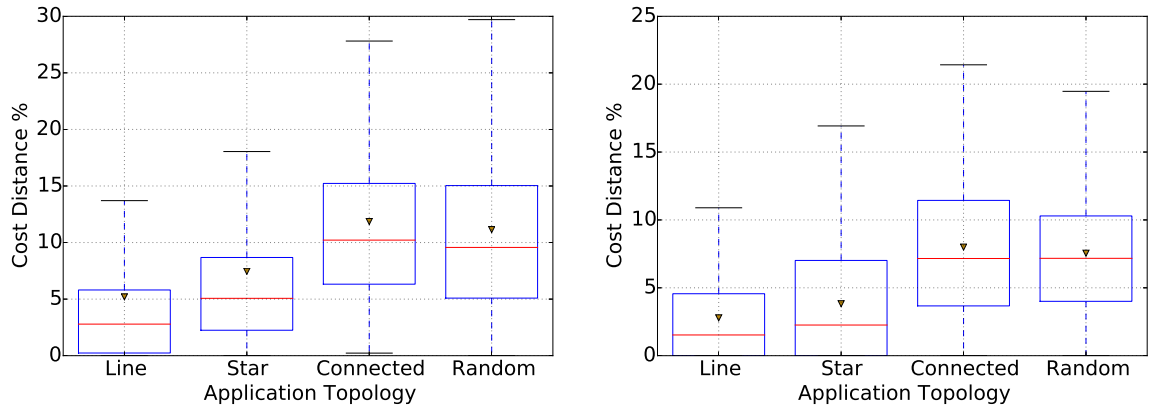


Figure 5.9: Cost distances between 2PCAP and relaxed CAPDAMP solutions for problem instances from classes B and C. Cost distance averages are illustrated by brown triangles.

Figure 5.9 illustrates the evolution of cost distances between 2PCAP and SA solutions. From left to right, we describe that metric for problem instances from classes B and C. We observe a similar pattern in every experiment, with cost distances varying between 0% and 35% but with the median always bellow 16.5%. The observed reducing cost distances between the experiences are partly due to the drop of the performance of S.A. as the size of the problem grows, which would require longer execution times to compute better solutions. Nevertheless, it is possible to notice that the cost distances – medians

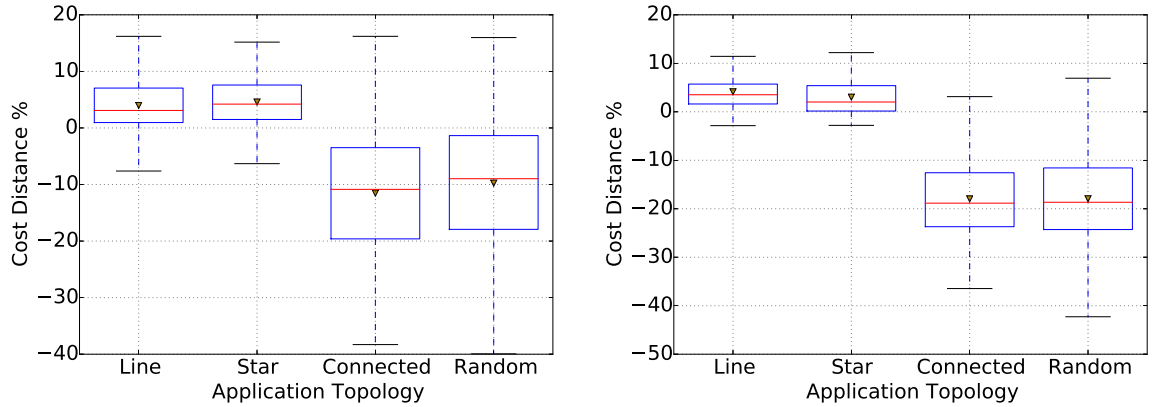


Figure 5.10: Cost distances between 2PCAP and S.A.1, a S.A. implementation that uses a modified version of 2PCAP to generate placement solutions. In summary, the composition phase of this algorithm was changed so it picks sub-placements randomly instead of the less expensive. Brown triangles indicate cost distance averages.

and averages, particularly – stay within similar intervals, indicating that even on large scenarios, the performance of 2PCAP is consistent.

Replacing S.A.'s Initialization Algorithm

As described in Section 2.4.2, S.A. builds a solution by improving *partial solutions* in a process that simulates the energy changes involved in cooling a material. In our implementation of S.A., there are two ways of generating a partial solution. It can be generated from a valid "neighbor solution", i.e. modifying a placement solution by moving a component from a VM to another VM, or it can be generated from scratch.

We observed that during our first tests, S.A. would spend most of its execution time trying to randomly generate valid partial solutions from the scratch due to the amount of invalid configurations in the search space.

Initializing S.A. With Random 2PCAP

To improve S.A. we implemented S.A.1, a modified S.A. that can generate partial solutions from the scratch using a modified version of 2PCAP which selects random sub-placements instead of the less expensive ones during composition phase. This allows for generating valid configurations very quickly. S.A.1 and 2PCAP were given one hour per problem instance to solve problems from Classes B and C. Thanks to 2PCAP, S.A.1 managed to generate solutions for 100% of valid problem instances. The results are depicted in Figure 5.10. We observe that in the worst case 2PCAP solutions are around 16% worse than S.A.1, and, in the best case, 2PCAP is around 40% better than S.A.1. Averages vary between around 3.5% and -11.5%, and medians between 4.2% and -11%. Notice that Class B and C experiments follow similar patterns.

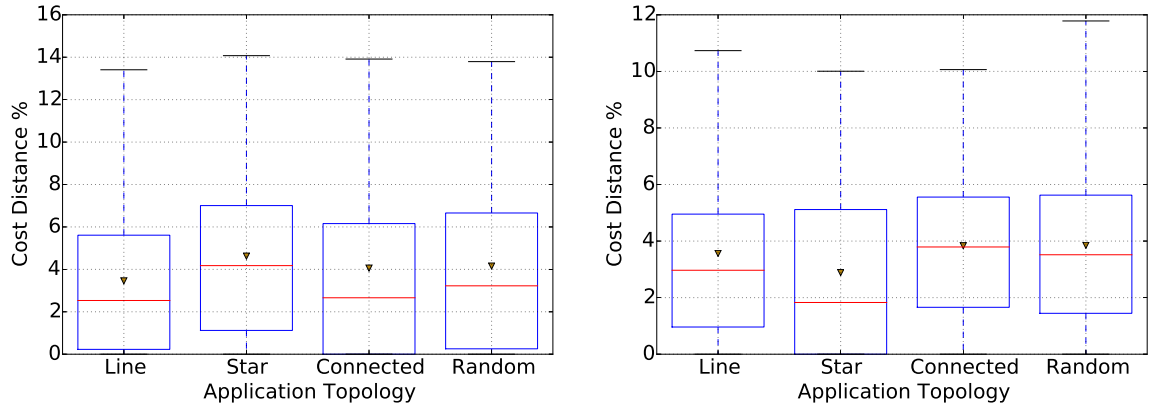


Figure 5.11: Cost distances between 2PCAP and S.A.2, a S.A. implementation that uses 2PCAP to calculate initial solutions. Brown triangles indicate cost distance averages.

Initializing S.A. With 2PCAP

In this section, we implemented S.A.2, a modified S.A. that uses 2PCAP to calculate its very first initial solution. Other partial solutions are generated randomly, if needed. This experience give us an insight about how much S.A. can improve a 2PCAP solution.

S.A.2 and 2PCAP were given one hour per problem instance to solve problems from Classes B and C. Figure 5.11 illustrates this metric.

S.A.2 managed to improve 2PCAP solutions by at most 14% and the median is always bellow 4.5%. This small improvement is a good indicator of 2PCAP's solution qualities.

2PCAP Execution Times

Figure 5.12 depicts execution times from 2PCAP, grouped by application topology, for Classes B and C. Despite having up to one hour to solve each problem, 2PCAP managed to compute solutions in at most around 140 seconds. Notice that there is a gap between execution times from problems with full connected and random application topologies and those with line and star topologies. This happens because of the high graph density of full connected and random topologies schemes. As a result, in the decomposition phase, 2PCAP generates less component subsets, leading to less bottom sub-placements and fewer calls to the bin packing heuristics.

5.8 Conclusion

In this chapter we presented an approach to calculate initial placements for component-based applications with the objective of minimizing costs while satisfying resource and communication constraints. This approach is based on a hierarchical model of the Cloud topology which allows the introduction of latency requirements despite the uncertainties inherent to Cloud networks, mainly due to virtualization. This model is used by 2PCAP, an efficient heuristic which, after an extensive evaluation, was shown to be capable of computing good quality solutions very quickly.

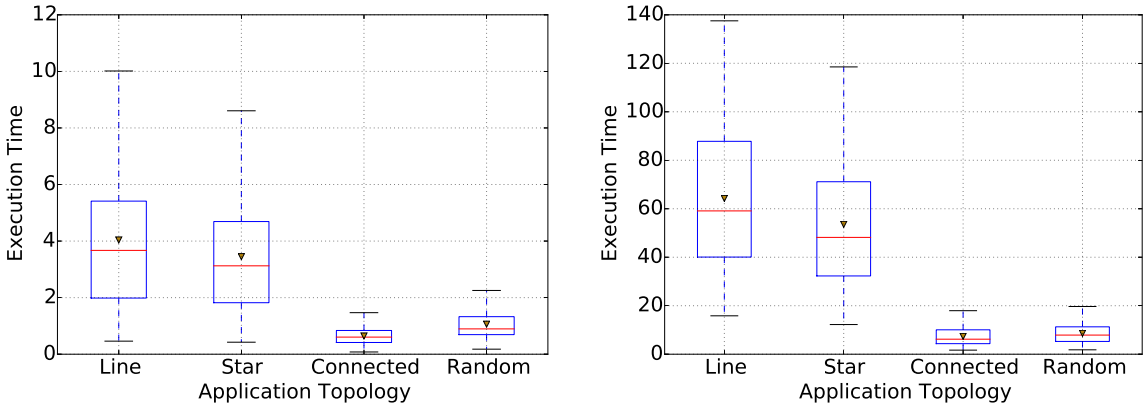


Figure 5.12: Execution times in seconds (per problem instance) aggregated by application topology taken by 2PCAP to compute solutions for problem instances from classes A, B and C. Execution time averages are illustrated by brown triangles.

It is important to notice, however, that 2PCAP is only capable of calculating *initial* placements. Reconfiguration placement scenarios, i.e. scenarios where parts of a distributed application have been previously deployed, cannot be solved by 2PCAP. This issue is addressed in Chapter 6, where we increment our models and propose a extended reconfiguration-aware version of 2PCAP.

Chapter 6

Communication and Cost-Aware Placement With Reconfiguration

In Chapter 5 we introduced the *communication-aware placement of distributed applications on multiple clouds problem* (CAPDAMP) and proposed the *two phase communication-aware placement heuristic* (2PCAP), a heuristic capable of calculating *initial* placements for the CAPDAMP. In this chapter, we extend our modeling of the CAPDAMP and propose an extension of 2PCAP which is able to calculate initial and *reconfiguration* placements.

6.1 Introduction

In Chapters 4 and 5, we proposed efficient heuristics able to calculate *initial* solutions for placement problems. Until now, we considered that there were no parts of a distributed application previously deployed at the moment the placement is computed.

In this chapter, we also consider *reconfiguration* placement scenarios. A reconfiguration scenario is characterized by the existence of previously deployed application components, and, consequently, instantiated VMs, at the moment a placement is computed. Hence, to achieve the objective of minimizing renting costs, it may be necessary to *migrate*, previously deployed application components to other VMs. However, moving application components involves costs and, to calculate them, it is necessary to take into account parameters such as connection latencies, amount of data being moved and expected application execution duration, for example.

In the next sections we present the main characteristics and issues related to reconfiguration placement problems and discuss solutions available in the state of the art. Then, we present a heuristic able to compute cost, communication and reconfiguration-aware placements called 2PCAP-REC and we close this chapter with its evaluation.

6.2 Problem Statement

In this section we present our modeling of distributed applications, Cloud based infrastructures, and reconfigurations. We formalize the problem of calculating cost, communication and reconfiguration-aware placements and describe our objectives.

6.2.1 Distributed Application and Cloud Computing Models

The Cloud and distributed application models used in this chapter are the same used in Chapter 5. Hence, in this section, we only summarize those models.

Distributed applications are modeled as component-based applications. Each component has resource requirements (e.g. CPU, memory, etc.) and may establish connection requirements in terms of latency with other components. Cloud infrastructures are modeled as interconnected sets of machine groups which encapsulate VMs. The latter are instanced from VM types and describe resource and connection capacities. The objective is to assign components to VMs aiming at minimizing costs while satisfying resource and communication constraints.

Let $\mathcal{I}$ be a set of components, $\mathcal{T}$ a set of VM types with D resources (or dimensions) of interest. Let $r_{i,d}$ be the requirements of component $i \in \mathcal{I}$ on dimension $d < D$, $c_{t,d}$ the capacity of VM type $t \in \mathcal{T}$ on dimension $d < D$ and p_t the price of renting a VM of type t per unit of time. Let $x_{i,j}^{comp}$ be the connection requirement between components $i \in \mathcal{I}$ and $j \in \mathcal{I}$.

Each VM type t is part of a *machine group* $s \in \mathcal{S}$, where $\mathcal{S}$ is a set of machine groups, a set of machines indistinguishable from the connection constraint point of view. This means that VMs belonging to the same machine group share the same *connection capacities*. Examples of machines group are clusters or *cloud provider sites*. Machine groups are interconnected and their connection capacities are represented as $x_{s,u}^{mg}$, where $s \in \mathcal{S}$ and $u \in \mathcal{S}$. The connection capacity between a VM $v \in \mathcal{V}$, that belongs to machine group s, and a VM $n \in \mathcal{V}$, that belongs to machine group u, is represented by $x_{v,n}^{vm}$ and, furthermore, $x_{v,n}^{vm} = x_{s,u}^{mg}$.

Let $\mathcal{V} = \{v_{k,t} \mid 1 \leqslant k \leqslant |\mathcal{I}|, t \in \mathcal{T}\}$ be the set containing all VMs that could be rented. Let $v \in \mathcal{V}$ and $c_{v,d}$ be the capacity of dimension $d \leqslant D$ of rented VM v, i.e., $c_{v,d} = c_{t,d}$ and p_v is the price paid for renting VM v, i.e., $p_v = p_t$.

6.2.2 Predicted Duration of Application Execution

We consider that an estimate δ of the duration of the execution of the distributed application is given in *units of time*. We also consider that each Cloud provider shares the same *billing time* interval of *one unit of time*. Thus, the cost of renting a VM $v \in \mathcal{V}$ will be $p_v.\delta$, i.e. its price p_v multiplied by the predicted duration δ.

6.2.3 Reconfiguration Specificities

In this chapter, besides *initial* placement scenarios, we also consider *reconfiguration scenarios*, that is, it is possible that a set of previously instantiated VMs, which host application components, may exist at the moment the placement is calculated. This impacts on *placement costs*. In our initial placement scenarios, the only source of costs comes from renting VMs. In reconfiguration scenarios renting costs exists too, however it is also necessary to take into consideration *component migration* costs, i.e., the costs of moving components from one VM to another. Those costs depend on many parameters such as type of deployed applications, migration technique, and the contract between customers and Cloud providers. For example, a deployed application unavailable to users due to a reconfiguration placement might result in extra fees for application owners. There may

also have extra costs for sending data, originating components or VM images, from a Cloud provider to another.

Let $\mathcal{W} \subseteq \mathcal{V}$ be a set of previously deployed VMs. Each VM $w \in \mathcal{W}$ hosts at least one application component. Let $\mathcal{H} \subseteq \mathcal{I}$ be the set of application components previously deployed on VMs.

Let $o_{j,w}$ be the mapping between a component $j \in \mathcal{H}$ to a VM $w \in \mathcal{W}$ on which it was previously deployed. A *reconfiguration* occurs when j is assigned to another VM $v \in \mathcal{V}$. Besides the cost $p_v.\delta$ of renting v for the next δ units of time, there is also the cost of moving i from w to v. The latter is described by a migration function $\mu(j, w, v)$.

Function μ, described in Equation 6.1, takes as arguments the component to be moved, the "source" VM and the "destination" VM.

$$\begin{aligned} \mu(j, w, v) &= \frac{\sum_{d \leqslant D} r_{i,d}}{x^{vm}_{w,v}}.\phi, \text{ for } j \in \mathcal{H}, w \in \mathcal{W}, v \in \mathcal{V} \\ \phi &= \frac{\sum_{i \in \mathcal{I}} \sum_{d \in D} r_{i,d}}{|\mathcal{I}|}.\frac{1}{min(p_v), \forall v \in \mathcal{V}} \end{aligned} \tag{6.1}$$

ϕ is a migration cost coefficient. In this work the value of ϕ is the average of the sum of all components dimensions divided by the price of the cheapest VM type available among all Cloud providers.

In this thesis, the value calculated by μ depends on the "size" of a component and the communication qualities between previous and current host VMs. Hence, we penalize large components and low latency connections, i.e. the cost of a migration is proportional to the ratio size and connection quality. We simplify the concept of size by considering it to be the sum of components' dimensions. Notice that μ could be easily replaced by other functions to model more specific placement scenarios.

We do not take into consideration explicit costs related to the interruption of service during migrations, however, this could be implicitly incorporated to ϕ. For example, in the case where each component has different interruption costs and they depend only on component's characteristics, μ could be extended to μ_i where $i \in \mathcal{I}$.

6.2.4 Placement Objective

Given the sets $\mathcal{I}$ of application components, $\mathcal{V}$ of VMs, $\mathcal{W}$ of rented VMs and $\mathcal{S}$ of machine groups, we want to compute a placement of all elements from $\mathcal{I}$ on VMs from $\mathcal{V}$ and $\mathcal{W}$. The objective is to minimize renting and migration costs while satisfying resource and communication constraints defined by the application. We illustrate the reconfiguration placement problem in Figure 6.1.

6.2.5 Optimization Problem Formulation

We formalize the placement problem described in the previous sections in Equation 6.2. A summary of the variables used in the problem is available in Table 6.1. This optimization problem formulation will be used as input to a MIP (Mixed Integer Programming) solver in Section 6.5.

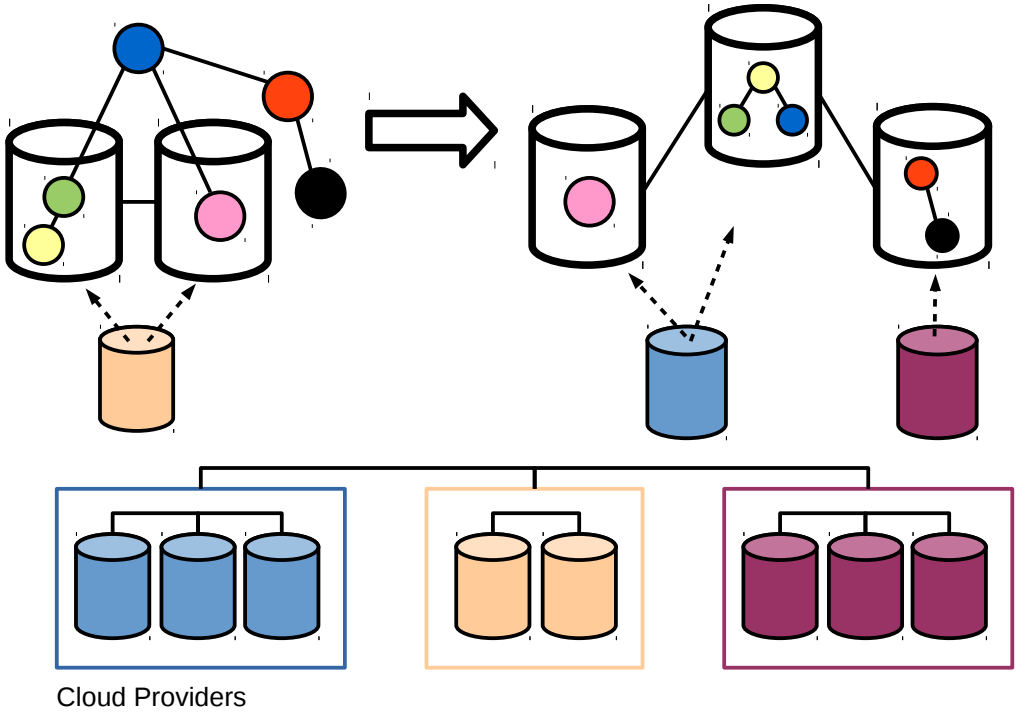


Figure 6.1: Reconfiguration placement scenario. At the bottom, we represent Cloud providers as rectangles and VM types as colored cylinders. VM instances are represented as transparent cylinders and dashed arrows indicate the types from which VMs are instanced. On the upper-left part, we illustrate a distributed application as a connected graph where nodes are application components and edges are connections between components. We illustrate the situation where components of the distributed application are already deployed (i.e. components inside VMs on the left part) and others are waiting to be deployed (i.e. components outside VMs). The objective, represented on the right, is to find a placement which minimizes costs originated from renting VMs and migrating components while satisfying resource and communication constraints.

$\mathcal{I}$	Set of components.
$\mathcal{T}$	Set of VM types.
$\mathcal{V}$	Set containing all potential VMs.
$\mathcal{W}$	Set of previously deployed VMs ($\mathcal{W} \subseteq \mathcal{V}$).
$\mathcal{H}$	Set of previously hosted components ($\mathcal{H} \subseteq \mathcal{I}$).
D	Number of resources/dimensions.
p_t	Price of VM type $t \in \mathcal{T}$.
p_v	Price of VM $v \in \mathcal{V}$.
a_v	$a_v = 1$ if $v \in \mathcal{V}$ is being used, 0 otherwise.
$v_{k,t}$	k-th VM of type $t \in \mathcal{T}$.
$o_{i,v}$	$o_{i,v} = 1$ if component $i \in \mathcal{I}$ was previously hosted on $v \in \mathcal{V}$, 0 otherwise.
$m_{i,v}$	$m_{i,v} = 1$ if component $i \in \mathcal{I}$ is assigned to VM $v \in \mathcal{V}$, 0 otherwise.
$r_{i,d}$	Requirement of component $i \in \mathcal{I}$ on dimension d.
$c_{v,d}$	Capacity of VM $v \in \mathcal{V}$ on dimension d.
$x^{comp}_{i,j}$	Requirement of connection between component $i \in \mathcal{I}$ and $j \in \mathcal{I}$.
$x^{mg}_{s,u}$	Capacity of connection between machine group $s \in \mathcal{S}$ and $u \in \mathcal{S}$.
$x^{vm}_{v,n}$	Capacity of connection between VM $v \in \mathcal{V}$ and $n \in \mathcal{V}$.

Table 6.1: Summary of variables used in this section and, more specifically, in Equation 6.2.

$$
\begin{aligned}
&\text{Minimize } \sum_{v \in \mathcal{V}} q_v + \sum_{i \in \mathcal{I}} g_i \\
&\text{s.t.} \\
&\quad \sum_{v \in \mathcal{V}} m_{i,v} = 1 && \forall i \in \mathcal{I} && (i) \\
&\quad \sum_{i \in \mathcal{I}} m_{i,v}.r_{i,d} \leqslant c_{v,d} && \forall v \in \mathcal{V}, 1 \leqslant d \leqslant D && (ii) \\
&\quad m_{i,n}.m_{j,v}.(x^{vm}_{n,v} - x^{comp}_{i,j}) \geqslant 0 && \forall n, v \in \mathcal{V}, \forall i \in \mathcal{I} && (iii) \\
&\quad g_i = \mu(i,n,v).m_{i,v}.o_{i,n} && \forall n,v \in \mathcal{V} | n \neq v, \forall i \in \mathcal{H} && (iv) \\
&\quad q_v = \begin{cases} \delta.p_v & \text{if } \sum_{i \in \mathcal{I}} m_{i,v} > 0 \\ 0 & \text{otherwise} \end{cases} && \forall v \in \mathcal{V} && (v) \\
&\quad m_{i,n}, o_{i,v} \in \{0,1\} && && (vi)
\end{aligned}
\tag{6.2}
$$

In the above equation, (i) guarantees that each component is assigned to at most one VM; (ii) ensures that no instantiated VM has more components than it can host; (iii) guarantees that network requirements are satisfied, (iv) sets migration costs for components that were migrated and (v) calculates renting costs of VMs for duration δ.

6.3 Related Work

Since we already presented an extensive literature concerning communication-aware placement in Chapter 5, in this section, we discuss related work specific to reconfiguration of

distributed applications.

In this thesis, our tool of choice for reconfiguring applications is *component migrations*. We migrate components instead of VMs because of the *multiple Clouds scenario* we are considering. As we discussed in Section 2.2.4, it is rare that different cloud providers share the same hypervisors or VM images format, causing what is called *vendor lock-in* [92]. We also concentrate on *live migrations*. They consist of transferring VM or components from a host machine to another one causing *very short to no interruption of service*.

In Section 5.3, we discussed many state of the art communication-aware approaches for the CAPDAMP. In this section he focus on the validation of our choice of migrating components instead of VMs (cf. Section 6.3.1) and the proposed migration model and cost function μ (cf. Section 6.3.2).

6.3.1 Migrating Components

There are many ways of encapsulating components across a distributed infrastructure. With the recent growth of popularity of *containers* (cf. Section 2.2.4), including Docker [29], OpenVZ [86] and LXC [67] among others, we have been observing a very fertile environment for work on component migration. Those containers, which can be hosted in VMs, can be used to run components and are capable of live migration similarly to VMs [73].

As examples of live migrations in the context of containers, we can cite two similar container live migration procedures based on logs [85, 115] to register actions performed during a migration. In summary, a duplicate of the container image to be moved is transfered to a new host and actions performed in the source container during the transfer are logged. Once the transfer is completed, the duplicate perform the action on the new host and the source container can be turned off. Similarly, in [81], authors propose a technique based on checkpointing for storing actions performed while transferring a container image.

It is not in our scope to go deeper into the details of live migration of containers, however, as established container implementations such as OpenVZ [86] start to *natively* allow for live migrations, we observe an indication that our choice of component migration is not just a theoretical option.

6.3.2 Modeling Migration Costs

Live migration costs are usually related to duration of service interruption [110], which is proportional to active memory on the source VM or container, available CPU at the source/destination host machine, co-location interference and dependent on the specific piece of software being run [110]. However, other parameters, such as software licenses, server maintenance, human resources, etc. may also have an influence on costs [99]. In this section we go through the state of the art on VMs and component live migration models and position our work in respect to them.

We call migration model the set of parameters of a placement problem taken in consideration to compute migration costs.

The migration model used in this chapter (cf. Section 6.2) takes into consideration components being migrated, source VM, destination VM, and resource and communication constraints. Furthermore, the proposed function μ (cf. Section 6.1) that compute

migration costs penalizes the transfer of "large" components over bad latency connections.

In [93], an exact algorithm based approach to calculate initial and reconfiguration placements with the objective of minimizing data transfer among VMs is proposed. The amount of data transfered between source and destination physical machine host is the migration cost. A very similar approach which aims at minimizing energy consumption of host machines instead of data is described in [31]. In that case, migrations costs are the difference between the energy consumption of source and destination physical machines. Another approach based on exact algorithms is proposed in [99]. In its very expressive VM migration model, the cost of migrating a VM is the predicted duration of the application execution and additional costs related to the maintenance during the reconfiguration (software licenses, human resources, etc.) which are previously defined by a manager. Authors in [97] let the application designers manually define migration costs for each VM in their approach. Finally, [117] presents a genetic meta-heuristic which is capable of calculating reconfiguration placements with the objective of minimizing renting costs of VMs while satisfying resource constraints. The migration cost is proportional to the memory and storage used by the application host.

We observe that the migration model proposed in Section 6.2 shares most of the characteristics discussed in the literature. The migration cost function we propose is dependent on the amount of resources used by the source hosting machine and on the communication between source and destination hosting machines. Besides of that, our model is flexible enough to be incremented with new parameters, such as host machine specific costs, and the migration function can be easily replaced.

Furthermore, our approach is cost and communication-aware and is based on a fast heuristic. Most of the work discussed in this section is based on exact algorithms or meta-heuristics. Exact algorithms are not scalable and meta-heuristics have their solution qualities proportional to the amount of time given to them to calculate. Hence, if there are narrow time constraints or, if problems are very large, the heuristic we propose may be an specially interesting option.

6.3.3 Discussion

In this section, our objective was to validate our decision of migrating components instead of VMs and our migration model. We presented interesting work which indicated that application components can be run inside of containers and those can be live-migrated in reconfiguration situations. We also compared the migration model proposed in this chapter with other migration models from the literature and observed that our model comprises many of the interesting features and that it could be extended if a more expressive migration model is needed.

6.4 The 2PCAP-REC Heuristic

In this chapter we propose the Two Phase Communication and Reconfiguration Aware Placement Heuristic (2PCAP-REC), an extension of the divide and conquer heuristic 2PCAP presented in Chapter 5.

2PCAP-REC computes not only *initial* solutions for the CAPDAMP, but it can also

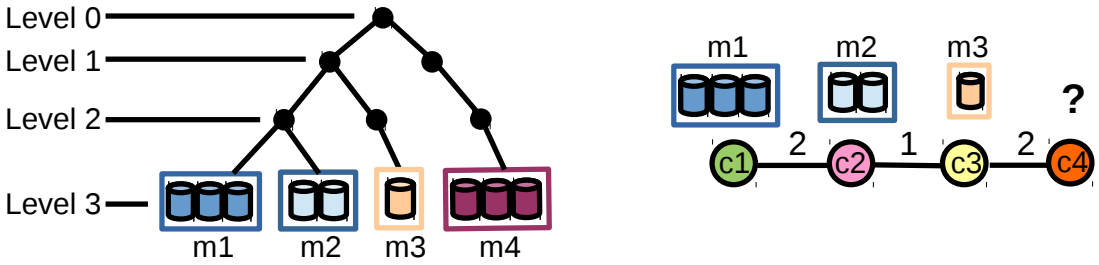


Figure 6.2: Addition of component $c4$. In the left part, we illustrate the Cloud topology and, in the right part, the distributed application modeled as a component-based application. Previously deployed components have the machine groups of their current hosting VMs identified above them ($c1$, $c2$, and $c3$). Non-deployed components have question marks "?" ($c4$).

compute solutions for *reconfiguration scenarios*, i.e. scenarios where part of the distributed application is deployed on VMs at the moment the placement is calculated.

Before presenting this heuristic in details, we need to discuss the application, Cloud and migration models used by 2PCAP-REC.

6.4.1 Application and Cloud Model

As 2PCAP-REC is an extension of 2PCAP, it shares the same application and Cloud models. Consequently, in this section, we briefly recall these models and invite the reader interested in more detail to refer to Sections 5.4 and 5.5.

Cloud Model

We model the Cloud topology as a hierarchy. In this topology (cf. Figure 6.2), *leaves* are *machine groups* and *inner nodes* represent *connections* between machine groups. The level of a inner node represents the *connection quality* of the machine groups having that inner node as sub-tree root. The highest internal node which interconnects two machine groups define their maximum connection quality.

The objective of representing the Cloud as a tree is to be able to describe *latencies* between machine groups even in situations where only uncertain or less precise information is available.

Distributed Application Model

Distributed applications are modeled using the *component-based paradigm* and are represented as a graph. Components are nodes and connection between components are weighted edges. Connection requirements are described by edge weights. Figure 6.2 illustrates an example of such distributed application.

6.4.2 Reconfiguration Model

To be able to describe reconfiguration scenarios, it is necessary to represent components previously deployed on the Cloud. We added this feature to the application model and also included migration cost functions μ and prevision of execution duration δ to the input of 2PCAP-REC. A scenario where a new component is added to an application is illustrated in Figure 6.2.

6.4.3 2PCAP-REC – Two Phase Communication and Reconfiguration Aware Placement Heuristic

The Two Phase Communication and Reconfiguration Aware Placement Heuristic (2PCAP-REC) is an extension of the Two Phase Communication Aware Placement Heuristic (2PCAP), described in details in Chapter 5. It also has two phases: a *decomposition* and *composition* phases.

The difference between 2PCAP-REC and 2PCAP lies in the decomposition step. 2PCAP-REC must take into consideration possible migrations of components when calculating *bottom sub-placements*. The rest of the algorithm is essentially the same. Thus, in this section, we summarize the similar parts and focus on the new ones. For the reader interested in a more extensive discussion on decomposition and composition phases, please refer to Section 5.5.

Decomposition

In the decomposition phase, 2PCAP-REC, similarly to 2PCAP, generates, at each level of the Cloud topology, component and machine group *subsets* that will serve as input to *sub-placements*. The decomposition is propagated until the *bottom sub-placement* level ℓ_{max} is found.

The application graph and the Cloud topology form the initial component and machine group subsets, respectively.

Component subsets are decomposed through the removal of edges weighting less than a given connection quality, or *level*, ℓ, from the graph generated by the elements of a component subset. The *nodes* of each resulting connected sub-graph is *a valid component subset*. An example of a component subset decomposition is illustrated in Figure 6.3(e).

Machine groups subsets are decomposed through the gathering of all inner nodes from the Cloud topology tree at a given connection quality, or level, ℓ. These nodes form a forest of subtrees and each set of leaves from each subtree is a valid machine group subset. Examples of machine group subset decomposition are illustrated in Figures 6.3(c), 6.3(d), and 6.3(e).

A *sub-placement* is the placement of a subset of components on a subset of machine groups. Sub-placements are similar to a "complete" placement since they also have the objective of minimizing costs while satisfying resource and communication constraints.

A *bottom sub-placement* is a sub-placement which can be calculated by communication *oblivious* heuristics while still *satisfying communication requirements*. Bottom sub-placements are generated at level ℓ_{max}, which is the highest communication quality requirement present in the application. This situation is illustrated in Figure 6.3(e). Ob-

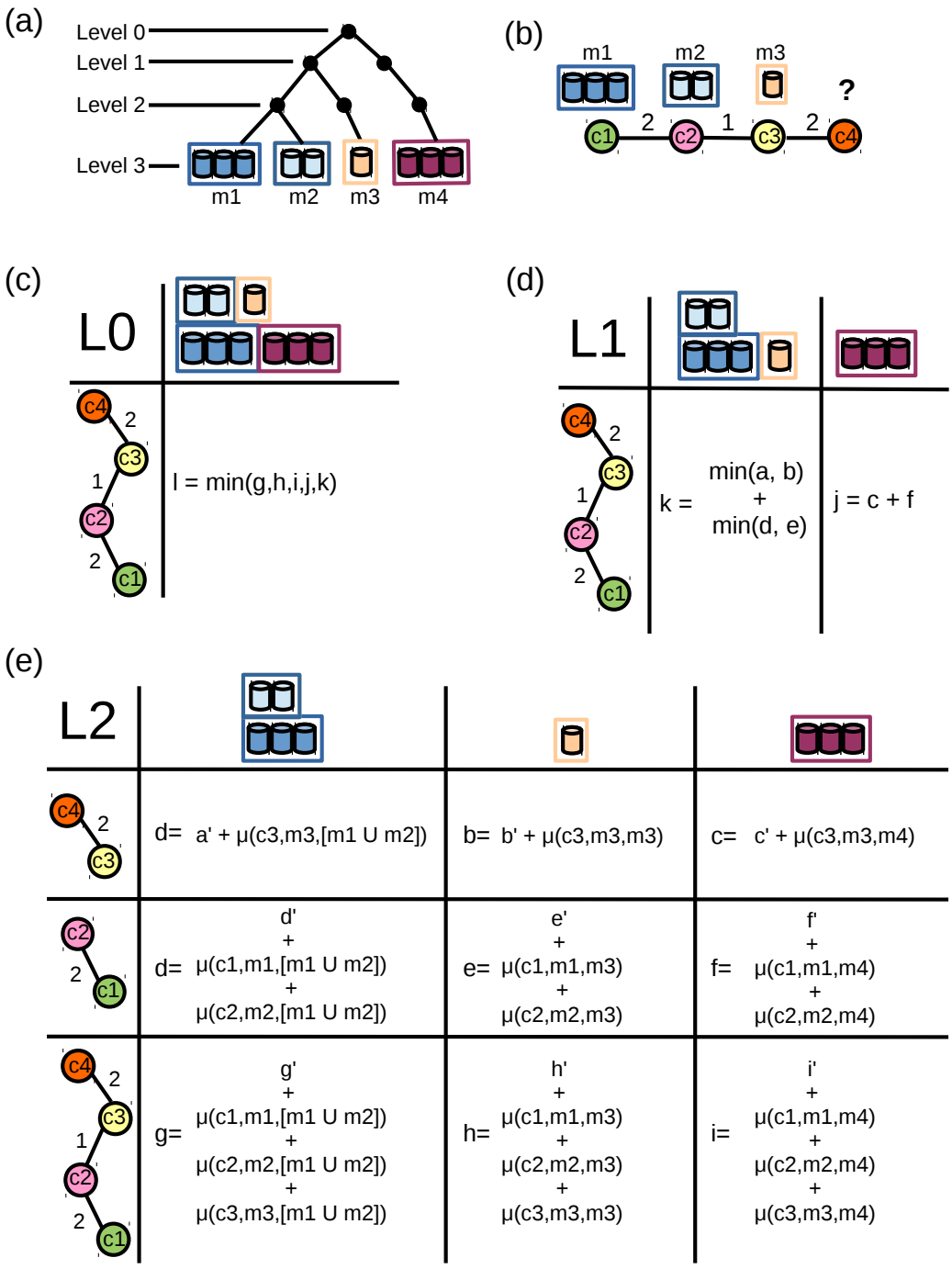


Figure 6.3: Reconfiguration placement example. We simulated the addition of component $c4$. In (a), we illustrate the Cloud topology and in (b) the distributed application modeled as a component-based application. Previously deployed components have the machine groups of their current hosting VMs identified above them ($c1$, $c2$, and $c3$). Non-deployed components have question marks "?" ($c4$). Tables (c), (d) and (e) illustrate the decomposition steps. Subtrees of machine groups are illustrated on their top and subgraphs on their left side (cf. Section 5.5). As we do not detail host VMs in this example, we slightly modified the input of μ to $\mu(i,s,u)|i \in \mathcal{H}$ and $s,u \in \mathcal{S}$. a', b',c',d',e' and f' are the costs of placing the respective component subsets on machine group subsets.

serve that it is not necessary to continue generating sub-placements up to level 3 since there are no communication requirements above 2.

Computing Bottom Sub-Placements

The last step of decomposition phase consists of calculating bottom sub-placements. The way it is done by 2PCAP-REC is different from 2PCAP because the former must take into account potential *component migrations*.

First, 2PCAP-REC calculates bottom sub-placements using communication oblivious heuristics similarly to 2PCAP. Then, it checks if there are any previously deployed components that were assigned to VMs *different* from their source VMs. If that is the case, the cost of migrating those component is calculated by a migration function $\mu(i,n,v) = \frac{\sum_{d \leqslant D} r_{i,d}}{x^{vm}_{n,v}}.\phi$ (cf. Section 6.2). Finally, the cost of each bottom sub-placement will be the cost of renting the VMs added to the cost of eventually migrating application components. Figure 6.3(e) illustrates the computation of bottom sub-placements a,b,c,d,e and f.

The packing improvement (Section 5.5.1) is also applied to 2PCAP-REC. It consists of creating bottom sub-placements of intermediary component subsets (i.e. generated at levels smaller than ℓ^{max}) on machine groups subsets of level ℓ^{max}. This is illustrated in Figure 6.3(e). Observe that, in the last line of table L2, the component subset $\{c1,c2,c3,c4\}$ was generated at level 0 (cf. Figure 6.3(c)).

Composition

The composition phase of 2PCAP-REC is identical to 2PCAP's (cf. Section 5.5.2).

After calculating all bottom sub-placements, the objective is to choose, at each step between levels ℓ_{max} and 0, the less expensive solutions from related sub-placements and compose them up to the final placement. This process is illustrated in Figure 6.3(d), whose sub-placements are chosen based on sub-placements calculated at level 2 (cf. Figure 6.3(e)) and in Figure 6.3(c) whose sub-placements are based on those calculated at level 1 (cf. Figure 6.3(d)).

6.4.4 2PCAP-REC Algorithm

2PCAP and 2PCAP-REC algorithms are mostly the same. As 2PCAP-REC supports component migrations, it also receives a duration δ and a migration function μ as input. Those two parameters are used to calculate potential migration costs stemming from bottom sub-placements. The function responsible for this task is the function **calculate** which is called in Line 5 from Algorithm 9 and illustrated in Algorithm 8.

In function **calculate** (cf. Algorithm 8), after storing a copy of a potential previous placement in variable *prev_plac* (cf. Line 1), a sub-placement is calculated using communication oblivious heuristics and the result stored in variable *sub_placement* (cf. Line 2), which is, then, compared to *prev_plac* in order to identify any component migrations (cf. Lines 4 to 9). If this is the case, the migration cost (cf. Section 7) is added to the cost of the sub-placement (cf. Line 11) and the computed sub-placement is returned.

Algorithm 8 calculate

Input: $comp_subset, mg_subset, \delta, \mu$
Output: $sub_placement$

```
 1: prev_plac ← get_prev_plac(comp_subset)
 2: sub_placement ← com_oblivious_heuristics(comp_subset, mg_subset)
 3: mig_cost ← 0
 4: for (comp_1, vm_1) in sub_placement do
 5:     for (comp_2, vm_2) in prev_plac do
 6:         if comp_1 == comp_2 AND vm1 ≠ vm2 then
 7:             mig_cost ← mig_cost + μ(comp_1, vm_1, vm_2)
 8:         end if
 9:     end for
10: end for
11: set_plac_cost(sub_placement, cost(sub_placement) + mig_cost)
12: return sub_placement
```

6.4.5 2PCAP-REC Complexity

The complexity of 2PCAP-REC is mostly the same as 2PCAP (cf. Section 5.5.4). Consequently, it is also dominated by operations of composition (**compose** function), decomposition (**decompose** function) and the computation of sub-placements (**calculate** function). 2PCAP's and 2PCAP-REC's functions **compose** and **decompose** are essentially the same. However, 2PCAP-REC's function **calculate** has the overhead of verifying if component migrations took place.

At level ℓ_{max} (bottom sub-placement level), there are two types of component subsets: those generated by decomposition at level ℓ_{max} and those which were decomposed at intermediary levels – a strategy to generate better packed solutions (Algorithm snippet between Lines 30 and 36).

There is an added complexity of verifying if component migrations were performed during bottom sub-placements computing. In the worst case, all components of the application would be previously deployed and there will be as many machine group subsets as machine groups. Thus, in this case, the complexity would be $O(|\mathcal{I}|^2.|\mathcal{S}|)$.

When it comes to calculating the added complexity related to component subsets decomposed at intermediate levels, it is necessary to consider that, in the worst case all connection requirements would be ℓ_{max} and, thus, every intermediate decomposition would result in the original application. Hence, the added complexity would be $O(|\mathcal{I}|^2.|\mathcal{S}|.(\ell_{max} - 1))$.

Thus, the total complexity overhead of calculating eventual component migrations is $O(|\mathcal{I}|^2.|\mathcal{S}|.\ell_{max})$ and the total complexity of function *calculate* will be $O(|\mathcal{I}|^2.|\mathcal{S}|.|\mathcal{T}|.\log|\mathcal{T}|)+$ $O(|\mathcal{I}|^2.|\mathcal{S}|.\ell_{max})$. As $|\mathcal{T}| \geqslant |\mathcal{S}|$ and it is expected that $\ell_{max} < |\mathcal{T}|$, the complexity of *calculate* will still be $O(|\mathcal{I}|^2.|\mathcal{S}|.|\mathcal{T}|.\log|\mathcal{T}|)$.

Algorithm 9 2PCAP-REC (Differences to Algorithm 7 are highlighted)

Input: $level, comp_subset, mg_subset, \delta, \mu$
Output: min_cost_plac

```
1: min_cost_plac ← ∞
2: if is_calculated(comp_subset, mg_subset) then
3:     return plac(comp_subset, mg_subset)
4: else if level = level_max then
5:     calculate(comp_subset, mg_subset, δ, μ)
6:     return plac(comp_subset, mg_subset)
7: else if level < level_max then
8:     comp_decomposition ← decompose(comp_subset, level)
9:     mg_decomposition ← decompose(mg_subset, level)
10:    if size(mg_decomposition) = 1 then
11:        plac ← null
12:        for cs in comp_decomposition do
13:            temp_plac ← 2PCAP-REC(level + 1, cs, mg_subset, δ, μ)
14:            plac ← compose(plac + temp_plac)
15:        end for
16:        min_cost_plac ← plac
17:    else if size(mg_decomposition) > 1 then
18:        plac ← null
19:        for cs in comp_decomposition do
20:            min_plac ← null
21:            for ms in mg_decomposition do
22:                temp_plac ← 2PCAP-REC(level + 1, cs, ms, δ, μ)
23:                if cost(temp_plac) < min_plac then
24:                    min_plac ← temp_plac
25:                end if
26:            end for
27:            plac ← compose(plac + min_plac)
28:        end for
29:        min_cost_plac ← plac
30:    end if
31:    for ms in mg_decomposition do
32:        temp_plac ← 2PCAP-REC(level_max, comp_subset, ms)
33:        if cost(temp_plac) < cost(min_cost_plac) then
34:            min_cost_plac ← plac
35:        end if
36:    end for
37: end if
38: return min_cost_plac
```

6.5 Evaluation

In this section we evaluate the performance of 2PCAP-REC. We do this through simulation and by comparing the proposed heuristic to other baseline algorithms based on state of the art approaches.

6.5.1 Methodology

In Chapter 3 we have discussed in detail the methodology, notation, and metrics used for evaluating our contributions. In this section, we summarize the concepts previously presented and focus on particularities of proposed heuristics' evaluation, such as experiment format, problem classes parameters or test platform characteristics.

As discussed in Section 3.3, the evaluation of 2PCAP-REC is based on experiments. An *experiment* is the resolution of a set of placement *problems* by a set of *algorithms* within a given *time*. In this chapter, we perform three experiments. Each one will have a distinct set of problems which we call *problem class*.

A *problem class* assembles problems that share similar characteristics allowing us to analyze different aspects of the evaluated algorithm.

Parameter	**Class A**	**Class B**	**Class C**
number of dimensions	2,4	4	4
initial number of components	2,3,4,5	10,20,30,40	50,75,100
initial number of VMs	100	1000	1000,5000,10000
number of sites	5	50	100
multi-cloud topology height	5	5	5
application topology	line, random	line, random	line, random
predicted duration	2, 5, 8	5, 10, 500	2, 5, 10, 500
migration cost function	f1	f1, f2	f1, f2
added/removed components	-1, 1, 2	-50%,-25%, +50%, +100%,	-50%, +50%
added/removed VMs	0	0, +50%	0, +50%
total problems	192	384	576

Table 6.2: Problem classes and their parameters.

We propose three classes, A, B, and C (cf. Table 6.2). Fundamentally, smaller and consequently easier to solve Class A problems are used to compare 2PCAP-REC to a MIP solver and the medium to large problems from Classes B and C are used to compare 2PCAP-REC to a Simulated Annealing (S.A.) meta-heuristic implementation. The metrics used to analyze the results of experiments are *cost distances* and the difference between *execution times* (cf. Section 3.3).

Problem classes' parameters are mostly the same used in Section 5.7 except for the last four parameters in Table 6.2: predicted duration, migration cost function, added/removed components and added VM types. Predicted duration refers to an estimate of the application execution duration in units of time; migration cost function refers to the function used to generate migration costs; added/removed components refers to the number of components added or removed in the reconfiguration scenario and added VM types refers

to the number of VM types added to the reconfiguration scenario. The two migration functions f_1 and f_2 use the same base migration cost function μ (cf. Equation 6.1), however f_1 considers that component migrations are free if they involve VMs from the same machine group and f_2 does not. We do this to simulate cost models practiced by many Cloud providers such as Amazon [4] and Azure [78] which do not charge data transfers inside the same Cloud provider site.

Essentially, a placement problem is composed of a set of possibly interconnected components and a set of VM types organized in interconnected Cloud sites. The parameters that guide the generation of those problems are defined by each problem class. The procedure of placement problem parameter generation is the same used in Section 5.7.

Component and VM type dimensions are generated by picking pseudo-random values from intervals pre-defined in Table 6.3. The price p_t of a VM type is dependent on its first four capacities and is generated as follows:

$$
\begin{aligned}
p_t &= \alpha + \beta + \gamma + \delta \text{ where} \\
\alpha &= c^*_{t,1} \times random(1,3) \\
\beta &= c^*_{t,2} \times random(8,20) \\
\gamma &= c^*_{t,3} \times random(5,8) \\
\delta &= \begin{cases} c^*_{t,4} \times random(10,15) & \text{if } c^*_{t,4} \leqslant 500 \\ c^*_{t,4} \times random(20,25) & \text{, otherwise} \end{cases} \\
c^*_{t,d} &= \frac{c_{t,d}}{max_d} \text{and } max_d \text{ is the upper bound of } d\text{'s data generation interval.}
\end{aligned}
\tag{6.3}
$$

The simulation of a reconfiguration situation has three steps. First the necessary data is generated following the experiment's problem class parameters and intervals of dimension data. At this step an initial placement problem is generated and 2PCAP computes a solution to that problem. Then, a certain number of components or VMs is added to that placement or removed from it (this amount is also defined by the problem classes). This constitutes the reconfiguration placement problem and it will be calculated by 2PCAP-REC and baseline algorithms.

Dimension	**Requirements**	**Capacities**
(i)	800 to 3000	1000 to 3500
(ii)	1 to 16	2 to 32
(iii)	1 to 32	2 to 40
(iv)	50 to 3500	150 to 4000
(v)	5 to 30	10 to 80
(vi)	1 to 8	1 to 16

Table 6.3: Intervals of dimension data generation. .

The 2PCAP-REC algorithm and the program necessary to automate its evaluation is implemented in Python. We used the SCIP suite [2] in conjunction with IBM's CPLEX [56] to generate optimal solutions and Python's simulated annealing framework Simaneal [91]. Experiments were conducted on Dell PowerEdge R430 (Intel(R) Xeon(R)

CPU E5-2620 v4 2.10GHz, 32GB RAM) nodes from cluster *Nova* of the *Grid'5000* experimental platform[1].

6.5.2 Small Problems (Class A Experiment)

Class A problems were designed to be used as input to a MIP solver which would allow for the generation of optimal points that would be compared to 2PCAP-REC solutions. After generating the problems, we used the SCIP optimization suite [2] in conjunction with the MIP solver IBM CPLEX [56] to solve them. The optimization problem formulation is described in Equation 6.2 and was programmed using ZIMPL [2], a high level language which translates a mathematical model to a format that CPLEX understands. Each individual problem had 12 hours to calculate a solution. During this time, SCIP and CPLEX were able to generate around 6% of Class A problems (around 13 problem instances).

This is an indicator of the difficulty of the problem and forced us to find other strategies to evaluate the performance of 2PCAP-REC.

6.5.3 Medium and Larger Problems (Class B and C Experiments)

In this section, we evaluate the performance of 2PCAP-REC in the larger scenarios described by Class B and C problems. To do that, we use as baseline algorithm an implementation of a simulated annealing meta-heuristic (S.A.), which is more scalable than the solver used in Section 6.5.2.

We discuss results related to two experiments, one using Class B problems (Class B Experiment) and another using Class C problems (Class C Experiment) as input. Both have a *one hour timeout* and use *three* S.A. variants as baseline algorithm.

Essentially, S.A. variants differ with regard to the way their *partial solutions* (cf. Section 2.4.2) are generated. In summary, the first variant, S.A.1 generates partial solutions pseudo-randomly; the second one, called S.A.2, generates solutions calculated by a modified 2PCAP-REC which, in the composition step, instead of choosing the less expensive sub-placements, choses them randomly; and the third one, named S.A.3, uses a solution computed by 2PCAP-REC.

2PCAP-REC vs. S.A.1

As described in Section 2.4.2, S.A. builds a solution by improving partial solutions in a process that simulates the energy changes involved in cooling a material. In our implementation of S.A., there are two ways of generating a partial solution. It can be generated from a valid "neighbor solution", which consists of moving a component from VM in a valid placement to another VM, or it can be generated from scratch, i.e. an entire new placement is generated randomly.

We observed that during our first tests, S.A. would spend most of its execution time trying to randomly generate valid partial solutions from scratch due to the amount of invalid configurations in the search space.

[1] cf. `https://www.grid5000.fr`

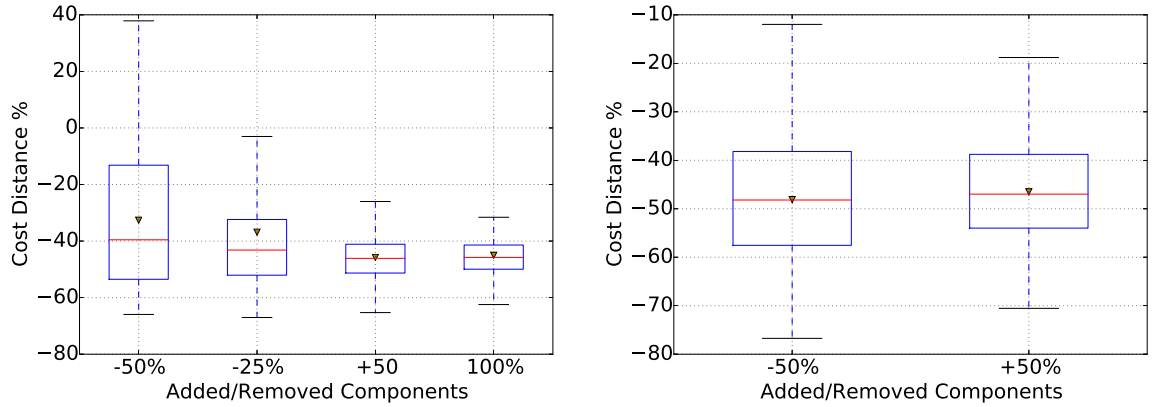


Figure 6.4: Cost distances between solutions from 2PCAP-REC and S.A.1 aggregated by number of application components. Results for Class B and Class C experiments are on the left and right sides, respectively. Triangles indicates the average of distances.

S.A.1 implements a different way of generating solutions from scratch. Instead of trying to generate valid placements randomly across multiple Cloud provider sites, it tries to place all components on same Cloud provider site. Observe that in this case all communication requirements would be satisfied. However, as this approach might miss many possible configurations, algorithm optimality is affected. If there is no site that could host the entire application, the original random procedure discussed in the beginning is performed.

The cost distances between 2PCAP-REC and S.A.1 are plotted in Figure 6.4. We aggregate all solutions by number of added or removed components. In general, most of distances are negative indicating that 2PCAP-REC managed to calculate better solutions than S.A.1. The median varies between -40% and -47% and the average between -35% and -45% for Class B experiment. For the Class C experiment, medians and averages are between -48% and -47%.

In Figure 6.4, S.A.1 has a better performance when the reconfiguration scenario involves the removal of components from the initial placement. This happens mainly because, by removing components, the search space becomes smaller and, thus S.A.1 will be able to explore a larger part of it.

2PCAP-REC vs. S.A.2

To improve that approach and introduce more of Cloud provider site heterogeneity to partial solutions, we implemented S.A.2, which can generate partial solutions from the scratch using a modified version of 2PCAP which selects random sub-placements instead of the less expensive ones during composition phase. This allows for generating valid configurations very quickly. The cost distances between 2PCAP-REC and S.A.2 for a Class B experiment aggregated by number of added or removed components is represented in Figure 6.5. We observe smaller and more balanced cost distances than those represented in Figure 6.4, specially in terms of medians and averages. Notice, however, that the last quartiles of situations where components are removed are worse than situations where components are added.

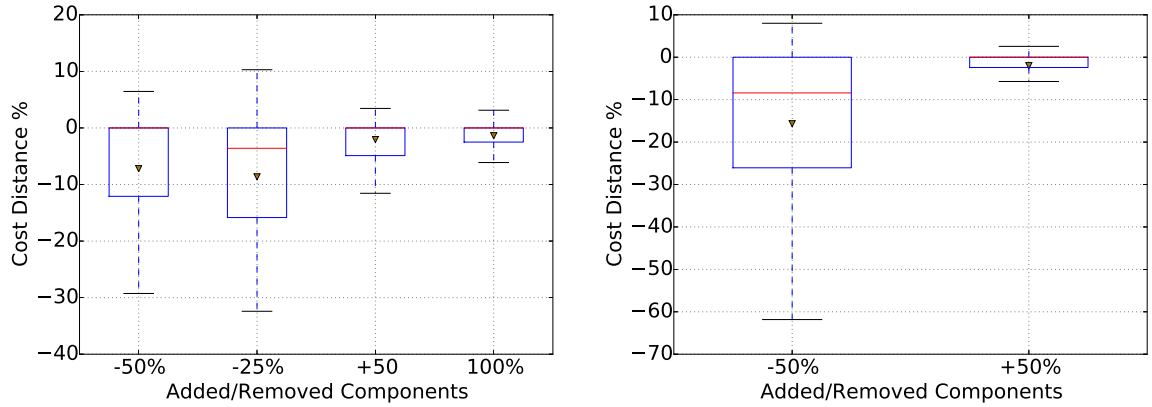


Figure 6.5: Cost distances between solutions aggregated by number of application components from 2PCAP-REC and S.A.2, a variant of S.A. that uses a randomized version of 2PCAP-REC to calculate partial solutions. Results for Class B and Class C experiments are on the left and right sides, respectively. Triangles indicates the average of distances.

We managed to improve the performance of S.A., however, 2PCAP-REC is still better in most of cases.

2PCAP-REC vs. S.A.3

The way initial configurations are calculated directly affects S.A. performance. Figure 6.6 illustrates the cost distance between 2PCAP-REC and S.A.3, a variation of S.A.1 in which initial solutions are calculated using 2PCAP-REC itself. This analysis helps us to understand by how much a solution calculated by 2PCAP-REC could be improved in one hour. It is possible to observe that there is an improvement specially in the last quartile of situations where components are removed during reconfiguration. In those cases it can reach an improvement of around 41%, however, the median and average are 0% and at most 3%, respectively. This reflects the difficulty to improve solutions calculated by 2PCAP-REC in large-scale scenarios.

2PCAP-REC vs. 2PCAP

The predicted duration of execution (δ) is an important reconfiguration problem parameter because it characterizes the *amortization* of migration costs. This amortization is determined by the ratio $\frac{migration\ cost}{\delta}$ and indicates the overhead per time unit caused by migration costs. For example, a \$10 migration will cost \$10 per time unit if $\delta = 1$; \$1 per time unit if $\delta = 10$; or \$0.1 per time unit if $\delta = 100$.

If δ is large enough to make migration costs become insignificant, the costs of reconfiguration placements and initial placements converge. To analyze that, we compare the performance of 2PCAP and 2PCAP-REC on Class B and C experiments with varying δ.

The results aggregated by predicted duration are depicted in Figure 6.7. As expected, cost distances reduce as the predicted duration grows indicating the aforementioned convergence. Notice that it is possible that a reconfiguration placement is cheaper than a reconfiguration placement calculated as an initial placement. This happens because pre-

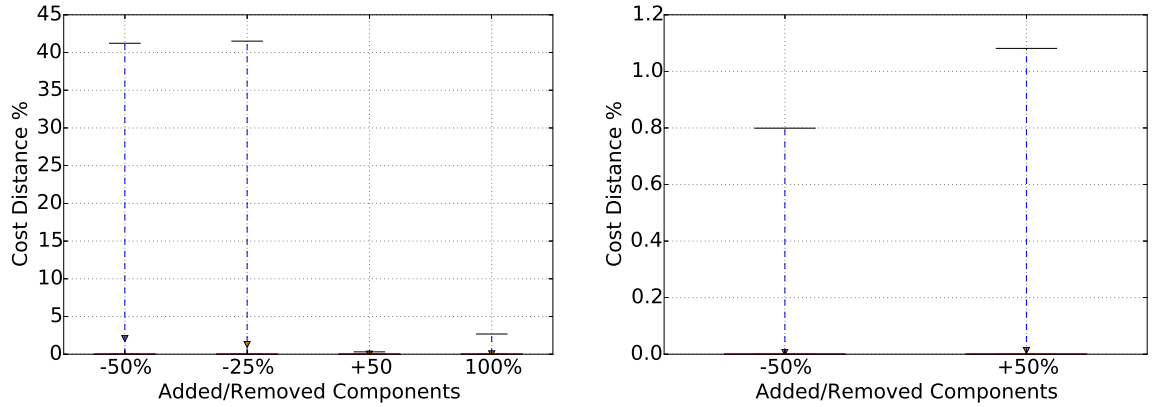


Figure 6.6: Cost distances between solutions from 2PCAP-REC and S.A.3, a version of S.A. meta-heuristic initialized by 2PCAP-REC aggregated by number of application components. In this analysis, the distances represent the improvement calculated by S.A. to a placement computed by 2PCAP-REC. Results for Class B and Class C experiments are on the left and right, respectively. Triangles indicate the average of distances.

viously deployed components and migration costs may allow for a different exploration of the search space that would not be considered by 2PCAP. This phenomena can be observed in Figure 6.7 where predicted duration is 500 for Class C experiment.

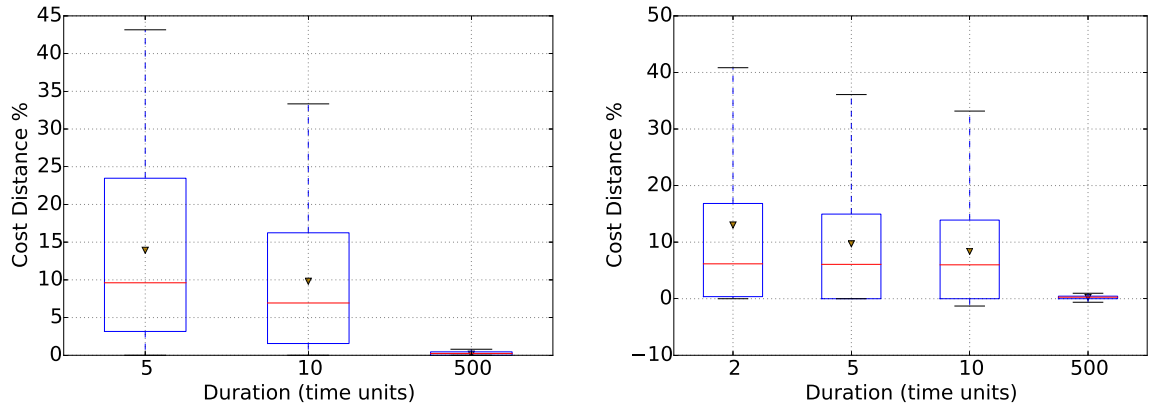


Figure 6.7: Cost distances aggregated by predicted duration of execution between solutions from 2PCAP-REC and 2PCAP (the reconfiguration placement problem is considered an initial placement problem in this case). Results for Class B and Class C experiments are on the left and right sides, respectively. Triangles indicates the average of distances.

Execution Times

S.A. variants were given one hour per problem to calculate a solution. Likewise, 2PCAP-REC was given the same timeout, however, as we can see in Figures 6.8, it has never taken more than 8 seconds per problem from Class B experiment and 80 seconds for Class C experiment problems. Notice that 2PCAP-REC's execution times follow the

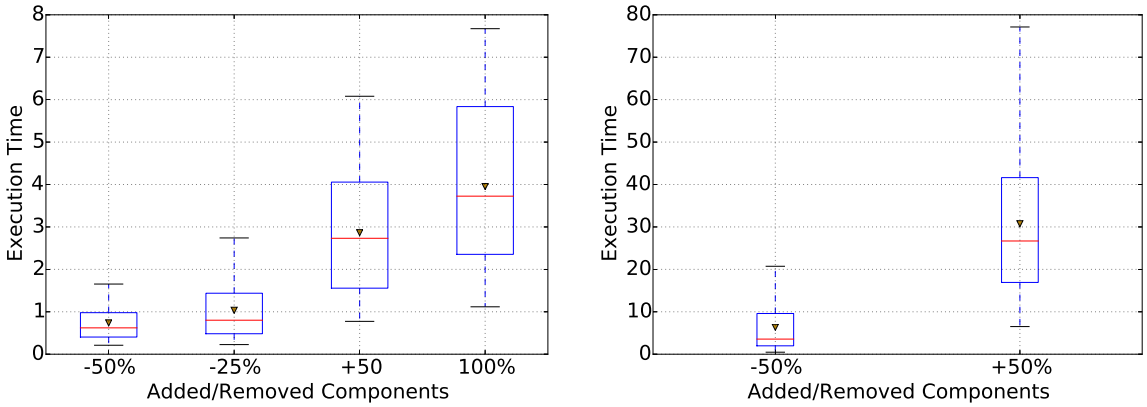


Figure 6.8: 2PCAP-REC execution times for Class B and Class C experiments (left and right sides, respectively). Triangles indicates the average of distances.

same pattern as 2PCAP's (cf. Section 5.7).

6.5.4 Discussion

In this section we evaluated the performance of 2PCAP-REC. This was accomplished through the comparison of reconfiguration placements calculated by 2PCAP-REC, S.A. meta-heuristic implementations and a MIP solver. We generated three different problem classes aiming at having a better understanding about the limitations of 2PCAP-REC. We observed that, in spite of having very reduced execution times, 2PCAP-REC managed to compute, most of the time, better placements than state of the art approaches or, at least, comparable to them. This is an indication that 2PCAP-REC is an interesting option when reconfiguration problems are large and when there are tight time constraints.

6.6 Conclusion

In this chapter we presented 2PCAP-REC, a communication and reconfiguration-aware heuristic. Being an extension of 2PCAP (cf. Chapter 5), 2PCAP-REC inherits the ability of calculating solutions for large-scale CAPDAMP problems and, furthermore, it is able to calculate solutions for reconfiguration scenarios.

We discussed a detailed related work specially focused on validating the component migration model presented in Section 6.2 and we evaluated 2PCAP-REC taking into consideration the main approaches of the state of the art. We observed in the experiments that 2PCAP-REC is an interesting option to be considered in large-scale scenarios or in situations where time constraints are tight or in both cases. In addition, 2PCAP-REC is also a good tool to quickly generate good quality initial solutions for meta-heuristics and other iterative algorithms.

Chapter 7

Conclusion and Perspectives

In this thesis we studied the problem of calculating initial and reconfiguration placements of distributed applications on multiple Cloud based infrastructures. We were interested in situations where the placement objective was to minimize renting costs while satisfying resource and communication constraints. This problem is difficult to solve and automate due to scalability issues resulting from the large amount of different Cloud providers and virtual machine (VM) types that we consider in the problem. There are many interesting approaches in the literature, however, despite their important contributions, their solutions do not scale well (such as exact algorithms), or may take too much time to calculate a good quality solution (such as meta-heuristics), or oversimplify placement scenarios (such as many heuristics).

In this context, we split that problem in three parts and propose heuristic-based approaches for each part incrementally. Each heuristic was extensively evaluated and compared to state of the art approaches.

The combination of proposed heuristics and their evaluation are the main contributions of this thesis. We summarize it as follows:

- **Cost + Resource Constraints:** In a first moment we considered only cost and resource constraints of distributed applications and Cloud infrastructure. Communication constraints were let aside. In this way, we modeled the placement problem as a generalization of a multi dimensional bin packing problem (or vector packing problem) and proposed efficient heuristics based on First Fit Decreasing heuristics which can compute *initial* placements.

- **Cost + Communication and Resource Constraints:** In a second moment, the objective was to improve the previously proposed communication-oblivious heuristics by adding communication constraints to the modeling. To do so, we used a hierarchical approach to model Cloud topologies and a graph based approach to model distributed applications. We proposed 2PCAP, a heuristic that uses these models to calculate communication-aware initial placements of distributed applications on multiple Clouds. The vector packing based heuristics previously proposed are part of 2PCAP.

- **Cost + Communication and Resource Constraints + Reconfiguration:** The last part of the split placement problem concerns reconfiguration scenarios.

We added ways of expressing migration costs and components which are already deployed to application and Cloud models. Then, we proposed 2PCAP-REC, an extension of 2PCAP, that can deal with reconfiguration scenarios, i.e. situations where part of a distributed application has previously been deployed at the moment a placement is calculated.

- **Evaluation:** Those three heuristics where evaluated separately against state of the art approaches in a very similar way. Placement problems with different sizes and characteristics were generated and solved by the considered heuristic(s) and baseline algorithms. The latter were, in summary, variations of mixed integer programming solvers and simulated annealing meta-heuristics, which are a good sample of state of the art approaches. The results pointed that the proposed heuristics were capable of calculating good quality solutions much faster than state of the art approaches, specially in large-scale placement scenarios.

In conclusion, the heuristics proposed in this thesis are valuable tools to quickly calculate placements for large-scale scenarios or cases where there are tight time constraints. Furthermore, they are still an interesting option as fast placement solution generators to be used to initialize algorithms, including branch-and-cut heuristics and meta-heuristics, or as good quality baseline algorithms.

7.1 Perspectives

The contributions of this thesis could be extended in many different ways. In this section, we discuss some research directions that could lead to interesting results.

7.1.1 Different Use Case Perspectives

Fog and Edge Computing

Today, a large part of Cloud computing community has its attention turned to the Fog and Edge computing. In summary, the objective of those platforms is to reduce the latency between the source of data or requests and service providers. This means that requests do not have to cross a country to reach a service hosted in a remote data center, but could be processed by servers (or any other connected computing device), responsible for attending a local area. Those servers could be hosted in data centers, inside cellphone transmission stations or even in personal use devices such as personal computers or cellphones. An example of utilization would be a mobile application delegating the heavy processing of some data to a near server and just getting its results. Another one would be a local infrastructure able to receive and process data from many near sensors, and to send them commands based on data analysis.

The challenge of how to place application components and replicas of application components over those Edge virtual servers in a given region is a very interesting placement problem. This scenario has a lot in common with those visited in Chapter 6, namely hosting infrastructure heterogeneity, resource, time and communication constraints, and problem scale. Hence, we think that 2PCAP-REC could be successfully adapted to that

context. It would benefit from improvements related to the modeling of resource and communication constraints and also some way of measuring resilience of virtual servers. In our opinion the core of both placement problems, on multiple Clouds or on the Edge, are very similar.

Virtual Machine Consolidation

Typically the virtual machine (VM) consolidation problem is characterized by calculating mappings between sets of VMs and sets of physical machines in a cluster. The objective is usually minimizing metrics, such as energy consumption or bandwidth usage. Observe that it may be necessary to reorganize VMs previously running on physical machines to allocate new VMs, thus VM migration may occur during the process.

2PCAP-REC could be adapted to this scenario. Firstly, because data centers can very commonly be modeled hierarchically. Furthermore the objective function of minimizing renting costs could be easily replaced by another one, such as energetic consumption minimization. Finally, it is possible to replace the current component migration model to a VM migration model.

7.1.2 Improvement of Application and Constraint Model Perspectives

In Chapter 3 we listed the hypotheses that we used to model the placement problems discussed throughout this thesis and to design our heuristics. In this section we propose removing or changing some of these hypotheses in order to bring more aspects from real world placement scenarios to the considered problems. The challenge is to improve application and infrastructure models while keeping a good performance of the proposed heuristics.

Performance Constraints

In our application and infrastructure models, we do not consider any interference effects resulting from components sharing a same VM. Currently it is not possible to describe performance metrics neither any relation between allocated resources and component performance. Consequently, it is not possible to describe performance requirements or capacities. Once a resource-performance correlation is established, we believe that 2PCAP-REC and the proposed application and Cloud models could be adapted or extended to take into account performance constraints.

Data Transfer Constraints

The proposed application and infrastructure models do not explicitly take into account data transfers between components. To adapt 2PCAP-REC to this scenario, it would be necessary to change the meaning of communication constraints. In the current model, we consider that these constraints are described in terms of latency. Considering that data transfer requirements are described in therms of throughput, it would be necessary to introduce the notion of communication bandwidth and correlate it to latency. Also, it would be necessary to explicitly introduce a "data transfer" dimension to components.

At this point whenever a new component is deployed to a VM, communication capacities of connections of the hosting VM to other VMs would be impacted. Finally, the way 2PCAP-REC's decomposition phase is implemented would have to be updated since the structure of component subsets would affect VM communication capacities.

Controlled SLA Violation

Throughout this thesis we considered that resource and communication constraints are hard constraints. It could be interesting extending 2PCAP-REC to allow violations of some constraints in exchange of smaller renting costs, for example. It could be a user defined parameter establishing allowed levels of degradation or minimal cost improvements.

7.1.3 Multi-objective Optimization

The objective of the heuristics presented in this thesis were minimizing renting and migration (where apply) costs while satisfying resource and communication (where apply) constraints. A very interesting research direction would be adding other optimization objectives to the problem, transforming it in a multi-objective optimization problem. For example, besides minimizing renting costs it could be interesting to also minimize resource utilization. The heuristics and application and infrastructure models proposed in this thesis could be a starting point for a solution.

Bibliography

[1] M. Abdelbaky, J. Diaz-Montes, M. Parashar, M. Unuvar, and M. Steinder. "Docker Containers across Multiple Clouds and Data Centers". In: *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. 2015.

[2] Tobias Achterberg. "SCIP: Solving Constraint Integer Programs". In: *Mathematical Programming Computation* (2009).

[3] *Amazon EC2 Container Service*. URL: `https://aws.amazon.com/ecs`.

[4] *Amazon Elastic Compute Cloud*. URL: `https://aws.amazon.com/ec2`.

[5] *Amazon Virtual Private Cloud*. URL: `aws.amazon.com/vpc`.

[6] *Amazon Web Services*. URL: `https://aws.amazon.com`.

[7] Greg R Andrews. *Foundations of Parallel and Distributed Programming*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[8] Apache. *Apache Cassandra*. URL: `http://cassandra.apache.org/`.

[9] *Apache Hadoop*. URL: `http://hadoop.apache.org/`.

[10] *Apache Spark*. URL: `https://spark.apache.org/`.

[11] *Autodesk Fusion 360*. URL: `ww.autodesk.com/products/fusion-360`.

[12] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Perez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. "Adding Virtualization Capabilities to the Grid'5000 Testbed". In: *CLOSER*. 2013.

[13] Nikhil Bansal, José R. Correa, Claire Kenyon, and Maxim Sviridenko. "Bin Packing in Multiple Dimensions: Inapproximability Results and Approximation Schemes". In: *Mathematics of Operations Research* (2006).

[14] D. Bernstein. "Containers and Cloud: From LXC to Docker to Kubernetes". In: *IEEE Cloud Computing* (2014).

[15] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. "Advanced Web Services". In: 2014. Chap. TOSCA: Portable Automated Deployment and Management of Cloud Applications.

[16] Ofer Biran, Antonio Corradi, Mario Fanelli, Luca Foschini, Alexander Nus, Danny Raz, and Ezra Silvera. "A Stable Network-Aware VM Placement for Cloud Systems". In: *CCGrid*. 2012.

[17] Blizzard. *World of Warcraft*. URL: `https://worldofwarcraft.com/`.

[18] Robert B. Bohn, John Messina, Fang Liu, Jin Tong, and Jian Mao. "NIST Cloud Computing Reference Architecture". In: *Proceedings of the 2011 IEEE World Congress on Services*. 2011.

[19] Adrian Bondy and M. Ram Murty. *Graph Theory*. Springer-Verlag, 2008.

[20] Ilhem BoussaïD, Julien Lepagnot, and Patrick Siarry. "A Survey on Optimization Metaheuristics". In: *Journal Information Sciences: an International Journal* (2013).

[21] Hinde Lilia Bouziane. "De l'abstraction des modèles de composants logiciels pour la programmation d'applications scientifiques distribuées". PhD thesis. 2008.

[22] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. "The FRACTAL component model and its support in Java". In: *Software: Practice and Experience* (2006).

[23] Drona Pratap Chandu. "A Parallel Genetic Algorithm for Three Dimensional Bin Packing with Heterogeneous Bins". In: *International Journal of Computer Trends and Technology* (2014).

[24] W. Chen, X. Qiao, J. Wei, and T. Huang. "A Profit-Aware Virtual Machine Deployment Optimization Framework for Cloud Platform Providers". In: *CLOUD*. 2012.

[25] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. "Live migration of virtual machines". In: *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*. 2005.

[26] Jens Clausen. *Branch and Bound Algorithms – Principles And Examples*. Tech. rep. Department of Computer Science, University of Copenhagen, 1999.

[27] Emanuel Ferreira Coutinho, Flávio Rubens de Carvalho Sousa, Paulo Antonio Leal Rego, Danielo Gonçalves Gomes, and José Neuman de Souza. "Elasticity in cloud computing: a survey". In: *Annals of telecommunications - Annales des télécommunications* (2015).

[28] Mehmet Deveci, Kamer Kaya, Bora Uçar, and Umit V. Catalyurek. "Fast and High Quality Topology-Aware Task Mapping". In: *IPDPS*. 2015.

[29] *Docker*. URL: `https://www.docker.com/`.

[30] *Documentation of vmware's vSphere*. URL: `https://pubs.vmware.com/vsphere-50/topic/com.vmware.vsphere.vm_admin.doc_50/GUID-CEFF6D89-8C19-4143-8C26-4B6D6734D2CB.html`.

[31] C. Dupont, T. Schulze, G. Giuliani, A. Somov, and F. Hermenier. "An energy aware framework for virtual machine placement in cloud federated data centres". In: *2012 Third International Conference on Future Systems: Where Energy, Computing and Communication Meet (e-Energy)*. 2012.

[32] *Evernote*. URL: `https://evernote.com`.

[33] Ulrich Faigle, Walter Kern, and Georg Still. "Algorithmic Principles of Mathematical Programming". In: Spring Netherlands, 2002.

[34] P. Fan, Z. Chen, J. Wang, Z. Zheng, and M. R. Lyu. "Topology-Aware Deployment of Scientific Applications in Cloud Computing". In: *CLOUD*. 2012.

[35] Eugen Feller, Louis Rilling, and Christine Morin. "Energy-Aware Ant Colony Based Workload Placement in Clouds". In: *GRID*. 2011.

[36] Md Hasanul Ferdaus, Manzur Murshed, Rodrigo N. Calheiros, and Rajkumar Buyya. "Virtual Machine Consolidation in Cloud Data Centers Using ACO Metaheuristic". In: *Europar*. 2014.

[37] L. R. Foulds. "Optimization Techniques An Introduction". In: Springer New York, 1981.

[38] Michaël Gabay and Sofia Zaourar. *Variable Size Vector Bin Packing Heuristics - Application to the Machine Reassignment Problem*. Tech. rep. INRIA, 2013.

[39] Michaël Gabay and Sofia Zaourar. "Vector Bin Packing with Heterogeneous Bins: Application to the Machine Reassignment Problem". In: *Annals of Operations Research* (2015).

[40] Yongqiang Gao, Haibing Guan, Zhengwei Qi, Yang Hou, and Liang Liu. "A Multi-objective Ant Colony System Algorithm for Virtual Machine Placement in Cloud Computing". In: *Journal of Computer and System Sciences* (2013).

[41] Mukund N. Thapa George B. Dantzig. "Linear Programming 1: Introduction". In: Springer New York, 1997.

[42] *Google*. URL: `https://www.google.com`.

[43] *Google App Engine*. URL: `https://cloud.google.com/appengine`.

[44] *Google Apps*. URL: `https://gsuite.google.com/`.

[45] *Google Cloud*. URL: `https://cloud.google.com`.

[46] Hadi Goudarzi and Massoud Pedram. "Multi-dimensional SLA-Based Resource Allocation for Multi-tier Cloud Computing Systems". In: *CLOUD*. 2011.

[47] Object Management Group. *Business Process Modeling And Notation (BPMN)*. URL: `http://www.omg.org/spec/BPMN/`.

[48] Object Management Group. *Common Object Request Broker Architecture (CORBA)*. URL: `http://www.omg.org/spec/CCM/`.

[49] Object Management Group. *Unified Modeling Language (UML) Version 2.0*. URL: `http://www.omg.org/spec/UML/2.0/`.

[50] Nikolay Grozev and Rajkumar Buyya. "Inter-Cloud architectures and application brokering: taxonomy and survey". In: *Software: Practice and Experience* (2014).

[51] L. Gu, D. Zeng, S. Guo, Y. Xiang, and J. Hu. "A General Communication Cost Optimization Framework for Big Data Stream Processing in Geo-Distributed Data Centers". In: *IEEE Transactions on Computers* (2016).

[52] BernardT. Han, George Diehr, and JackS. Cook. "Multiple-Type, Two-Dimensional Bin Packing Problems: Applications and Algorithms". In: *Annals of Operations Research* (1994).

[53] Pavol Hell and Jaroslav Nešetřil. *Graphs and Homomorphisms*. Oxford University Press, 2004.

[54] Pavol Hell and Jaroslav Nešetřil. "On the complexity of H-coloring". In: *Journal of Combinatorial Theory, Series B* (1990).

[55] Chris Hyser, Bret Mckee, Rob Gardner, and Brian J Watson. *Autonomic Virtual Machine Placement in the Data Center*. Tech. rep. HPL-2007-189. HP Laboratories, 2007.

[56] International Business Machines Corporation (IBM). *IBM ILOG CPLEX Optimizer*. URL: `https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`.

[57] *International Business Machines Corporation (IBM)*. URL: `https://www.ibm.com`.

[58] Manar Jammal, Ali Kanso, and Abdallah Shami. "High Availability-Aware Optimization Digest for Applications Deployment in Cloud". In: *ICC*. 2015.

[59] D. Jayasinghe, C. Pu, T. Eilam, M. Steinder, I. Whally, and E. Snible. "Improving Performance and Availability of Services Hosted on IaaS Clouds with Structural Constraint-Aware Virtual Machine Placement". In: *SCC*. 2011.

[60] Emmanuel Jeannot, Guillaume Mercier, and Francois Tessier. "Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques". In: *IEEE Transactions Parallel Distributed Systems* (2014).

[61] Brendan Jennings and Rolf Stadler. "Resource Management in Clouds: Survey and Research Challenges". In: *Journal of Network and Systems Management* (2014).

[62] David Karger and Krzysztof Onak. "Polynomial Approximation Schemes for Smoothed and Random Instances of Multidimensional Packing Problems". In: *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '07. 2007.

[63] Oliver Kopp, Tobias Binz, Uwe Breitenbücher, and Frank Leymann. "Winery - A Modeling Tool for TOSCA-Based Cloud Applications." In: *Proceedings of the 11th International Conference on Service-Oriented Computing, ICSOC 2013, Berlin, Germany*. 2013.

[64] Katrina LaCurts, Shuo Deng, Ameesh Goyal, and Hari Balakrishnan. "Choreo: Network-aware Task Placement for Cloud Applications". In: *IMC*. 2013.

[65] Vincent Lanore. "On Scalable Reconfigurable Component Models for High-Performance Computing". PhD thesis. Ecole normale supérieure de lyon - ENS LYON, 2015.

[66] William Leinberger, George Karypis, and Vipin Kumar. "Multi-Capacity Bin Packing Algorithms with Applications to Job Scheduling Under Multiple Constraints". In: *ICPP*. 1999.

[67] *Linux Containers*. URL: `https://linuxcontainers.org`.

[68] Jose Luis Lucas-Simarro, Rafael Moreno-Vozmediano, Ruben S. Montero, and Ignacio M. Llorente. "Scheduling Strategies for Optimal Service Deployment across Multiple Clouds". In: *Future Generation Computer Systems* (2013).

[69] T.V. Lakshman M. Alicherry. "Network Aware Resource Allocation in Distributed Clouds". In: *INFOCOM* (2012).

[70] Zoltán Ádám Mann. "Allocation of Virtual Machines in Cloud Data Centers&Mdash;A Survey of Problem Models and Optimization Algorithms". In: *ACM Comput. Surv.* (2015).

[71] Sunilkumar S. Manvi and Gopal Krishna Shyam. "Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey". In: *Journal of Network and Computer Applications* 41 (2014).

[72] Ching Chuen Teck Mark, Dusit Niyato, and Tham Chen-Khong. "Evolutionary Optimal Virtual Machine Placement and Demand Forecaster for Cloud Computing". In: *IEEE AINA* (2011).

[73] Violeta Medina and Juan Manuel García. "A Survey of Migration Mechanisms of Virtual Machines". In: *ACM Computing Surveys* (2014).

[74] Peter M. Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. Tech. rep. 2011.

[75] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. "Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement". In: *INFOCOM*. 2010.

[76] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. "Equation of State Calculations by Fast Computing Machines". In: *The Journal of Chemical Physics* (1953).

[77] *Microsoft*. URL: `http://www.microsoft.com`.

[78] *Microsoft Azure*. URL: `https://azure.microsoft.com`.

[79] *Microsoft Azure Stack*. URL: `https://azure.microsoft.com/en-us/overview/azure-stack/`.

[80] *Microsoft HD Insight*. URL: `https://azure.microsoft.com/en-us/services/hdinsight/`.

[81] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. "Containers checkpointing and live migration". In: *In Ottawa Linux Symposium*. 2008.

[82] *Mongo DB*. URL: `https://www.mongodb.com/`.

[83] Nintendo. *Pokemon Go*. URL: `http://www.pokemongo.com/`.

[84] L. Nonde, T. E. H. El-Gorashi, and J. M. H. Elmirghani. "Energy Efficient Virtual Network Embedding for Cloud Networks". In: *Journal of Lightwave Technology* (2015).

[85] A. Omezzine, S. Yangui, N. Bellamine, and S. Tata. "Mobile Service Micro-containers for Cloud Environments". In: *2012 IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*. 2012.

[86] *OpenVZ – Virtuozzo Containers*. URL: `https://openvz.org/`.

[87] Computational Infrastructure for Operations Research (COIN-OR). *COIN-OR Branch and Cut (CBC)*. URL: `https://www.coin-or.org/Cbc/cbcuserguide.html`.

[88] Manfred Padberg and Giovanni Rinaldi. "A Branch-and-cut Algorithm for the Resolution of Large-scale Symmetric Traveling Salesman Problems". In: *SIAM Review* (1991).

[89] C. Pahl and B. Lee. "Containers and Clusters for Edge Cloud Architectures – A Technology Review". In: *2015 3rd International Conference on Future Internet of Things and Cloud*. 2015.

[90] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder. *Heuristics for Vector Bin Packing*. Tech. rep. Microsoft Research, 2011.

[91] Matthew Perry. *Simanneal: Python Module for Simulated Annealing Optimization*. URL: `https://github.com/perrygeo/simanneal`.

[92] Dana Petcu. "Portability and Interoperability between Clouds: Challenges and Case Study". In: *Towards a Service-Based Internet: 4th European Conference, ServiceWave 2011, Poznan, Poland, October 26-28, 2011. Proceedings*. 2011.

[93] Jing Tai Piao and Jun Yan. "A Network-aware Virtual Machine Placement and Migration Approach in Cloud Computing". In: *Proceedings of the 2010 Ninth International Conference on Grid and Cloud Computing*. GCC '10. 2010.

[94] Gerald J. Popek and Robert P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures". In: *Communications of the ACM* (1974).

[95] *Rackspace*. URL: `https://www.rackspace.com`.

[96] M.A. Rodriguez and R. Buyya. "Deadline Based Resource Provisioning and Scheduling Algorithm for Scientific Workflows on Clouds". In: *IEEE Transactions on Cloud Computing* (2014).

[97] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. "Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting". In: *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing*. CLOUD '11. 2011.

[98] Cihan Seçinti and Tolga Ovatman. "On Optimizing Resource Allocation and Application Placement Costs in Cloud Systems". In: *CLOSER*. 2014.

[99] Mina Sedaghat, Francisco Hernandez-Rodriguez, and Erik Elmroth. "A Virtual Machine Re-packing Approach to the Horizontal vs. Vertical Elasticity Trade-off for Cloud Autoscaling". In: *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*. CAC '13. 2013.

[100] Amazon Web Services. *What are Containers?* URL: `https://aws.amazon.com/containers/`.

[101] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. "Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors". In: *SIGOPS Operating Systems Review* (2007).

[102] Bart Spinnewyn, Bart Braem, and Steven Latre. "Fault-Tolerant Application Placement in Heterogeneous Cloud Environments". In: *CNSM*. 2015.

[103] Advancement of Structured Information Standards (OASIS). *Web Services Business Execution Language*. URL: `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[104] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. 2002.

[105] El-Ghazali Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.

[106] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. 2006.

[107] Francois Tessier, Guillaume Mercier, and Emmanuel Jeannot. "Process Placement in Multicore Clusters:Algorithmic Issues and Practical Techniques". In: *IEEE Transactions on Parallel and Distributed Systems* ().

[108] Hien Nguyen Van, F.D. Tran, and J.-M. Menaud. "SLA-Aware Virtual Resource Management for Cloud Infrastructures". In: *CIT*. 2009.

[109] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. "A Break in the Clouds: Towards a Cloud Definition". In: *SIGCOMM Computer Communication Review* (2009).

[110] A. Verma, G. Kumar, R. Koller, and A. Sen. "CosMig: Modeling the Impact of Reconfiguration in a Cloud". In: *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*. 2011.

[111] Werner Vogels. "Beyond Server Consolidation". In: *ACM Queue* (2008).

[112] Rafael Weingärtner, Gabriel Beims Bräscher, and Carlos Becker Westphall. "Cloud resource management: A survey on forecasting and profiling models". In: *Journal of Network and Computer Applications* 47 (2015).

[113] Roland Wunderling. "Paralleler und objektorientierter Simplex-Algorithmus". PhD thesis. Technische Universität Berlin, 1996.

[114] Andrew Chi-Chih Yao. "New Algorithms for Bin Packing". In: *J. ACM* (1980).

[115] Chenying Yu and Fei Huan. "Live Migration of Docker Containers through Logging and Replay". In: *3rd International Conference on Mechatronics and Industrial Informatics Cite this publication*. 2015.

[116] Minyi Yue. "A Simple Proof of the Inequality $FFD(L) \leqslant \frac{11}{9}OPT(L)+1, \forall L$ for the FFD Bin-Packing Algorithm". In: *Acta Mathematicae Applicatae Sinica* (1991).

[117] Z. I. M. Yusoh and M. Tang. "Clustering Composite SaaS Components in Cloud Computing using a Grouping Genetic Algorithm". In: *CEC*. 2012.

[118] Qian Zhu and Gagan Agrawal. "Resource Provisioning with Budget Constraints for Adaptive Applications in Cloud Environments". In: *HPDC*. 2010.

[119] Y. Zhu, J. Xu, Q. Zhang, X. Wang, P. Palacharla, and T. Ikeuchi. "Game Theory Based Reliable Virtual Network Mapping for Cloud Infrastructure". In: *ICC*. 2016.

[120] B. Zong, R. Raghavendra, M. Srivatsa, X. Yan, A. K. Singh, and K. W. Lee. "Cloud Service Placement via Subgraph Matching". In: *ICDE*. 2014.