Chapter 6

Initial communication and cost aware placement

In this thesis our objective is to study the placement automation of distributed applications over Cloud based infrastructures. In Chapter 5, we proposed cost-aware and communication oblivious heuristics capable of calculating initial placements of component-based application over the Cloud. In this Chapter, we improve our previously application and infrastructure models to make it possible to describe communication constraints and propose a heuristic capable of calculating initial communication and cost aware placements.

6.1 Introduction

We are interested in the placement of *distributed applications* on the Cloud based infrastructures. More specifically, we address the challenge of calculating an initial placement of component-based applications on multiple clouds with the objective of minimizing renting costs while satisfying an applications' *resource* and *communication* constraints. Our hypothesis is that users describe their applications' communication constraints because they want a placement that respects their application *latency* requirements. Thus, to support this constraint and to overcome the lack of information about the network topologies of public cloud providers, we introduce a flexible approach that allows an application designer to describe communication constraints using a less accurate view of the Cloud topology, as well as a more precise schema in the context of private cloud providers. Benefiting from this model, we propose an efficient and scalable heuristic that mixes graph partitioning and clustering concepts and which is able to compute good quality placements very quickly for small and large scenarios. This paper extends the approach presented in Chapter 5, where we proposed multi dimensional bin packing based greedy heuristics to solve a *communication-oblivious* placement problem.

In Section 6.2 we formalize the problem and model it as a *mixed integer programming.* In Section 6.3 we discuss the state of the art. Section 6.4 presents our application and cloud models. Section 6.5 details the proposed heuristic which is evaluated in Section 6.6.

6.2 Communication Aware Placement of Component-Based Applications on Multiple Clouds

In this chapter we tackle the problem of finding an *initial* placement for a *component-based* application on multiple clouds (multi-cloud).

Components or software components can be seen as independent binary units of deployment that interact to form a functioning system (cf. Section 2.3.4). In summary, a component exposes what it provides and what it requires through interfaces, hiding its implementation to reduce coupling and enhance reusability.

Each application *component* has *resource requirements* as well as *communication requirements* for its *connections* to other components. We also consider that there is a description of the *capacities* and *renting price* of each available *virtual machine (VM) type* and a description of their *inter-connections*. Resource requirements may be number of CPU cores, RAM or disk usage, for example. Communication requirements may be described in terms of bandwidth or latency, for example.

Our objective is to map each application component to an instance of a VM type in a way that renting *costs are minimized* and resource and communication *constraints are satisfied*. This means that the resource capacities of a VM instance *must be larger than or equal to* the sum of resource requirements of the components it hosts. We call those resource capacities or requirements *dimensions*. Similarly, the connection requirements between each pair of components must be inferior to the connection capacities between the virtual machine instances hosting them. We suppose that connection requirements and capacities are used to characterize communication latencies and that they can be expressed in a numerical way.

In this chapter we only calculate *initial placements*, i.e., we consider that the application or part of it is not previously deployed.

Figure 6.1 illustrates an example of a communication aware placement and, in Section 6.2.1 we formalize this problem.

6.2.1 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. Let $x^{comp}_{i,j}$ be the connection requirement between components i and j, both elements of $\mathcal{I}$.

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*. Examples of machines group are a cluster, a *cloud provider site*, etc. Machine groups are inter-connected and their connection capacities are represented as $x^{mg}_{s,u}$, both s and u element of $\mathcal{S}$.

Each VM type can be instanced as many times as needed. A component must be assigned to exactly one VM and each VM may host several components.

The objective is to assign all components to VMs in a way that resource and communi-

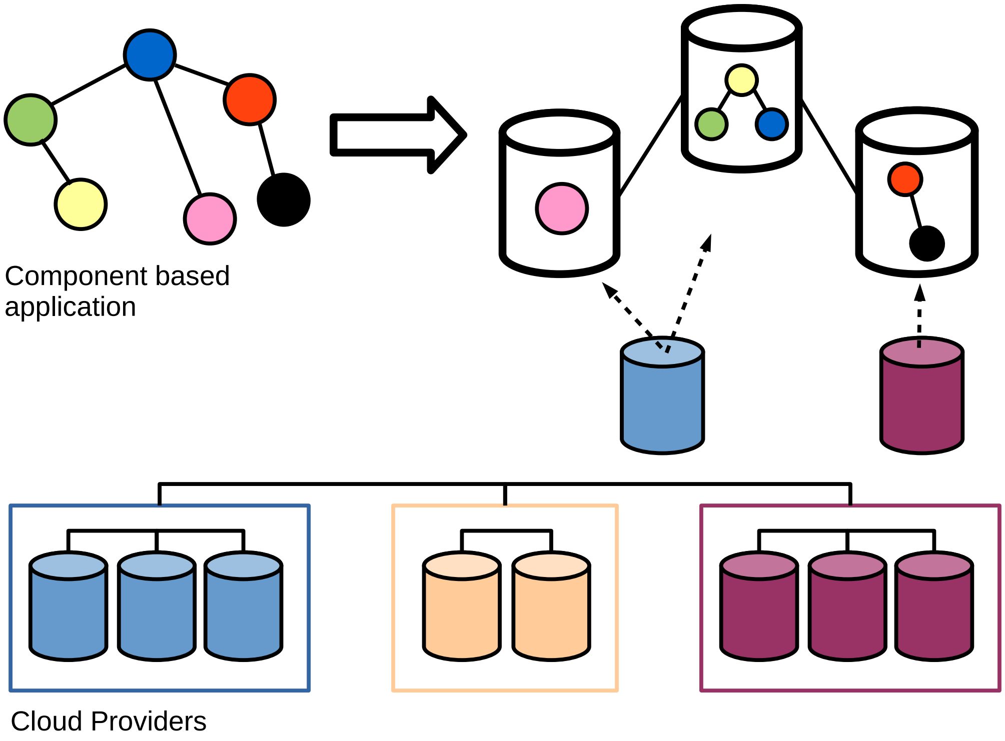

Figure 6.1: Communication aware placement example. The component based application is represented as a graph where nodes represent components and edges represent connections or dependencies between components. Colored boxes represent cloud providers and colored cylinders inside of them are VM types. 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.

cation requirements of components are satisfied, the capacities of each VM are respected, and the renting costs are minimized.

As this work considers the initial placement, we do not assume *a priori* information concerning expected workload, dynamic actors that would allow online modifications of the placement, and renting times.

6.2.2 Optimization Problem Formulation

The problem of finding an initial placement for a component-based application on multiple clouds can be modeled using 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 6.6.

Let $v_{k,t}$ be the k-th rented VM of type t. The set containing all possible rented VMs $\mathcal{V} = \{v_{k,t} \mid 1 \leqslant k \leqslant |\mathcal{I}|, 1 \leqslant t \leqslant |\mathcal{T}|\}$ has size $|\mathcal{V}| = |\mathcal{I}| \times |\mathcal{T}|$. Indeed, 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. To simplify the notation, let $v \in \mathcal{V}$. Hence, $\exists!\ k$ where $1 \leqslant k \leqslant |\mathcal{I}|$ and $\exists!\ t \in \mathcal{T}$ such that $v = v_{k,t}$. Consequently, the price of v is $p_v = p_t$, and while v is empty (i.e. no component is assigned to it), its capacity is $c_{v,d} = c_{t,d}$.

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.

Finally, let $x^{vm}_{v,w}$ be the connection capacity between VMs $v, w \in \mathcal{V}$. If v and w are VMs instanced from VM types belonging to machine groups $s1, s2 \in S$, respectively, then $x^{vm}_{v,w} = x^{mg}_{s_1,s_2}$.

The optimization problem is described in Equation 6.1 and all variables and constants are summarized in Table 6.1.

$$
\begin{aligned}
\text{Minimize } & \sum_{v \in \mathcal{V}} p_v . a_v \\
\text{s.t.} & \\
& \sum_{v \in \mathcal{V}} m_{i,v} = 1 && \forall i \in \mathcal{I} \quad (i) \\
& \sum_{i \in \mathcal{I}} m_{i,v} . r_{i,d} \leqslant c_{v,d} && \forall v \in \mathcal{V} \quad (ii) \\
& && 1 \leqslant d \leqslant D \\
& a_v = \begin{cases} 1 & \text{if } \sum_{i \in \mathcal{I}} m_{i,v} > 0 \\ 0 & \text{otherwise} \end{cases} && \forall v \in \mathcal{V} \quad (iii) \\
& m_{i,n} . m_{j,v} . (x^{vm}_{n,v} - x^{comp}_{i,j}) \geqslant 0 && \forall v, n \in \mathcal{V} \quad (iv) \\
& m_{i,n}, m_{j,v} \in \{0,1\}
\end{aligned}
\tag{6.1}
$$

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 $a_v = 1$ 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 possible rented VMs.
D	Number of resources/dimensions.
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.
$x^{vm}_{v,w}$	Connection capacity between VM $v \in \mathcal{V}$ and $w \in \mathcal{V}$.
$x^{comp}_{i,j}$	Connection requirement between VM $i \in \mathcal{I}$ and $j \in \mathcal{I}$.

Table 6.1

6.3 Related Work

Even though the problem of placing distributed applications on the cloud has been extensively investigated, there are still many open challenges. As discussed in Section 6.2, this work addresses the problem of minimizing the costs of the initial placement of large distributed applications on multiple clouds. We take into account application resource and communication constraints that the rented VMs must satisfy. For brevity, we call this problem CAPDAMP, for *communication aware placement of distributed applications on multiple clouds problem.*

The CAPDAMP can be seen as the combination of two NP-hard problems: a packing problem (the placement of application components on VMs satisfying resource constraints) and a graph partitioning problem (the placement of application components on VMs satisfying communication constraints). Consequently, the CAPDAMP is also NP-hard, which means that there is not any known polynomial time algorithm to solve it.

Approaches to the CAPDAMP, exactly as described in Section 6.2, are not very common. Usually previous work targets either the packing or the graph partitioning problem.

Previous work is classified into three groups based on the approach used to tackle the CAPDAMP and related *communication-aware* problems: exact/optimal approaches, graph partitioning approaches, and heuristic based approaches. A more extensive bibliography about the *packing* problem is provided in our previous work [72].

PS: too much blabla?

We divide the related work into three groups based on the approach used to tackle the CAPDAMP and related *communication-aware* problems: exact approaches, metaheuristic approaches, and heuristic based approaches. Then, we discuss them with respect to the CAPDAMP.

6.3.1 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*. The main advantage of this approach is the capability of producing optimal solutions. Another 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 [73], 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, [42] proposes a MIP to minimize application downtime. Both approaches neither consider renting cost minimizations nor allow for more than two dimensions of interest. In [51], 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 [44], 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, contratry to our work.

The main inconvenient of exact algorithms in the context of the CAPDAMP is scalability. However, the ability of calculating optimal solutions makes this approach very attractive for small instances of the CAPDAMP.

6.3.2 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 [53].

In [16] and [85], 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 [39], 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 [26]. In [22] another ant colony based approach for the VM consolidation problem is proposed.

The main issue in using meta-heuristics for communication aware problems is that the solution quality is, in summary, proportional to the time given to the meta-heuristic to calculate it. 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. Lastly, it is important to highlight that depending on the CAPDAMP characteristics, using meta-heuristics may not be recommended. Especially when the search space is rugged and/or plain [76], meaning that it may contain too many *local optimal solutions* and low correlation between them.

PS: last paragraph really necessary?

6.3.3 Heuristics

Due to the NP-hardness of the CAPDAMP, there are many approaches based on heuristics able to quickly calculating placements.

Graphs are recurrently used as a representation of application and cloud topologies. In this context, finding a valid placement may be seen as a graph partition problem, i.e., finding a subgraph from the communication graph which satisfies weight constraints from the application graph.

A graph based greedy heuristic for calculating the task mapping on supercomputer clusters is presented in [17]. Also using a greedy heuristic, an approach to place tasks on the Cloud is proposed in [48]. Using a max-clique based approach, [10] and [52] describe algorithms for the consolidation of VM types. A similar problem is addressed in [64], 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, [57] 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 [20]. In [88], a graph matching algorithm based on a *graph query* approach for the service placement on the cloud is proposed. In [87], a game theory approach is proposed to the allocation of virtual network for VMs from multiple data centers. Finally, [37] 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 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 tend to be more difficult. As a result, issues like VM/machine heterogeneity, renting costs and multi-dimensionality of resources are not addressed.

The authors of [43] 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.

6.3.4 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 proportional 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 chapter, we propose an efficient and scalable heuristic which addresses the aforementioned problems. Using a graph clustering and multidimensional bin packing strategies, it manages to calculate good quality solutions very quickly, as described in Section 6.6.

6.4 Modeling the Application and Cloud Platforms

To calculate solutions for the CAPDAMP, it is necessary to model resource and communication requirements from the distributed application and, in the same way, resource and communication capacities from the Cloud infrastructure. In Chapter 5 we presented an approach to model resource constraints and, in this section, we complement this model with a communication constraint modeling approach.

The main contribution of this chapter is the heuristic proposed in Section 6.5 which strongly relies on the models presented in this section.

PS: create a splitted version of Figure 6.2

6.4.1 Cloud Network Topology

The main challenge about modeling communication constraints for the CAPDAMP 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 2.2, this issue happens because, commonly, cloud providers does not make available details concerning their internal network infrastructures 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. For example, we previously discussed that, in general, network latencies are better between machines hosted in the same cloud provider than between machines from distinct cloud providers. In the same way, machines sharing the same cloud provider site have better latencies than machines belonging to the same cloud provider but hosted in different sites.

Thus, we are interested in a solution that is able to make use of *uncertain* or *less precise* information concerning cloud network latencies. Our approach to this issue is the modeling of the cloud *communication* or *network* topology as a tree. An example of this modeling is illustrated in Figure 6.2(a).

In the multi-cloud topology tree, *leaves* are sets of VM types which we call *machine groups*. Each *inner node m* represents a *connection* between all machine groups available in the sub-tree having m 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.

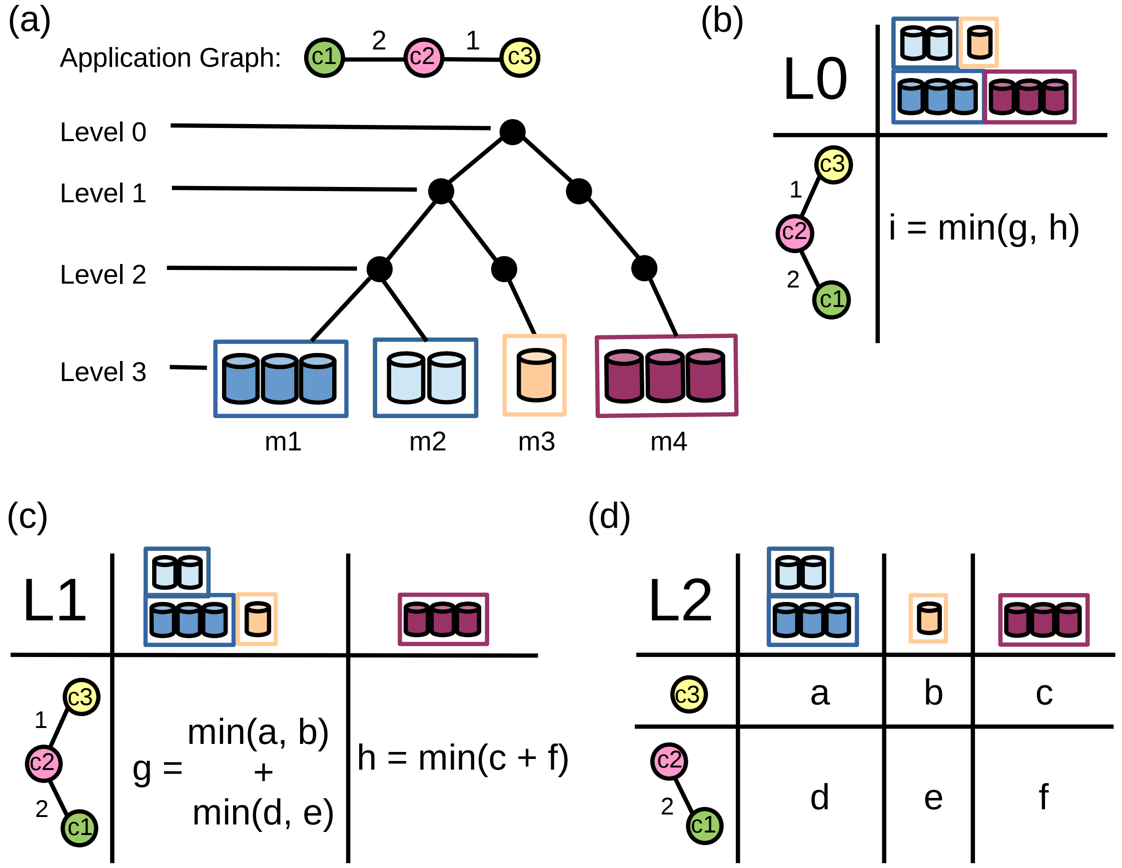

Figure 6.2: Placement example.

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 6.2(a), 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 in the same machine group are always connected with a connection quality equals to the level of the leaves. In the example illustrated in Figure 6.2(a), all connections between VM types inside the same machine group have quality 3. Thus, the closest an internal node is to the leaves, the smallest will be the latency.

Although our objective with this model is to make possible the description of uncertain or less accurate information network capacities when modeling communication constraints, it can also be employed in situations where details about the network topology are known.

6.4.2 Distributed Application Communication Topology

To describe distributed applications we extend the component based model presented in Chapter 5 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 6.4.1, matching the Cloud topology.

In the example illustrated in Figure 6.2, components $c1$ and $c2$ communicate and require a connection quality of *at least* 2, while $c2$ and $c3$ require at least 1.

6.5 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 6.4.1) to calculate solutions for the initial placement of component-based 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 subsets of components on subsets of machine groups are calculated using fast *communication-oblivious* multi-dimensional bin packing heuristics (cf. Chapter 5). Finally, the placement of the entire application is obtained by composing those subset placements that we call *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 with 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 6.2 illustrates the execution of 2PCAP and will used throughout this section. We use tables to represent the *decomposition* steps of that heuristic. The name of the tables refer to the level of decomposition (in the example 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 those two phases in details .

6.5.1 Phase 1: Decomposition

The decomposition phase has fundamentally 2 steps: the decomposition of component and machine group subsets and the computing of sub-placements. In the next paragraphs we present the necessary concepts to understand this phase.

Component Subset

A *component subset* $i_\ell \in \mathcal{I}_\ell$ 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 a set containing all component subsets *constructed on* level ℓ.

In the example illustrated in Figure 6.2, table (b) and (c) have both only one component subset composed by components $c1$, $c2$ and $c3$, i.e. $\mathcal{I}_0 = \mathcal{I}_1 = \{\{c1,c2,c3\}\}$. Table (d), however, has two component subsets: one composed by c and $c2$ and another one by $c3$. This is equivalent to $\mathcal{I}_2 = \{\{c1,c2\},\{c3\}\}$.

Observe that the union all component subsets is equal to the set of components, i.e. $\cup_{i_\ell \in \mathcal{I}_\ell} = \mathcal{I}$.

Machine Group Subset

A *machine group subset* $s_\ell \in \mathcal{S}_\ell$ 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. $\mathcal{S}_\ell$ contains all subsets built on level ℓ.

For example, in Figure 6.2, in table (b), at root level, there is only one machine group subset which is composed by all machine groups, namely $m1$,$m2$,$m3$ and $m4$, i.e. $\mathcal{S}_0 = \{\{m1,m2,m3,m4\}\}$. In table (c), at level 1 there are two machine groups, one composed by $m1$,$m2$ and $m3$ an another one by $m4$, thus $\mathcal{S}_1 = \{\{m1,m2,m3\},\{m4\}\}$. Finally, in table (d), at level 2, there are three machine groups, i.e. $\mathcal{S}_2 = \{\{m1,m2\},\{m3\},\{m4\}\}$.

Notice that the union all machine group subsets is equal to the set of components, i.e. $\cup_{i_\ell \in \mathcal{S}_\ell} = \mathcal{S}$.

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 root of the Cloud topology.

In the example presented in Figure 6.2, table (b) level 0 (L0) has only one machine group subset $\mathcal{S}_0 = \{\{m1,m2,m3,m4\}\}$. The decomposition of this subset on level 1 (L1), represented in table (c), has two machine group subsets, i.e. $\mathcal{S}_1 = \{\{m1,m2,m3\},\{m4\}\}$, given that there are two possible subtrees 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 split. Each resulting connected sub-graph is a valid component subset. Notice that, at level 0, there will always be only one component subset.

In the example illustrated in Figure 6.2, in table (b), at level 0 (L0), there is only one component subset $\mathcal{I}_0 = \{\{c1,c2,c3\}\}$. Given that in the example application there is no connection quality requirement smaller than one, at level 1 (L1), represented in table (c), there is also only one component subset $\mathcal{I}_1 = \{\{c1,c2,c3\}\}$. However, at level 2 (L2), removing edges with connection quality inferior to 2 will result in two subgraphs and consequently, two component subsets, i.e. $\mathcal{I}_2 = \{\{c1,c2\},\{c3\}\}$. This is represented in table (d).

Sub-placement

A *sub-placement* is the placement of a subset of components on a subset of machine groups. Exactly as an "usual" placement (cf. Section 6.2) a sub-placement aims to minimize VM renting costs while satisfying resource and communication constraints established by the application.

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 ℓ. The sub-placement of i_ℓ on s_ℓ can only be calculated if it is a *bottom sub-placement* or if the sub-placements generated by the decomposition of i_ℓ and s_ℓ were *calculated*.

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 as defined in Section 6.2. Thus, a bottom sub-placement has the following property:

$$x^{comp}_{i,j} \leqslant x^{mg}_{m,n}, \; \forall i, j \in i_\ell, \forall m,n \in s_\ell. \tag{6.2}$$

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 Figure 6.2, in tables (b), (c) and (d), each intersection between a machine group and component subsets is a sub-placement. The sub-placement at level 0, in table (b), is the sup-placement of $\{c1,c2,c3\}$ on $\{m1,m2,m3,m4\}$. One can notice that it is not a bottom sub-placement once the constraint described in Equation 6.2 does not holds. For instance, if $c1$ and $c2$ are placed over $m4$ and $c3$ over $m1$, thus communication constraints between $c2$ and $c3$ would not be satisfied. Likewise, sub-placements in table (c) are not bottom sub-placements, because Equation 6.2 is not satisfied. For example, $c1$ is placed on $m3$ and $c1$ and $c2$ are placed on $m1$ the connection constraint between $c1$ and $c2$ will not be satisfied. Finally, in table (d), at level 2, Equation 6.2 is satisfied, thus, it is not possible to find a sub-placement which would fail to satisfy communication quality requirements from component subsets.

Let ℓ_{max} be the highest connection quality requirement present in the component graph. We can observe that every sub-placement of $i_\ell \in \mathcal{I}_\ell$ for $0 \leqslant \ell < H$ on $s_{\ell_{max}} \in \mathcal{S}_{\ell_{max}}$ is a valid bottom sub-placement. Decomposing component and machine group subsets at level $\ell > \ell_{max}$ 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 to higher renting costs. Hence, there is no reason to continue the decomposition process beyond ℓ_{max}.

In the example illustrated in Figure 6.2(a), the highest communication quality requirement is 2. Hence, $\ell_{max} = 2$ and the decomposition stops at level 2 (cf. table (d)).

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.

6.5.2 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 $s_{\ell+1}$. In Figure 6.1, this case is illustrated in tables (c) and (d). Sub-placements c and f, described in table (d), compose the sub-placement h, as depicted in table (b).

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. Two examples are given in Figure 6.2 with the composition of sub-placement i in Table L0 using sub-placements g and h from table (c), as well as in the composition of sub-placement g from Table L1 using sub-placements a, b, d and e from Table (d).

6.5.3 2PCAP Algorithm

A synthetic pseudo-code version of 2PCAP algorithm is presented in Algorithm 6. 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.

Algorithm 6 Pseudo-code of 2PCAP.

```
Input: level, comp subset, mg subset
Output: min cost plac
min cost plac ← ∞
if is calculated(comp subset, mg subset) then
    return plac(comp subset, mg subset)
else if level = level max then
    calculate(comp subset, mg subset)
    return plac(comp subset, mg subset)
else if level < level max then
    comp decomposition ← decompose(comp subset, level)
    mg decomposition ← decompose(mg subset, level)
    if size(mg decomposition) = 1 then
        plac ← null
        for cs in comp decomposition do
            temp plac ← 2PCAP(level + 1, cs, mg subset)
            plac ← compose(plac + temp plac)
        end for
        min cost plac ← plac
    else if size(mg decomposition) > 1 then
        plac ← null
        for cs in comp decomposition do
            min plac ← null
            for ms in mg decomposition do
                temp plac ← 2PCAP(level + 1, cs, ms)
                if cost(temp plac) < min plac then
                    min plac ← temp plac
                end if
            end for
            plac ← compose(plac + min plac)
        end for
        min cost plac ← plac
    end if
    for ms in mg decomposition do
        temp plac ← 2PCAP(level max, comp subset, ms)
        if cost(temp plac) < cost(min cost plac) then
            min cost plac ← plac
        end if
    end for
end if
return min cost plac
```

The functions used inside Algorithm 6 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*. Internally, *calculate* makes use of the multi-dimensional bin packing based heuristics proposed in Chapter 5.

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 and returns as argument and returns its size.

Between lines 30 and 36, 2PCAP computes the sub-placement of every generated component subset, which is not part of a bottom sub-placement, on machine group subsets generated at level ℓ_{max}. Those component subsets tend to be larger than the habitual component subsets from bottom sub-placements, opening the opportunity to compute better *packed* solutions.

PS: do you think it is necessary to give the pseudo of those small functions?

Case 1 is described between lines 2 and 3, Case 2 between lines 10 and 16, and Case 3 between lines 17 and 30.

Notice that a bottom sub-placement may not have a solution, and in this case, 2PCAP marks that bottom sub-placement as invalid and gives it an infinity cost.

6.5.4 Discussion

2PCAP heuristic does not compute, during the decomposition phase, all possible sub-placements. For example, in Figure 6.2, there are no bottom sub-placements computed using subsets composed by nodes $\{c_1c_2,c_3\}$, $\{c_1,c_3\}$, or $\{c_2,c_3\}$. Doing so would result in a factorial complexity, and would lead to excessively long execution times for large problems. Thus, as a result, the heuristic does not consider some sub-placements that may have an impact in the final solution quality. The approach illustrated between lines 30 and 36 of Algorithm 6, consists of calculating the sub-placement of every generated component subset, which is not part of a bottom sub-placement, on machine group subsets generated at level ℓ_{max}. Doing this will allow 2PCAP to explore further the solution space without increasing too much the algorithm‘s time complexity, even so, optimality cannot be guaranteed.

Observe, however, that 2PCAP decomposes machine groups subsets in a manner that solutions for the graph partition part (or the communication problem) of the placement problem are always found if they exist. Therefore, if a problem instance is valid, i.e., it has at least one solution, 2PCAP is able to calculate a solution.

Another 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.

PS: example if have time

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 and connecting it to the old roots.

PS: example if have time

6.5.5 2PCAP Complexity

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

Let $\mathcal{I}$ and $\mathcal{S}$ bet the sets of components and machine groups, respectively. When decomposing a component subset cs at level ℓ, *decompose* function breaks the graph G, formed by components from cs, by removing edges having weight larger than ℓ. In the worst case, *decompose* will have to verify each edge of G, 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. In the worst case, *decompose* function would return $|mg|$ other subsets, meaning that, each level, except for level 0, has $|mg|$ internal nodes. This happens when machine groups only connect at level 0. In this case, *decompose* would have to visit $|\mathcal{S}|.(\ell_{max} - (\ell - 1))$ nodes, resulting in a complexity of $O(|S|.(\ell_{max}))$.

The *decompose* function is called $\ell_{max} - 1$ times. This results in a total complexity of *decompose* for component subsets of $O(\ell_{max}.(|\mathcal{I}^2|)$. For machine group subsets, we have an arithmetic progression $a_n = a_0 + (n-1).d = |\mathcal{S}|.(\ell_{max} - (l-1))$. Hence during its execution it will have to visit $\frac{1}{2}.(\ell_{max}+1)(|\mathcal{S}|.\ell_{max} + 2.|\mathcal{S}|)$ nodes and, consequently, its complexity will be $O(|\mathcal{S}|.\ell^2_{max})$. The total complexity of *decompose* will be $O(|\mathcal{S}|.\ell^2_{max} + |\mathcal{I}^2|.\ell_{max})$.

In the composition phase of 2PCAP, sub-placement solutions are compared and only the less expensive are used to compose the final placement. To do so, it is necessary to verify each of the sub-placements computed. The complexity of that operation is $O(|\mathcal{I}|.|\mathcal{S}|)$.

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 5). Those heuristics have a complexity of $O(t.\log t.c)$, 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.

As we previously discussed, 2PCAP can try sub-placements of component subsets over machine group subsets decomposed at ℓ_{max}. This is illustrated between lines 30 and 36 of Algorithm 6. The complexity of this snippet is $O(|\mathcal{S}|.|\mathcal{I}|.\ell_{max})$.

6.5.6 Examples

In this section, we present some examples of placements using 2PCAP.

PS: use latex like counter

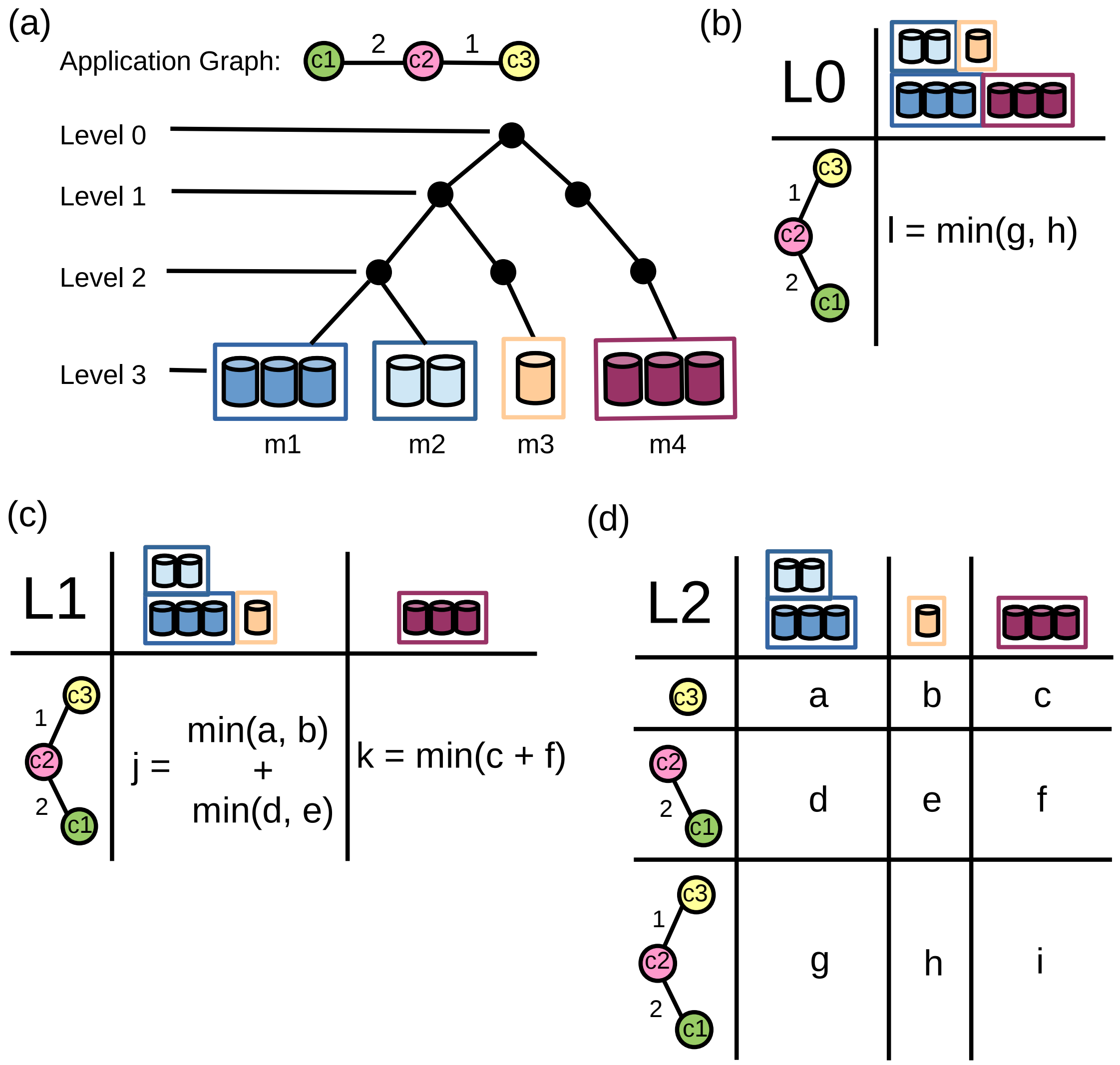

Figure 6.3: Complete Placement Example #1: Placement previously described in Figure 6.2. In this example the solution space exploration extension discussed in Section 6.5.3 is added.

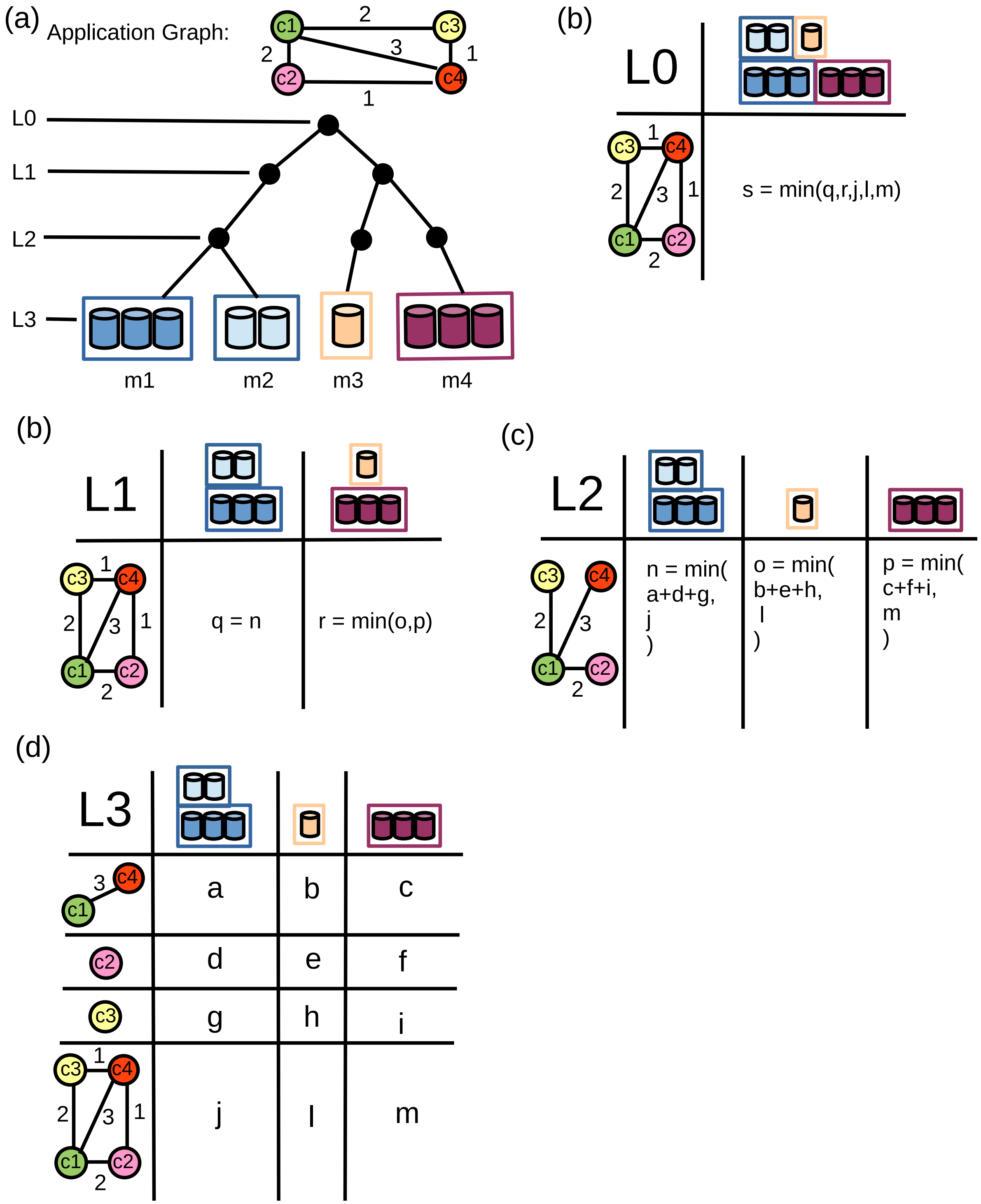

Figure 6.4: Complete Placement Example #2.

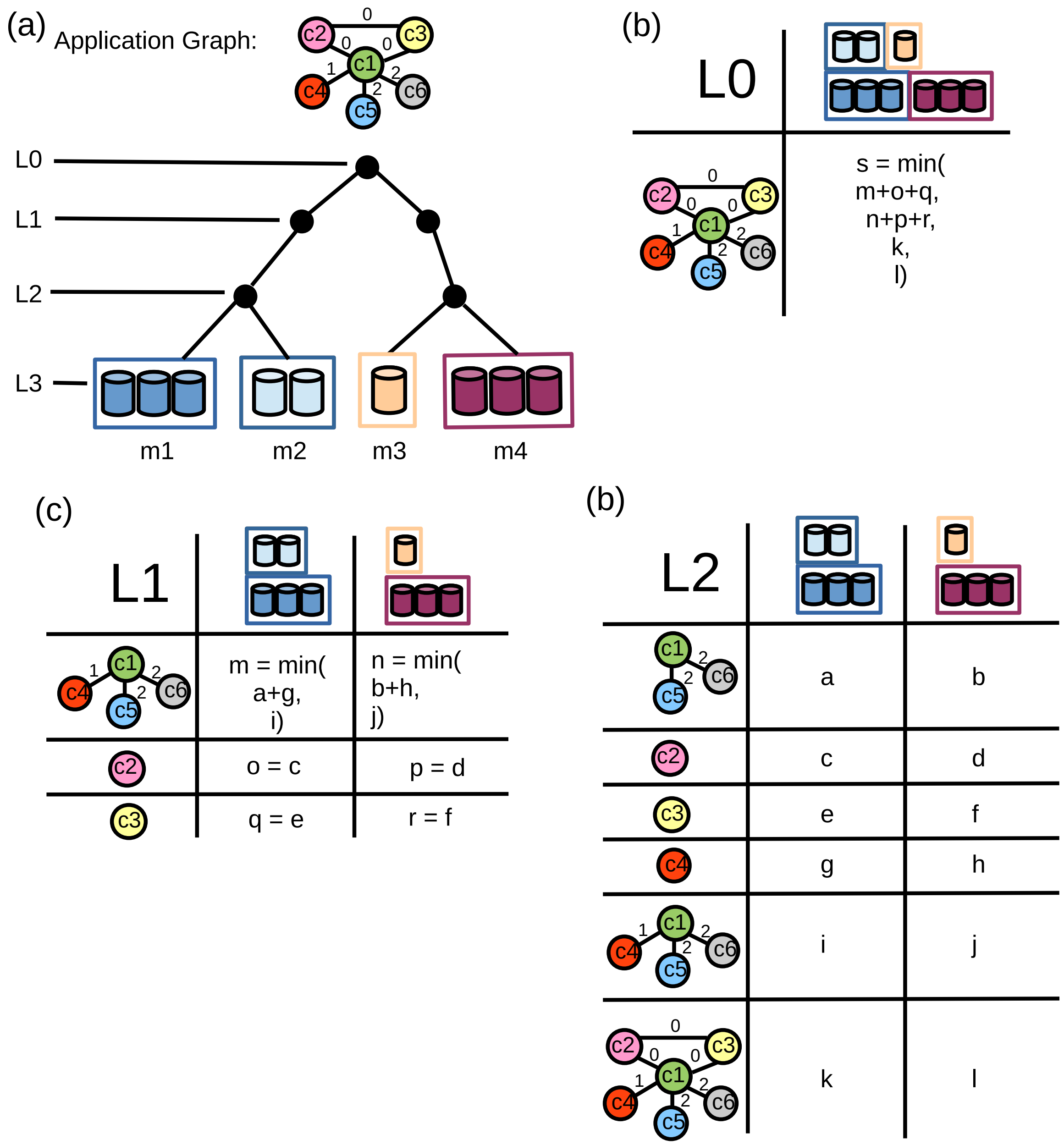

Figure 6.5: Complete Placement Example #3.

6.6 Evaluation

It would be interesting to compare the solutions computed by 2PCAP to optimal ones. However, as we previously discussed in Section 6.3, the communication aware placement of distributed applications on the multiple clouds problem (CAPDAMP) is NP-Hard. 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 ones. 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.

6.6.1 Methodology

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 6.2 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 6.2: 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 6.3). 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

Dimension	(i)	(ii)	(iii)	(iv)	(v)	(vi)
Requirements	$800 \sim 3k$	$1 \sim 16$	$1 \sim 32$	$50 \sim 3.5k$	$5 \sim 30$	$1 \sim 8$
Capacities	$1k \sim 3.5k$	$2 \sim 32$	$2 \sim 40$	$150 \sim 4k$	$10 \sim 80$	$1 \sim 16$

Table 6.3: Intervals of dimension data generation.

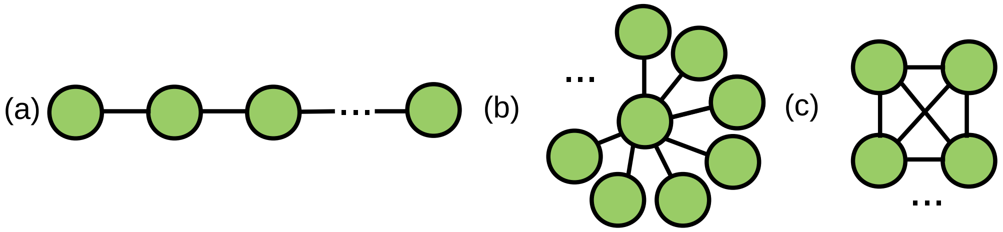

Figure 6.6: Schemas of part of the generated application topologies. (a) *line*, (b) *star* and (c) *full connected*.

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 6.6), 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 value for dimension d (cf. Table 6.3). 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 where 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[1].

6.6.2 2PCAP Performance on Small Problems

This section makes use of Class A problems instances (cf. Table 6.2) to calculate a set of optimal points and to compare them to solutions computed by 2PCAP.

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

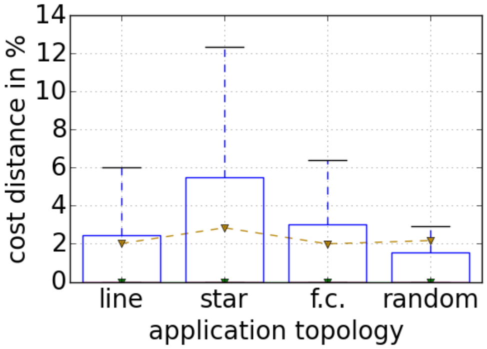

Figure 6.7: Cost distances as percentage of optimal between 2PCAP solutions and optimal solutions aggregated by application topology type. The median is represented by a green solid line and the average by a brown dashed line.

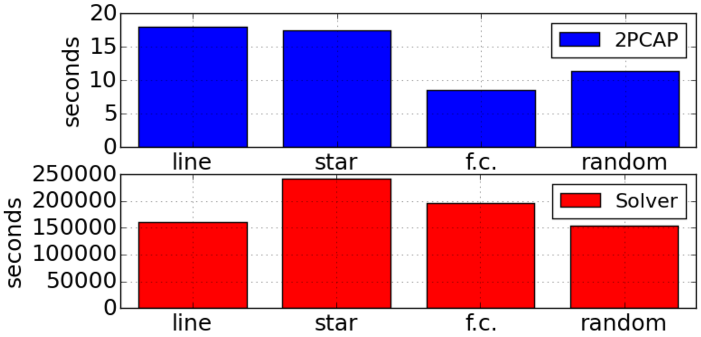

Figure 6.8: Sum of execution times in seconds from 2PCAP and SCIP solver executions aggregated by application topology. Only the execution time for problem instances successfully solved by SCIP are computed.

We integrated to our test platform the *SCIP* solver [1], a framework for constraint integer programming and branch-cut-and-price (the formulation is described in Section 6.2.2). 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 6.7 illustrates the *cost distance* from solutions computed by 2PCAP to the optimal ones as a percentage of the latter for problem instances successfully solved by SCIP. Cost distances are grouped by application topology.

In Figure 6.7 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 6.8 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, in the median, the solutions are optimal.

6.6.3 2PCAP Performance on Large Problems

This section evaluates 2PCAP on larger scenarios. To overcome the scalability issues of generating optimal solutions for those problem instances, we use more scalable approaches as baseline algorithms.

As a first approach, we implemented a Simulated Annealing (SA) meta-heuristic for the CAPDAMP. We used the Python module *Simanneal* [67] and computed placements for the medium sized Class B problem instances. Despite giving a timeout of 1 hour to SA per problem instance, the meta-heuristic manages to solve only around 12% of all Class B problem instances. This happens mainly because of the size of CAPDAMP‘s search space which has a large amount of invalid solutions and many isolated local optima solutions.

In Section 6.3, we discussed some heuristics from the state of the art which used *relaxed* versions from NP-hard problems to quickly compute solutions. In a similar way, we use 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 SA algorithm initialized with the best solution among those calculated by 2PCAP and other multi dimensional bin packing based heuristics, discussed in Chapter 5, with one hour timeout per problem instance.

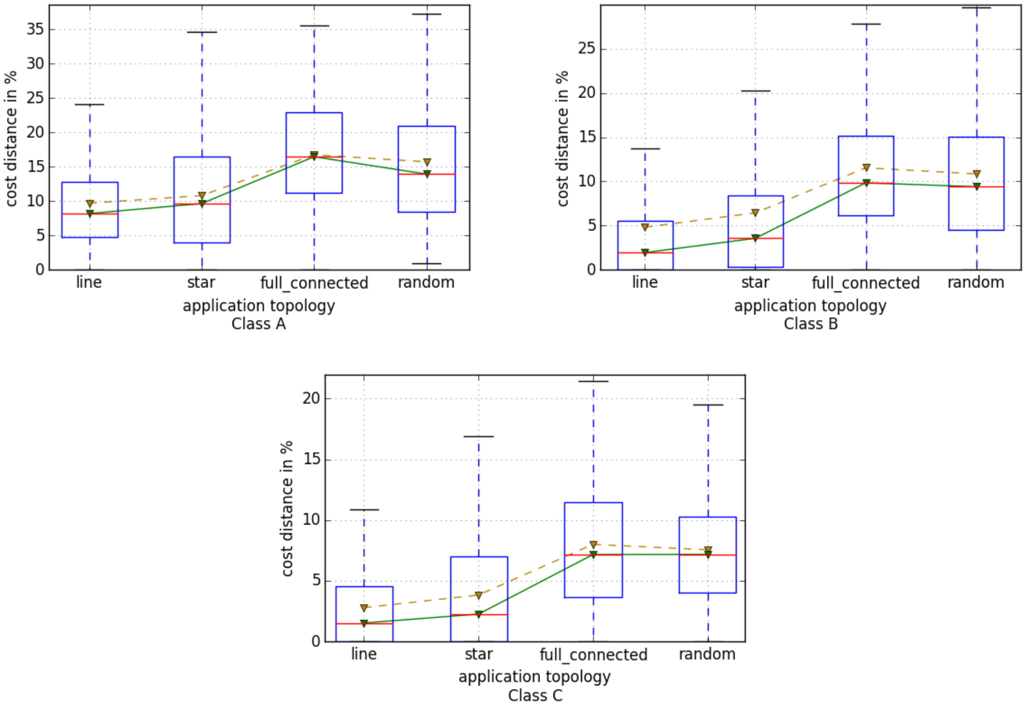

Figure 6.9: Cost distances between 2PCAP and Simulated Annealing solutions for problem instances from classes A, B and C. The green solid line is the median and the brown dashed line is the average.

Figure 6.9 illustrates the evolution of cost distances between 2PCAP and SA solutions. From left to right, we describe that metric for problem instances from classes A, 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 SA 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 and averages, particularly stay within similar intervals, indicating that even on large scenarios, the performance of 2PCAP is consistent. Also, as illustrated in Figure 6.10, the execution times from 2PCAP, grouped by application topology, for the three experiences scenarios is still reduced. It manages to compute solutions in at most 82 seconds per problem instance. In Figure 6.10, we can observe a gap between the 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 connectivity from full connected and random topologies schemas. 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.

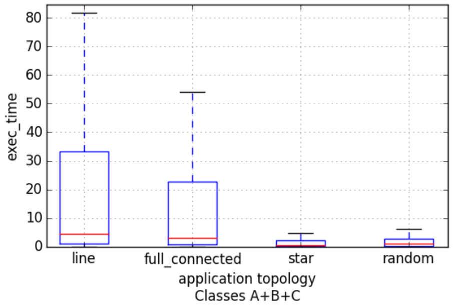

Figure 6.10: Execution times in seconds (per problem instance) aggregated by application topology taken by 2PCAP to compute solutions for problem instances from classes A, B and C.

6.7 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 producing good quality solutions very quickly.

Naturally, in spite of being designed originally for Cloud based infrastructures, 2PCAP could be used to compute placements over other distributed infrastructures, like Grid and Cluster infrastructures. Furthermore, other communication based metrics like bandwidth or throughput could replace latency for representing communication qualities.

It is important to notice, however, that 2PCAP is only capable of calculating *initial* placements. This issue is addressed in the next chapter where we increment our models and propose a extended reconfiguration-aware version of 2PCAP capable of taking into consideration pre-deployed components when calculating placements.