Combining hardware nodes and software components ordering-based heuristics for optimizing the placement of distributed IoT applications in the fog

Ye Xia, Xavier Etchevers, Loic Letondeur, Thierry Coupaye, Frédéric Desprez

Combining Hardware Nodes and Software Components Ordering-based Heuristics for Optimizing the Placement of Distributed IoT Applications in the Fog

Ye Xia
Orange Labs
Meylan, France
ye.xia@orange.com

Xavier Etchevers
Orange Labs
Meylan, France
xavier.etchevers@orange.com

Loïc Letondeur
Orange Labs
Meylan, France
loic.letondeur@orange.com

Thierry Coupaye
Orange Labs
Meylan, France
thierry.coupaye@orange.com

Frédéric Desprez
UGA, Inria, CNRS, LIG
Grenoble, France
Frederic.Desprez@inria.fr

ABSTRACT

As fog computing brings compute and storage resources to the edge of the network, there is an increasing need for automated placement (i.e., selection of hosting devices) to deploy distributed applications. Such a placement must conform to applications' resource requirements in a heterogeneous fog infrastructure. The placement decision-making is further complicated by Internet of Things (IoT) applications that are tied to geographical locations of physical objects/things. This paper presents a model, an objective function, and a mechanism to address the problem of placing distributed IoT applications in the fog. Based on a backtrack search algorithm and accompanied heuristics, the proposed mechanism is able to deal with large scale problems, and to efficiently make placement decisions that fit the objective—to lower placed applications' response time. The proposed approach is validated through comparative simulations of different combinations of the algorithms and heuristics on varying sizes of infrastructures and applications.

CCS CONCEPTS

• **Software and its engineering → Distributed systems organizing principles**;

KEYWORDS

placement, heuristics, fog computing, IoT

ACM Reference Format:
Ye Xia, Xavier Etchevers, Loïc Letondeur, Thierry Coupaye, and Frédéric Desprez. 2018. Combining Hardware Nodes and Software Components Ordering-based Heuristics for Optimizing the Placement of Distributed IoT Applications in the Fog. In *Proceedings of SAC 2018: Symposium on Applied Computing (SAC 2018).* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3167132.3167215

SAC 2018, April 9–13, 2018, Pau, France

ACM ISBN 978-1-4503-5191-1/18/04…$15.00
https://doi.org/10.1145/3167132.3167215

1 INTRODUCTION

After a decade of continuous growth of cloud computing, fueled by mega data centers that concentrate computing and storage resources for on-demand enterprise and web applications, we can now witness the emergence of more distributed paradigms, such as fog computing. Fog computing [1, 2] is typically motivated by Internet of Things (IoT) applications for which it appears more adequate to put computing, storage, interaction, and control close to sensors and actuators rather than only in remote mega data centers. By making use of devices in the (extreme) edge of the network, fog computing gains the ability to satisfy IoT applications that require low response time, data privacy enforcement, control over the amount of data commuting by the core network, and so on. However, satisfying these requirements must be based on an intelligent selection of hosting devices, in other words, the applications must be placed properly.

Placement decision-making (i.e., to decide the way to host distributed applications on a set of devices) is known to be an NP-hard problem [3, 4]. Compared with the cloud, placement problems are even more complex in the context of fog computing and IoT applications: i) The fog contains numerous heterogeneous devices (in terms of network position, hardware configuration, operating system, etc), which makes it harder to get a proper placement. ii) IoT applications are based on sensing and actuating services provided by physical objects, which demands placement approaches to take these objects' locations into account. iii) Specific (e.g., time-sensitive, privacy-sensitive) applications' requirements must be satisfied.

This paper makes the following contributions:

- A model that formalizes the problem of placing a set of IoT applications in a fog infrastructure, and an objective function that aims to minimize applications' average response time.
- Two placement algorithms which guarantee to find a placement that satisfies considered applications' requirements if such a placement exists.
- Two combinable heuristics that accelerate the placement decision-making process, make the placement algorithm much more scalable, and improve placement result quality according to the objective function.

- A simulation-based evaluation that benchmarks result quality and execution time of different combinations of proposed algorithms and heuristics under growing scales of infrastructures and applications.

Compared with other placement approaches, one of the innovations introduced in this work concerns the proposition of "Dedicated Zone" and "Anchor". These concepts specifically address IoT applications tied to physical objects that are geographically localized in the real world.

The rest of this paper is organized as follows. Section 2 introduces a motivating example. Section 3 discusses related works. Section 4 details the proposed model, objective function, placement algorithms, and heuristics. The proposal evaluation is presented in Section 5. Section 6 concludes and discusses future works.

2 MOTIVATING EXAMPLE

Inspired from [5], our motivating example concerns the placement of a distributed IoT application "Smart Bell" in a fog infrastructure.

2.1 Fog Infrastructure

As depicted in Figure 1, the fog infrastructure provides resources in three network layers: *cloud layer*, *edge layer*, and *extreme edge layer*. Each layer contains devices connected by network links. Generally, from *extreme edge layer* to *cloud layer*, devices are more and more capable and stable, while losing locality and reactivity.

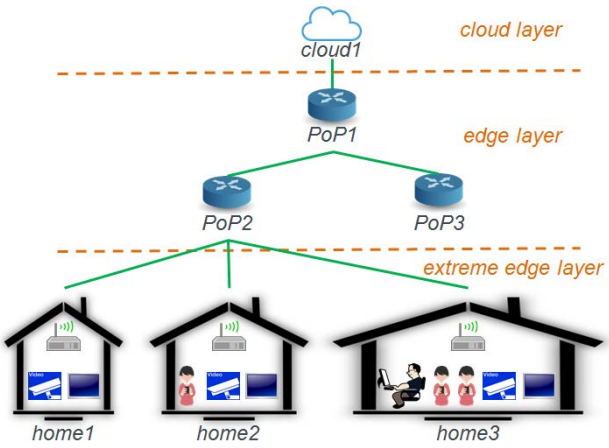

Figure 1: Infrastructure Example (each home is equipped with a camera, a screen, and a wireless gateway. Moreover, *home*2 has also a mobile, *home*3 has two mobiles and a PC)**.**

The infrastructure example in Figure 1 contains a cloud in *cloud layer*, three network Points of Presence (PoP[1]) in *edge layer* and a number of end devices in three homes in *extreme edge layer*. End devices in each home are connected via a wireless gateway (box). Cameras and screens do not have any resource to host applications, but they respectively provide image capturing services and displaying services. Except cameras and screens, each device provides certain amounts of processing and storage resources. Each link is characterized by a certain network latency and a certain bandwidth.

[1]A PoP designates a set of telecom operator's equipment in charge of routing data flow between networks. As a simplification, a PoP is considered as a single device in this paper.

2.2 Smart Bell Application

As a smart home application, Smart Bell notifies a home when it gets a visitor. By trying to recognize visitors, each notification made by Smart Bell is based on the relationship between the visitor and visited home inhabitants. Through accessing each home's database (DB) that stores images of the home inhabitants and their friends, a visitor can be recognized as an inhabitant, a friend, or neighbor's friend. If a recognition fails, the visitor will be identified as a stranger. Considering a given home, when a visitor rings at the door, Smart Bell verifies if he/she is an inhabitant or a friend according to the DB of the visited home; if not, through communicating with neighbors' DBs, it verifies if the visitor is a neighbor's friend; otherwise, the visitor is considered as a stranger. Smart Bell makes a specific notification for each identified relationship. It also activates an alarm when a stranger keeps ringing doorbells around.

Technically speaking, a distributed application is made up with software elements known as "components". Each component acts as a functional and deployment unit. Smart Bell contains several types of components, as listed in Table 1.

Comp Type	Functionality
Extractor	extracting human faces in captured images
DB	storing inhabitants' and friends' information (e.g., face images) for recognition
Recognizer	trying to recognize visitors
Decider	making reaction decisions to visitors
Executer	generating and sending commands to inform inhabitants through screens
Recorder	storing strangers' information and counting how many times they appear

Table 1: Component Types of Smart Bell.

An example instance of Smart Bell is illustrated in Figure 2. It serves three homes in a same neighborhood. Smart Bell needs cameras to capture images, and screens to display notifications. In the graph, each node in blue is a component, each node in green is a camera or a screen, each edge is a communication channel.

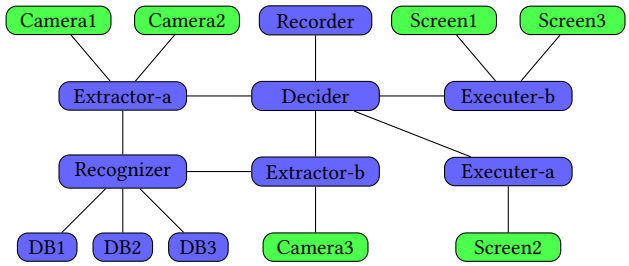

Figure 2: Example Instance of Smart Bell.

Smart Bell is privacy-sensitive: each home's DB stores private data, and must be placed in devices of the home. Without special privacy requirement, other components can be placed anywhere.

To be executed properly, each component of Smart Bell requires certain processing and storage resources, and each communication channel needs certain bandwidth and network latency. In terms of network latency, traditional clouds in the core network are too far from end devices to satisfy Smart Bell. Benefiting from close proximity to end devices, the fog can be well suitable.

2.3 Challenges

The placement of distributed IoT applications, such as Smart Bell, in a fog infrastructure exhibits the following challenges:

- **Heterogeneity**: a fog infrastructure is composed of heterogeneous devices and links whose resource capabilities strongly differ. Moreover, these devices are in different network layers and positions.
- **Constraint diversity**: IoT applications are subject to many kinds of constraints/requirements in order to execute properly. These constraints concern i) consumable resources (e.g., processing and bandwidth capabilities) and non-consumable properties (e.g., network latency, privacy), ii) different kinds of entities (e.g., component, communication channel), which result in high diversity.
- **Locality and geo-distribution**: IoT applications are by definition tied to physical objects. As these objects can be spread over different geographical locations, an IoT application can span multiple localization areas.
- **Multi-tenancy**: as a fog infrastructure is not dedicated to a single application, the placement of multiple applications (and/or applicative instances) needs to be managed simultaneously.
- **Scalability**: in order to be reactive to deployment requests, placement decisions must be made time-efficiently. However, the complexity of placement problem dramatically increases with the infrastructure size (i.e., number of devices) and applications' size (i.e., number of components), which makes it hard to deal with large-scale problems.

3 RELATED WORK

A number of prior researches are related to this study. However, most of them are in the context of distributed clouds. Only [4] and [6] focus on fog computing; as generic approaches, [3] and [7] suit both fog and cloud. From application's point of view, only [4] deals with IoT applications, the others are rather for traditional cloud applications. These related works are classified into exact algorithms, metaheuristics, and heuristics.

3.1 Exact Algorithms

Many works, such as [3, 6], formulate the placement problem with Integer Linear Programming (ILP). ILP expresses a problem with mathematical constraints and an objective function. Such an expressed ILP problem can be solved by generic ILP-solvers which guarantee to return the optimal result.

Another exact algorithm is discussed in [4]. The algorithm finds out all valid (i.e., conforming to constraints) placements based on a backtrack search algorithm. Then among found valid placements, it returns the one that best fits the objective.

Given a problem, exact algorithms always deliver the optimal placement. However, suffering from high execution time that is exponential with respect to the problem size, exact algorithms are hardly scalable.

3.2 Metaheuristics

High level metaheuristics have also been used to solve placement problems. Based on Hill Climbing algorithm, [8] and [9] improve an initial valid placement iteratively. At each iteration, one component is re-placed to better fit a predefined objective function. This algorithm can be much faster than exact algorithms. Whereas it assumes that the initial valid placement can be found easily in a random manner. Such an assumption can work for cloud applications designed to execute on homogeneous resource-rich data centers. However, in the context of fog, stricter constraints of new applications and numerous heterogeneous devices make it hard to get a valid placement, and the random manner can be much more time-consuming. Furthermore, starting from a random initial placement, this approach usually traps into a local optimum.

[10] and [11] make placement decisions using Ant Colony Optimization, in which placements are generated and tested dynamically. The placement generation is based on each component's probabilities to be placed in each device. These probabilities are tuned dynamically according to generated placements' test results. This approach returns the best placement among tested ones when a predefined timeout is reached. Consequently, its result quality is strongly related to the timeout value. Furthermore, it has no guarantee to find an existing valid placement.

3.3 Heuristics

Many approaches, such as [7, 12], deal with the placement problem as a bin-packing problem without considering network resource constraints. Heuristics like Best-Fit, Worst-Fit are proposed in these approaches to consolidate or balance hosts' load. Another heuristic [13] chooses candidate hosts with higher cost-efficiency priorly to minimize financial renting cost. Unfortunately, these heuristics do not consider applications' response time.

In the context of distributed clouds, [14] selects a subset of data centers (DC) to host an application. In order to minimize application's response time, it tries to simultaneously minimize two values: i) selected DCs' distance to considered application's end users, ii) selected DCs' diameter (i.e., the longest communication path between selected DCs). A proposed heuristic selects firstly the closest DC to the end users. Then, as long as resource requirements of the application are not met, it selects a new DC that increases the diameter the least. This work minimizes the diameter to limit maximal latency between application's components, but the communication details between the components are not taken into account. Moreover, using a single location to represent all end users, this approach is not suitable for geo-distributed applications.

[15] uses geographic coordinates to locate DCs, end users, and application's components. Problems concerning multiple end user

locations are supported. Initially, each component is assigned to the geographic center of end users it communicates with. Then, through an iterative optimization process, each component's location is updated iteratively to the center of end users and components it communicates with. Lastly, each component is placed in the closest DC with enough resource to host it. This work makes placement decisions with the help of city-level geographic coordinates obtained from IP addresses. It can work in the context of distributed clouds, but not in the fog which has finer granularity of device locations.

None of these works seems to be able to tackle all the challenges introduced in Section 2.3. New mechanisms must be developed to address them.

4 CONTRIBUTIONS

This section is devoted to the proposal to overcome the challenges in Section 2.3. Section 4.1 proposes models, constraints, and an objective function to formulate placement problems. Section 4.2 introduces two placement algorithms: Exhaustive Search and Naive Search. Section 4.3 and Section 4.4 improve Naive Search with two heuristics that respectively order devices and components. Section 4.5 discusses how the placement algorithms and heuristics can be combined.

4.1 Model and Problem Formulation

A fog infrastructure is composed of devices and network links. There are two kinds of devices: i) *fog nodes* (e.g., cloud, PC), which provide processing and storage resources; ii) *appliances* (e.g., camera, screen), which provide sensing and actuating services. Only fog nodes can be used as hosting devices, appliances do not provide resources to host applications. All the devices are connected by links which provide network resources.

An application consists of applicative components, logical bindings, and appliances[2]. A *component* is a software element that can be executed on one fog node. A *binding* is a communication channel that connects a couple of components or a component and an appliance.

An infrastructure (resp., application) is modeled as a graph. Each node of the graph is a device (resp., a component or an appliance). Each edge is a network link (resp., binding). The model is defined and summarized in Table 2.

In the model, each component is characterized with some requirements. *ReqCPU*, *ReqRAM*, and *ReqDISK* respectively indicate CPU, RAM, and disk capabilities that a component needs. In order to respect components' specific requirements on privacy (e.g., in a certain home) and software/hardware properties (e.g., operating system, being equipped with bluetooth), each component has a *Dedicated Zone* (DZ). A DZ is a deployment area specified by application developer. If not specified, a component's DZ contains the whole infrastructure $Infra$. Actually, each DZ can be interpreted to be a set of fog nodes. Bindings are characterized with requirements as well. $bind_i.ReqBW$ designates $bind_i$'s bandwidth requirement. $bind_i.ReqLAT$ indicates maximal network latency that $bind_i$ can accept. Components and appliances of an application app_i are denoted by $app_i.components$ and $app_i.appliances$, respectively. *Apps* is regarded as a single placement request, all the applications in *Apps* will be placed or refused together.

[2]Considering that a sensing/actuating service is tied to and must be executed on its appliance, both services and hardware of appliances are named "appliance" in this paper.

$Infra$	the target infrastructure
$node_i$	a fog node of $Infra$
$node_i.CPU$	$node_i$'s available CPU capacity
$node_i.RAM$	$node_i$'s available RAM capacity
$node_i.DISK$	$node_i$'s available disk capacity
$appliance_i$	an appliance of $Infra$
$link_i$	a network link of $Infra$
$link_i.LAT$	$link_i$'s network latency
$link_i.BW$	$link_i$'s available bandwidth capacity
Apps	a set of applications to deploy
$comp_i$	a component of an application in *Apps*
$comp_i.ReqCPU$	$comp_i$'s CPU requirement
$comp_i.ReqRAM$	$comp_i$'s RAM requirement
$comp_i.ReqDISK$	$comp_i$'s disk requirement
$comp_i.DZ$	$comp_i$'s Dedicated Zone
$bind_i$	a binding of an application in *Apps*
$bind_i.ReqLAT$	$bind_i$'s latency requirement
$bind_i.ReqBW$	$bind_i$'s bandwidth requirement
app_i	an application in *Apps*
$app_i.components$	all the components of app_i
$app_i.appliances$	all the appliances of app_i
$(comp_i, node_j)$	a mapping of $comp_i$ to $node_j$
$(Apps, Infra)$	a placement of *Apps* in $Infra$

Table 2: Summary of Notations.

As denoted by $(Apps, Infra)$, a placement is a mapping of a set of applications onto the infrastructure. More precisely, a placement is a mapping of all the components onto fog nodes of $Infra$, in which each component is mapped to one and only one fog node. Therefore, $(Apps, Infra)$ can be expressed as a set containing each component $comp_i$'s mapping $(comp_i, node_j)$. For an *Apps* with n components,

$$(Apps, Infra) = \begin{pmatrix} comp_1, node_i \\ comp_2, node_j \\ \dots \\ comp_n, node_k \end{pmatrix}$$

When all the components are placed in fog nodes, correspondingly, each binding is placed in a communication path composed of a set of network links. A *binding's latency* is the network latency of the communication path in which this binding is placed.

An applicable placement must satisfy the following constraints: (1) each component is placed in its DZ; (2) required CPU, RAM, and DISK in each fog node do not exceed its capacities; (3) required bandwidth in each network link does not exceed its capacity; (4) each binding's latency does not exceed its requirement. A placement that satisfies all these constraints is a *solution* to the placement problem. A problem can have multiple solutions. Within the solutions, only one can be selected as the final placement decision.

This work aims to minimize average response time of a set of applications. Response time of a request is the time spent from the request sending until reception of its response. It is composed of communication time (i.e., time spent to transfer messages) and execution time (i.e., time spent within components for calculation). By assuming that the execution time does not change with placement, this work focuses on minimizing communication time. Higher bandwidth and lower network latency help to reduce messages' transferring time. In the model, each binding's bandwidth is predefined, whereas bindings' latencies depend on applied placement. To make a selection among solutions, our objective function is proposed as follows:

$$min : WAL = \sum_{bind \in Apps} \frac{bind.ReqBW}{total\_BW} \times bind.Lat$$

$$total\_BW = \sum_{bind \in Apps} bind.ReqBW$$

$total\_BW$ is the total bandwidth requirement of all the bindings. $bind.Lat$ is $bind$'s latency. The objective function aims to minimize *Weighted Average Latency* (WAL) of considered applications. Considering that a binding with higher bandwidth requirement impacts more on an application's response time, each binding's latency is weighted by the proportion of the binding's bandwidth requirement in $total\_BW$. Through minimizing WAL, the latencies of bindings, especially the ones with high bandwidth requirement, can be minimized, which helps to reduce communication time spent for each request. Ideally, given a placement problem, the solution with minimal WAL should be selected as the placement decision. The evaluation of this objective function is given in Section 5.2.

4.2 Exhaustive Search and Naive Search

For a placement problem with m fog nodes and n components, there exists m^n possible placements. The search space composed with all these placements can be represented using a tree data structure. In such a tree, each placement is a leaf whose depth is n. Any branch from the tree root to a leaf describes the way to build a placement by mapping successively each component to a fog node. An example of the tree data structure is illustrated in Figure 3.

A backtrack search process is used to find solutions in such trees. This process deals with a set of applications: components of all considered applications are mixed and placed one by one. When it tries to place a component, all kinds of constraints are verified for this component and previously placed ones. As depicted by the orange curve in Figure 3, when the process passes a constraint verification, it continues with the next component. If all the components are successfully placed, it implies that a solution has been found. When a constraint verification fails, the process tests the next fog node to place the current component. If all possible fog nodes are tested for a component, and no suitable fog node is found, the process backtracks to the previous component to test other possibilities (i.e., change hosting fog node of the previous component). When the process tries to backtrack from the first component, it means that the search space has been traversed.

Because a component must be placed in its DZ, fog nodes out of this DZ are not tested when trying to place considered component.

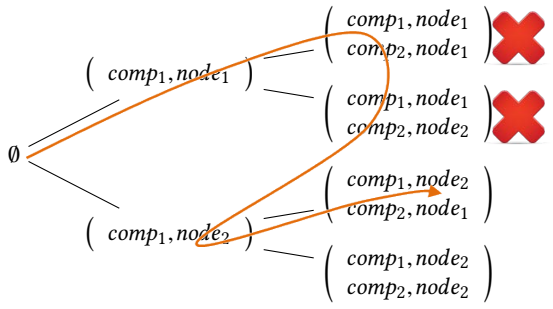

Figure 3: Search Process Example (the backtrack search begins by testing to place $comp_1$ in $node_1$, which passes; then it fails to place $comp_2$ in $node_1$ and $node_2$ because of constraint violations; having tested all possible fog nodes for $comp_2$, the search backtracks to $comp_1$ and tries to place it in $node_2$. The test passes, and the search continues to place $comp_2$).

If a constraint verification fails on a tree node during the search process, all subtrees of this tree node are pruned (i.e., not tested any more). Effectively, all the placements in pruned subtrees are based on the failed tree node, which can not lead to any solution.

Based on the depth-first search process above, two search algorithms are developed: i) Exhaustive Search, which always returns the optimal solution with minimal WAL; ii) Naive Search, which returns the first solution found.

Exhaustive Search ends only when the search space is traversed, so that it always visits all existing solutions and guarantees to select the one that minimizes WAL. Naive Search ends as soon as a solution is found. If no solution exists, it ends when the search space is traversed. By continuing the search until a solution is found, Naive Search guarantees to find an existing solution.

Regarding the search space size, Exhaustive Search is not feasible for large-scale problems. Naive Search can avoid always traversing the search space, but it has no guarantee on returned solution's quality (i.e., WAL value). To improve Naive Search, two heuristics are designed and will be introduced in the following.

4.3 Anchor-based Fog Nodes Ordering

In Naive Search, fog nodes are tested in random order. As a result, WALs of returned solutions distribute randomly. Through taking care of fog nodes test order, the heuristic Anchor-based Fog Nodes Ordering (AFNO) aims at lowering WAL of returned solutions.

Consisting of fog nodes in different network positions, the fog has the potential to localize applications. In a localized application, each component should be close to appliances and components it communicates with.

Within a component $comp$'s DZ, the fog node that best localizes $comp$ in terms of minimizing WAL is $comp$'s *anchor*. An anchors-based placement (i.e., map each component to its anchor) does not guarantee to satisfy constraints concerning hardware resources (i.e., constraints 2, 3, and 4 in Section 4.1). Before the search process, AFNO sorts each component's candidate fog nodes (i.e., fog nodes in its DZ) in ascending order of network latency to the component's anchor. Through AFNO, the first solution found will be

close to the anchors and thus helps to minimize WAL.

Anchors of each application are calculated through Algorithm 1. Inputs of the algorithm contain an infrastructure model $infra$ and an application *app*.

Algorithm 1: GetAnchors

```
   Input: infra, app
1  center ← infra.centerNode( app.appliances() );
2  for each comp ∈ app.components() do
3      anc[comp] ← comp.closestNodeInDZ( center );
4  compList ← app.components();
5  while compList ≠ ∅ do
6      comp ← compList.firstElement();
7      compList.remove(comp);
8      newAnc ← calculateAnc( comp, anc, infra );
9      if anc[comp] ≠ newAnc then
10         anc[comp] ← newAnc;
11         compList ← compList ∪ comp.boundComps();
12 return anc;
```

Line 1–3 localizes *app* based on its appliances. In line 1, *app.appliances*() returns all the appliances of *app*. Through $infra.centerNode()$, *center* is assigned as the fog node that minimizes average network latency to *app*'s appliances. In line 2, *app.components*() returns all the components of *app*. In line 3, *comp.closestNodeInDZ*() returns the fog node that minimizes network latency to *center* in *comp*'s DZ. Through line 1–3, each component's anchor is assigned as the closest fog node to *center* in the component's DZ.

Line 4–11 further localizes each component. *compList* stores the components to get anchor update tests, and it is initialized to contain all the components of *app* (line 4). Components in *compList* are tested one by one. At each iteration (line 6–11), the first component in *compList* is selected and removed (line 6–7), then the algorithm tries to update selected component *comp*'s anchor (line 8–10). When *comp*'s anchor changes, bound components of *comp* (i.e., components that *comp* communicates with) can change with it. Therefore, these bound components are added into *compList* again to test later (line 11). Once *compList* is empty, the calculation of anchors finishes and the algorithm returns (line 12).

According to the definition of WAL, for a component *comp*, if a new anchor leading to smaller WAL exists, the new anchor must be closer to one of the devices *comp* communicates with. Therefore, in each anchor update test (line 8), *CalculateAnc*() evaluates all the fog nodes between *comp* and *comp*'s bound components (i.e., fog nodes on communication paths from *comp*'s current anchor to bound components' anchors) and between *comp* and *comp*'s bound appliances. Finally, *CalculateAnc*() returns the fog node that minimizes WAL within evaluated ones.

AFNO guides the search process to minimize WAL of the first solution found. Moreover, it can also accelerate such a search. The influence of AFNO is evaluated in Section 5.

4.4 Dynamic Components Ordering

Naive Search has two test orders: the order of fog nodes and the order of components. The former is treated by AFNO. Dynamic Components Ordering (DCO) is responsible for the later to further accelerate the search.

In Naive Search, components are placed one by one, and placing a component depends on placed ones[3]. As a result, components' placement order has an impact on the test number in Naive Search, especially when it has to backtrack. For example, considering n components ordered as $c_1, c_2, \ldots, c_i, \ldots, c_j, \ldots, c_n$. If former placed c_i makes it impossible to place c_j, after finding out that no fog node suits c_j, Naive Search has to backtrack from c_j until c_i before being able to change c_i's host. Without the knowledge that failures of placing c_j concern c_i, before arriving at c_i, Naive Search must test all possible placements of $\{ c_{i+1}, c_{i+2} \ldots c_j \}$, which lead to $\|c_{i+1}\| \times \|c_{i+2}\| \times \cdots \times \|c_j\|$ tests in the worst case ($\|c\|$ is the number of fog nodes in c's DZ). Such a huge amount of tests is possible to be avoided if c_j is ordered in front of c_i. Under the new order, c_j is placed without constraints introduced by c_i. However, c_i gets stronger constraints compared with the original order. If c_i can still get a suitable fog node, Naive Search will successfully place c_i and c_j without backtrack. When Naive Search succeeds to place n components without backtrack, it needs at most $\|c_1\| + \|c_2\| + \cdots + \|c_n\|$ tests.

Since fog nodes and links can be resource-constrained in the fog, it is quite probable to encounter backtracks during a search process. To avoid backtracks, DCO adjusts components' order dynamically. When the search process fails to place a component *comp*, without knowing which components constrain *comp*, DCO sets *comp* as the first component to place. Then, DCO redoes the search process all over again under this new component order. Because such a new order can still cause backtracks, it is possible that DCO gets multiple rounds of reordering. Even if so, regarding the huge amounts of tests in backtracks, DCO can still outperform Naive Search.

To avoid infinite loop, each component order can be tested only once. When DCO is going to produce an order already tested, instead of reordering the components, it backtracks as in Naive Search, so that the possibility to traverse the search space is retained, and the guarantee to find an existing solution is kept. The search process with DCO is depicted in Figure 4.

DCO accelerates the search for finding out a solution, the performance of searches with/without DCO are compared in Section 5.3.

4.5 Combination of Algorithms and Heuristics

Two placement algorithms Exhaustive Search and Naive Search are discussed in Section 4.2. Based on Naive Search, two heuristics AFNO and DCO are proposed, which enable three placement algorithms: heuristic search based on AFNO / DCO / combined AFNO and DCO.

AFNO and DCO improve and accelerate only the search for finding one solution. However, Exhaustive Search traverses the search space and finds out all existing solutions. As a result, AFNO and DCO do not suit Exhaustive Search, and the heuristic searches to be discussed are all combination of the heuristics and Naive Search.

[3]The dependency is caused by concurrent consumption of limited resources in the infrastructure, and by constraints on binding's maximal latency.

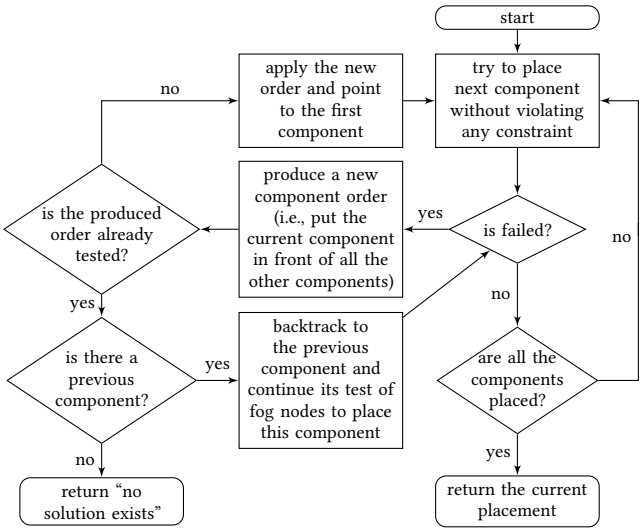

Figure 4: Search Process with DCO.

Through AFNO, local fog nodes are tested priorly when placing a component. Regarding that the localization is in terms of minimizing WAL, the first solution found must get lower WAL compared with Naive Search. A localized component is close to the appliances and components it communicates with, so that its bindings' latency requirements are prone to be satisfied. Based on localization, AFNO can guide components to network positions close to a solution, and thereby accelerates the search. However, because of anchors-calculating and fog nodes-ordering, AFNO introduces an overhead to localize the components.

With DCO, backtracks that result in huge amounts of tests can be avoided. For each avoided backtrack, a new search process under a new component order must be launched. These new orders help to find a solution without backtrack, which can reduce search's test number by orders of magnitude especially for large-scale problems.

The combination of AFNO and DCO should make the placement decision-making process more scalable. Moreover, this combination also allows to get solutions with lower WAL.

5 EVALUATION

According to Section 4.5, there are five placement algorithms to evaluate and compare—Exhaustive Search (Exhaustive), Naive Search (Naive), heuristic search based on AFNO (AFNO) / DCO (DCO) / AFNO and DCO (AFNO-DCO). In the following, Section 5.1 introduces common setup of the evaluations, Section 5.2 and Section 5.3 respectively evaluate result quality and execution time of the algorithms.

5.1 Common Setup

For evaluating proposed algorithms, the motivating example discussed in Section 2 is reused. Models of fog infrastructures and Smart Bell applications are generated as input to the algorithms. This subsection details the attributes of the model generation.

Each fog node provides certain available resource capacities. To test different infrastructure configurations, in a generated infrastructure model, the capacities of a fog node distribute randomly in ranges related to its device type. Resource ranges of each device type are listed in Table 3.

Device Type	**CPU** (GFlops)	**RAM** (GB)	**DISK** (GB)
cloud	infinite	infinite	infinite
pop	0 ~ 100	0 ~ 500	0 ~ 5000
box	0 ~ 1	0 ~ 1	0 ~ 100
pc	0 ~ 2	0 ~ 4	0 ~ 200
mobile	0 ~ 1	0 ~ 2	0 ~ 50

Table 3: Capacities of Each Fog Node Type[4].

Similarly, resources of network links also follow uniform distribution. The ranges of network latency and available bandwidth for each link type are listed in Table 4.

Link Type	**LAT**(ms)	**BW**(Mbps)
cloud – pop	30 ~ 100	0 ~ 1000
pop – pop	3 ~ 7	0 ~ 5000
box – pop	1 ~ 20	0 ~ 100
pc – box mobile – box camera – box screen – box	1 ~ 2	0 ~ 1000

Table 4: Capacities of Each Link Type.

For applications, each component relies on certain processing and storage resources. Resource requirements of each component type of Smart Bell are given in Table 5. As stated in Section 2, considering privacy requirement, each DB should be placed in its corresponding home, other component types can be placed in any fog node.

Component Type	**ReqCPU** (GFlops)	**ReqRAM** (GB)	**ReqDISK** (GB)	**DZ**
DB	0.1	0.1	0.1	Home
Recorder	0.5	0.5	20	Infra
Extractor	0.2	0.2	0	Infra
Recognizer	0.3	0.3	0	Infra
Decider	0.2	0.1	0	Infra
Executer	0.1	0.2	0	Infra

Table 5: Requirements of Each Component Type.

Network resource requirements of each binding type of Smart Bell are given in Table 6.

[4]A device dedicates only part of its total hardware capacities to the fog infrastructure for hosting applications, rest capacities are reserved to ensure its original functionality.

Binding Type	ReqLAT(ms)	ReqBW(Mbps)
Camera – Extractor	25	0.6
Screen – Executer	25	0.01
Extractor – Recognizer	25	0.1
DB – Recognizer	25	0.3
Decider – Recorder	25	0.2
Extractor – Decider	50	0.1
Decider – Executer	50	0.01

Table 6: Requirements of Each Binding Type.

5.2 Algorithms' Result Quality

This subsection evaluates the proposed objective function and quality of placement decisions made by the algorithms. As introduced in Section 4.1, our objective function is to minimize Weighted Average Latency (WAL).

In this evaluation, Smart Bell instance in Figure 2 is used as as the application to place. To cover different infrastructure configurations, following the distribution defined in Section 5.1, 10 models of the infrastructure in Figure 1 are generated, each generated infrastructure model implies a placement problem. For each problem, 21 sample solutions (i.e., placements subject to all constraints) are selected[5]. Based on Simgrid simulation platform [16], Smart Bell's response time is simulated for each selected solution under corresponding infrastructure configuration. Figure 5 depicts simulated response time of Smart Bell versus WAL for all the 21×10 selected solutions.

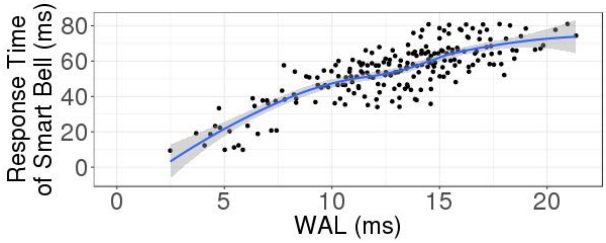

Figure 5: Simulated Response Times of Smart Bell When Applying Selected Solutions.

According to the simulation results in Figure 5, a solution with lower WAL leads to a lower response time, which validates the proposed objective function.

To compare result quality of proposed algorithms, the 10 placement problems generated above are reused. Each problem is solved 10 times by each of the algorithms. Returned solutions' WALs are normalized based on optimal solutions obtained by Exhaustive. WAL of each optimal solution is regarded as 1 (100%). Average normalized WAL and Standard Deviation (SD) of placement decisions made by each algorithm are listed in Table 7.

As DCO aims only at accelerating the search, it gets an average WAL close to Naive. AFNO and AFNO-DCO get lower WAL and lower variance than Naive and DCO. Their results are even close to Exhaustive's. As discussed in Section 4.2, Exhaustive guarantees to return optimal solutions. With a difference lower than 1%, solutions returned by AFNO and AFNO-DCO are near to optimal ones in this evaluation.

[5]Given a placement problem with n solutions, all existing n solutions are found through Exhaustive Search and then sorted in ascending order of WAL. In the sorted list, selected solutions position at 0%×n, 5%×n, . . . , 100%×n.

Algorithm	Average WAL	SD
Exhaustive	100%	0%
Naive	172.4%	53.6%
DCO	173.0%	50.8%
AFNO	100.6%	1.0%
AFNO-DCO	100.4%	0.8%

Table 7: Placement Decision Quality of Each Algorithm.

5.3 Execution Time and Scalability

By generating infrastructure and application models to simulate placement problems encountered in large scale testbeds, this subsection evaluates execution time (i.e., physical processing duration) of proposed algorithms. The test environment is detailed in Table 8.

CPU	Intel i7 - 7820HQ @ 2.90 GHZ
RAM	16GB
OS	Debian 8.8
JAVA	1.8.0_131

Table 8: Execution Time Test Environment.

In order to evaluate the execution time under changing problem sizes, in the following two groups of evaluations, the infrastructure size and considered applications' size increase respectively. The execution times are measured with a timeout of 30 minutes for each benchmark.

a Evaluation with Growing Infrastructure. Smart Bell instance in Figure 2 is used as the application to place all along this evaluation. The infrastructure is enlarged in three phases:

- *phase 1*: based on the infrastructure in Figure 1, to enlarge considered application's specific DZs, one after another, the three homes get a PC or mobile in each round. This enlargement stops when each home gets 10 new fog nodes.
- *phase 2*: based on the final state of *phase 1* (a cloud, a high-level PoP *PoP*1, two low-level PoPs *PoP*2 and *PoP*3, 7+30 fog nodes in 3 homes), new homes are added and connected to *PoP*2. This enlargement stops when the infrastructure contains 200 homes.
- *phase 3*: based on the final state of *phase 2* (a cloud *cloud*1, a high-level PoP *PoP*1, two low-level PoPs, 200 homes), for each increase, a high-level PoP, 5 low-level PoPs, and 1000 homes are added. Each group of added devices form a tree topology. Each added low-level PoP connects with simultaneously added high-level PoP and 200 homes. All high-level PoPs in the infrastructure are linked as a random connected

graph, and each high-level PoP also connects with *cloud*1. This enlargement stops at the 10th round.

In *phase 2* and *phase 3*, each added home has a camera, a screen, a box, a PC or mobile. For each infrastructure size, 10 infrastructure models are generated following the distribution defined in Section 5.1. Each of these generated models implies a placement problem, which is solved 10 times by each placement algorithm. Measured execution times are illustrated in Figure 6.

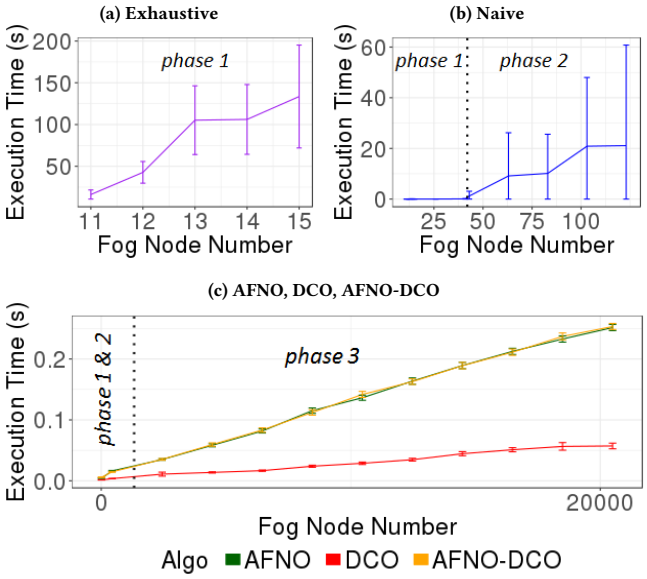

Figure 6: Execution Times with Growing Infrastructure.

Exhaustive performs the worst and exceeds the timeout rapidly in *phase 1*. As shown in Figure 6 (a), execution times of Exhaustive are obtained only for the five smallest sizes. Considering its execution time can be higher than 30 minutes when there are 16 fog nodes, the execution time increase is rather dramatic. Each number of fog nodes in x-axis of the graph corresponds to 10 infrastructure models. Because resource capacities of fog nodes and links distribute randomly, under each infrastructure size, the 10 models' solution numbers differ strongly. Considering that Exhaustive must visit all existing solutions, the solution number difference results in high variances and the fluctuation in the graph.

Better than Exhaustive, Naive reaches the timeout in *phase 2* when 51 homes are added (i.e., when the infrastructure contains 143 fog nodes). As shown in Figure 6 (b), both execution time and variance of Naive increase with infrastructure size. In terms of network latency, homes added in *phase 2* are relatively far from specific DZs (*home*1 ∼ *home*3) and appliances of the considered application. With fog nodes tested in random order, Naive can probably place some components in distant fog nodes and incur backtracks. Such probable backtracks lead to high execution times, high variances, and finally the timeout.

As in Figure 6 (c), AFNO, DCO, and AFNO-DCO successfully place the application till the end of *phase 3*. Compared with Exhaustive and Naive, they get much lower execution times and variances, as well as slow execution time increase. Because of the overhead introduced by anchors-calculating and fog nodes-ordering, AFNO and AFNO-DCO get similar execution times, and perform worse than DCO. Along with the infrastructure enlargement, AFNO and AFNO-DCO have to order more and more fog nodes, which results in an increase of the overhead. However, even with more than 20000 fog nodes, the overhead is lower than 0.3s, which is rather unremarkable.

In this evaluation, AFNO and AFNO-DCO always get similar WAL. Based on problems commonly solved by Exhaustive and AFNO / AFNO-DCO, AFNO / AFNO-DCO's average WAL is only 0.4% higher than Exhaustive's. Similar comparisons show that the average WAL of AFNO / AFNO-DCO is about 42% lower than that of DCO / Naive.

b Evaluation with Growing Applications. Based on the final infrastructure of the evaluation with growing infrastructure (20431 fog nodes and 20400 appliances in 10200 homes, 52 low-level PoPs, 11 high-level PoPs, and 1 cloud), an infrastructure model is generated and used all along this evaluation. To increase application size, Smart Bell instances are added repeatedly for unserved homes. Each 3∼10 homes are in the same neighborhood that can be served by a Smart Bell instance. In each Smart Bell instance, DB number equals to the number of homes served by the instance, component numbers of other types distribute randomly from 1 to 3. Each algorithm is launched 10 times for each application size. Measured execution times are depicted in Figure 7.

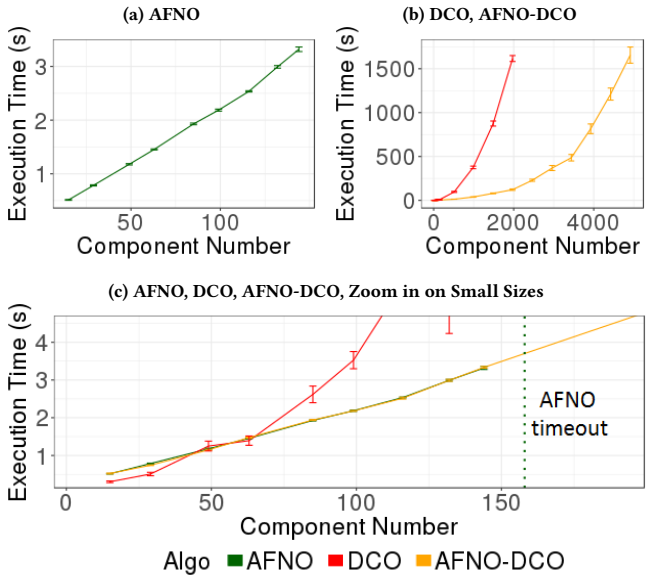

Figure 7: Execution Times with Growing Applications.

Due to the large infrastructure scale, Exhaustive and Naive explode at the very beginning. As component number increases, AFNO, DCO and AFNO-DCO reach the timeout one by one.

As shown in Figure 7 (a), AFNO performs well initially. However, it exceeds the timeout in a sudden fashion when placing 158

components. With more and more applications concurrent to limited resources, there can be resource-constrained situations. Such situations can make it hard to place certain components and thus cause backtracks, high execution times, and the sudden timeout of AFNO.

DCO avoids backtracks by reordering the components, which allows DCO to get a better scalability than AFNO. According to Figure 7 (b), DCO arrives to place 1972 components. To compare the algorithms when application size is small, Figure 7 (c) zooms in on 0~200 components. DCO gets better performance than AFNO and AFNO-DCO initially. However, with more components to place, the overhead of components reorderings becomes significant.

By testing local fog nodes priorly, AFNO-DCO reduces the number of times of components reorderings, and avoids re-launching too many search processes. As a result, AFNO-DCO outperforms DCO rapidly and gets higher scalability. As in Figure 7 (b), AFNO-DCO succeeds to place 301 applications with 4913 components in 30 minutes. Furthermore, based on commonly solved problems, the average WAL of AFNO-DCO is about 40% lower than that of DCO.

Through evaluations of both quality (Section 5.2) and scalability (Section 5.3), it can be found that both AFNO and DCO help to accelerate the placement decision-making process especially for large-scale problems. The overheads they introduce are relatively unremarkable. In addition, through AFNO, returned solutions always get much lower WAL (more than 40% reduction in evaluated cases), and are even close to optimal ones (less than 1% higher than the optimal in evaluated cases), which leads to lower response times of considered applications. To sum up, AFNO-DCO appears as the best compromise in terms of quality and scalability.

6 CONCLUSION AND FUTURE WORKS

This paper tackles the placement of distributed IoT applications in fog infrastructures, that is how to map a set of software components onto a set of fog nodes.

The work focuses on quality and scalability of the placement. Quality is expressed as average response time of considered applications. Scalability is assessed through execution time of placement algorithm under growing numbers of fog nodes and of applicative components.

Contributions of this paper are i) a model and an objective function to formalize the placement problem, ii) two backtrack placement algorithms (Exhaustive and Naive) and two heuristics (Anchor-based Fog Nodes Ordering and Dynamic Components Ordering), and iii) a simulation-based evaluation of different combinations of proposed placement algorithms and heuristics.

The evaluation shows that i) the proposed objective function helps to minimize application's response time, ii) the proposed heuristics highly improve placement results according to the objective function, iii) proposed heuristics, especially their combination, accelerate the placement decision-making process, and make the algorithm much more scalable.

Future works include i) enhancement of proposed placement approaches so as to better handle the intrinsic volatility of fog infrastructures (typically churn and mobility of devices in the extreme edge) and ii) to deal with placement re-optimization considering the migration cost of deployed applications when new deployment requests arrive, and iii) implementation and experimentation on industrial testbeds (such as the Orange Labs internal testbed introduced in [5]).

ACKNOWLEDGMENTS

We thank Prof. Adrien Lèbre for his comments on a preliminary version of this work. Thanks also go to Dr. François-Gaël Ottogalli for fruitful discussions.

REFERENCES

[1] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[2] Luis M Vaquero and Luis Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *ACM SIGCOMM Computer Communication Review*, 44(5):27–32, 2014.

[3] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Optimal operator placement for distributed stream processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 69–80. ACM, 2016.

[4] Antonio Brogi, Stefano Forti, and Ahmad Ibrahim. How to best deploy your Fog applications, probably. In *International Conference on Edge and Fog Computing*, 2017.

[5] Letondeur Loïc, Ottogalli François-Gaël, and Coupaye Thierry. A demo of application lifecycle management for IoT collaborative neighborhood in the fog - Practical experiments and lessons learned around docker. In *Fog World Congress*. IEEE, 2017.

[6] Mohammed Islam Naas, Philippe Raipin, Jalil Boukhobza, and Laurent Lemarchand. iFogStor: an IoT Data Placement Strategy for Fog Infrastructure. In *IEEE 1st International Conference on Fog and Edge Computing*, Madrid, Spain, May 2017.

[7] Mina Sedaghat, Francisco Hernández-Rodriguez, and Erik Elmroth. Autonomic resource allocation for cloud data centers: A peer to peer approach. In *Cloud and Autonomic Computing (ICCAC), 2014 International Conference on*, pages 131–140. IEEE, 2014.

[8] Emiliano Casalicchio, Daniel A Menascé, and Arwa Aldhalaan. Autonomic resource provisioning in cloud systems with availability goals. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, page 1. ACM, 2013.

[9] Kuan-yin Chen, Yang Xu, Kang Xi, and H Jonathan Chao. Intelligent virtual machine placement for cost efficiency in geo-distributed cloud systems. In *Communications (ICC), 2013 IEEE International Conference on*, pages 3498–3503. IEEE, 2013.

[10] Md Hasanul Ferdaus, Manzur Murshed, Rodrigo N Calheiros, and Rajkumar Buyya. Virtual machine consolidation in cloud data centers using ACO metaheuristic. In *European Conference on Parallel Processing*, pages 306–317. Springer, 2014.

[11] 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. *Journal of Computer and System Sciences*, 79(8):1230–1242, 2013.

[12] Wubin Li, Johan Tordsson, and Erik Elmroth. Virtual machine placement for predictable and time-constrained peak loads. In *International Workshop on Grid Economics and Business Models*, pages 120–134. Springer, 2011.

[13] Pedro Silva and Christian Perez. An Efficient Communication Aware Heuristic for Multiple Cloud Application Placement. In *European Conference on Parallel Processing*, pages 372–384. Springer, 2017.

[14] Jiaxin Li, Dongsheng Li, Jing Zheng, and Yong Quan. Location-aware multi-user resource allocation in distributed clouds. In *Advanced Computer Architecture*, pages 152–162. Springer, 2014.

[15] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. Volley: Automated Data Placement for Geo-Distributed Cloud Services. In *NSDI*, volume 10, pages 28–0, 2010.

[16] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, 2014.