Placement Module for Distributed SDN/NFV Network Emulation

Giuseppe Di Lena, Andrea Tomassilli, Frédéric Giroire, Damien Saucez, Thierry Turletti, Chidung Lac

Placement Module for Distributed SDN/NFV Network Emulation

G. Di Lena, A. Tomassilli, F. Giroire, D. Saucez, T. Turletti, C. Lac

RESEARCH
REPORT
N° 9391
February 2021
Project-Team Coati and Diana

ISSN 0249-6399   ISRN INRIA/RR--9391--FR+ENG

Placement Module for Distributed SDN/NFV Network Emulation

G. Di Lena, A. Tomassilli, F. Giroire, D. Saucez, T. Turletti, C. Lac

Project-Team Coati and Diana

Research Report n° 9391 — February 2021 — 32 pages

Abstract: With the increased complexity of today's networks, emulation has become an essential tool to test and validate a new proposed networking solution. As these solutions also become more and more complex with the introduction of softwarization, network function virtualization, and artificial intelligence, there is a need of scalable tools to carry out resource intensive emulations. To this end, distributed emulation has been proposed. However, distributing a network emulation over a physical platform requires to choose carefully how the experiment is run over the equipment at disposal.

In this work, we evaluate the placement algorithms which were proposed for, and implemented in, existing distributed emulation tools. We show that they may lead to bad placements in which several hardware resources such as link bandwidth, CPU, and memory are overloaded. Through extensive experiments, we exhibit the impact of such placements on important network metrics such as real network bandwidth usage and emulation execution time, and show that they may lead to unreliable results and to a waste of platform resources.

To deal with this issue, we propose and implement a new placement module for distributed emulation. Our algorithms take into account both link and node resources and minimize the number of physical hosts needed to carry out the emulation. Through extensive numerical evaluations, simulations, and experiments, we show that our placement methods outperform existing ones leading to reliable experiments using a minimum number of resources.

Key-words: virtualization, emulation, placement, approximation algorithms

This work was supported by the French government through the UCA JEDI [ANR-15-IDEX-01], the FIT EquipEx [ANR-10-EQPX-31] and EUR DS4H [ANR-17-EURE-004] Investments in the Future Projects, and by the Inria associated team EfDyNet.

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Module de Placement pour une émulation distribuée de réseaux SDN/NFV

Résumé : Avec la complexité croissante des réseaux actuels, l'émulation est devenue un outil essentiel pour tester et valider une nouvelle solution réseau. Comme ces solutions deviennent également de plus en plus complexes avec l'introduction des réseaux logiciels, de la virtualisation des fonctions réseau et de l'intelligence artificielle, il est nécessaire de disposer d'outils qui passent à l'échelle pour réaliser des émulations intensives en ressources. À cette fin, il a été proposé de distribuer les émulations. Cependant, la distribution d'une émulation de réseau sur une plate-forme physique nécessite de choisir avec soin la manière dont l'expérience est menée sur l'équipement à disposition.

Dans ce travail, nous évaluons les algorithmes de placement qui ont été proposés et implémentés dans les outils d'émulation distribuée existants. Nous montrons qu'ils peuvent conduire à de mauvais placements dans lesquels plusieurs ressources matérielles telles que la bande passante de liens, le processeur et la mémoire peuvent être surchargés. Grâce à des expériences approfondies, nous montrons l'impact de ces placements sur des paramètres importants du réseau tels que l'utilisation réelle de la bande passante et le temps d'exécution de l'émulation, et nous montrons qu'ils peuvent conduire à des résultats peu fiables et à un gaspillage des ressources de la plate-forme.

Pour traiter cette question, nous proposons et implémentons un nouveau module de placement pour l'émulation distribuée. Nos algorithmes prennent en compte les ressources des liens et des nœuds et minimisent le nombre d'hôtes physiques nécessaires pour réaliser l'émulation. Grâce à des évaluations numériques, des simulations et des expériences poussées, nous montrons que nos méthodes de placement sont plus performantes que les méthodes existantes, ce qui permet de réaliser des expériences fiables en utilisant un nombre minimal de ressources.

Mots-clés : virtualisation réseau, émulation distribuée, algorithmes de placement

Contents

Placement Module for Distributed SDN/NFV Network Emulation

February 5, 2021

1 Introduction

The complexity of networks has greatly increased in the last years. Networks currently rely massively on software frameworks and virtualization, and their performances become implementation dependent. This makes them hard to model, or even to simulate, to obtain relevant predictions of the behavior of their protocols. As a complementary approach, we have seen the rise of prototyping, experimentation in real systems, and emulation. In particular, emulation is frequently used now to evaluate new networking proposals. Indeed, this allows one to experiment network prototypes on a single computer, but using the same real software that would be used in production. As an example, Mininet is a popular and broadly used emulator for network experiments with a simple yet powerful Python API [1]. It emulates network topologies on the experimenter's computer with virtual environments, network namespaces, and virtual switches. This approach is particularly useful as it allows fast prototyping of network experiments.

However, using a single machine for a rapid emulation becomes limiting to handle resource intensive experiments, e.g., needing heavy memory, processing, input/output, or specific hardware, to emulate, for instance, networks with virtual network functions or artificial intelligence algorithms. To tackle this issue, distributed emulation tools were proposed: Maxinet [2], Mininet Cluster Edition [3], Distrinet [4, 5, 6]. These tools allow to run experiments of different types on a large number of machines with different hardware configurations.

Carrying distributed emulation rises several challenges. First, facing an experiment, is there a need to distribute it? In other words, how to know if the experiment exceeds the capacity of a single node? Then, if yes, onto how many nodes and on which nodes should it be distributed? If it has to be distributed onto m machines, how should the experiments be executed on these machines? Actually, a networking experiment can be seen as a virtual network or a graph with node and link demands in terms of CPU, memory, network capacity, etc. A fundamental problem that arises in this context is how to map virtual nodes and links to a physical network topology while minimizing a certain objective function without exceeding the available resources.

Existing tools have placement modules answering partially these questions. `Mininet Cluster Edition` implements three simple algorithms (Round Robin, Random, and Switch Bin [3]), while `Maxinet` uses an algorithm from `METIS` [7], a library for partitioning graphs. However, these placement methods have several important limitations. Firstly, they do not take into account the nodes' resources and the links' capacities. This means that they do not verify if nodes or links capacities are exceeded. Consequently, experiments may run with overloaded links and nodes leading to unreliable results, as we show in this paper. Secondly, they do not minimize the number of needed machines, and use all machines at their disposal. This is especially important for public clusters, where physical resources are shared.

To solve these limitations, we studied placement algorithms to map an experiment onto a set of physical machines (e.g., in private testbeds or clusters). The experimental infrastructure topology is taken into account. The goal is to provide a mapping such that the physical resources of the nodes (i.e., processing and memory) and links (i.e., capacity) are not exceeded. This ensures a trusty emulation. The combination of nodes and links constraints makes finding a feasible solution a difficult task. Indeed, by using a small number of physical nodes, we might exceed their physical resources, while by using too many nodes, we may exceed the available rate of the physical links. Our objective consists in minimizing the number of reserved machines to run an experiment, motivated by the fact that scientific clusters such as *Grid5000* [8] require to reserve a group of machines before running an experiment [9] and an excess in these terms may lead to usage policy violations or to a large waiting time to obtain the needed resources.

The problem can be seen as a variant of a Virtual Network Embedding problem. However, only exact methods based on linear programming were proposed to deal with it in the literature and such solutions do not scale well and have long execution times for large networks, which are our targets in this work. This motivates the need for fast algorithms that can provide near optimal solutions.

Contributions. We proposed new tailored placement algorithms and compared them with the ones used in existing tools. We built a *placement module* for distributed emulators to efficiently solve this problem in practice. This module first decides if the experiment has to be distributed. Then, given a pool of available machines, it computes the deployment using the minimum number of machines to run the experiments in such a way that physical resources are not exceeded. The placement module can be used with any emulator. However, to test it in the wild, we integrated it in `Distrinet`. Through this approach, the experiment is automatically distributed over several nodes using the optimal allocation. To summarize, our contributions are:

- We studied placement algorithms to distribute an experiment onto the machines of a testbed. We proposed several efficient algorithms to deal with the problem.
- We built a placement module [10] for distributed emulators with all the algorithms implemented. The placement module can currently be used in

`Distrinet`, but the algorithms may potentially be integrated by any tool.

- We compared our algorithms with the ones implemented in existing tools using extensive simulations. We show that they succeed in ensuring that no link or node capacity are exceeded, while the same experiments running with other tools would lead to resource overload.
- We then carried experimentation in a private cluster with the goal of evaluating the impact of such resource overload on the emulation. We show that overloading a link, the CPU, or the memory may lead to respectively important drops of measured bandwidth, the increase of execution time, and emulation crashes.

The rest of this paper is organized as follows. In Section 2, we review related works on placement methods to carry out distributed emulation. We then formally state the problem and propose algorithms to deal with it in Section 3. We evaluate the algorithms against the existing placement modules with extensive simulations in Section 4 and by experimentation in Section 5. Last, we conclude and present future work on placement algorithms in Section 6.

2 Related Work

Placement for Distributed Emulations. Existing tools for distributed large-scale emulations adopt different strategies to map the virtual topology to the physical one.

`Mininet Cluster Edition` provides 3 different placement algorithms [3]:

- *SwitchBinPlacer* first distributes the virtual switches (*vSwitches*) (and the controllers if some are assigned) around the infrastructure, such that each physical host (also called server) has the same amount of vSwitches assigned. It then places the virtual hosts (*vHosts*) on the server to which its connected vSwitches are assigned.
- *RoundRobinPlacer* is the implementation of the classic RoundRobin algorithm that assigns a vSwitch or a vHost, choosing each time the next physical host in the list.
- *RandomPlacer* is the simplest placer: for each vHost or vSwitch to be assigned, it chooses a random physical node.

`Maxinet` uses the Multilevel Recursive Bisectioning algorithm [11] to partition the virtual switches and the virtual hosts into the physical machines. In `Maxinet`, there is no notion about the physical infrastructure (hosts resources or network topology). This means that the partition will not change if we deploy the virtual network in different physical topologies (i.e., spanning tree, clique, etc.). The virtual network, given as an input to the partition algorithm, does not have notion of virtual CPU (vCPU) or virtual RAM (vRAM); i.e., a virtual node (vNode) requiring 1 vCPU is treated like a vNode requiring 10 vCPUs.

The partitioning algorithm is not directly implemented in `Maxinet`. `Maxinet` uses `METIS` [7], a set of tools for partitioning graphs. The goal of the algorithm is to find a partitioning of the nodes such that the sum of the nodes weights (e.g., workload) in each partition is balanced and the sum of all the edges in the cuts are minimized.

In all these algorithms, the physical infrastructure is not taken into account (or partially with `METIS`). This means that a *physical link or a physical machine can be overloaded and become a bottleneck* for the emulation *without the user being notified*. In fact, we show that the existing placement solutions behave well when the physical infrastructure is an homogeneous environment. When the physical environment is heterogeneous (different types of machines or a complex physical network), they often return solutions with overloaded resources. Moreover, the existing solutions do not evaluate the minimum number of machines needed to run the experiment (and in particular if the experiment has to be distributed). These solutions use all the machines put at their disposal by the user. On the contrary, our placement module provides the user the smallest number of physical hosts in order to run the experiment without any overloaded physical resource.

Virtual Network Embedding (VNE) Problem. The solution we propose is based on the investigation of a VNE problem. Such problems have been widely studied in the literature. We refer to [12] for a comprehensive survey of the existing works. Many different settings have been considered. Minimization of the resource allocation cost [13, 14, 15, 16], of the energy consumption [17], of the maximum load [18] or revenue maximization [19] are just few examples. In our settings, we aim at finding a mapping which uses the smallest number of substrate nodes. Thus, our objective can be seen as a variant of the energy-aware VNE problem in which we aim to minimize the number of activated substrate nodes.

Exact solutions which provide optimal techniques to solve small instances have been proposed (see, e.g., [20, 21]). They are mainly based on exact approaches such as Integer Linear Programming (ILP) and formulate the VNE problem as virtual nodes and links mapping. These approaches are not suitable in our use case as runtime is a crucial factor and the delay in the embedding of a virtual request should be minimized. A heuristic approach to find an acceptable solution in a short execution time is to be preferred. We thus propose heuristic approaches able to provide near-optimal solutions in a reasonable computation time. Beyond the general case, we additionally study specific settings often encountered when carrying out emulations in real cluster environments. They are characterized by a homogeneous computing environment or by a spanning tree routing protocol, and they have not been fully addressed in the literature.

3 Problem and Algorithms

When emulating large data center networks with hundreds or thousand of nodes, it is necessary to distribute the emulation over multiple physical machines in order not to overload the physical resources.

We study here the problem of mapping virtual nodes and links to a physical network topology, while minimizing the number of used machines and without exceeding the available physical resources (CPU cores, memory, and links' rates). We first define formally the optimization problem which is considered. We then propose algorithms to deal with it. In order to evaluate the performances of the algorithms, we run numerical evaluations and compare their solutions with the optimal one found using an ILP.

We consider several specific settings often found in real cluster environments: homogeneous topologies or physical network arranged as a tree.

3.1 Problem Statement

The VNE problem can be formally stated as follows.

Substrate Network. We are given a substrate network modeled as an undirected multigraph $G^S = (N^S, L^S)$ where N^S and L^S refer to the set of nodes and links, respectively. G^S is a multigraph as there may be multiple links between a pair of nodes. Each node $n^S \in N^S$ is associated with the CPU capacity (expressed in terms of CPU cores) and memory capacity denoted by $c(n^S)$ and $m(c^S)$, respectively. Also, each link $e^S(i, j) \in L^S$ between two substrate nodes i and j is associated with the bandwidth capacity value $b(e^S)$ denoting the total amount of rate that can be supported.

Virtual Network Request. We use an undirected graph $G^V = (N^V, L^V)$ to denote a virtual network, where N^V is the set of virtual nodes and L^V the set of virtual links. Requirements on virtual nodes and virtual links are expressed in terms of the attributes of the nodes (i.e., CPU cores $c(n^V)$ and memory $m(c^V)$) and links (i.e., the rate to be supported $b(e^V)$) of the substrate network. If there are no sufficient substrate resources available, the virtual network request should be rejected or postponed. When the virtual network expires, the allocated substrate resources are released.

The problem consists in mapping the virtual network requests to the substrate network, while respecting the resource constraints of the substrate network. The problem can be decomposed into two major components: (1) Node assignment in which each virtual node is assigned to a substrate node, and (2) Link assignment in which each virtual link is mapped to a substrate path.

The combination of nodes and links constraints makes the problem extremely hard for finding a feasible solution. Indeed, if on one hand by using a small amount of substrate nodes, we may exceed physical resources capacities, such as CPU and memory, on the other hand, by using too many nodes, we may exceed the available rate of the substrate links. Rost et al. [22] show that the problem of finding a feasible embedding is NP-Complete, even for a single request.

Algorithm 1 k-balanced

```
Input: Virtual network $G^V$, Substrate network $G^S$.
Output: a mapping of virtual nodes to substrate nodes $m : N^V \rightarrow N^S$.
for $k = 1, 2, \ldots, \min(|N^S|, |N^V|)$ do
    sol ← Compute an approximate solution of the $k$-balanced partitioning problem for $G^V$.
    if sol is feasible (see Sec. 3.2.3) then return sol
return $\emptyset$
```

3.2 Algorithms

We propose three algorithms to tackle this problem. Our algorithms are able to provide near-optimal solutions in a reasonable computation time. Solutions are compared with the optimal ones computed using an ILP approach.

The first two algorithms, k-balanced and DivideSwap, have two phases. Firstly, virtual nodes are mapped into the physical topology and, secondly, physical paths are found to map virtual links. The third proposed algorithm, GreedyPartition, mixes both nodes and links mapping.

3.2.1 Homogeneous Case

If the substrate nodes within the cluster are homogeneous in terms of physical resources (or if there is a subset of homogeneous nodes from the entire cluster), an assignment strategy may consist in carrying out a partition of the tasks to be done by the physical machines while minimizing the network tasks that would be necessary to be done. We refer to this algorithm as k-balanced. We use as a subroutine an algorithm for the k-balanced partitioning problem. Given an edge-capacitated graph and an integer $k \geq 2$, the goal is to partition the graph vertices into k parts of equal size, so as to minimize the total capacity of the cut edges (i.e., edges from different partitions). The problem is NP-Hard even for $k = 2$ [23]. k-balanced solves a k-partitioning problem for $k = 1, \ldots, \min(|N^S|, |N^V|)$, and tests the feasibility of the computed mapping $m : N^V \rightarrow N^S$ of virtual nodes on the substrate network. The smallest k for which a feasible k-partitioning exists will be the output of the algorithm. The corresponding pseudo-code is given in Algorithm 1.

The best known approximation factor for the k-balanced partitioning problem is due to Krauthgamer et al. [24] and achieves an approximation factor of $O(\sqrt{\log n \log k})$, with n being the number of nodes in the virtual network. Nevertheless, as their algorithm is based on semi-definite programming and would lead to long execution time, to deal with the problem, we use the $O(\log n)$ approximation algorithm described in [25]. The main idea consists in solving recursively a Minimum Bisection Problem. To this end, we use the Kernighan and Lin heuristic [26].

Algorithm 2 DivideSwap

```
Input: Virtual network $G^V$, Substrate network $G^S$.
Output: a mapping of virtual nodes to substrate nodes $m : N^V \to N^S$.
for $k = 1, 2, \ldots, \min(|N^S|, |N^V|)$ do
    Divide the nodes from $N^S$ in $k$ balanced subsets $V_1$,    $V_2$, ..., $V_k$.
    Take a random sample of $k$ physical nodes $P_1$, $P_2$, ...,    $P_k$
    Assign nodes in $V_j$ to $P_j$ for $j = 1, ..., k$.
    for $i = 1, 2, \ldots, N\_SWAPS$ do
        Choose at random two nodes $u, v \in N^V$ assigned to two distinct
        physical nodes.
        If by swapping $u$ and $v$ the cut weight decreases, swap them in sol
        and update the cut weight
    if sol is feasible (see Sec. 3.2.3) then return sol.
return $\emptyset$
```

k-balanced has theoretical guarantees of efficiency for its node mapping phase, but only when the number of parts takes some specific values (powers of 2). Indeed, the procedure is based on merging two small partitions until the number of partitions is greater than the desired one (e.g., if $k = 3$ and the algorithm computes 4 partitions, then the result is unbalanced as 2 will be merged).

We thus propose a new algorithm, DivideSwap, efficient for any values of k. The global idea of DivideSwap is to first build an arbitrary balanced partition dividing randomly the nodes in balanced sets, and then swapping pairs of nodes to reduce the cut weight (or required rate for the communications). Its pseudocode is given in Algorithm 2.

3.2.2 General Case

When the substrate nodes are associated with different combinations of CPU, memory, and networking resources, k-balanced and DivideSwap may have difficulties in finding a good assignment of virtual nodes to substrate nodes which respects the different capacities. To prevent this, we define a general procedure referred to as GreedyPartition. Again, we first build a bisection tree. We compute it by recursively applying the Kernighan-Lin bisection algorithm (see the discussion above on bisection algorithm). Then we test the use of an increasing number of physical machines, from 1 to N. We take the j most powerful ones, with powerful defined as a combination of CPU and memory. Next, we perform a BFS visit on the bisection tree. For each considered node, we find a physical node such that the resources are enough and the communication can be performed considering the already placed virtual nodes. If for a node the conditions hold, then the node is removed from the tree with all its subtrees. If not, we consider the next node of the bisection tree. If at any point the tree is

Algorithm 3 GreedyPartition

```
Input: Virtual network $G^V$, Substrate network $G^S$.
Output: a mapping of virtual nodes to substrate nodes $m : N^V \to N^S$.
T ← Compute a bisection tree of $G^V$.
for $j = 1, 2, \dots, |N^S|$ do
    Select the $j$ most powerful machines, $P_1, \dots, P_j$
    Perform a BFS on the bisection tree $T$.
    for each Node $v$ of $T$ do
        if $\exists\ P \in P_1, \dots, P_j$ with enough resource to host the virtual nodes in $v$ then
            v is assigned to $P$
            Remove $v$ and the subtree rooted at $v$ from $T$
    if T is empty then return sol
return $\emptyset$
```

empty, we return the solution.

3.2.3 Link Mapping

DivideSwap and k-balanced are based on assigning virtual nodes to physical nodes, then checking the feasibility of the problem by trying to assign all the virtual links between virtual nodes assigned to two different substrate nodes to a substrate path. This problem is solved differently according to the structure of the physical substrate network.

Tree Topologies. Even if the substrate network is assumed to be a tree, there are still decisions to be made in terms of how the network interfaces of a substrate compute node should be used by the virtual nodes. Indeed, if we allow the traffic associated to a single virtual link to be sent using more than one network interface (e.g., 50% on `eth0` and 50% on `eth1`), then multiple links between a compute node and a switch can be considered as a single link with an associated rate which corresponds to the sum of rates on all the node network interfaces. As a consequence, once the mapping between virtual and substrate nodes has been selected, checking if a substrate compute node has enough resources on its interfaces to send or receive a given set of rates can be done *exactly* in polynomial time with a visit (either BFS or DFS) in time $O(|N^S| + |L^S|)$, as there exists only one path between each source and destination pair. Conversely, if the virtual link rate to be supported can be mapped on only a single network interface, then the situation can be reduced to the *Bin Packing Problem* and is thus NP-hard. To deal with it, we use the First-fit decreasing heuristic which has been shown to be $\frac{11}{9}$-approximated for this problem [27] (i.e., it guarantees an allocation using at most $\frac{11}{9}$OPT + 1 bins, with OPT being the optimal number of bins).

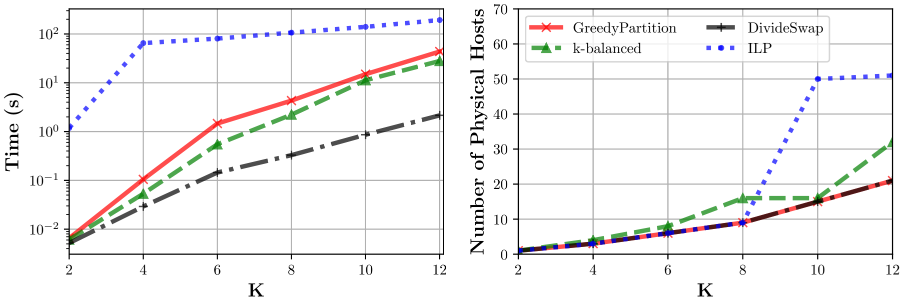


Figure 1: Performances of the heuristics and ILP solver for a K-ary Fat Tree. Time needed to find the solution (left) and value of the solution found (right).

General Topologies. Also in this case, we need to distinguish two cases according to the desired strategy for mapping a virtual link of the virtual network to a physical path in the substrate network. If path splitting is supported by the substrate network, then the problem can be solved in polynomial time by using a multi-commodity flow algorithm [19]. On the other hand, if path splitting is not supported, then the situation can be reduced to the *Unsplittable Flow Problem*, which is NP-hard [28]. In such case, we use the following approach. We consider the virtual links in non-increasing order. Given the remaining capacities, we find the shortest path in the residual network in which we remove all links with an available rate smaller than the rate to be mapped. If we succeed to find a physical path for all the virtual links to be mapped (i.e., between nodes assigned to distinct physical machines), then the problem is considered feasible.

3.3 Numerical Evaluation

The experiment consists in mapping data center servers interconnected by a K-Fat Tree onto the Nancy node of the academic cluster Grid'5000 [8]. Each logical switch and server require 2 cores and 8 GB of memory, and each server is sending 0.2 Gbps of traffic. The physical nodes are machines with 32 cores and 132 GB of RAM. The physical links are Ethernet links of capacity 10 Gbps. The goal is to minimize the number of machines of the cluster used for the experiment for different data center sizes. In Fig. 1, we compare the solutions given by three heuristics with the ones computed using an ILP approach (with a running time limit set to 2 minutes). The experiments are run on an Intel Core i5 2.9 GHz with 16 GB of memory. For the ILP approach, we use CPLEX 12.8 as a solver. When the time limit is reached, we report the best solution found so far.

First, we verify that our solutions map a Fat Tree with K=2 onto a single machine as the requirements in terms of cores and memory are low enough. However, this is no longer the case for 4-Fat Trees and larger topologies. Second, we see that the best proposed algorithms, DivideSwap and GreedyPartition, can compute optimal or near-optimal solutions within a few seconds, showing they can be used for fast experimental deployment.

Second, we see that DivideSwap and GreedyPartition distribute the experiment very efficiently for large Fat Trees. Indeed, they obtain solutions using the same number of machines as the ILP when it gives optimal solutions (Fat Trees with $K \leq 8$). The ILP does not succeed to find efficient solutions for Fat Trees with K=10 and K=12 in less than 2 minutes, while our algorithms succeed. For the 10-Fat Tree, our algorithms return a solution using 16 physical machines (when the solution of the ILP is using 50). Note that it corresponds to a case (power of 2) for which k-balanced is guaranteed to use near optimal partitions of the experiment graph. This confirms the efficiency of DivideSwap and GreedyPartition which obtain the same result in this case. Our algorithms are very fast and can map large Fat Trees in few seconds. DivideSwap is the fastest. However, GreedyPartition better handles more complex scenarios as it is shown in the next sections. To sum up, we see that the best proposed algorithms, DivideSwap and GreedyPartition, can compute optimal or near-optimal solutions within a few seconds, showing they can be used for fast experimental deployment.

4 Evaluation of the Placement Modules

In this section, we compare our placement algorithms with the ones used by `Mininet Cluster Edition` and `Maxinet`. In order to make the comparison as meaningful as possible and to understand the advantages and disadvantages of each algorithm, we considered different scenarios with *homogeneous* or *heterogeneous* virtual and physical topologies. Scenarios with homogeneous virtual and physical infrastructures are the most favorable for simple placement algorithms. Scenarios with homogeneous physical infrastructures should be the most favorable ones for the placement modules of the existing tools as they do not take into account the physical infrastructure. We show that our algorithms outperform them even in this scenario. The heterogeneous scenario represents a more complex case to show the importance of taking into consideration the capacities of the physical infrastructure.

Physical Topologies. We used two physical infrastructures, one homogeneous, and one heterogeneous. The first one is a simple star topology corresponding to the one of the *Gros* cluster in Grid'5000 [29]. We used 20 physical machines, each equipped with an Intel Gold 5220 (Cascade Lake-SP, 2.20 GHz, 1 CPU/node, 18 cores/CPU) and 96 GB of RAM. The machines are connected by a single switch with 25 Gbps links. The second is represented in Fig. 2: this infrastructure is made of a subset of 25 hosts of the *Rennes* cluster in Grid'5000. There are four types of servers with different numbers of cores and memory sizes: Parapide (2 servers with 8 cores and 24 GB of RAM); Parapluie (8 servers with 24 cores and 48 GB of RAM); Parasilo (5 servers with 32 cores and 128 GB

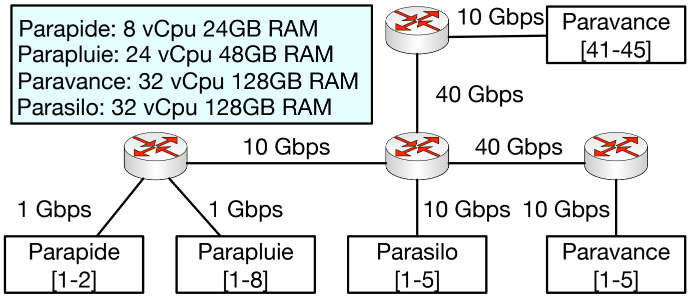


Figure 2: *Rennes* topology cluster

of RAM); Paravence (10 servers with 32 cores and 128 GB of RAM) for a total of 25 machines. The servers are interconnected using a small network with 4 switches and links with capacities of 1 , 10 or 40 Gbps.

Virtual Topologies. We used two different families of virtual topologies: Fat Trees and Random. We chose the first one as it is a traditional family of data center topologies: this corresponds to the homogeneous scenario. Indeed, Fat Trees present symmetries and all servers are usually similar.

Fat Trees. We tested Fat Trees with different parameters:

- *K*: the number of ports that each switch contains (2, 4, 6, 8 or 10);
- *Number of CPU cores*: the number of virtual cores to assign to each vSwitch or vHost (1, 2, 4, 6, 8 or 10);
- *Memory*: the amount of RAM required by each vSwitch or vHost (100, 1000, 2000, 4000, 8000 or 16000 MB);
- *Links' rates*: the rate associated to each virtual link (1, 10, 50, 100, 200, 500 or 1000 Mbps).

Random Topologies. We used a generator of random topologies which takes as inputs the number of vSwitches and the link density between them. Half of the vSwitches are chosen to be the core network (meaning that no host is attached to them). The other half are the edge switches (vHosts are connected to them). The generator then chooses a random graph to connect the vSwitches, making sure that all of them are connected. The random graph is obtained by generating Erdös-Renyi graphs using the classical `networkx` Python library till we obtain a connected one that can be used as vNetwork. After setting up the switch topology, a vHost is connected to a single edge switch selected uniformly at random. The capacity of the vLinks (1, 10, 50, 100, or 200 Mbps), the number of virtual cores (vCores) (1, 2, 3, 4, 5, 6, 7, or 8) and the RAM (1000, 2000, 4000, 6000, 8000, or 16000 MB) required for each vNode are then selected uniformly at random. For our experiments, for each pair (N, Density) with N in (10, 15, 20, 25, 30, 35, 40, 45, and 50) and with density in (0.1, 0.2, 0.3, 0.4, and 0.5), we generated 100 random networks.

A fundamental difference between the two families of networks is that, while for Fat Trees all the virtual networks have homogeneous resource requirements (i.e., all nodes and links have the same physical requirements among them), for Random Networks the requirements associated to the virtual nodes and virtual links may be different.

4.1 Results

In this section, we extensively study the performances of the different placement algorithms. To this end, we considered more than 70,000 test instances, corresponding to the mapping of each generated virtual topology on the two physical topologies presented above.

While the existing placement algorithms always return solutions as they do not take into account node and link capacities constraints, our algorithms don't as they make sure that resources are not overloaded. To assess the impact of such a difference, we first analyze the cases where feasible solutions are found. We then study the cases where physical constraints are not respected. Finally, we discuss how the algorithmic choices translate in the number of physical hosts needed to run the experiments.

Finding a Feasible Solution. When comparing the results of the placement algorithms, we only considered the virtual instances for which at least one of them was able to find a feasible solution. In total, we report the experiments made for more than 5,000 instances.

Table 1 shows the percentage of instances solved by each algorithm (over the set of feasible instances). We provide the percentage for each family of virtual topologies: *vFT* (virtual Fat Tree) and *vRD* (virtual Random) topologies. For each family of virtual topology, the tests have been performed on both physical topologies. In particular, the number of feasible solutions analyzed are 761 for *Gros* vFT (Homogeneous-Homogeneous), 4500 for *Gros* vRD (Homogeneous-Heterogeneous), 708 for *Rennes* vFT (Heterogeneous-Homogeneous), and 4436 *Rennes* vRD (Heterogeneous-Heterogeneous). We indicate in the last column of the table (Intersection) the percentage of virtual instances for which all algorithms return a feasible solution. First, we observe that a large number of instances cannot be solved by all the algorithms. Second, the results confirm

Algorithm		GreedyP	k-balanced	DivideSwap	Metis	Random	RoundRobin	SwitchBin	Intersection
Cluster	**vTopo**								
Gros	vFT	100.0%	91.72%	97.76%	85.02%	72.01%	98.81%	83.18%	68.59%
	vRD	100.0 %	83.31 %	96.64%	58.31%	52.64%	93.15%	37.86%	32.09%
Rennes	vFT	100.0%	83.90%	92.09%	65.81%	43.22%	68.36%	44.49%	34.18%
	vRD	99.98%	73.51%	74.41%	32.75%	20.37%	34.67%	28.40%	11.47%

Table 1: Percentage of solutions found using different algorithms, virtual topologies, and different clusters.

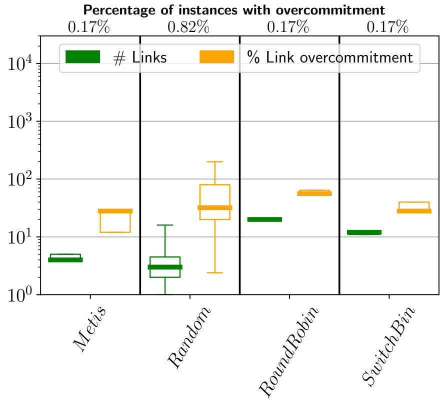


Figure 3: *Gros* Link overcom.

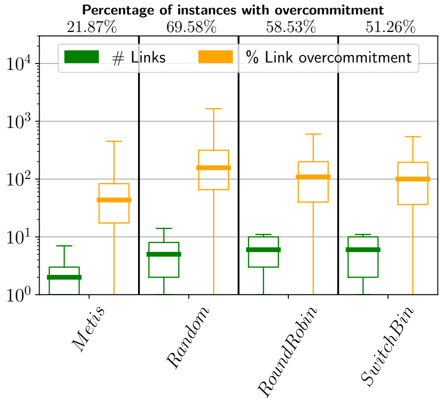


Figure 4: *Rennes* Link overcom.

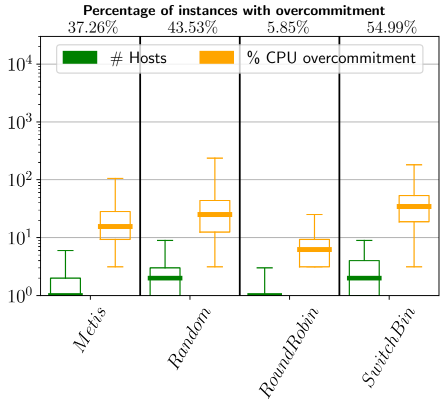


Figure 5: *Gros* CPU overcom.

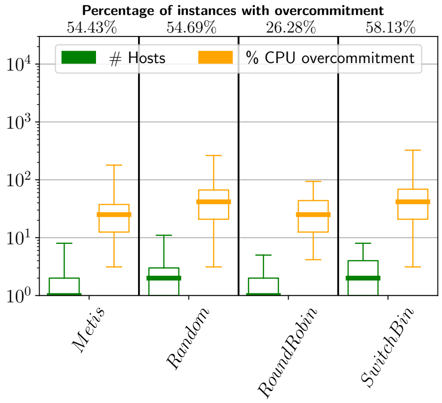


Figure 6: *Rennes* CPU overcom.

that heterogeneous (whether virtual or physical) topologies are a lot harder to solve (in particular for the algorithms of the existing tools). Note that only 11.7% of the vRD were solved by all algorithms on the *Rennes* cluster.

From a high level, two of the algorithms proposed in this paper reach the higher success ratio in terms of number of solved instances. In particular, GreedyPartition succeeds to find a feasible solution for almost all the feasible virtual networks when mapped to the *Gros* cluster, and vFT when mapped to the *Rennes* cluster, while it finds a feasible solution for 99.98% of the instances in the vRD case. The second best algorithm is DivideSwap which solved more than 90% of the instances in the *Gros* cluster and in the *Rennes* Cluster for the vFT topology. Note that k-balanced has a lower percentage (76.7%). This is expected as this algorithm is efficient when the solution is mapped on specific numbers of physical hosts (powers of 2), a case for which it has some theoretical

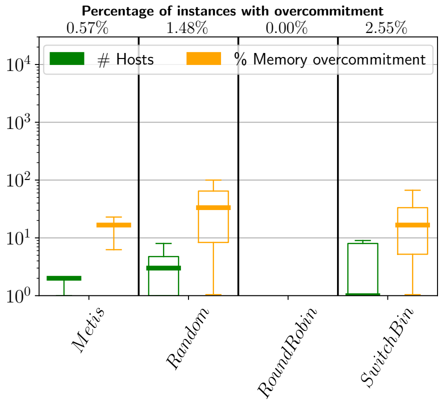


Figure 7: *Gros* Memory overcom.

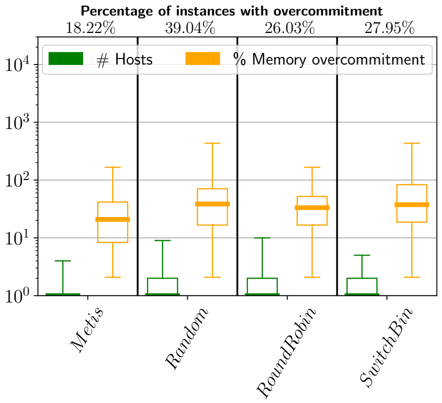


Figure 8: *Rennes* Memory overcom.

guarantees. But it is behaving as well for other values.

Then, the algorithm used by `Maxinet` (`METIS`) finds 85.02% of the solutions for *Gros* vFT. As expected, the algorithm drastically changes its performances when the physical environment is non heterogeneous (i.e., *Rennes* vFT), or when the network to emulate has different vNodes requirements or vLinks requirements (i.e., virtualizing a random network in *Gros*). The worst scenario that we have with `METIS` is in the case of homogeneous infrastructure virtualizing a random topology (i.e., *Rennes* vRD) where only 32.75% of the returned solutions are feasible. We tested the 3 other algorithms that are directly implemented in `Mininet Cluster Edition`. `Random` solves 72 % of the instances in *Gros* vFT, while the performances drop drastically in the homogeneous case and when virtualizing a random topology. The same behavior can be observed for `Round Robin` and `Switch Bin`.

Analysis of Solutions not Respecting Capacities. Even though the existing algorithms do not always succeed to find solutions respecting the physical capacity constraints, they still return solutions. Here, we study how severely overloaded links and nodes can belong to the computed solutions.

Figs. 3–8 take into account the virtual instances for which an overloaded solution is returned (in terms of CPU, memory, or link overcommitment) for all the Mininet Cluster and Maxinet algorithms, using the different clusters.

The *overcommitment* is the estimation (in %) of how much the CPU, memory, or bandwidth is assigned in excess to a physical node or a physical link. From the percentages in the plots, we can observe a strong difference in terms of feasible solutions returned for the two different clusters.

If we focus our attention on the link overcommitment (Figs. 3 and 4), we can observe that less than 1% of the returned solutions lead to links overloaded in the *Gros* cluster. This is expected as the physical topology is a simple star. Conversely, in *Rennes*, a high percentage of the solutions lead to links over-

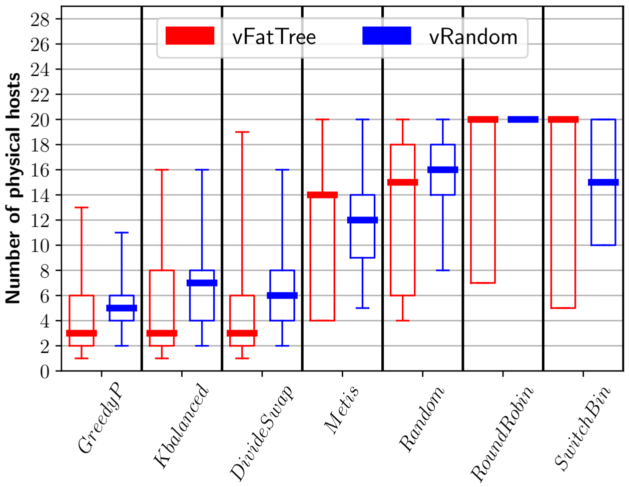


Figure 9: Number of physical hosts used by the placement algorithms. Homogeneous *Gros* cluster.

loaded: from 21.7% for Metis to almost 70% for Random, and 51% and 57% for SwitchBin and RoundRobin, respectively. Moreover, for such instances, the overcommitment is important: 4 or 5 links are overloaded of around 100% in average for Random, SwitchBin, and RoundRobin. The overload factor may reach several hundred percents and even more than 1000% for some instances with Random. The overcommitment is lower for Metis, the median case being 2 links with a 40% overload. The explanation of the better behavior of the latter algorithm is that it is using a partitioning graph algorithm minimizing the cuts between partitions placed in different machines.

Considering now CPU overcommitment (Figs. 5 and 6), we see that it is frequent both for the *Gros* and *Rennes* clusters. RoundRobin is the algorithm handling the best CPU resources: only 5.85% of its returned solutions are overloaded compared to 35%, 43%, and 55% for Metis, Random, and SwitchBin, respectively. The explanation is that RoundRobin tries to distribute the load evenly to the physical hosts. Metis behaves a bit better than Random, and SwitchBin, in *Rennes* with fewer overloaded nodes.

Memory overcommitment rarely happens in *Gros* (see Figs. 7 and 8). In *Rennes*, from 18% to 35% of the instances are overloaded. Metis is the best algorithm in this case.

Number of Physical Hosts Needed. An important additional advantage of the algorithms that we propose in this paper is that they minimize the number of physical hosts needed to emulate the experiments. This helps reducing the use of testbed resources and even making feasible some large experiments that would not be able to run without well optimizing hardware usage, as we show

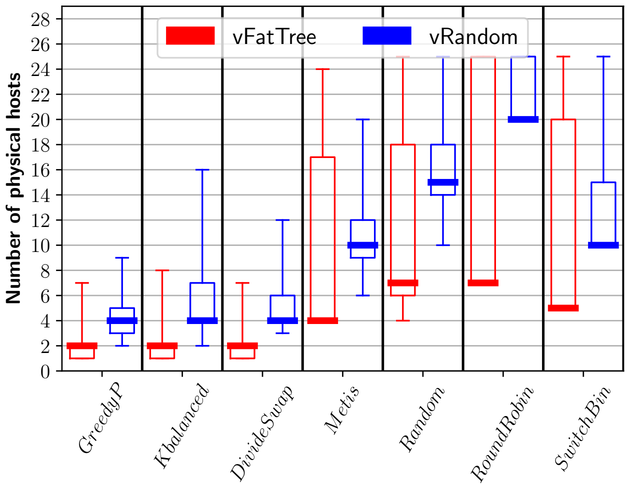


Figure 10: Number of physical hosts used by the placement algorithms. Heterogeneous *Rennes* cluster.

below.

We report in Figs. 9 and 10 the distributions of the number of hosts used by the algorithms over all the virtual topologies for which all algorithms found a feasible solution (Intersection subset). Note that this subset of experiments does not contain many large topologies, as the less efficient placement algorithms were not able to find solutions for them.

As expected, the proposed algorithms use much fewer physical hosts. For the *Gros* cluster (Fig. 9), the general tendency is that GreedyPartition uses between 1 and 13 hosts (median is 3) in case it is emulating a vFT, and between 2 and 11 hosts (median is 5) in case it emulates a vRD. `METIS` uses a minimum of 4 instances with a maximum of 20 instances (medians are 4 and 12). `Round Robin` (medians are 13 and 14), `Random` (medians are 7 and 20), and `Switch Bin` (medians are 5 and 15) use in general more hosts than Metis.

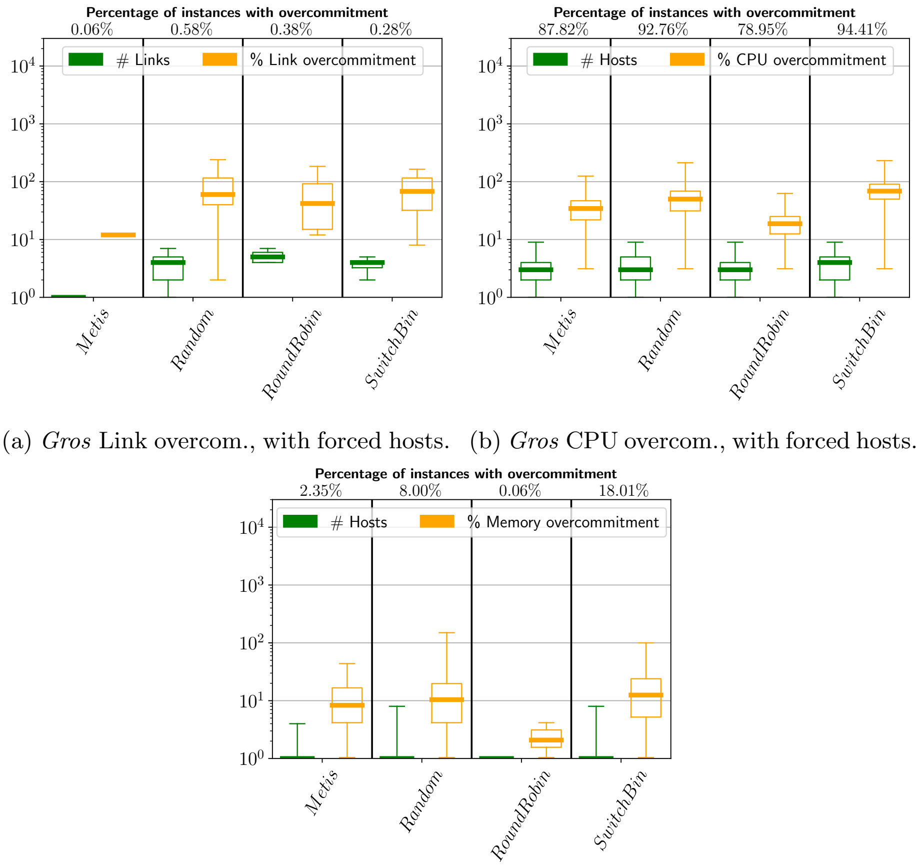
The differences are even more important for the heterogeneous topology, *Rennes*, especially in terms of maximum number of hosts used (Fig. 10). The numbers of hosts used by our algorithms are in general lower (e.g., between 1 and 7 hosts for GreedyPartition) while the ones for the existing algorithms are higher (e.g., between 4 and 24 hosts for `METIS`, and between 7 and 26 for `Round Robin`). This is explained by the fact that the same virtual topology is harder to solve on a heterogeneous physical topology than on a homogeneous one. So, the set of topologies for which all algorithms find a solution is smaller for the *Rennes* topology and contains smaller virtual topologies on average. Our algorithms thus find solutions using a lower number of hosts, while other algorithms have troubles mapping them efficiently on the heterogeneous physical topology. Again, GreedyPartition is the best algorithm in terms of resource usage.

Alg.	METIS	RANDOM	ROUNDROBIN	SWITCHBIN
vFT	59.42%	33.52%	96.38%	32.0%
vRD	5.51%	2.71%	11.82%	1.40%

Table 2: Percentage of feasible solutions found by the algorithms from `Maxinet` and `Mininet Cluster Edition` when the number of physical hosts to use is set to the minimum one found by GREEDYPARTITION.

Limiting the Number of Physical Hosts. The algorithms adopted by `Maxinet` and `Mininet Cluster Edition` do not optimize the number of physical hosts but tend to use all the ones they have at their disposal. However, the fact that they are using more hosts does not mean that they would not be able to use fewer of them. To check this, we carried out another study. We first computed the minimum number of hosts needed to run the experiment (as given by GREEDYPARTITION). We then put this number of hosts at the disposal of the algorithms used by `Maxinet` and `Mininet Cluster Edition`. We check (*i*) if the algorithm finds a solution and (*ii*) if this solution overloads physical hosts or links. The results are reported in Table 2 for vFT and vRD topologies for the *Gros* cluster and in Figures 11a, 11b, and 11c. We tested 525 different vFT instances and 4500 vRD topologies. For the vFT, the most favorable scenario for the existing algorithms (homogeneous virtual and physical topologies), the first observation is that `METIS`, `Round Robin`, `Random`, and `Switch Bin` are only able to find feasible (with no node or link overcommitment) solutions with the same number of physical hosts for 59.42%, 33.52%, 96.38%, and 32.0% of the instances, respectively. The percentage is low for `METIS` and very low for `Random` and `Switch Bin`. For `Round Robin`, the results are good for this scenario because each physical machine has the same capabilities and each vNode has the same requests. In homogeneous cases and when considering the number of hosts found by our algorithm, if the Round Robin strategy returns an unfeasible solution, it can only be because of link overcommitment. As the link capacities are very high in *Gros* (25 Gbps), it explains why `Round Robin` is able to solve 96.38% of instances. If we consider now the more complex scenario with vRT, the percentage of solved instances by the existing algorithms drops to values between 1.4% and 11.82%. This shows the efficiency of GREEDYPARTITION to find efficient solutions even in hard cases. Note that this advantage would be even more important when doing experiments on heterogeneous clusters.

We analyze now the solutions returned when no solution satisfying node and link constraints could be found. The goal is to see if such solutions are “close” to be feasible in the sense that only a node or link is a little bit overloaded. Link, CPU, and memory overcommitments are reported in Figures 11a, 11b, and 11c, respectively. `Random`, `Round Robin`, and `Switch Bin` do not return close solutions when they have some overloaded links. Indeed, they have multiple overloaded links (with a median of 4 and 5) with an overcommitment between 2% and 200%, and median values of 60%, 42%, and 68%, respectively.



(a) *Gros* Link overcom., with forced hosts. (b) *Gros* CPU overcom., with forced hosts.

(c) *Gros* Memory overcom., with forced hosts.

Figure 11: Gros overcommitments with forced hosts.

For Metis, the link overcommitment is lower and involves a single link overloaded at around 10%. The overcommitment in terms of CPU tends to be similar within the algorithms. The number of overloaded hosts is between 1 and 10 (with a median of 3 or 4) and the median overload is between 20% and 90%. The maximum overload reaches more than 100% for three of the four algorithms.

5 Experimental Evaluation

To show the importance of placement for distributed emulations and to evaluate the impact of violating the limitations in terms of physical resources, we carried out experiments using a placement module implemented in `Distrinet` [4].

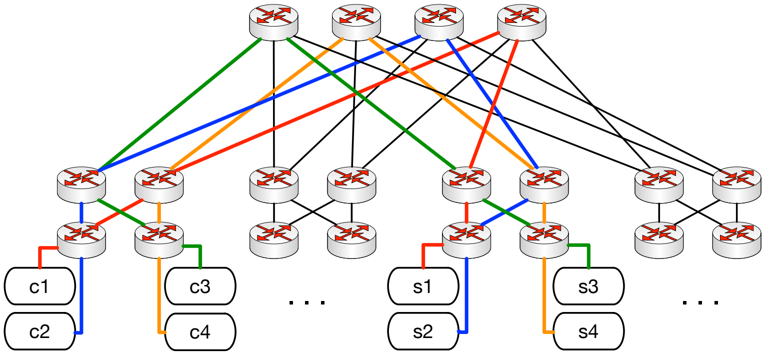
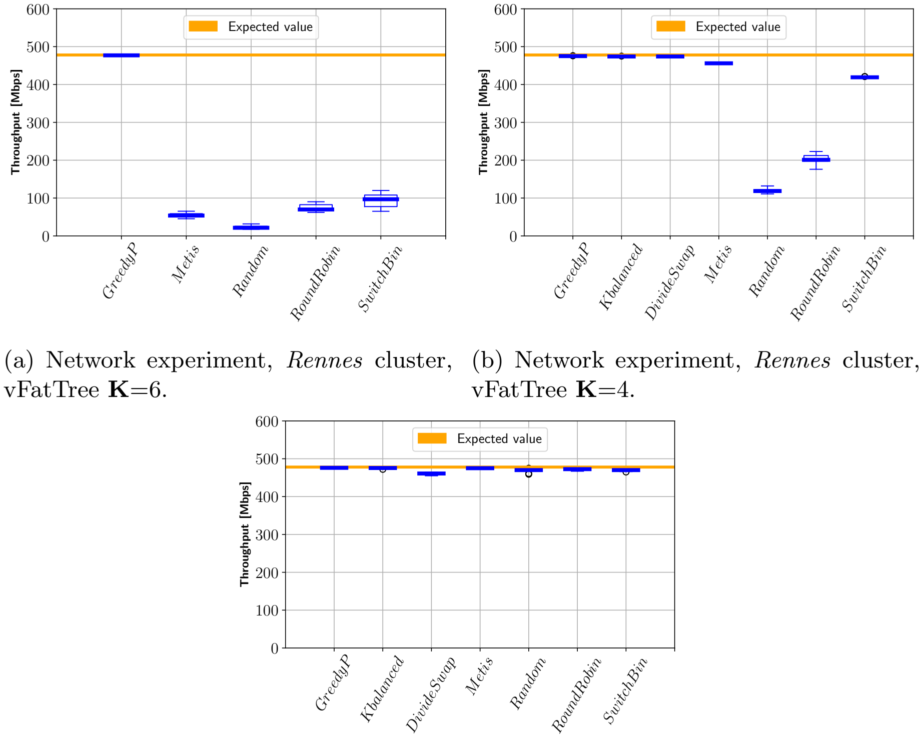


Figure 12: vFatTree K=4, bandwidth experiment.

They were performed using Grid'5000 [29] on two different clusters (*Gros* and *Rennes*). We performed three kinds of experiments: bandwidth, CPU, and memory intensive experiments discussed in Sections 5.1, 5.2, and 5.3, respectively. We show the very strong impact of placement on bandwidth usage and, thus, on emulation reliability, as well as the one on crashes and increased emulation completion time due to lack of memory or because of CPU overloading.

5.1 Bandwidth Intensive Experiments

The first experiment shows the performances of our placement module in a network intensive scenario. The networks we emulate are virtual Fat Trees with K=4 and K=6. They are composed of vHosts and vSwitches requiring 1 vCore and 1 GB of RAM, while all the vLinks are set to 500 Mbps. Half of the vHosts are clients and the other ones are servers. The experiment consists in running TCP `iperf` [30] between each pair of client/server. The total aggregated demand to be served is 4 Gbps and 13.5 Gbps, while the total network traffic generated is 24 Gbps and 81 Gbps, for K=4 and K=6, respectively. The traffic is forwarded in a way guaranteeing that each client is theoretically able to send at full speed (see Fig. 12 for an example). This is possible as a Fat Tree is a permutation network. Each experiment was performed 10 times, enough to obtain reliable results as the variability is small. To avoid the impact of the installation of the forwarding rules by the controller to the measurements, rules have been installed manually on each switch before starting sending the traffic. The first half of vHosts acts as a client, while the second half corresponds to the servers. Each vLink is set to 500 Mbps, so in theory each client should be able to send 500 Mbps to its corresponding virtual server, if the routes in the switch are correctly set. To check what the expected traffic is, we set up a simple experiment with `Mininet`. We create a simple topology with one switch and two vHosts and we measure the traffic with `iperf` for 60 s. The result with `Mininet` is 478 Mbps, so we will use this value as a baseline for the distributed



(a) Network experiment, *Rennes* cluster, vFatTree **K**=6.

(b) Network experiment, *Rennes* cluster, vFatTree **K**=4.

(c) Network experiment, *Gros* cluster, vFatTree **K**=6.

Figure 13: Network experiments

experiments.

Homogeneous Case. The first experiment shows the bandwidth measured in *Gros* cluster (see Section 4) with 20 physical nodes, virtualizing a vFT with K=6. The physical network in this cluster is a simple star topology with 25 Gbps links connecting all physical nodes to the central nodes. The achieved throughput values for all placement solutions are summarized in Fig. 13c by means of boxplots representing the distributions over all client/server pairs. Since each link has a capacity of 500 Mbps, the maximum `iperf` speed is slightly lower than this value (as for Mininet). We observe that in this simple case, all the algorithms return a solution that is not overloading any physical link nor any physical machine. The emulation is working as expected for each of the studied placement solutions.

Heterogeneous Case. We now consider the heterogeneous physical (yet simple) physical topology of the *Rennes* cluster (see Fig 2). Results using this experimental platform are illustrated in Figs. 13b and 13a. The experiments show how the emulation can return unexpected results when the placement of the virtual nodes does not take into account links' rates. Note that, as this topology is not homogeneous like the *Gros* cluster, finding a good placement is significantly harder, as discussed in Sec. 4.

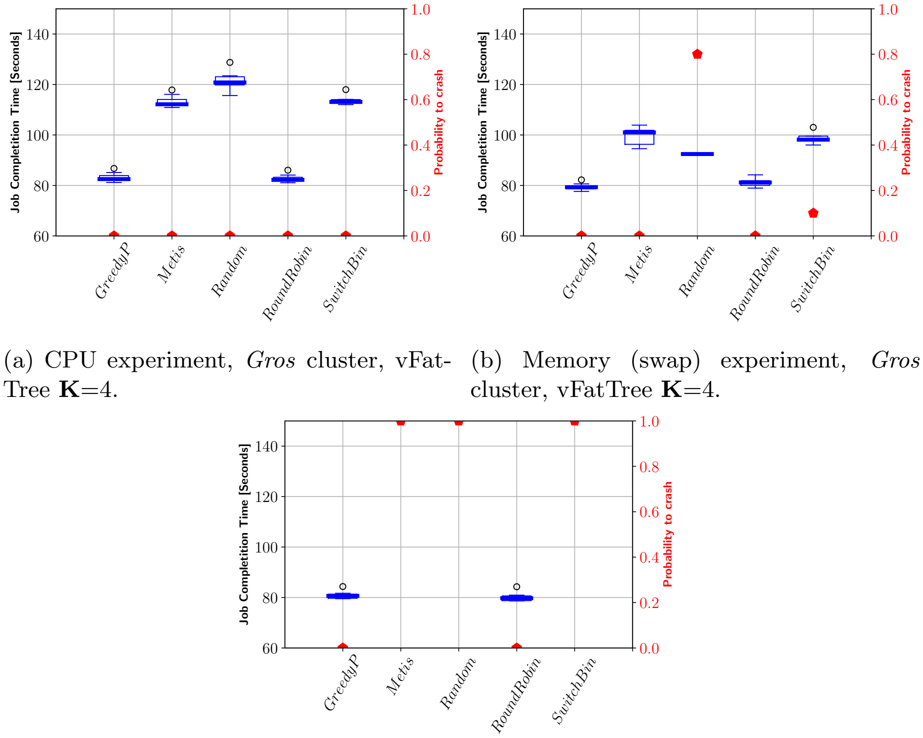
The first test in *Rennes* consists in emulating a vFT topology K=4 in this cluster using all the algorithms. In this case, the emulation creates a vFT with 16 vHosts and 20 vSwitches (see Fig. 12). Fig. 13b reports the bandwidth performance of a vFT (with K=4) obtained for the different placement algorithms when the emulation is performed in the *Rennes* cluster. As we can observe, the bandwidth results using GreedyPartition, k-balanced, DivideSwap, and Metis are the ones expected from the emulation, while the results obtained by other algorithms, Random and RoundRobin, are far from the expected ones. Indeed, some of the links are overloaded in the placement returned by the latter algorithms. This means that the paths of two demands are using the same links. For these links, the throughput drops to 120 Mbps and 200 Mbps, respectively. When running a larger emulation for a vFT with K=6, we can observe in Fig. 13a that k-balanced, DivideSwap do not find a feasible solution and the results returned by Metis and SwitchBin are not trustworthy anymore. The measured throughput on some overloaded links has fallen to very low values between 20 and 100 Mbps, to be compared with the expected 478 Mbps. On the contrary, the emulation distributed with GreedyPartition returns exactly the expected results, showing the efficiency and reliability of the proposed algorithm.

5.2 CPU Intensive Experiments

We now study the behavior of emulations for which a placement algorithm returns a solution with CPU overcommitment. We evaluate the impact on emulation execution time.

We built a CPU intensive scenario using Hadoop Apache [31]. The cluster used for this test is *Gros*, and the virtual topology is again a vFT with K=4. One vHost in the vFT is the Master, while the other vHosts are the Workers. The experiment consists in running a classical Hadoop benchmarking test. This test computes π using a quasi-Monte Carlo method and MapReduce, with 2000 maps and 1000 samples for each map. Each vHost requires 24 vCPUs and 32 GB of RAM, while the vSwitches only require 1 vCPU and 1 GB of RAM. Each machine of the cluster provides 32 vCores and 96 GB of RAM.

Fig. 14a reports the Hadoop Job Completion times for each placement algorithm. The boxplots provide their distributions over 10 experiments. The best performances are the ones of GreedyPartition and RoundRobin. The reason is that their solutions are not overloading any physical machine nor link. On the contrary, Metis and SwitchBin return the same placement overloading 8 physical machines in terms of CPU by 50% (48 vCores are assigned to 8 hosts with 32 cores available). The impact of this overcommitment on job completion time is an increase of around 40% as seen in the figure. Random returns a different placement for each experiment. For most of them, at least one physical machine hosts 3 virtual hosts leading to a CPU overcommitment



(a) CPU experiment, *Gros* cluster, vFatTree **K**=4.

(b) Memory (swap) experiment, *Gros* cluster, vFatTree **K**=4.

(c) Memory (no swap) experiment, *Gros* cluster, vFatTree **K**=4.

Figure 14: Resource-intensive experiments in Gros cluster.

of 100%, explaining why Random has the highest job completion time.

5.3 Memory Intensive Experiments

For the memory intensive tests, we create an experiment in which the nodes run at the same time an in-memory file storage and the Hadoop benchmark computing π used in the previous section. The first application was chosen as it is memory-hungry. The second application is used to assess the impact of memory overload on execution time in case the experiment does not crash due to a lack of memory. We chose the parameter of the experiments to ensure that each algorithm may return solutions with only memory overcommitment, but not CPU or network ones. We tested a scenario with low memory overcommitments, as large ones would induce direct crashes. We considered two scenarios for the impact of memory overcommitment: swap memory enabled or disabled in the physical machines. Note that `Distrinet` is dynamically allocating virtual memory. So, if the vHost is not using it, it is available for other vHosts or for other tasks in the hosting machine. The experiment creates a vFT with

K=4, in which each vHost requires 12 vCPU and 50 GB of RAM. A physical machine has 32 cores and 96 GB of RAM. Hence, if we assign 2 vHosts that use all 50 GB of RAM, the physical machine is overloaded in terms of memory by 4.2%. In these tests, just like for the CPU overcommitment ones, one vHost is the Master, while the other vHosts are the Workers.

Swap Enabled Case. The physical machines also provide 4 GB of swap memory. So, if the RAM is completely full (depending on the swappiness parameter in the kernel. For the tests we use the default value), the machine starts to use the swap memory (*Gros* cluster uses SSDs for the storage). Fig. 14b presents the results using the different placement algorithms. In this case, we observe that the overcommitment slows down the job completion time using Metis. We also notice that a large fraction of runs crashed, especially using Random (80% of the experiments did not succeed). This is due to a bad assignment of resources made by the algorithm (often 3 vHosts were assigned to the same physical host, leading to a memory overcommitment of 56%).

Swap Disabled Case. When the swap memory is disabled, if the assignment overloads the memory, the machine cannot rely on the SSD memory. In this case, we see in Fig. 14c that, with only 4% of memory overcommitment, the emulation cannot run with Metis or SwitchBin. We observed during the same experiment different behaviors. Sometimes, `Distrinet` containers simply crashed, while other times the in-memory files generated were corrupted during the emulations. In both cases, the emulation is considered crashed. Similar to the experiments with swap enabled, GreedyPartition and RoundRobin do not overcommit the physical machines and manage to complete the task without issues.

6 Conclusion

In this paper, we propose a placement module for tools enabling distributed network emulation. Indeed, large scale or resource intensive emulations have to be distributed over several physical hosts to avoid overloading hosts carrying out the emulation. A network experiment can be seen as a virtual network with resources needed on nodes (e.g., CPU or memory) and links (e.g., bandwidth). This network has to be mapped to the physical clusters on which the emulation is done. As we show in this paper, a bad mapping may lead to overload physical resources leading to untrustworthy experiments. This is the reason why we propose and evaluate in this paper placement algorithms that provide trustworthy mapping ensuring that resources are never overloaded. As a bonus, our algorithms also minimize the number of physical machines needed. We show that they outdo existing solutions in all aspects. With the ever-growing need of resources for experiments, we are convinced that having such a fast, efficient, and trustworthy placement module is essential for our community.

In this work, we consider the case of private testbeds for which we know and control the infrastructure. However, not all emulations are done on such platforms. Experiments may be done in private testbeds for which the infrastructure is known but the control is not total (e.g., the routing cannot be chosen), or in public clouds (e.g., Amazon EC2 or Microsoft Azure) in which the characteristics of available machines are known, but not the network interconnecting them. It would be interesting to study how to adapt the methods and algorithms developed in this work to scenarios in which the knowledge of, and the control over, the experimental infrastructure is partial.

References

[1] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: ACM, 2010, pp. 19:1–19:6. [Online]. Available: http://doi.acm.org/10.1145/1868447.1868466

[2] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. H. Zahraee, and H. Karl, "Maxinet: Distributed emulation of software-defined networks," in *2014 IFIP Networking Conference*, June 2014, pp. 1–9.

[3] "Cluster edition prototype," https://github.com/mininet/mininet/wiki/Cluster-Edition-Prototype.

[4] "Distrinet website," https://distrinet-emu.github.io.

[5] G. Di Lena, A. Tomassilli, D. Saucez, F. Giroire, T. Turletti, and C. Lac, "Mininet on steroids: exploiting the cloud for mininet performance," in *IEEE International Conference on Cloud Networking (CloudNet)*, 2019.

[6] ——, "Distrinet: a mininet implementation for the cloud," in *ACM SIGCOMM Computer Communication Review (CCR)*, Jan. 2021.

[7] "Metis documentation," http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.

[8] "Nancy ethernet network topology," https://www.grid5000.fr/w/Nancy:Network.

[9] P. Vicat-Blanc, B. Goglin, R. Guillier, and S. Soudan, *Computing networks: from cluster to cloud computing.* John Wiley & Sons, 2013.

[10] G. Di Lena, A. Tomassilli, F. Giroire, D. Saucez, T. Turletti, and C. Lac, "A right placement makes a happy emulator: a placement module for distributed sdn/nfv emulation," in *IEEE International Conference on Communications (ICC)*, 2021.

[11] G. Karypis and V. Kumar, "Multilevel algorithms for multi-constraint graph partitioning," in *SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing.* IEEE, 1998, pp. 28–28.

[12] A. Fischer, J. F. Botero, M. T. Beck, H. De Meer, and X. Hesselbach, "Virtual network embedding: A survey," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, pp. 1888–1906, 2013.

[13] N. M. K. Chowdhury, M. R. Rahman, and R. Boutaba, "Virtual network embedding with coordinated node and link mapping," in *IEEE INFOCOM 2009.* IEEE, 2009, pp. 783–791.

[14] M. Rost and S. Schmid, “Virtual network embedding approximations: Leveraging randomized rounding,” *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 2071–2084, 2019.

[15] G. Even, M. Medina, and B. Patt-Shamir, “On-line path computation and function placement in sdns,” *Theory of Computing Systems*, vol. 63, no. 2, pp. 306–325, 2019.

[16] M. Rost, E. Döhne, and S. Schmid, “Parametrized complexity of virtual network embeddings: Dynamic & linear programming approximations,” *ACM SIGCOMM Computer Communication Review*, vol. 49, 2019.

[17] J. F. Botero, X. Hesselbach, M. Duelli, D. Schlosser, A. Fischer, and H. De Meer, “Energy efficient virtual network embedding,” *IEEE Communications Letters*, vol. 16, no. 5, pp. 756–759, 2012.

[18] M. Chowdhury, M. R. Rahman, and R. Boutaba, “Vineyard: Virtual network embedding algorithms with coordinated node and link mapping,” *IEEE/ACM Transactions on Networking (TON)*, vol. 20, no. 1, pp. 206–219, 2012.

[19] M. Yu, Y. Yi, J. Rexford, and M. Chiang, “Rethinking virtual network embedding: substrate support for path splitting and migration,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 17–29, 2008.

[20] M. Melo, J. Carapinha, S. Sargento, L. Torres, P. N. Tran, U. Killat, and A. Timm-Giel, “Virtual network mapping–an optimization problem,” in *International Conference on Mobile Networks and Management.* Springer, 2011, pp. 187–200.

[21] I. Houidi, W. Louati, W. B. Ameur, and D. Zeghlache, “Virtual network provisioning across multiple substrate networks,” *Computer Networks*, vol. 55, no. 4, pp. 1011–1023, 2011.

[22] M. Rost and S. Schmid, “Charting the Complexity Landscape of Virtual Network Embeddings,” in *Proc. IFIP Networking*, 2018.

[23] M. R. Garey, D. S. Johnson, and L. Stockmeyer, “Some simplified np-complete problems,” in *Proceedings of the sixth annual ACM symposium on Theory of computing.* ACM, 1974, pp. 47–63.

[24] R. Krauthgamer, J. Naor, and R. Schwartz, “Partitioning graphs into balanced components,” in *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms.* SIAM, 2009, pp. 942–949.

[25] H. D. Simon and S.-H. Teng, “How good is recursive bisection?” *SIAM Journal on Scientific Computing*, vol. 18, no. 5, pp. 1436–1445, 1997.

[26] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell system technical journal*, vol. 49, no. 2, pp. 291–307, 1970.

[27] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham, "Worst-case performance bounds for simple one-dimensional packing algorithms," *SIAM Journal on computing*, vol. 3, no. 4, pp. 299–325, 1974.

[28] S. G. Kolliopoulos and C. Stein, "Improved approximation algorithms for unsplittable flow problems," in *Proceedings 38th Annual Symposium on Foundations of Computer Science*. IEEE, 1997, pp. 426–436.

[29] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, ser. Communications in Computer and Information Science, I. I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan, Eds. Springer International Publishing, 2013, vol. 367, pp. 3–20.

[30] A. Tirumala, "Iperf: The tcp/udp bandwidth measurement tool," *http://dast. nlanr. net/Projects/Iperf/*, 1999.

[31] "Apache hadoop documentation," http://hadoop.apache.org.

A Reproducibility

We have 3 kinds of experiments that can be reproduced. All the code is currently available in the repository:
https://github.com/distrinet-emu

To simplify the process, we use Vagrant to have a uniform environment. The tool can be installed in your system following the instructions in the official website:
https://www.vagrantup.com/docs/installation

First set of experiments. The first experiments that can be reproduced are the performance comparison between ILP and the algorithms GreedyPartition, k-balanced, and DivideSwap introduced in Sec. 3.

CPLEX needs to be installed if you are using your system to run the code or you have to install it manually in Vagrant.

The first experiment repository is:
https://github.com/distrinet-emu/algo_solver
After cloning the repo, you should go to the main directory and turn on the virtual machine with:

- *cd algo_solver*
- *vagrant up*
- *vagrant ssh*

Install CPLEX inside vagrant, then run and plot the experiment with:

- *cd /vagrant/algo_solver*
- *python3 exp.py*
- *python3 exp_plot.py*

To clean the environment:

- *exit*
- *vagrant destroy*

Second set of experiments. The second set of experiments are the simulations of Sec. 4. We first explain (i) how to plot the results pre-computed for the paper, then (ii) how to perform personalized simulations.

You have to clone the repository:
https://github.com/distrinet-emu/algo_simulations
(i) Note that it can take days to recompute all results. You need Anaconda installed on your system. Then, open the playbook located in:
algo_simulations/bin/algo_comparision.ipynb
If you run all the notebook, you obtain all the plots and the results reported in Sec. 4.

(ii) To run new simulations, you first need to turn on Vagrant:

- *vagrant up*
- *vagrant ssh*
- *sudo -i*
- *cd /vagrant/bin/*
- *python3 Comparison.py*

The pkl files will be created inside the directory. It can take some hours or days to have all the simulations (depending on your machine).

To clean the environment:

- *exit*
- *vagrant destroy*

Third set of experiments. The last set of experiments that can be reproduced are the emulations done on a real environment, i.e., iperf, Hadoop, and memory tests presented in Sec.5. Like the previous experiments, we first explain (i) how to plot the precomputed results and then (ii) how to run the real experiments.

(i) You have to clone the repository:
https://github.com/distrinet-emu/algo_experiments
Then, go to the directory algo_experiments and run:

- *vagrant up*
- *vagrant ssh*
- *cd /vagrant/algo_experiments/plot_results*
- *python3 box_plot.py*

(ii) Since the configuration and the installation of the environment for the real emulation are more complicated, we invite the user to follow the step by step tutorial present in the README.md file of the repository.

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399