Resilient application placement for geo-distributed cloud networks

Resilient Application Placement for Geo-Distributed Cloud Networks

Bart Spinnewyn[a,**], Ruben Mennes[a,*], Juan Felipe Botero[b,*], Steven Latré[a,*]

[a] *Department of Mathematics and Computer Science, University of Antwerp - iMinds, Antwerp, Belgium*
[b] *Department of Electronics and Telecommunications, University of Antioquia, Medellin, Colombia*

Abstract

The strong uptake of cloud computing has led to an important increase of mission-critical applications being placed on cloud environments. Those applications often require high levels of availability coupled with guarantees on a minimum level of throughput and a maximum level of response time. To achieve the lowest response time possible, clouds are more and more decentralized, leading to a heterogeneous network of micro clouds positioned on the edge of the network and possibly interconnected by best-effort links. This heterogeneous environment introduces important challenges for the management of these clouds as the heterogeneity results in an increased failure probability. In this paper, we address these challenges by providing a resilient placement of mission-critical applications on geo-distributed clouds. We present an exact solution to the problem, which is complemented by two heuristics: a near-optimal distributed genetic meta-heuristic and a scalable centralized heuristic based on subgraph isomorphism detection. A detailed performance evaluation shows that, with the newly proposed heuristic based on subgraph isomorphism detection, we can double the amount of applications satisfying availability requirements, in cloud environments comprising over 100 nodes, while keeping the time required to calculate the solution under 20 seconds.

Keywords: Cloud Computing, Quality of Service, Application Placement, Reliability

2010 MSC: 00-01, 99-00

*Corresponding author
**Principal corresponding author
Email addresses: bart.spinnewyn@uantwerpen.be (Bart Spinnewyn), ruben.mennes@uantwerpen.be (Ruben Mennes), juanf.botero@udea.edu.co (Juan Felipe Botero), steven.latre@uantwerpen.be (Steven Latré)

1. Introduction

Cloud computing offers computing resources as a utility: one no longer has to manage and maintain its own private computing infrastructure, instead computing infrastructure is time-shared and can be accessed on-demand. Cloud technology allows elastic scaling of resources when the demand for an application changes. While traditionally a cloud infrastructure is located within a data-center, recently, there is a need for geographical distribution. Consider for instance the case of cloud robotics [1]. Modern robots are capable of adapting to changing conditions, however they need a massive amount of intelligence to function properly, resulting in complex machines and control systems. Because of latency constraints, great care must be taken with the placement and management of the intelligence. On the one hand, time-critical control services such as navigation and sensor information processing must be placed close to the base station serving the working area. On the other hand, non time-critical services, such as the facility management function controlling the plant which houses the robots, can be placed remotely, as their actions do not affect real-time behavior.

Lately, this need for geo-distribution has led to a new evolution of decentralization. Most notably, the extension of cloud computing towards the edge of the enterprise network, is generally referred to as fog or edge computing [2]. In fog computing, computation is performed at the edge of the network at the gateway devices, reducing bandwidth requirements, latency, and the need for communicating data to the servers. Second, mist computing pushes processing even further to the network edge, involving the sensor and actuator devices [3]. Closely related to mist computing is the cloud robotics architecture put forward by Hu et al. [4]. The architecture leverages the combination of a virtual ad-hoc cloud formed by machine-to-machine (M2M) communications among participating robots, and an infrastructure cloud enabled by machine-to-cloud (M2C) communications.

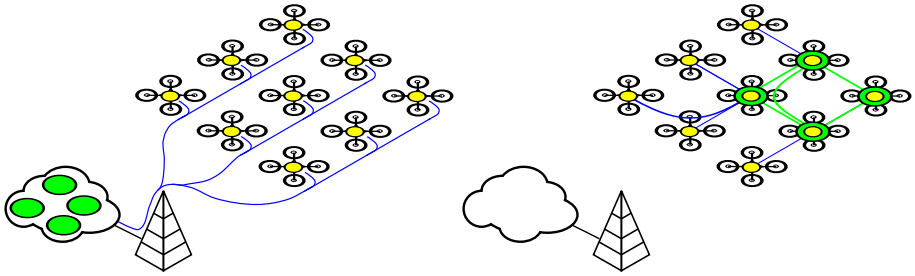
Compared to a traditional cloud computing environment, a geo-distributed cloud environment is less well-controlled and behaves in an ad-hoc manner. Devices may leave and join the network, or may become unavailable due to unpredictable failures or obstructions in the environment.

Additionally, while in a data-center heterogeneity is limited to multiple generations of servers being used, there is a large spread on capabilities within a geo-distributed cloud environment. Memory and processing means range from high (e.g. servers), over medium (e.g. cloudlets, gateways) to very low (e.g. mobile devices, sensor nodes). While some communication links guarantee a certain bandwidth (e.g. dedicated wired links), others provide a bandwidth with a certain probability (e.g. a shared wired link), and others do not provide any guarantees at all (wireless links).

Reliability is an important non-functional requirement, as it outlines *how* the software systems realizes its functionality [5]. The unreliability of substrate resources in a heterogeneous cloud environment, severely affects the reliability of the applications relying on those resources. Therefore, it is very challenging to host reliable applications on top of unreliable infrastructure [6].

Moreover, traditional cloud management algorithms cannot be applied here, as they generally consider powerful, always on servers, interconnected over wired links. Many algorithms do not even take into account bandwidth limitations. While such an omission can be justified by an appropriately overprovioned network bandwidth within a data-center, it is not warrented in the above described geo-distributed cloud networks.

As a motivating example, consider the problem introduced by Saska et al. [7]. In the investigated scenario, the formation of multiple Micro Air Vehicles (MAVs) has to reach a desired target region in a complex environment with obstacles, while keeping predefined relative positions between the robots. The authors present a Model Predictive Control (MPC) based algorithm for maintenance of leader-follower formations of MAVs. MPC is normally implemented in a centralized fashion. In large-scale interconnected systems a centralized control scheme may not be possible, and decentralized or distributed control is required [8]. In Figure 1, sense and actuation services (yellow) are placed



(a) Offloading to a remote cloud. (b) Execution on the drones.

Figure 1: Distribution of MPC services in a leader follower scenario for MAVs.

on each MAV. Four distributed MPC services (green) are needed for real-time control. A first way to distribute the MPC services is to offload them to a remote infrastructure cloud (Figure 1a). In this configuration, all communications between the actuation and sensing services happen via M2C communications, routed over one single base station, effectively forming a single point of failure. A second way of distribution is to execute each MPC service on a MAV (1b). In this configuration we only make use of M2M communications. While the first configuration can benefit from the virtually infinite computing capability that resides within the remote cloud infrastructure, the second has to make do with the limited computing capabilities within the MAVs. However, while the first configuration requires all sensing data to be uploaded to the cloud in real-time, the second configuration requires no up-link capabilities at all, as the sensing data is kept within the M2M network. Additionally, while the first configuration requires the MAVs, the remote cloud infrastructure and their M2C interconnections to be on-line, the second configuration requires the MAVs, and their M2M interconnections to be on-line. Whichever configuration results in the best availability will ultimately depend on the failure behavior of the physi-

cal resources used. To decide which is the best configuration, we need intelligent cloud management algorithms that, next to computing resources and bandwidth limitations, consider the availability of mission-critical applications.

Current management approaches fail to provide availability guarantees in these geo-distributed cloud networks for one of following reasons. First, most approaches do not consider availability at all. Second, there are cloud management solutions that consider failures, but lack a model to calculate availability. Finally, there are approaches that model availability, but make too limiting assumptions.

In this paper, we consider the problem of processing an initial collection of application requests upon startup of a cloud environment, which is referred to as the off-line Application Placement Problem (APP) [9]. The APP has two integral parts: first, there is admission control, which selects the application requests that are accepted. Second, there is placement control, which determines how to distribute the application components in the cloud. The off-line APP differs from the on-line version, in that the application requests are all considered simultaneously, and not sequentially. We model the application requests as Virtual Networks (VNs), consisting of services and their required communication channels. The cloud infrastructure is modeled as a Substrate Network (SN) consisting of physical nodes and their interconnecting links. The problem of mapping the VNs to a SN, whilst considering failures in the SN is known as the Survivable Virtual Network Embedding (SVNE) problem [10]. Given the failure behavior of the cloud infrastructure, we solve the problem of initial distribution of application components over the cloud environment, whilst satisfying a minimum level of total availability for each application.

To the best of our knowledge, this paper is the first that provides a computationally feasible approach to place applications in a realistic and large-scale failure prone cloud environment that provides guarantees on the availability. More specifically, the contributions of this paper are four-fold. First, we present a novel approach of placing applications in a failure prone cloud environment: by adding additional replicas we are able to provide availability guarantees. This is formulated as an Integer Linear Program (ILP), which can be used to find an exact solution. Second, we propose a distributed fault-tolerant meta-heuristic based on genetic programming: a distributed set of workers concurrently search for the best placement of applications (including the definition of replicas). Third, we present a scalable centralized algorithm using the paradigm of subgraph isomorphism: this heuristic approach provides ultra-fast placement of applications with a cost in optimality. Fourth, based on an extensive performance evaluation that investigates the performance of different application types, we provide clear guidelines on when and how to apply which application placement algorithm.

The remainder of the paper is organized as follows. Section 2 discusses related works on cloud management algorithms and survivability. In Section 3 our model for availability is introduced, and in Section 4 the APP is formulated as an Integer Linear Program (ILP), which will be used to find exact solutions. In Section 5 a near-optimal distributed genetic meta-heuristic, and a scalable

centralized heuristic based on subgraph isomorphism detection are presented. Section 6 presents simulations results that evaluate the proposed heuristics. Finally, Section 8 concludes the paper and summarizes the key findings.

2. Related work

In this section, the state of the art with regard to the APP in cloud environments is discussed. Early work on application placement merely considers nodal resources, such as Central Processing Unit (CPU) and memory capabilities. Deciding whether requests are accepted and where those virtual resources are placed then reduces to a Multiple Knapsack Problem (MKP) [11]. An MKP is known to be NP-hard and therefore optimal algorithms are hampered by scalability issues. A large body of work has been devoted to finding heuristic solutions. For instance, Xu et al. focus on the multi-objective Virtual Machines (VMs) placement problem [12]. They propose a genetic algorithm with fuzzy multi-objective evaluation for efficiently searching the large solution space and conveniently combining possibly conflicting objectives. While Yi et al. propose an evolutionary game theoretic framework for adaptive and stable application deployment in clouds [13]. Other works include Network Interface Card (NIC) capabilities as a dimension in the MKP [14] and assumes an over-provisioned inner-network. While plausible within the boundaries of one data-center, this condition rarely holds when a combination of multiple clouds or even a wireless environment is considered.

When the application placement not only decides where computational entities are hosted, but also decides on how the communication between those entities is routed in the SN, then we speak of *network-aware* APP. Network-aware application placement is closely tied to Virtual Network Embedding (VNE) [15].

An example of a network-aware approach is the work from Moens et al. [16]. It employs a Service Oriented Architecture (SOA), in which applications are constructed as a collection of communicating services. This optimal approach performs node and link mapping simultaneously. In contrast, other works try to reduce computational complexity by performing those tasks in distinct phases [17], [18].

While the traditional VNE problem assumes that the SN network remains operational at all times, the SVNE problem does consider failures in the SN. For instance, Ajtai et al. try and guarantee that a virtual network can still be embedded in a physical network, after k network components fail. They provide a theoretical framework for fault-tolerant graphs [19]. However, in this model, hardware failure can still result in service outage as migrations may be required before normal operation can continue.

Mihailescu et al. try to reduce network interference by placing Virtual Machines (VMs) that communicate frequently, and do not have anti-collocation constraints, on Physical Machines (PMs) located on the same racks [20]. Additionally, they uphold application availability when dealing with hardware failures by placing redundant VMs on separate server racks. A major shortcoming

is that the number of replicas to be placed, and the anti-collocation constraints are user-defined.

Csorba et al. propose a distributed algorithm to deploy replicas of VM images onto PMs that reside in different parts of the network [21]. The objective is to construct balanced and dependable deployment configurations that are resilient. Again, the number of replicas to be placed is assumed predefined.

SiMPLE allocates additional bandwidth resources along multiple disjoint paths in the SN [22]. This proactive approach assumes splittable flow, i.e. the bandwidth required for a Virtual Link (VL) can be realized by combining multiple parallel connections between the two end points. The goal of SiMPLE is to minimize the total bandwidth that must be reserved, while still guaranteeing survivability against single link failures. However, an important drawback is that while the required bandwidth decreases as the number of parallel paths increases, the probability of more than one path failing goes up exponentially, effectively reducing the VL's availability.

Chowdhury et al. propose Dedicated Protection for Virtual Network Embedding (DRONE) [23]. DRONE guarantees VN survivability against single link or node failure, by creating two VNEs for each request. These two VNEs cannot share any nodes and links.

Aforementioned SVNE approaches [19], [20], [21], [22], [23] lack an availability model. When the infrastructure is homogeneous, it might suffice to say that each VN or VNE need a predefined number of replicas. However, in geo-distributed cloud environments the resulting availability will largely be determined by the exact placement configuration, as moving one service from an unreliable node to a more reliable one can make all the difference. Therefore, geo-distributed cloud environments require SVNE approaches which have a computational model for availability as a function of SN failure distributions and placement configuration.

The following cloud management algorithms have a model to calculate availability. Jayasinghe et al. model cloud infrastructure as a tree structure with arbitrary depth [24]. Physical hosts on which VMs are hosted are the leaves of this tree, while the ancestors comprise regions and availability zones. The nodes at bottom level are physical hosts where VMs are hosted. Wang et al. were the first to provide a mathematical model to estimate the resulting availability from such a tree structure [25]. They calculate the availability of a single VM as the probability that neither the leaf itself, nor any of its ancestors fail. Their work focuses on handling workload variations by a combination of vertical and horizontal scaling of VMs. Horizontal scaling launches or suspends additional VMs, while vertical scaling alters VM dimensions. The total availability is then the probability that at least one of the VMs is available. While their model suffices for traditional clouds, it is ill-suited for a geo-distributed cloud environment as link failure and bandwidth limitations are disregarded.

In contrast, Yeow et al. define reliability as the probability that critical nodes of a virtual infrastructure remain in operation over all possible failures [26]. They propose an approach in which backup resources are pooled and shared across multiple virtual infrastructures. Their algorithm first determines the re-

quired redundancy level and subsequently performs the actual placement. However, decoupling those two operations is only permissible when link failure can be omitted and nodes are homogeneous.

In previous work [27], an availability model for geo-distributed cloud networks was introduced, which considers any combination of node and link failures, and supports both node and link replication. The aforementioned model was employed to study the problem of guaranteeing a certain level of availability for applications. Using an ILP formulation of the problem and an exact solver, an increased placement ratio was demonstrated, compared to naive approaches which lack an availability model. While the ILP solver can find optimal placement configurations for small scale networks, its computation time quickly becomes unmanageable when the substrate network dimensions increase. In [28] a first heuristic is presented. This distributed evolutionary algorithm employs a pool-model, where execution of computational tasks and storage of the population database (DB) are separated.

Compared to previous work, this paper presents the following novelties. First, a fast new algorithm, based on subgraph isomorphism detection, is introduced. In contrast to previous work, this new algorithm is scalable and is the only one that is applicable for real-life large-scale environemnts. Second, a much more extensive evaluation is provided, considering multiple SN topologies and dimensions, and application types. In comparison to our previous work, next to a flat SN, now also real-world Internet type topologies, generated by a transit-stub model, are studied. Additionally, not only unstructured applications, but also more practical application models, namely 3-Tier and MapReduce, are simulated. Third, we carry out a detailed comparative study to the performance of the presented heuristics, relative to traditional placement algorithms, and homogeneous survivability methods. This study provides clear guidelines about the applicability of each algorithm.

3. Resilient cloud placement model

3.1. Application requests

This paper considers a SOA, which is a way of structuring IT solutions that leverage resources distributed across the network [29]. In a SOA, each application is described as its composition of services. Throughout this work, the collected composition of all requested applications will be represented by the instance matrix ($\boldsymbol{I}$).

Services have certain CPU ($\boldsymbol{\omega}$) and memory requirements ($\boldsymbol{\gamma}$). Additionally, bandwidth ($\boldsymbol{\beta}$) is required by the VLs between any two services. A sub-modular approach allows sharing of memory resources amongst services belonging to multiple applications.

3.2. Cloud infrastructure

Consider a substrate network consisting of nodes and links. Nodes have certain CPU ($\boldsymbol{\Omega}$) and memory capabilities ($\boldsymbol{\Gamma}$). Physical links between nodes

Symbol	Description
$\mathbf{A}$	set of requested applications
$\mathbf{S}$	set of services
ω_s	CPU requirement of service s
γ_s	memory requirement of service s
β_{s_1,s_2}	bandwidth requirement between services s_1 and s_2
$I_{a,s}$	instantiation of service s by application a: 1 if instanced, else 0
$\mathbf{N}$	set of physical nodes comprising the substrate network
$\mathbf{E}$	set of physical links (edges) comprising the substrate network
Ω_n	CPU capacity of node n
Γ_n	memory capacity of node n
p_n^N	probability of failure of node n
B_e	bandwidth capacity of link e
p_e^E	probability of failure of link e
R_a	required total availability of application a: lower bound on the probability that at least one of the duplicates for a is available
δ	maximum allowed number of duplicates

Table 1: Overview of input variables to the Cloud Application Placement Problem (CAPP).

are characterized by a given bandwidth ($\boldsymbol{B}$). Both links and nodes have a known probability of failure, $\boldsymbol{p^N}$ and $\boldsymbol{p^E}$ respectively. Failures are considered to be independent.

3.3. The VAR protection method

Availability not only depends on failure in the SN, but also on how the application is placed. Non-redundant application placement assigns each service and VL at most once, while its redundant counterpart can place those virtual resources more than once. The survivability method presented in this work, referred to as VAR, guarantees a minimum availabity by application level replication, while minimizing the overhead imposed by allocation of those additional resources. VAR uses a static failure model, i.e. availability only depends on the current state of the network. Additionally, it is assumed that upon failure, switching between multiple application instances takes place without any delay. These separate application instances will be referred to as duplicates. Immediate switchover yields a good approximation, when the duration of switchover is small compared to the uptime of individual components. A small switchover time is feasible, given that each backup service is preloaded in memory, and CPU and bandwidth resources have been preallocated. Furthermore, immediate switchover allows condensation of the exact failure dynamics of each component, into its expected availability value, as long as the individual components fail independently (a more limiting assumption).

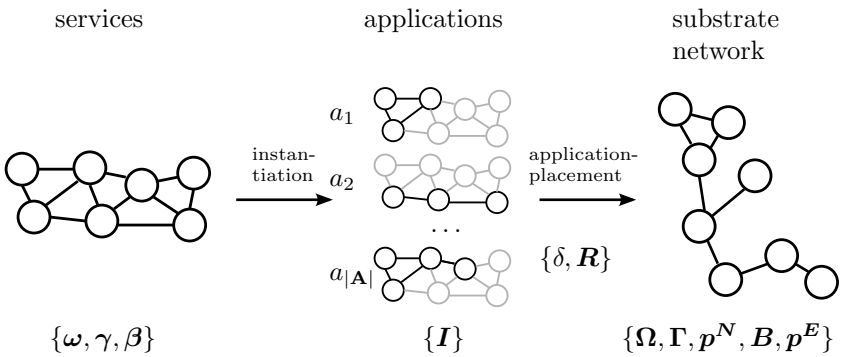
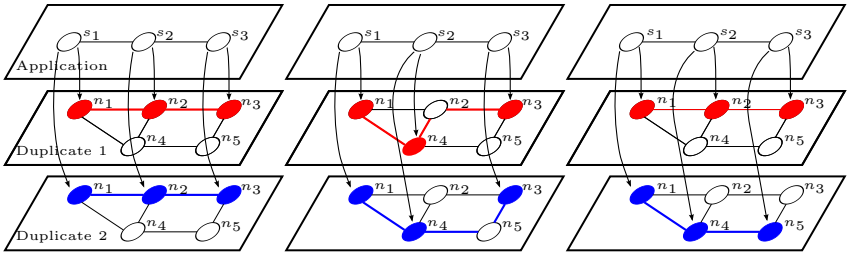


Figure 2: Overview of this work: applications $\{\boldsymbol{\omega}, \boldsymbol{\gamma}, \boldsymbol{\beta}\}$, composed of services $\{\boldsymbol{I}\}$, are placed on a substrate network where node $\{\boldsymbol{p^N}\}$ and link failure $\{\boldsymbol{p^E}\}$ is modeled. By increasing the redundancy δ, a minimum availability $\boldsymbol{R}$ can be guaranteed.

	sharing of resources		
	CPU	**memory**	**bandwidth**
within application	yes	yes	yes
amongst applications	no	yes	no

Table 2: An overview of resource sharing amongst identical services and VLs.

In the VAR model, an application is available if at least one of its duplicates is on-line. A duplicate is on-line if none of the PMs and Physical Links (PLs), that contribute its placement, fail. Duplicates of the same application can share physical components. An advantage of this reuse is that a fine-grained tradeoff can be made between increased availability, and decreased resource consumption. An overview of resources' reuse is shown in Table 2. In Figure 3



(a) 0 replicated services (b) 1 replicated VL (c) 2 replicated services

Figure 3: Illustration of the VAR protection method.

three possible placement configurations using two duplicates are shown for one application. In Figure 3a both duplicates are identical, and no redundancy is introduced. The nodal resource consumption is minimal, as CPU and memory for s_1, s_2, and s_3 are provisioned only once. Additionally, the total bandwidth

required for (s_1, s_2), and (s_2, s_3) is only provisioned once. The bandwidth consumption of this configuration might not be minimal, if consolidation of two or three services onto one PM is possible. This placement configuration does not provide any fault-tolerance, as failure of either n_1, n_2 or n_3, or $(n_1, n_2), (n_2, n_3)$ results in downtime.

When more than one duplicate is placed and the resulting arrangements of VLs and services differ, then the placement is said to introduce redundancy. However, this increased redundancy results in a higher resource consumption. In Figure 3b the application survives a singular failure of either (n_4, n_2), (n_2, n_3), (n_4, n_5), or (n_5, n_3). The placement configuration depicted in Figure 3c survives all singular failures in the SN, except for a failure of n_1.

Duplicates can be seen as a generalization of the placement configuration model defined by Chowdhury et al. [23]. The authors place a primary and backup VN to guarantee survivability of single node or link failures, which can be considered two duplicates of the same VN. There are three key differences between their model and VAR's. First, Chowdhury et al. require the placement of exactly two duplicates, while VAR supports any number of duplicates. Second, their approach requires all services for one duplicate to be located on different nodes. In our approach, services of one duplicate can be consolidated onto one physical node. Consolidation offers the possibility to increase availability of a duplicate, and avoids wasting precious bandwidth resources. Third, their model does not allow duplicates of the same application to share PMs or PLs. In our model, duplicates of the same application can either have no parts of the SN in common, have some parts of the SN in common (and possibly share resources), or even have completely identical placement configurations (and require no additional resources). Those three differences mean that their model cannot yield any feasible solution for the problem depicted in Figure 3.

3.4. Availability calculation

In the previous section, an application was defined available, if at least one of its duplicates is on-line. Hence, the total availability of an application is then the probability that at least one of its duplicates is available. When at most δ duplicates are considered, then the total availability of application a is given by

$$Z(a) = \mathbf{P}\left[\bigcup_{d=1}^{\delta} D_d^a\right],$$

where D_d^a denotes the event that duplicate d, of application a is available. The event that this duplicate is not available is denoted by $\overline{D_d^a}$.

4. Formal problem description

In this section, the problem is formulated as a binary ILP. The input variables to the model were already described throughout Section 3. Given those input variables, the algorithm finds a value for the decision variables listed in

Section 4.1 that minimizes the objective function (Section 4.3). The optimization is subject to the constraints listed in Section 4.2.

Symbol	Description
$\mathbf{C}$	set of physical components in the SN, i.e. nodes and edges ($\mathbf{C} = \mathbf{N} \cup \mathbf{E}$)
$\mathbf{D}$	set of duplicates
$\mathbf{M}$	set of minterms
$\mathbf{X}$	set of all possible states
$\mathbf{X}(m)$	particular state of the substrate network, the state of each component follows from m according to Equations 4 and 5
χ_c	state of physical component c
$b_c(m)$	value of χ_c for component c in minterm m
$\zeta(d, a)$	availability of duplicate d for application a
$Z(a)$	joint availability of application a

Table 3: Overview of auxiliary symbols used throughout the formulation of the ILP.

4.1. Decision variables

The decision variables are described in Table 4. O indicates which application requests are accepted, while G provides detailed information about which duplicates are actually placed. Information about the assignment of services to physical nodes is contained in π, Π and U, while υ and Υ tell us how the VLs are routed over the PLs. K, τ and T are directly used for availability calculation. Auxiliary variables are described in Table 3.

4.2. Constraints

4.2.1. Admission control

At most, δ duplicates can be placed for each application:

$$|\mathbf{D}| = \delta.$$

An application can only be accepted if at least one of its duplicates is placed:

$$\forall a \in \mathbf{A} : O^a \leq \sum_{d \in \mathbf{D}} G^{d,a}.$$

4.2.2. Node-embedding

Nodal resources are only assigned to duplicates if they are considered placed:

$$\forall a \in \mathbf{A}, s \in \mathbf{S}, n \in \mathbf{N}, d \in \mathbf{D} : \pi_{s,n}^{d,a} \leq G^{d,a} \times I_{a,s}.$$

The number of services hosted for each accepted duplicate equals the total number of instantiated services. If a duplicate is not placed, no services are

Symbol	Description
O^a	acceptance of application a: 1 i.f.f. accepted
$G^{d,a}$	placement of duplicate d of application a: 1 i.f.f. placed
$\pi_{s,n}^{d,a}$	placement of service s for duplicate d of application a on node n: 1 i.f.f. hosted
$\Pi_{s,n}^{a}$	use of node n for hosting of service s by application a: 1 i.f.f. used
$U_{s,n}$	hosting of service s on node n: 1 i.f.f. hosted
$\upsilon_{s_1,s_2}^{d,a}(e)$	placement of VL between services s_1 and s_2 on physical link e for duplicate d of application a: 1 i.f.f. placed
$\Upsilon_{s_1,s_2}^{a}(e)$	use of physical link e by at least one duplicate of application a for the placement of the VL between s_1 and s_2: 1 i.f.f. placed
$K_c^{d,a}$	use of physical component c by duplicate d of application a: 1 i.f.f. used
$\tau_m^{d,a}$	coverage of minterm m by duplicate d of application a: 1 i.f.f. covered m
T_m^a	availability of application a when the state of the network equals $\mathbf{X}(m)$: 1 i.f.f. available

Table 4: Overview of decision variables to the binary ILP: variables can only assume 0 or 1.

instantiated:

$$\forall a \in \mathbf{A}, d \in \mathbf{D} : G^{d,a} \times \sum_{s \in \mathbf{S}} I_{a,s} = \sum_{s \in S} \sum_{n \in N} \pi_{s,n}^{d,a}.$$

If a service is hosted on a node for any of its duplicates, then $\Pi_{s,n}^{a}$ equals 1:

$$\forall a \in \mathbf{A}, d \in \mathbf{D}, s \in \mathbf{S}, n \in \mathbf{N} : \pi_{s,n}^{d,a} \leq \Pi_{s,n}^{a}.$$

For each duplicate a service is hosted on at most one node:

$$\forall a \in \mathbf{A}, d \in \mathbf{D}, s \in \mathbf{S} : \sum_{n \in \mathbf{N}} \pi_{s,n}^{d,a} \leq 1.$$

Conservation of CPU and memory resources dictates:

$$\forall n \in \mathbf{N} : \sum_{a \in \mathbf{A}} \sum_{s \in \mathbf{S}} \Pi_{s,n}^{a} \times \omega_s \leq \Omega_n$$

and

$$\forall n \in \mathbf{N} : \sum_{s \in S} U_{s,n} \times \gamma_s \leq \Gamma_n.$$

A service must be hosted on a node, as soon as it is used by one of the duplicates:

$$\forall s \in \mathbf{S}, \forall n \in \mathbf{N} : \sum_{a \in \mathbf{A}} \sum_{d \in \mathbf{D}} \pi_{s,n}^{d,a} \leq U_{s,n} \times |\mathbf{D}| \times \sum_{a \in \mathbf{A}} I_{a,s}.$$

4.2.3. Link-embedding

Multi Commodity Flow (MCF) constraints on each node can be expressed as: $\forall a \in \mathbf{A}, s_1, s_2 \in \mathbf{S}, d \in \mathbf{D}, n_1 \in \mathbf{N}$:

$$\sum_{(n_1,n_2) \in \mathbf{E}} \upsilon_{s_1,s_2}^{d,a}(n_1, n_2) - \sum_{(n_2,n_1) \in \mathbf{E}} \upsilon_{s_1,s_2}^{d,a}(n_2, n_1) = \pi_{s_1,n_1}^{d,a} - \pi_{s_2,n_1}^{d,a}.$$

$\Upsilon_{s_1,s_2}^{a}(e)$ indicates if at least one of an application's duplicates uses e for this VL: $\forall a \in \mathbf{A}, s_1 \in \mathbf{S}, s_2 \in \mathbf{S}, e \in \mathbf{E}, d \in \mathbf{D}$:

$$\upsilon_{s_1,s_2}^{d,a}(e) \leq \Upsilon_{s_1,s_2}^{a}(e).$$

The total bandwidth used per link cannot exceed the total link capacity:

$$\forall e \in \mathbf{E} : \sum_{s_1 \in \mathbf{S}} \sum_{s_2 \in \mathbf{S}} \sum_{a \in \mathbf{A}} \Upsilon_{s_1,s_2}^{a}(e) \times \beta_{s_1,s_2} \leq B_e.$$

4.2.4. Availability-awareness

For a duplicate to be available, each of the individual components it uses must be available. A component is used by a duplicate if it hosts any of the duplicate's services or VLs: $\forall a \in \mathbf{A}, d \in \mathbf{D}, c \in \mathbf{C}, s_1, s_2 \in \mathbf{S}$:

$$K_c^{d,a} \geq \begin{cases} \pi_{s_1,c}^{d,a} & \text{if } c \in \mathbf{N} \\ \Upsilon_{s_1,s_2}^{d,a}(c) & \text{if } c \in \mathbf{E} \end{cases}. \tag{1}$$

The state of an individual component is described as:

$$\forall c \in \mathbf{C} : \chi_c = \begin{cases} 0 & \text{if } c \text{ fails} \\ 1 & \text{if } c \text{ does not fail} \end{cases}. \tag{2}$$

The probability that a component fails is given by:

$$\forall c \in \mathbf{C} : \mathrm{P}\left[\chi_c = 0\right] = \begin{cases} p_c^N & \text{if } c \in \mathbf{N} \\ p_e^N & \text{if } c \in \mathbf{E} \end{cases}. \tag{3}$$

The state of the substrate network can then be described as:

$$\mathbf{X} = (\chi_1, \chi_2, \ldots, \chi_{|\mathbf{C}|}).$$

To facilitate systematical description of all possible SN states, which will be further referred to as minterms, the following notation is introduced:

$$\mathbf{M} = \{0, 1, \ldots, 2^{|\mathbf{C}|} - 1\}$$

and

$$\forall m \in \mathbf{M} : \mathbf{X}(m) = \mathbf{X} | \forall c \in \mathbf{C} : \chi_c = b_c(m), \tag{4}$$

where $b_c(m) \in \{0, 1\}$ is defined by:

$$\forall m \in \mathbf{M} : m = \sum_{c \in \{0,1,\ldots,|\mathbf{C}|-1\}} b_c(m) \times 2^c. \tag{5}$$

As component failures are assumed independent, the probability of each minterm is given by:

$$\forall m \in \mathbf{M} : \mathrm{P}\left[\mathbf{X} = \mathbf{X}(m)\right] = \prod_{c \in \mathbf{C} | b_c(m)=0} \mathrm{P}\left[\chi_c = 0\right] \times \prod_{c \in \mathbf{C} | b_c(m)=1} \mathrm{P}\left[\chi_c = 1\right].$$

As stated earlier, a duplicate is available, if all physical components that contribute to its placement are on-line:

$$\begin{aligned} \forall a \in \mathbf{A}, d \in \mathbf{D} : \zeta(a, d) &= \mathrm{P}\left[\bigcap_{c \in \mathbf{C}} (\chi_c{=}1) \cup (K_c^{d,a}{=}\,0)\right] \\ &= \sum_{m \in \mathbf{M}} \tau_m^{d,a} \mathrm{P}\left[\mathbf{X}{=}\mathbf{X}(m)\right], \end{aligned}$$

where the law of total probability was used:

$$\tau_m^{d,a} = \mathrm{P}\left[\bigcap_{c \in \mathbf{C}} (\chi_c{=}1) \cup (K_c^{d,a}{=}0) | \mathbf{X}{=}\mathbf{X}(m)\right].$$

For the ILP this is reformulated as Equation 6. An additional $G^{d,a}$ term ensures that no minterm is covered when a duplicate is not placed ($G^{d,a}{=}0$): $\forall m \in \mathbf{M}, d \in \mathbf{D}, c \in \mathbf{C}, a \in \mathbf{A}$:

$$\tau_m^{d,a} \leq G^{d,a} + (b_c(m) - 1)\, K_c^{d,a}. \tag{6}$$

Finally, an application is available if at least one of its duplicates is available:

$$\forall a \in \mathbf{A}, m \in \mathbf{M} : T_m^a = \mathrm{P}\left[\bigcup_{d \in \mathbf{D}} \tau_m^{d,a}\right],$$

which can be formulated as:

$$\forall m \in \mathbf{M}, a \in \mathbf{A} : T_m^a \leq \sum_{d \in \mathbf{D}} \tau_m^{d,a}.$$

The total availability of an application is then given by:

$$\forall a \in \mathbf{A} : Z(a) = \sum_{m \in \mathbf{M}} T_m^a \mathrm{P}\left[\mathbf{X}{=}\mathbf{X}(m)\right].$$

Finally the condition that an application is only placed if the joint availability exceeds R_a can be written as:

$$\forall a \in \mathbf{A} : 1 - O^a + \sum_{m \in \mathbf{M}} T_m^a \mathrm{P}\left[\mathbf{X}{=}\mathbf{X}(m)\right] \geq R_a.$$

4.3. Objective function

The placement is sequentially optimized in multiple steps. In each step an objective function is minimized and results of previous steps are added as equality constraints. The objective functions are listed in the order in which they are used by the algorithm.

Maximize acceptance:

$$f_1(\mathbf{A}) = -\sum_{a \in \mathbf{A}} O^a.$$

Minimize bandwidth usage:

$$f_2(\mathbf{A}, \mathbf{E}, \mathbf{S}, \boldsymbol{\beta}) = \sum_{a \in \mathbf{A}} \sum_{e \in \mathbf{E}} \sum_{s_1, s_2 \in \mathbf{S}} \Upsilon_{s_1,s_2}^a(e) \times \beta_{s_1,s_2}.$$

Minimize CPU resources usage:

$$f_3(\mathbf{A}, \mathbf{N}, \mathbf{S}, \boldsymbol{\omega}) = \sum_{n \in \mathbf{N}} \sum_{a \in \mathbf{A}} \sum_{s \in \mathbf{S}} \Pi_{s,n}^a \times \omega_s.$$

Minimize the number of duplicates used:

$$f_4(\mathbf{A}, \mathbf{D}) = \sum_{a \in \mathbf{A}} \sum_{d \in \mathbf{D}} G^{d,a}.$$

The last objective function ensures that multiple duplicates of the same application are only placed if beneficial to maximize the placement ratio, or minimize resource usage.

5. Heuristic approaches for real-time calculation

The ILP formulation presented in the previous chapter can be used to find exact solutions to the problem. In Section 6, it will be shown that when the

dimensions of the problem increase, even for small instances finding an exact solution to the problem quickly becomes computationally intractable. Therefore two heuristic algorithms, which can find "good-enough" solutions within a reasonable time-frame, were developed.

5.1. GRECO: Genetic Reliable ClOuds application placement algorithm

A scalable algorithm to search for a good placement solution, by using a distributed Genetic Algorithm (GA), is defined. This algorithm is validated in a pool-based framework, which allows a completely decentralized computation of the solution and can even survive when the nodes that calculate the solution fail. In this section, a Genetic Reliable ClOuds application placement algorithm (GRECO) is described. Firstly, Section 5.1.1 explains the foundations of a GA. Secondly, the chromosome and its decoding, are explained in 5.1.2 and 5.1.3, respectively. Finally, Section 5.1.4 describes the pool model.

5.1.1. Genetic Algorithm

GAs are a common tool to solve hard optimization problems. A GA uses a chromosome representation to represent solutions in the solution space. A chromosome representation of one particular solution is referred to as an individual. In a population-based GA multiple individuals are maintained at each time during execution of the algorithm. The GA starts by creating a random set of individuals, referred to as the seed population. Additionally, through several iterations, the GA selects the best individuals (based on a selection operator). New solutions are generated in each iteration by combining chromosomes two by two, producing (hopefully better) children. With a small probability there can be a small mutation on one of the chromosomes. The last iteration step is to check the end condition, which is highly problem-specific. The key idea behind a GA is the use of the laws of natural selection and survival of the fittest to generate the solution [30].

5.1.2. Chromosome definition

The key challenge in creating a GA is finding a suitable chromosome representation, because this representation will largely determine its performance. A Biased-Random-key chromosome is proposed, which is an array of floating point values between 0 and 1, and is used to make decisions in the decoding phase of the chromosome [31]. A biased-random-key allows definition of a decoder, which can't possibly create invalid solutions. Hence, one can be sure that the offspring of two valid parent solutions, will also be valid solutions.

$$C = \Big[\underbrace{A_1, A_2, \cdots, A_{|\mathbf{A}|}}_{\text{Order of the applications}} , \underbrace{S_1^{1,1}, S_2^{1,1}, \cdots, S_{|\mathbf{S}_1|}^{1,1}, S_1^{1,2}, \cdots, S_{|\mathbf{S}_1|}^{1,\delta_1}, \cdots, S_{|\mathbf{S}_{|\mathbf{A}|}|}^{|\mathbf{A}|,\delta_{|A|}}}_{\text{Order of the services for each duplicate}}, \underbrace{N_1^{1,1}, N_2^{1,1}, \cdots, N_{|\mathbf{S}_1|}^{1,1}, N_1^{1,2}, \cdots, N_{|\mathbf{S}_1|}^{1,\delta_1}, \cdots, N_{|\mathbf{S}_{|\mathbf{A}|}|}^{|\mathbf{A}|,\delta_{|A|}}}_{\text{PM number for each service in each duplicate}} \Big] \tag{7}$$

The chromosome described in Equation 7 consists of three main parts:

1. $A_1, \cdots, A_{|\mathbf{A}|}$, describes the order in which the applications are placed.
2. $S_1^{1,1}, \cdots, S_{|\mathbf{S}_{|\mathbf{A}|}|}^{|\mathbf{A}|,\delta_{|A|}}$ describes the order in which the services are placed within each duplicate.
3. $N_1^{1,1}, \cdots, N_{|\mathbf{S}_{|\mathbf{A}|}|}^{|\mathbf{A}|,\delta_{|A|}}$ determine which PM hosts which service for which duplicate.

5.1.3. Deterministic Decoding

When using a biased-random-key, there is no straight-forward way to interpret the chromosome. Therefore, a decoding algorithm is used to translate the chromosome into a solution in the solution space.

Within GRECO, the approach shown in Algorithm 1, is taken. First, the applications are ordered by the value of the chromosome's first part ($A_1, \cdots, A_{|\mathbf{A}|}$), as this will define the order in which applications are prioritized in the actual placement, the first step of Algorithm 1. Placement of an application, starts by allocating its first duplicate. If that duplicate is allocated, then the realized availability is calculated (line 6). If the realized availability exceeds the requested availability, then the application has succesfully been allocated, and the decoder proceeds to the next application. On the other hand, if the realized availability is lower than the requested availability, and the maximum number of duplicates for this application has not been exceeded, then the decoder tries to place an additional duplicate. If the required availability level is not met and the maximum number of duplicates has been placed, then the application request is declined and its placement removed.

To place duplicate d of application a, first its services are ordered by the value of the second part of the chromosome $S_1^{a,d}, S_2^{a,d}, \cdots S_{|S_a|}^{a,d}$, (line 8). For each service s a list (L) is created, containing only the PMs that can run service s, while satisfying all necessary constraints (line 10). For each PM it must be verified if the remaining CPU and memory capacity suffice, and if there is sufficient bandwidth to the PMs, that were added to the placement configuration previously. Only PMs that satisfy those three constraints are included in L. If list L is empty, then application a is removed from the placement because there is no valid way to place a (line 12). Else, if $|L| > 0$, then L is ordered and the

n^{th} PM is selected (line 15). This PM will host service s for duplicate d and the required bandwidth for the VLs between s and the previously handled services is allocated. The VLs is routed over the shortest path with sufficient remaining bandwidth.

It is mandatory that a certain genome maps to one and only one placement configuration. Therefore the PMs must have a total deterministic order, independent of the use of the PMs. The PMs are sorted by id, this is easy, deterministic and generally results in a good random order for the last step. The decoding algorithm is illustrated in Algorithm 1.

Algorithm 1 Decoding algorithm

```
1: procedure DECODING(A_1.., S_1^{1,1}.., N_1^{1,1}..)
2:     sortByChromosome(A, A_1..A_{|A|})
3:     for each a ∈ A do
4:         δ = 0
5:         r = 0
6:         while d ≤ δ_a ∧ r < R_a do
7:             d+ = 1
8:             sortByChromosome(S_a, S_1^{a,d}..S_{|S_a|}^{a,d})
9:             for each s ∈ S_a do
10:                L = allPossiblePM(s)
11:                if |L| == 0 then
12:                    break
13:                end if
14:                sortByID(L)
15:                n = ⌈N_s^{a,d}|L|⌉
16:                placeService(s, n)
17:            end for
18:            r = calculateAvailability(a)
19:        end while
20:        if r < R_a then removeApplication(a)
21:        end if
22:    end for
23: end procedure
```

5.1.4. Framework: Pool model

There exist various models to distribute the population amongst the computing nodes. Amongst those distribution models for population-distributed GAs, the pool-model offers the best combination of scalability and fault-tolerance. While in other distribution models, worker nodes (called workers) are responsible of both executing GA operations, and storing part of the population database, the pool-model maintains the population in a separate DB. A pool model deploys a set of autonomous processors working on a shared resource pool. The processors are loosely coupled, do not know of each other's existence and interact with only the pool.

The workflow of a GRECO worker is shown in Figure 4. First, the worker loads a configuration file from the DB, which describes the application requests, and the SN. Subsequently, the worker contacts the DB and requests a bucket,

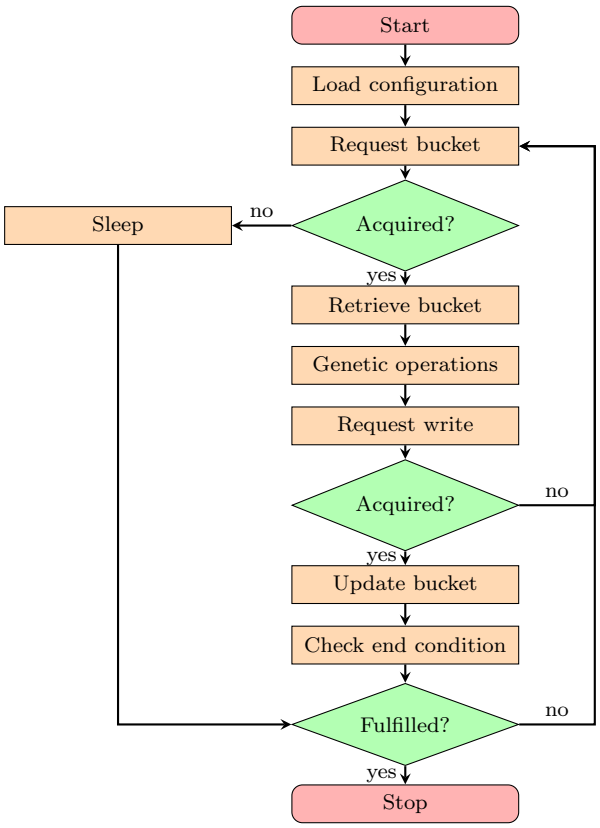


Figure 4: Workflow of a GRECO worker

which holds a random partition of the population. If a bucket cannot be acquired, because all buckets have already been assigned, then the worker sleeps, and checks the end-condition of the GA. If a bucket can be acquired, then the worker retrieves it and performs the evolutionary operators (selection, crossover, mutation, elite conservation). Next, the worker requests permission to update the bucket in the DB. When the worker cannot update the bucket, e.g. because it took too long to process it, then the worker will try to continue processing other buckets. If the worker gets write permission then it will proceed to update the bucket, and check the end-condition. If the end-condition is met (e.g. certain number of generations exceeded), then there is no more work to do, and the worker stops, else the worker proceeds to request new work.

5.2. Subgraph isomorphism

The APP algorithm presented in this section is based on subgraph isomorphism detection: it maps nodes and links during the same stage.

Lischka et al. show that this method results in better mappings and is faster than the two stage approach, especially for large virtual networks with

high resource consumption which are hard to map [32]. The advantage of this single stage approach is that link mapping constraints are taken into account at each step of the mapping. When a bad mapping decision is detected it can be revised by simply backtracking to the last valid mapping decision, whereas the two stage approach has to remap all links which is very expensive in terms of run-time. vnmFlib, the backtracking algorithm presented by Lischka et al. allows the mapping of links to paths shorter than a predefined distance value ϵ (in terms of hops).

A modified version of vnmFlib, referred to as vnmFlibm, which is represented in pseudo-code in Algorithm 2, is used in this paper.

An overview of the variables used in vnmFlibm is shown in Table 5.

variable	**definition**
G^V	graph representing application requests, comprising services and VLs that need to be placed
G^V_{sub}	subgraph of G^V, containing the services and VLs that have been added to the placement
G^P	the graph, representing the SN, comprising PMs and PLs
C	list of (service-PM) pairs
$M(\cdot)$	mapping of its argument to G^P
$\mathrm{M}(G^V_{max})$	mapping for which most applications are placed
w	total number of mappings that have been tried
W	upper-limit to the number of mappings

Table 5: An overview of the variables used in the vnmFlibm routine.

The algorithm tries to build a valid VNE solution by successively adding nodes and links of G^V to an initially empty subgraph G^V_{sub} of G^V. During the mapping process the algorithm ensures that $M(G^V_{sub})$ is a valid mapping of G^V_{sub} onto G^P. In each step of the algorithm, a list of possible (service, PM)-pairs is generated by the function *Genneigh* (line 5). The pairs are ordered by *Genneigh* as follows. First, the (service, PM)-pairs are ordered application-wise, which assures that at most one partially placed application is present in $M(G^V_{sub})$. The application order is determined subsequently by total increasing CPU requirements, memory requirements, and finally by application number. Second, the (service, PM)-pairs are ordered by service index. Third, they are ordered by duplicate number. Finally, they are ordered by the id of the PM.

The algorithm loops over this sorted list, and when a valid mapping is encountered, a new subgraph G^V_{sub} and a corresponding mapping are generated (line 9). The validity of the mapping is checked by the *valid* function. The *valid* function (line 7) checks if PM n^P has sufficient remaining nodal resources to host service n^V, and if sufficient bandwidth remains in the SN to route all VLs to and from the services already included in $M(G^V_{sub})$. The VLs are routed using a shortest path algorithm.

Additionally, the *valid* function verifies if, the availability requirement can be met. During availability calculation, the placement of an application is most of

the time incomplete. The availability of those unplaced components is assumed 100%. If the potential availability increments, brought by future placement of an additional replica were not taken into account, then the algorithm would not be able to place applications which require multiple duplicates to reach their availability goal. The (intermediate) availability of a mapping will be referred to as $f(M(G^V_{sub}))$ in the example at the end of this section.

The algorithm can terminate in one of two ways. On the one hand, if the maximum number of mappings W has been exceeded (line 3), or if all (service, PM)-pairs have been exhausted (line 22), then the algorithm returns null. In the calling procedure, the best intermediate result can be used, which was stored in $M(G^V_{max})$. $M(G^V_{max})$ is updated each time a placement is found which holds a higher number of completely placed applications (line 11). On the other hand, when G^V_{sub} fully covers G^V (line 18), the algorithm returns $M(G^V_{sub})$ (line 19), which is a valid mapping of G^V on G^P.

Algorithm 2 vnmFlibm procedure.

```
1:  procedure VNMFLIB(G^V_sub, M(G^V_sub), G^V, G^P, M(G^V_sub), M(G^V_max), w, W)
2:      if w ≥ W then
3:          return null
4:      end if
5:      C← GENNEIGH(G^V_sub, G^V, G^P)
6:      for each (n^V, n^P) ∈ C do
7:          if VALID(M(G^V_sub), (n^V, n^P), G^V) then
8:              w ← w + 1
9:              create G^V_sub and M(G^V_sub) by adding (n^V, n^P)
10:             if PLACED(M(G^V_sub)) >PLACED(M(G^V_max)) then
11:                 M(G^V_max) ← M(G^V_sub)
12:             end if
13:             M_T ← VNMFLIB(G^V_sub, M(G^V_sub), G^V, G^P)
14:             if M_T !=null then
15:                 return M_T
16:             end if
17:         end if
18:         if G^V_sub == G^V then
19:             return M(G^V_sub)
20:         end if
21:     end for
22:     return null
23: end procedure
```

The algorithm's operation is illustrated using the problem shown in Figure 3. An availability A, equal to 0.9853, is assumed for each SN component. For the sake of clarity, the bandwidth in the SN is considered not to be a critical constraint, this condition holds if each VL requires 1 unit of bandwidth, and each PL has a bandwidth capability of at least 1 unit. The required availability is 97%. It is assumed that the memory constraints limit the number of services hosted per PM to 1. In Figure 5 the different steps of the algorithm are shown. Note that only the mapped (service, PM) pairs are explicitly indicated, to ease notation. The routing is performed using a shortest path algorithm. For the availability calculation, these paths must be taken into account. Distinction

is made between the services of the two duplicates, by adding a tilde to the services corresponding to the second duplicate.

The inputs to vnmFlibm take following values: $G^V_{sub} = (\emptyset, \emptyset)$, $G_V = (V^V, E^V)$, where $V^V = \{s_1, \tilde{s}_1, s_2, \tilde{s}_2, s_3, \tilde{s}_3\}$, and $E^V = \{(s_1, s_2), (\tilde{s}_1, \tilde{s}_2), (s_2, s_3), (\tilde{s}_2, \tilde{s}_3))\}$. $G^P = (V^P, E^P)$, where $V^P = \{n_1, n_2, n_3, n_4, n_5\}$, and
$E^P = \{(n_1, n_2), (n_1, n_4), (n_2, n_3), (n_2, n_4), (n_3, n_5), (n_4, n_5)\}$
$M(G^V_{sub}) = \emptyset$, $M(G^V_{max}) = \emptyset$, $w = 1$, $W = 4 \times 6 = 24$.

Now the execution commences. In step 1, G^V_{sub} is empty and the algorithm tries to place the first service, namely s_1, on n_1, which is a valid placement. In step 2, (s_1, n_1) is added to the mapping, and the algorithm attempts to place $\tilde{s}_1$ on n_1, which is also a valid placement. In step 3, $c = (s_2, n_1)$ is not valid because n_1 does not have enough resources. Therefore the first valid option is (s_2, n_2). In step 4, $c = (\tilde{s}_2, n_1)$ is not valid because n_2 does not have enough resources. $c = (\tilde{s}_2, n_2)$ is not valid because $f(M(G^V_{sub}))$ would become $A^3 = 0.9565$, which is lower than 97%. n_3 is a valid option. In step 5, n_1 through n_3 do not have enough resources to host s_3. The availability when using either s_4, or s_5 would be too low, as $f(M(G^V_{sub}))$ would be respectively $A^5 + A^4 - A^7 = 0.9696$, and $A^6 + A^4 - A^7 = 0.9559$. Hence, there is no valid (service, PM)-pair in C, and the algorithm backtracks (rip up $(\tilde{s}_2, n_3)$). In step 6, the algorithm continues where it left off in step 4, and tries (s_2, n_4), which is a valid combination. In step 7, n_1 and n_2 do not have sufficient resources to host s_3, but n_3 does. In step 8, n_1 and n_2 cannot host $\tilde{s}_3$. n_3 is also not a valid candidate, as the resulting $f(M(G^V_{sub}))$ would be $A^5 + A^6 - A^7 = 0.9421$. n_4 is also not a valid option, because it does not have sufficient remaining resources. n_5 is a valid option. In step 9, $M(G^V_{sub})$ contains one fully placed application, therefore $M(G^V_{max})$ is overwritten. G^V equals G^V_{sub} and the algorithm terminates. The resulting placement is the one depicted in Figure 3c.

6. Performance evaluation

In this section the performance of the proposed heuristics is compared against the ILP formulation presented in Section 4, and two other placement methods found in literature. For this evaluation a custom simulation platform is developed in Java. First, this software platform simulates the selected type of application requests and SN configurations. The result is a json file per problem description, holding the input parameters shown in Table 1. Subsequently, the selected application placement algorithm is applied to these input json files and a placement configuration is generated and stored in an output json file. These output files are then further analysed to compile the graphs shown throughout this section.

This section is structured as follows. First, the evaluated algorithms are discussed. Second, the models for workload and cloud environments are presented. Then, the effect of multiple parameters is analyzed. For each input parameter, the types of simulated workload and cloud environment are described and an analysis is provided.

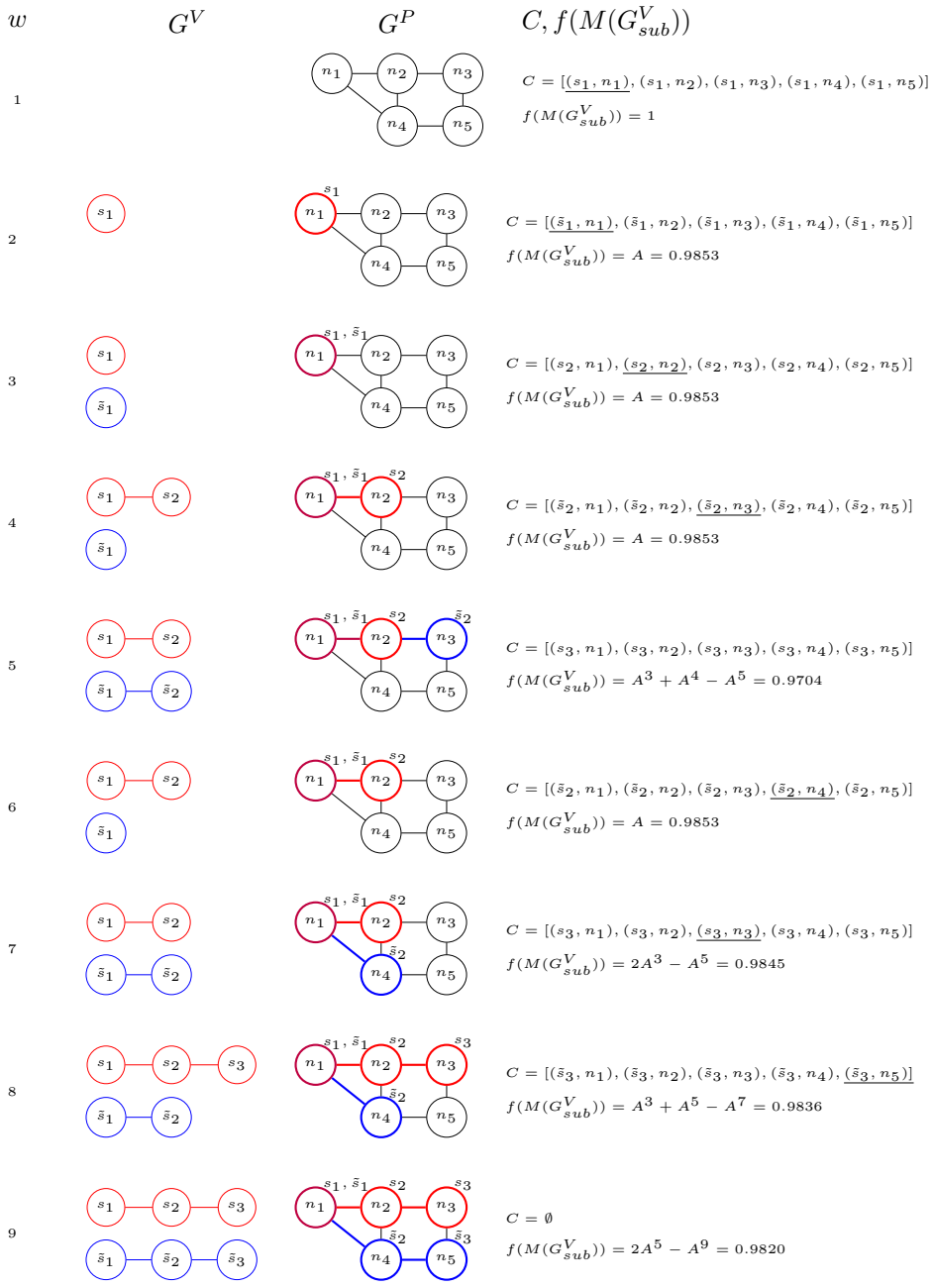


Figure 5: Steps taken by vnmFlibm when solving the problem depicted in Figure 3.

6.1. Evaluated algorithms

In this section the suffixes OPT, SUB, and GA indicate an exact algorithm based on an ILP-formulation, a heuristic based on subgraph isomorphism detection, and a genetic algorithm respectively. An overview of the compared placement methods is given in Table 6. VAR refers to the protection method

Method	**Duplicates per accepted application**	**Consolidation onto one PM**	**Availability**
VAR (this work)	at least 1, and at most δ	any two services can be consolidated	checked before each mapping
MOENS [16]	exactly 1	any two services can be consolidated	checked after execution has finished
DRONE [23]	exactly 2	only services of differerent applications can be consolidated	checked after execution has finished

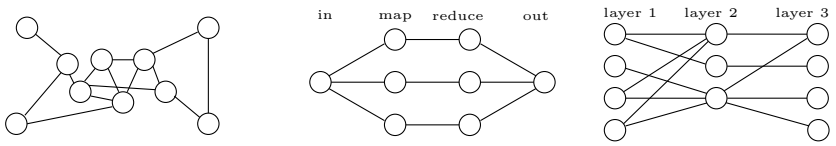
Table 6: Overview of the evaluated placement methods.

described in Section 3.3, which can vary the amount of redundancy introduced. The ILP-formulation presented in Section 4, is referred to as VAR-OPT, the GRECO algorithm (Section 5.1.1) as VAR-GA, and the heuristic based on sugraph isomorphism detection (5.2) as VAR-SUB. To the best of our knowledge, there exist no other placement algorithms, introducing a model for availability which considers both node and link failures. Therefore, a comparison is made to methods which do not have a model for availability. After these placement algorithms have finished execution, the applications whose availability requirements have not been met, are removed. Two other placement methods are considered. On the one hand Moens et al. place each application at most once [16]. MOENS-OPT is an exact algorithm based on ILP. MOENS-SUB is a self-defined heuristic based on subgraph isomorphism detection for MOENS-OPT. The only differences between MOENS-SUB and VAR-SUB, are that MOENS-SUB does not have an availability requirement, and that it places at most 1 duplicate. On the other hand, DRONE-OPT is based on the ILP-formulation from Chowdhury et al [23]. In their approach, each accepted application request must be placed exactly twice and services belonging to the same application cannot be consolitated onto one PM. The embeddings of the primary and backup VN must be fully disjoint (i.e. they cannot have any PM orPL in common), which guarantees survivability against single node or link failure. For comparison sake, their model is modified so that memory resources can be shared amongst applications which use the same service. DRONE-SUB is a self-defined algorithm based on subgraph isomorphism detection for DRONE-OPT. Compared to VAR-SUB,

this algorithm does not check availability during placement, and it always tries to place two duplicates. Additionally, DRONE-SUB allows an application to make use of each PM and PL at most once, while VAR-SUB has no such restrictions.

6.2. Application model

The SOA model described in Section 3.1 can be used to represent a wide range of applications. In the following, three application types are presented: one with a flat hierarchy, and two with a special structure, which are commonly used in software engineering. These application types are used throughout the performance evaluation.



(a) Random (b) MapReduce (c) 3 Tier

Figure 6: An illustration of multiple application models.

Random. This application type is similar to the simulation setup used by Moens et al. [16]. In this model, application requests are generated as follows. First, a certain number of services is generated, which have a certain probability to be interconnected pairwise by a VL. Then, each application randomly picks services out of those services. In this model, multiple applications can share the same service. The application model is illustrated in Figure 6a.

MapReduce. MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key [33]. The application model shown in Figure 6b comprises 1 mapper and 1 reducer stage, and assumes the input files and output files to be located on one input, and one output node respectively. It is assumed that resources cannot be reused between services of multiple applications. Each mapper communicates with the input node and exactly one reducer node. The reducer nodes communicate with the output node, and exactly one mapper node. When there are m mappers in the mapper stage, then there are also m reducers, which totals $2 + 2 \times m$ services per application.

3 Tier. A Multi-tier Architecture is a software architecture in which different software components, organized in tiers (layers), provide dedicated functionality. The most common occurrence of a multi-tier architecture is a three-tier system consisting of a data management tier (mostly encompassing one or several database servers), an application tier (business logic) and a client tier (interface

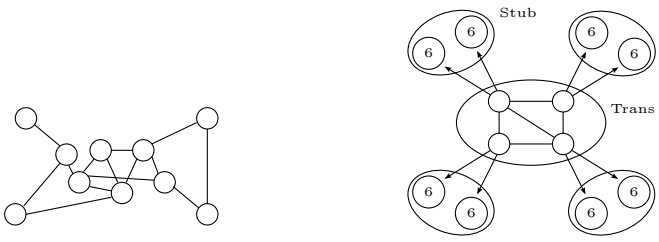
functionality). Conceptually, a multi-tier architecture results from a repeated application of the client/server paradigm. A component in one of the middle tiers is client to the next lower tier and at the same time acts as server to the next higher tier [34]. The simulations assume a VL between any pair of services in two subsequent layers, and an equal amount of services per layer.

6.3. Cloud infrastructure

In this section, two SN models are presented.

Random SN. In this model a random graph, with a predefined number of PLs and VLs, is generated. The procedure by Broder et al. is used to generate a minimal spanning tree [35], subsequently non-neighbouring PMs are interconnected at random, until the desired number of PLs has been reached. An illustration of such a topology is shown in Figure 7a.

Trans-stub model SN. Transit-stub network topologies are generated using the GT-ITM topology generator [36]. An illustration of this SN model, using 4 transit nodes, is shown in Figure 7b. Any two nodes in the transit network are connected by a PL with a probability of 80%. Within a cluster in the stub-network this probability equals 40%. Each PM in the transit network is connected to 2 clusters, each comprising 6 PMs. The total number of nodes in the substrate graph is then equal to thirteen (=transit node + 2×6 stub nodes) times the number of transit nodes. For 1, 2, 3, and 4 transit nodes, the SN comprises 13, 26, 52 and 104 PMs respectively.



(a) Random (b) Trans-stub

Figure 7: An illustration of SN types.

6.4. Algorithmic parameters

The VAR-GA stops after it has computed 100 generations or it has the same best solution for 20 generations. The algorithms based on subgraph isomorphism detection attempt at most 4 placements per virtual node in G^V, as proposed by Lischka et al. The ILP models are solved by Gurobi 6.5.1. All tests were executed on the High Performance Cluster (HPC) core facility CalcUA at the University of Antwerp. Each test used one machine with 20 cores. For the GA the MongoDB database was allocated at a server at the university and not at the HPC. The latency between HPC and the MongoDB server is about 1.740 milliseconds.

6.5. Key observations

Before analysing the performance of the algorithms, two metrics must be introduced. First, the placement ratio is the ratio of applications that meet the availability requirement to the total of application requests. Second, the execution time is the time it takes to execute the placement algorithm. While also an important perforamance metric, the application response time, has not been simulated. One could incorporate response time requirements by appropriately dimensioning the CPU and bandwidth requirements according to the expected user workload. Additionally, VL latency requirements could easy be added as an additional constraint to the model. Third, the CPU Load Factor (CLF) expresses the loading on the cloud environment, i.e. the ratio of total CPU demand of all application requests, to the total amount of available CPU resources:

$$\text{CLF} = \frac{\sum_{s \in \mathbf{S}} \sum_{a \in \mathbf{A}} I_{a,s} \times \omega_s}{\sum_{n \in \mathbf{N}} \Omega_n}.$$

In the following, a wide range of parameters is swept. Unless explicitly stated otherwise, for each parameter setting, the CLF is varied from 0.1 to 1, in increments of 0.1. For each CLF level 100 input files are generated, each containing a certain workload and SN. The simulation results are shown in Figures 8 - 11, where markers indicate averages and errorbars represent the standard error of the mean.

6.5.1. Influence of required availability

Generated workload. In this experiment, the workload consists of 10 *random* applications. The services of each application are chosen (at random) out of a set of three services with a probability of 60%. For each service $s : \omega_s$ is uniformly distributed in the interval $[0.2; 1]$, and γ_s is uniformly distributed in the interval $[0.75; 1]$. Any two services of the same application are interconnected by a VL, requiring a bandwidth which is uniformly distributed in the interval $[0.02; 0.04]$.

Used substrate network. A random SN is generated, consisting of 5 PMs and 8 PLs. Each PL has a bandwidth of 1. For each PM $n : \Omega_n \in \{0.5, 2, 10, 50\}$, and $\Gamma_n \in \{1, 1.5, 2\}$. For each PL and PM the failure rate is a uniformly distributed random choice out of the $\{0\%, 2.5\%, 5\%\}$.

Results. It can be seen in Figure 8 that for a required availability level of 0% the placement ratio of MOENS-OPT and VAR-GA are very close to optimal (VAR-OPT). The placement ratios for VAR-SUB and MOENS-SUB are about 6% lower, which can be attributed to differences in routing. While the GA and the ILP formulation can select the best path (for bandwidth or availability) between any two PMs, the algorithms based on subgraph isomorphism detection always use shortest path routing. The placement ratios for DRONE-OPT and DRONE-SUB, are very low because these algorithms must always find two completely

disjoint mappings while there are only 5 PMs in total. As the required availability level increases from 0% to 90%, the placement ratios for the algorithms which are availability-aware (VAR-OPT, VAR-SUB, VAR-GA) remain relatively constant. Additionally, the placement ratios for DRONE-OPT and DRONE-SUB remain constant, as the backup VN embedding protects against single node and link failures. In contrast to that, the placement ratio for MOENS-SUB decreases by 4%, and the placement ratio for MOENS-OPT even decreases by 33%. The differences between MOENS-SUB and MOENS-OPT, can be attributed to the fact that MOENS-SUB generally uses less PLs and PMs per application, yielding a higher availability. When the required availability level increases further from 90% to 99% the placement ratios drop significantly for most algorithms. Only for DRONE-OPT and DRONE-SUB the drop is less than 1%, as almost all of the placed applications fulfil the availability requirement. For an availability requirement of 99% there is a huge benefit associated to considering 2 duplicates instead of 1 (for VAR-SUB an increase of 81% in placement ratio), at the cost of an increase of 128% in computation time. Given the drastically improved placement ratio, 2 duplicates will be used in the following experiments, unless explicitely stated otherwise.

While the computation time of DRONE-OPT and MOENS-OPT is below 100ms, for VAR-OPT it increases dramatically when the required availability goes from 0% to 90%. Most of the configurations with a required availability level of 99% did not finish and were therefore excluded from the graph. It can be seen that, while the GA provides a dramatic speedup compared to VAR-OPT for non-zero required availability levels (741x at 90% required availability), the subgraph algorithm is up to 169x faster than the GA. However, the placement ratio of VAR-SUB is up to 14% lower than for VAR-GA. Because finding the exact solution takes too much time for moderate scale problems, VAR-OPT will be excluded from the remaining experiments.

6.5.2. Influence of CLF

Generated workload. In this setup, 10 *random* applications are generated, comprising 24 services in total. Each application is assigned 12 out of those 24 services at random. For each service $s : \omega_s$ is uniformly distributed in the interval $[0; \omega_{max}]$, and γ_s is uniformly distributed in the interval $[0; \gamma_{max}]$, where

$$\omega_{max} = \frac{CLF_{target}.|\mathbf{N}|.2.\overline{\Omega}}{\sum_{a \in \mathbf{A}} \sum_{s \in \mathbf{S}} I_{a,s}} \tag{8}$$

and

$$\gamma_{max} = \Gamma_{max} \times \frac{\omega_{max}}{\Omega_{max}}. \tag{9}$$

Equation 8 ensures that the expected CLF equals CLF_{target}. Equation 9 scales the memory requirements proportionally with the CLF.

Used substrate network. A trans-stub network comprising 13 PMs, is generated. The PM capabilities are a uniformly distributed random choice of t2, m3, and m4, Amazon EC2 instance specifications [37]. The PL bandwidth is uniformly

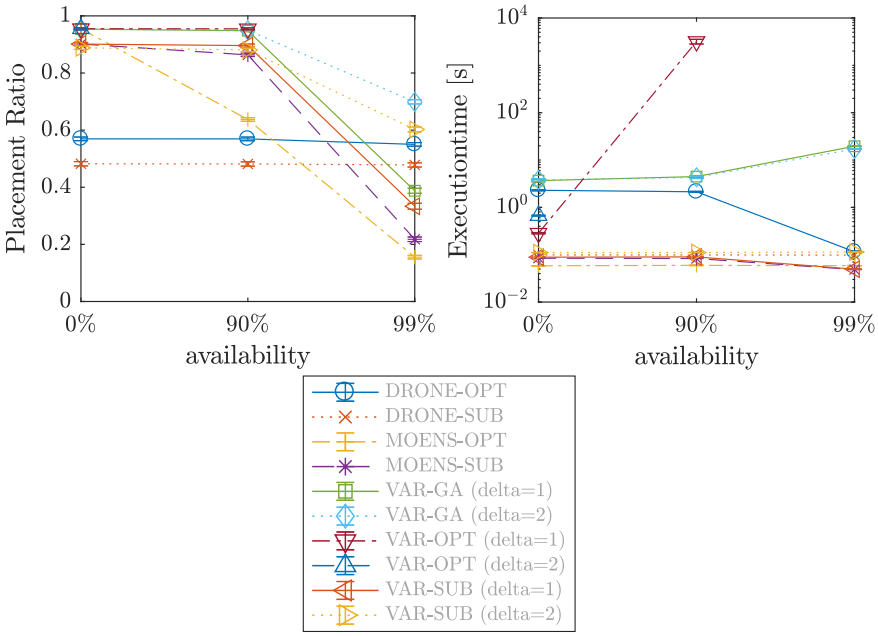


Figure 8: Influence of the required availability level.

distributed in $[0; 100]$. Both PM and PL failure rates are uniformly distributed in $[0\%; 1\%]$.

Results. The results in Figure 9 show that the placement ratio decreases, for all algorithms, as the CLF increases from 0.1 to 1.0. Again, the placement ratio of MOENS-OPT and MOENS-SUB is very low (up to 89% and 71% lower than VAR-SUB) as these algorithms do not introduce any redundancy. For low CLF values the performance of DRONE-OPT and DRONE-SUB is comparable to that of VAR-SUB (less than 1% difference) . However, when the CLF increases, then insufficient resources remain to place each VN twice and the placement ratios of DRONE-OPT and DRONE-SUB are up to 27% and 41% lower than for VAR-SUB. While VAR-SUB performs about 1% better for CLFs up to 0.6, the GA performs 28% better for a CLF of 1.0. This difference can be explained by how the algorithms handle the application requests. While the subgraph isomorphism algorithm tries to place the applications in a predetermined order, and backtracks as soon as an application cannot be placed, the GA tries different orderings of the applications, and when an application cannot be placed, it proceeds to the next one. A similar reasoning can be applied to the differences between DRONE-OPT and DRONE-SUB. For CLF values up to 0.3 DRONE-SUB achieves a higher placement ratio than DRONE-OPT, as typically its embeddings are more compact, realizing a higher availability and dito placement ratio. However, for CLF values higher than 0.3, DRONE-OPT realizes up to 23% higher placement ratios, as the ILP solver can try different

orderings of applications, while DRONE-SUB backtracks as soon as an application cannot be placed.

While the influence of CLF on execution time is not so clear, it is clear that VAR-SUB is much faster than VAR-GA: a speed-up of 400x, up to even 900x is observed. The GA is much slower in our experiments, because of the large overhead in communicating with the database. A detailed breakdown of the computation time of the GRECO algorithm shows that up to 93% of the time is spent communicating with the remote DB, even though the workers for one optimization are all running on one and the same PM [28]. Because of administrative limitations it was not possible to host the DB within the HPC network. Given the dramatic differences in execution time, compared to the other algorithms, only the heuristics based on subgraph isomorphism detection (VAR-SUB, MOENS-SUB, and DRONE-SUB) are included in the remaining experiments.

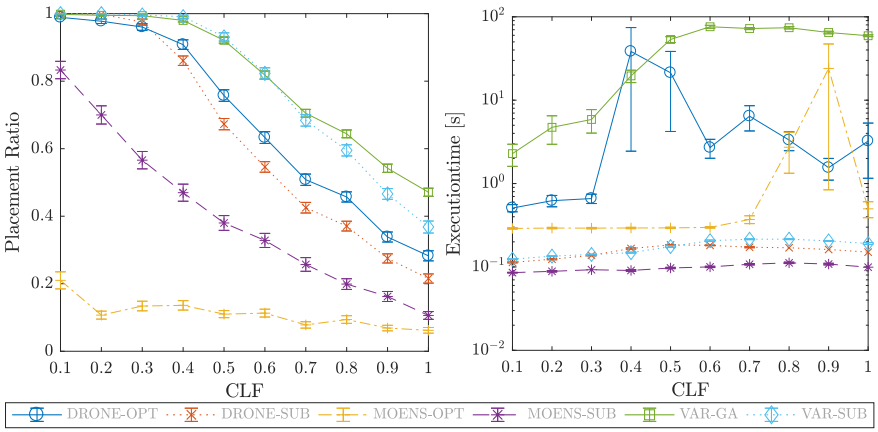


Figure 9: Influence of the CLF, for a required availability of 99.0%.

6.5.3. Influence of application requests and SN dimensions

Generated workload. For this setup, separate test runs are generated, each considering only *random*, *MapReduce*, or *3 Tier* applications requests. The number of services per application is 12, resulting in 5 mappers and 5 reducers, and 4 services per layer, in the MapReduce and 3 Tier model respectively. In Section 6.5.2 the number of applications was fixed to 10, and the CLF was increased by generating increasingly CPU heavy applications. However, in this setup the CLF is varied by changing the number of application requests, while keeping the service requirements constant. For a CLF_{target} of 0.1, 0.2, and 0.3, the simulation platform generates 10, 20, and 30 applications respectively. Again for each service the CPU and memory requirements are uniformly distributed, with maximum values dictated by Equation 8, and 9. Also, VL bandwidth requirements are uniformly distributed in $[0; 1]$.

Used substrate network. A trans-sub model is used with the same specifications as in Section 6.5.2, only now the SN comprises 1, 2, 3, or 4 transit nodes.

Results. Figure 10 shows that the placement ratio decreases as the SN size increases. Again, the placement ratio for MOENS-SUB is the lowest of all three algorithms. The placement ratio for 13 and 26 nodes is roughly the same for DRONE-SUB and VAR-SUB. However, when the SN grows further, then the performance for VAR-SUB is up to 2x better than for DRONE-SUB. An explanation is that as the number of nodes increases, the CPU resources get more fragmented (constant CLF), which makes it harder to always place 2 disjoint duplicates. Additionally, an increased SN size brings a higher number of reliable nodes. Hence, there is a higher probability that a combination of high available PLs and PMs can be made, avoiding the need for a second duplicate. The execution time for DRONE-SUB is clearly smaller than for VAR-SUB, as the number of possible mappings is smaller (no consolidation possible within application) and as the availability conditions are only checked once (versus before each mapping step). Figure 11 shows that the performance of the algorithm

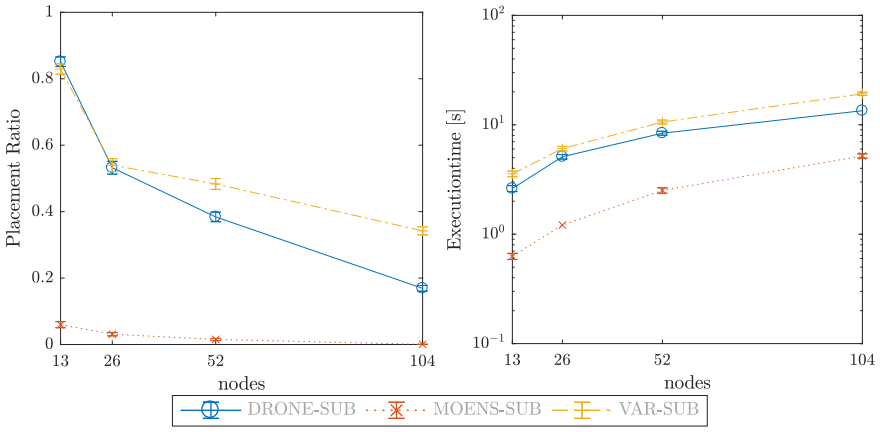


Figure 10: Influence of the SN dimensions for a required availability of 99.9% and application type "random".

is strongly dependent on the type of application considered. While the three application types require the same number of services, and have the same service requirements, the performance for the random and structured applications (MapReduce and 3Tier) differs significantly. The random application type takes significantly longer to place. Additionally, structured applications can benefit more from using more than one duplicate. For the random application type, the placement ratio even decreases slightly when the number of duplicates increases from 2 to 3. Intuitively these observations can be explained by the fact that the services of structured applications are easier to consolidate. This effect becomes more pronounced when redundancy levels go up, causing increased competition for CPU resources.

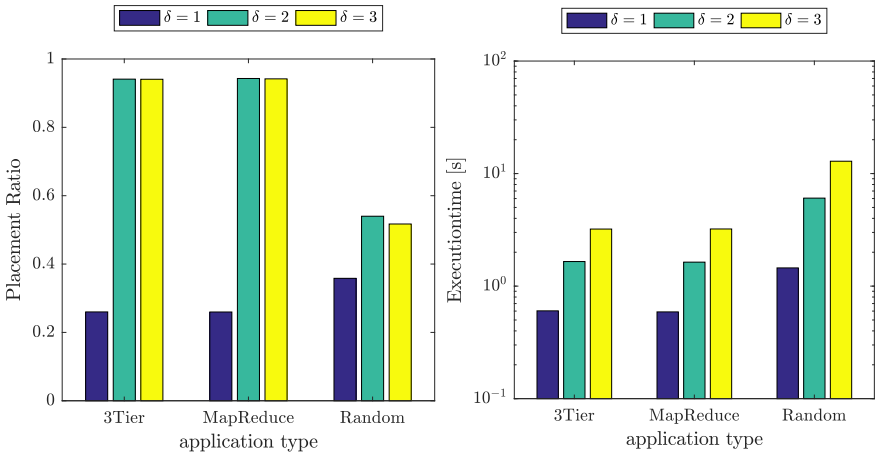


Figure 11: Influence of application type for 26 nodes, for a required availability of 99.9%, using VAR-SUB.

Figure 12 shows that the placement ratio decreases and the execution time increases, when the number of application requests goes up. Increasing the

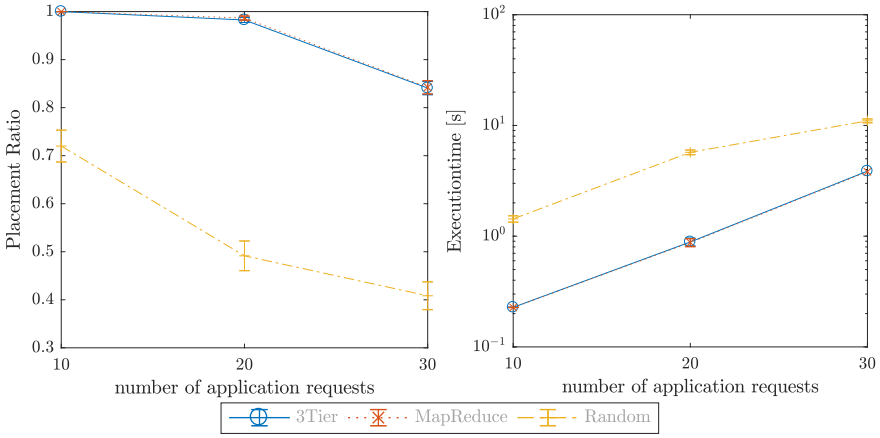


Figure 12: Influence of the number of application requests for 26 PMs, a required availability of 99.9%, using VAR-SUB.

number of applications requests causes the memory usage of the subgraph algorithm to increase, as the recursion depth goes up. Therefore it is good practice to limit the number of mappings to be considered.

Figure 13 shows that the placement ratio increases for the structured applications (MapReduce, 3Tier) as the number of nodes increases, while the placement ratio decreases for the random application. This can be explained as follows. On the one hand, when the number of nodes increases, the nodal capabilities get more fragmented, as the CLF is kept constant. On the other hand, as the

SN dimensions increase it become easier to meet the availability requirements, as long as the nodal capabilities are not too fragmented to fit the services. It is clear that the computation time increases when the number of nodes increases. Additionally the computation time for the random application is higher than for the structured applications.

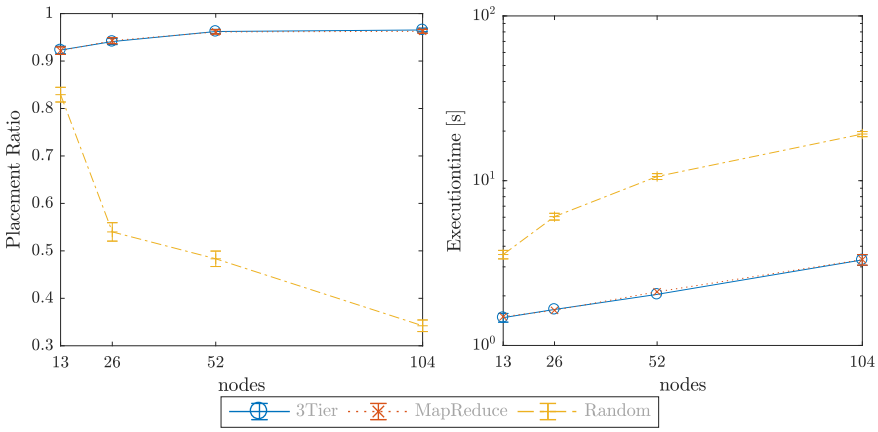


Figure 13: Influence of the SN dimensions for for a required availability of 99.9%, using VAR-SUB.

7. Results discussion

In the previous section, it was shown that in heterogeneous cloud networks, VAR can bring a dramatic increase in placement placement ratio, relative to the state-of-the-art, currently lacking an availability model. While exact solution of the ILP formulation scales badly [27], the GA [28] can place 10 applications on a SN comprising 13 nodes within 100 seconds. The newly proposed heuristic based on subgraph isomorphism detection can even place up to 30 applications on a SN comprising 104 nodes, all within 20 seconds. When the placement algorithm can be offloaded to a (remote) reliable server, then VAR-SUB can be used. If this offloading is not possible, requiring execution of the placement algorithm within the heterogeneous cloud network, then either the fault-tolerant distributed GA must be used, or alternatively the protection methods laid out throughout this work can be applied to VAR-SUB, to make it survive failure of the management nodes.

In reliable cloud environments (or equivalently, under low availability requirements) it is often acceptable to place each VN only once, and not bother about availability [16]. However, when the frequency of failures is higher (or if availability requirements increase), then one of the following measures should be taken. First, one can improve the availability by placing additional backups, which fail independently of one another. However, this approach works best in homogeneous cloud environments, where one can use the same number of

backup VN embeddings, regardless of the exact placement configuration. In heterogeneous environments a fixed redundancy level for each application either results in wasted SN resources, or a reduced placement ratio. The same reasoning can be made for applications, having heterogeneous availability requirements. Second, one can already achieve an increased placement ratio by considering the realized availability during placement (VAR-SUB, $\delta = 1$). In this case, the placement algorithm can place applications with higher availability requirements on more reliable parts of the infrastructure, and at the same time avoid wasting resources on applications whose availability requirements cannot be fulfilled. Third, one can consider the realized availability during placement and decide on the appropriate redundancy level. This decision depends on both the precise availability requirements and the actual placement configuration.

For example in an Amazon cloud environment, where heterogeneity is limited to multiple generations of servers being used, and resources are virtually infinite, it makes sense to disperse a predefined number of copies across multiple availability zones. Now, consider again the distributed MPC example shown in Figure 1, where computation tasks must be deployed close to the edge of the network. In a such a heterogeneous, unreliable computational environment, one cannot afford to waste precious resources. While previously intractable, now VAR-SUB can find an intelligent placement configuration, tailored to the specific availability requirements of each application, in a matter of seconds.

When focussing on the placement ratio obtained by VAR-SUB, following observations can be made. First, when the required availability level increases, the placement ratio goes down and the computation time increases. Additionally, more stringent availability requirements require a higher number of duplicates to be used. Second, increased loading of the SN, either caused by resource-heavier applications, or an increased number of application requests, decreases the fraction of application requests that can be accepted. While the resource requirements of individual applications has little effect on the execution time, increasing the number of application requests significantly affects computation time. Third, the fraction of application requests that can be accepted largely depends on the type of application requests. Applications that can be consolidated more easily achieve better placement ratios.

8. Conclusion

Cloud environments are becoming increasingly decentralized, leading to a heterogeneous network of micro-clouds which are positioned on the edge of the network and possibly interconnected by best-effort links. This heterogeneous environment introduces important challenges for the management of these clouds as the heterogeneity results in an increased failure probability. In this paper, we study the problem of simultaneous placement of a set of mission-critical applications on such an unreliable network, while guaranteeing a certain level of availability for each application. Three algorithms are presented that make use of intelligent application level replication: an optimal algorithm using an ILP solver (VAR-OPT), and two heuristics. The first heuristic (VAR-GA) is a

fault-tolerant distributed GA which uses a distributed pool-model to distribute the population. The second (VAR-SUB) is a fast centralized algorithm, based on subgraph isomorphism detection. VAR-GA performs near optimal for small problem instances, and outperforms the subgraph algorithm up to 28% when its placement ratio drops below 0.7. However, when the problem size grows, the VAR-SUB algorithm scales better and becomes the algorithm of choice. While previous solutions were computationally too complex to allow a timely calculation in real-life large-scale environments, the newly presented algorithm based on subgraph isomorphism detection (VAR-SUB), effectively removes this barrier. A detailed performance evaluation shows that, in comparison to algorithms that protect against single node or link failure, VAR-SUB can doubles the placement ratio in cloud environments comprising over 100 nodes, while keeping the time required to calculate the solution under 20 seconds.

Acknowledgments

This work was supported by VLAIO and iMinds [iFEST and EMD project]; and the University of Antioquia [CODI sustainability strategy 2014-2015].

References

[1] A. O. Marzio Puleri Roberto Sabella, Cloud robotics: 5G paves the way for mass-market autmation, in: Charting the future of innovation, 5th Edition, Vol. 93, Ericsson, The address of the publisher, 2016.

[2] F. Bonomi, R. Milito, J. Zhu, S. 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, ACM, 2012, pp. 13–16. doi:10.1145/2342509.2342513.
URL http://doi.acm.org/10.1145/2342509.2342513$\delimiter"026E30F$npapers2://publication/doi/10.1145/2342509.2342513

[3] J. Al-Muhtadi, R. Campbell, A. Kapadia, M. D. Mickunas, S. Yi, Routing through the mist: privacy preserving communication in ubiquitous computing environments, in: Proceedings 22nd International Conference on Distributed Computing Systems, 2002, pp. 74–83. doi:10.1109/ICDCS.2002.1022244.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1022244

[4] G. Hu, W. P. Tay, Y. Wen, Cloud Robotics: Architecture, Challenges and Applications, IEEE Network 26 (June) (2012) 21–28. doi:10.1109/MNET.2012.6201212.

[5] ISO/IEC-25010, Systems and software engineering Systems and software Quality Requirements and Evaluation (SQuaRE) System and software quality models., Standard, International Organization for Standardization, Geneva, CH (mar 2010).

[6] B. Spinnewyn, S. Latré, Towards a fluid Cloud: An extension of the Cloud into the local Network, in: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 9122, Ghent, Belgium, 2015, pp. 61–65. doi:10.1007/978-3-319-20034-7_7.

[7] M. Saska, Z. Kasl, L. Peucil, Motion planning and control of formations of micro aerial vehicles, IFAC Proceedings Volumes 47 (3) (2014) 1228–1233. doi:10.3182/20140824-6-ZA-1003.02295.

[8] E. Camponogara, D. Jia, B. Krogh, S. Talukdar, Distributed model predictive control, in: IEEE Control Systems Magazine, Vol. 22, IEEE, 2002, pp. 44–52. doi:10.1109/37.980246.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=980246

[9] B. Jennings, R. Stadler, Resource management in clouds: Survey and research challenges, Journal of Network and Systems Management 23 (3) (2014) 567–619.
URL http://link.springer.com/article/10.1007/s10922-014-9307-7

[10] M. R. Rahman, R. Boutaba, SVNE: Survivable virtual network embedding algorithms for network virtualization, IEEE Transactions on Network and Service Management 10 (2) (2013) 105–118. doi:10.1109/TNSM.2013.013013.110202.

[11] R. Camati, A. Calsavara, L. L. Jr, Solving the Virtual Machine Placement Problem as a Multiple Multidimensional Knapsack Problem, in: ICN 2014, The Thirteenth ..., no. c, 2014, pp. 253–260.
URL http://www.thinkmind.org/index.php?view=article{&}articleid=icn{_}2014{_}11{_}10{_}30065

[12] J. Xu, J. a. B. Fortes, Multi-objective Virtual Machine Placementin Virtualized Data Center Environments, in: 2010 IEEE/ACM Int'l Conference on \& Int'l Conference on Cyber, Physical and Social Computing (CPSCom), GREENCOM-CPSCOM '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 179—-188. doi:10.1109/GreenCom-CPSCom.2010.137.

[13] Y. Ren, J. Suzuki, A. Vasilakos, S. Omura, K. Oba, Cielo: An evolutionary game theoretic framework for virtual machine placement in clouds, in: Proceedings - 2014 International Conference on Future Internet of Things and Cloud, FiCloud 2014, 2014, pp. 1–8. doi:10.1109/FiCloud.2014.11.

[14] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, F. De Turck, Cost-Effective feature placement of customizable multi-tenant applications in the cloud, Journal of Network and Systems Management 22 (4) (2014) 517–558. doi:10.1007/s10922-013-9265-5.

[15] A. Fischer, J. F. Botero, M. T. Beck, H. De Meer, X. Hesselbach, Virtual network embedding: A survey, IEEE Communications Surveys and Tutorials 15 (4) (2013) 1888–1906. doi:10.1109/SURV.2013.013013.00155.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6463372

[16] H. Moens, B. Hanssens, B. Dhoedt, F. De Turck, Hierarchical network-aware placement of service oriented applications in clouds, in: IEEE/IFIP NOMS 2014 - IEEE/IFIP Network Operations and Management Symposium: Management in a Software Defined World, 2014, pp. 1–8. doi:10.1109/NOMS.2014.6838230.

[17] X. Cheng, S. Su, Z. Zhang, H. Wang, F. Yang, Y. Luo, J. Wang, Virtual network embedding through topology-aware node ranking, ACM SIGCOMM Computer Communication Review 41 (2) (2011) 38. doi:10.1145/1971162.1971168.

[18] Y. Zhu, M. Ammar, Algorithms for assigning substrate network resources to virtual network components, in: Proceedings - IEEE INFOCOM, 2006, pp. 1–12. doi:10.1109/INFOCOM.2006.322.

[19] M. Ajtai, N. Alon, J. Bruck, R. Cypher, C. Ho, M. Naor, E. Szemeredi, Fault tolerant graphs, perfect hash functions and disjoint paths, in: Proceedings., 33rd Annual Symposium on Foundations of Computer Science, 1992, pp. 693–702. doi:10.1109/SFCS.1992.267781.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=267781

[20] M. Mihailescu, S. Sharify, C. Amza, Optimized application placement for network congestion and failure resiliency in clouds, in: 2015 IEEE 4th International Conference on Cloud Networking, CloudNet 2015, 2015, pp. 7–13. doi:10.1109/CloudNet.2015.7335272.

[21] M. J. Csorba, H. Meling, P. E. Heegaard, Ant system for service deployment in private and public clouds, in: Proceeding of the 2nd workshop on Bio-inspired algorithms for distributed systems - BADS '10, ACM, 2010, p. 19. doi:10.1145/1809018.1809024.
URL http://portal.acm.org/citation.cfm?doid=1809018.1809024

[22] M. M. A. Khan, N. Shahriar, R. Ahmed, R. Boutaba, SiMPLE: Survivability in multi-path link embedding, in: Proceedings of the 11th International Conference on Network and Service Management, CNSM 2015, 2015, pp. 210–218. doi:10.1109/CNSM.2015.7367361.

[23] S. Chowdhury, R. Ahmed, M. M. ALAM KHAN, N. Shahriar, R. Boutaba, J. Mitra, F. Zeng, Dedicated Protection for Survivable Virtual Network Embedding, in: IEEE Transactions on Network and Service Management, 2016, pp. 1–1. doi:10.1109/TNSM.2016.2574239.
URL http://ieeexplore.ieee.org/document/7480798/

[24] D. Jayasinghe, C. Pu, T. Eilam, M. Steinder, I. Whalley, E. Snible, Improving performance and availability of services hosted on IaaS clouds with structural constraint-aware virtual machine placement, in: Proceedings - 2011 IEEE International Conference on Services Computing, SCC 2011, IEEE, 2011, pp. 72–79. doi:10.1109/SCC.2011.28.

[25] W. Wang, H. Chen, X. Chen, An availability-aware virtual machine placement approach for dynamic scaling of cloud applications, in: Proceedings - IEEE 9th International Conference on Ubiquitous Intelligence and Computing and IEEE 9th International Conference on Autonomic and Trusted Computing, UIC-ATC 2012, 2012, pp. 509–516. doi:10.1109/UIC-ATC.2012.31.

[26] W.-L. Yeow, C. Westphal, U. Kozat, Designing and embedding reliable virtual infrastructures, Proceedings of the second ACM SIGCOMM workshop on Virtualized infrastructure systems and architectures - VISA '10 41 (2) (2010) 33. arXiv:1005.5367, doi:10.1145/1851399.1851406.
URL http://portal.acm.org/citation.cfm?doid=1851399.1851406

[27] B. Spinnewyn, B. Braem, S. Latre, Fault-tolerant application placement in heterogeneous cloud environments, in: Proceedings of the 11th International Conference on Network and Service Management, CNSM 2015, IEEE, 2015, pp. 192–200. doi:10.1109/CNSM.2015.7367359.

[28] R. Mennes, B. Spinnewyn, S. Latré, J. F. Botero, {GRECO:} A Distributed Genetic Algorithm for Reliable Application Placement in Hybrid Clouds, in: 2016 IEEE 5th International Conference on Cloud Networking (CloudNet) (CLOUDNET'16), IEEE, 2016.

[29] K. B. Laskey, K. Laskey, Service oriented architecture, Wiley Interdisciplinary Reviews: Computational Statistics 1 (1) (2009) 101–105. doi:10.1002/wics.8.
URL http://dx.doi.org/10.1002/wics.8

[30] P. D. Justesen, Multi-objective Optimization using Evolutionary Algorithms, Vol. Confirmati, John Wiley & Sons, 2009.

[31] J. F. Gonçalves, M. G. C. Resende, Biased random-key genetic algorithms for combinatorial optimization, Journal of Heuristics 17 (5) (2011) 487–525. doi:10.1007/s10732-010-9143-1.
URL http://link.springer.com/10.1007/s10732-010-9143-1

[32] J. Lischka, H. Karl, A virtual network mapping algorithm based on subgraph isomorphism detection, in: Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures VISA 09, ACM, 2009, p. 81. doi:10.1145/1592648.1592662.
URL http://portal.acm.org/citation.cfm?doid=1592648.1592662

[33] J. Dean, S. Ghemawat, Simplified data processing on large clusters, Sixth Symp. Oper. Syst. Des. Implement. 51 (1) (2004) 107–113. arXiv:10.1.1.163.5292, doi:10.1145/1327452.1327492.

[34] C. Caragea, V. Honavar, P. Boncz, P. Boncz, P.-Å. Larson, S. W. Dietrich, G. Navarro, B. Thuraisingham, Y. Luo, O. Wolfson, S. M. Beitzel, E. C. Jensen, O. Frieder, C. S. Jensen, N. Tradišauskas, E. V. Munson, A. Wun, K. Goda, S. E. Fienberg, J. Jin, G. Liu, N. Craswell, T. B. Pedersen, C. Pautasso, M. M. Moro, S. Manegold, S. Manegold, B. Carminati, M. Blanton, S. Bouchenak, N. de Palma, W. Tang, C. Quix, W. Tang, M. A. Jeusfeld, R. K. Pon, D. J. Buttler, M. A. Jeusfeld, W. Meng, P. Zezula, M. Batko, V. Dohnal, J. Domingo-Ferrer, J. Domingo-Ferrer, D. Barbosa, I. Manolescu, J. Xu Yu, J. Domingo-Ferrer, E. Cecchet, V. Quéma, X. Yan, O. Wolfson, G. Santucci, D. Zeinalipour-Yazti, P. K. Chrysanthis, C. Quix, A. Deshpande, C. Guestrin, S. Madden, C. K.-S. Leung, R. H. Güting, R. H. Güting, A. Gupta, T. B. Pedersen, H. Tao Shen, G. Weikum, B. Thuraisingham, G. Weikum, R. Jain, J. X. Yu, P. Ciaccia, K. S. Candan, M. L. Sapino, J. X. Yu, R. Jain, C. Meghini, F. Sebastiani, U. Straccia, F. Nack, V. S. Subrahmanian, M. V. Martinez, D. Reforgiato, J. X. Yu, T. Westerveld, M. Sebillo, G. Vitiello, M. De Marsico, K. Voruganti, C. Parent, S. Spaccapietra, C. Vangenot, E. Zimányi, P. Roy, S. Sudarshan, E. Puppo, P. Kröger, M. Renz, H. Schuldt, S. Kolahi, A. Unwin, W. Cellary, Multi-Tier Architecture, Springer US, Boston, MA, 2009, pp. 1862–1865. doi:10.1007/978-0-387-39940-9_652.
URL http://www.springerlink.com/index/10.1007/978-0-387-39940-9{_}652

[35] A. Broder, Generating random spanning trees, in: 30th Annual Symposium on Foundations of Computer Science, 1989, pp. 442–447. doi:10.1109/SFCS.1989.63516.
URL http://www.computer.org/portal/web/csdl/doi/10.1109/SFCS.1989.63516

[36] E. Zegura, K. Calvert, S. Bhattacharjee, How to model an internetwork, in: Proceedings of IEEE INFOCOM '96. Conference on Computer Communications, Vol. 2 of INFOCOM'96, IEEE Computer Society, Washington, DC, USA, 1996, pp. 594–602. doi:10.1109/INFCOM.1996.493353.
URL http://dl.acm.org/citation.cfm?id=1895868.1895900

[37] Amazon EC2 Instance Comparison, \url{https://aws.amazon.com/ec2/instance-types}.
URL http://www.ec2instances.info

Appendix

Acronyms

APP Application Placement Problem.

CAPP Cloud Application Placement Problem.

CLF CPU Load Factor.

CPU Central Processing Unit.

DB database.

GA Genetic Algorithm.

ILP Integer Linear Program.

M2C machine-to-cloud.

M2M machine-to-machine.

MAV Micro Air Vehicle.

MCF Multi Commodity Flow.

MKP Multiple Knapsack Problem.

MPC Model Predictive Control.

NIC Network Interface Card.

PL Physical Link.

PM Physical Machine.

SN Substrate Network.

SOA Service Oriented Architecture.

SVNE Survivable Virtual Network Embedding.

VL Virtual Link.

VM Virtual Machine.

VMs Virtual Machines.

VN Virtual Network.

VNE Virtual Network Embedding.