On Dynamic placement of resources in Cloud Computing-Technical Report

Yuval Rochman
Tel-Aviv University
Tel Aviv, Israel
yuvalroc@post.tau.ac.il

Hanoch Levy
Tel-Aviv University
Tel Aviv, Israel
hanoch@cs.tau.ac.il

Eli Brosh
Vidyo
New Jersey, USA
eli@vidyo.com

ABSTRACT

We address the problem of dynamic resource placement in general networking and cloud computing applications. We consider a large-scale system faced by time varying and regionally distributed demands for various resources. The system operator aims at placing the resources across regions to minimize costs (maximize revenues), and to address the problem of how to dynamically reposition the resources in reaction to the time varying demand. Large Software-as-a-Service (SaaS) providers that rent server resources (across multiple datacenters) from a cloud provider, face the problem of where to place various server resources and naturally fall under this paradigm.

The main challenge posed by this setting is the need to deal with arbitrary multi-dimensional (of high-dimensionality) stochastic demands which *vary over time*. Under such settings one should provide a tradeoff between optimizing the resource placement as to meet its demand, and minimizing the number of added and removed resources to the placement.

Our analysis and simulations reveal that optimizing the resource placement may inflict huge resource repositioning costs, even if the demand has small fluctuations. We therefore propose an algorithmic framework that overcomes this difficulty and yields very efficient dynamic placements with bounded repositioning costs.

Our solutions are based on new theoretical techniques using graph theory methodologies that can be applied to other optimization/combinatorial problems. Our solution is developed under a very wide cost model that allows accommodation of many systems.

1. INTRODUCTION

Cloud computing has emerged as an attractive solution for the delivery of computing resources over the Internet. Existing cloud computing platforms like Amazon EC2 [2] and Microsoft Azure [12] organize a shared pool of servers in geographically distributed datacenters to enable on-demand rental of compute resources at scale. Many Software-as-a-Service (SaaS) solutions, such as online web services, streaming applications, and social networks, often feature demands that are highly dynamic and geo-distributed, and leverage cloud computing to serve users more efficiently and reliably.

A common approach used by large-scale SaaS providers is to place (server) resources at various geographical areas to be close to their users and guarantee adequate levels of service quality, e.g., low response times. It is typically preferred to serve a demand by a resource located in the same area rather than by a remotely located resource. At the same time, placing and maintaing a resource incurs a cost such as server rental cost. For example, Amazon EC2 offers several server instance types[1] natively (and few thousands more through a marketplace). Amazon bills instance usage on a pay-per-use basis, e.g., according to an on-demand price plan that varies with instance type and datacenter location.

Engineering such a service in a cost-effective manner is challenging due to the need to deal with regionally distributed demands for various resources, namely, finding the right (e.g., a low cost) placement of resources at the regions while providing adequate service quality. The problem is further complicated by the fact that the demand is time varying and can incur large spikes [7]. A simple solution adopted by many providers is to dynamically scale the number of resources to match the average demand. However, the uncertainty of variable demands can lead to insufficient resources, degrading service quality. To mitigate this scenario, operators often maintain an extra pool of active server resources at the expense of an additional cost overhead.

In this paper, we address the challenging problem of how to place resources over geographically distributed areas so as to minimize the total operation cost while providing adequate level of performance; and how to dynamically reposition the resources in reaction to demand fluctuations. To this end, we provide a framework and algorithms wherein demand is captured as a multi dimensional distribution which *varies over time*, consisting of the demand distribution for each resource and in each area.

We develop a general model that captures the major cost parameters that affect the design of dynamic resource placement in SaaS applications, including: (a) the cost of placing a resource (e.g, renting an on-demand server) and that of repositioning (moving) it, and (b) the benefit of satisfying the demand, e.g., the revenue associated with reducing user response times.

This paper makes two contributions. First, our analysis and simulations reveal that changing the optimal placement (across two adjacent time epochs) may inflict huge repositioning costs, even when reacting to a small change in the demand. Thus, a practical treatment of the dynamic placement problem should provide a tradeoff between optimizing

[1]An EC2 instance is a virtual machine that runs a specific application image.

the resource placement as to meet the demand, and minimizing repositioning costs, i.e., the number of added and removed resources.

Second, we propose an algorithmic solution addresses the above concern and yields very efficient dynamic placements with bounded repositioning costs. The algorithm consists of two components.

The first one is a **Lazy Algorithm (LA)** and is driven by our sensitivity analysis. The analysis shows that small changes in the demand distribution result in a bounded placement cost deviation (though it can incur very large repositioning costs). Consequently, the algorithm avoids unnecessary resource reconfigurations by maintaining the current placement when placement deviation cost is small.

The second component is a **Shortest Cycle Canceling algorithm (SCC)**, which modifies a (previously determined) placement and adapts it to new demand distribution. The algorithm efficiently improves a placement (reduces its cost) under a bounded reposition cost. It is based new theoretical techniques using graph theory methodologies that can be applied to other optimization/combinatorial problems. The algorithm uses the following methodologies and principles: 1) A reduction of the static placement problem into a min cost flow problem. 2) Finding improved (lower cost) placements by finding augmenting flows through negative cycles in the corresponding flow graph. 3) Achieving improved placements whose reposition costs are minimal by finding negative cycles whose length is minimal.

Combining these two parts, we propose a **Hybrid algorithm** that uses LA to avoid placement reconfigurations until needed, and then applies SCC to allow for controlled non-abrupt changes in the configuration of resources.

We evaluate our algorithms using practical conditions, i.e., using Amazon's EC2 on-demand image costs and realistic demand arrival rates. Our results demonstrate that the proposed Hybrid algorithm can achieve near optimal placement (its cost is higher up to 1.4% from the optimal placement cost), while maintaining a low reposition cost. In particular, the reposition cost of the proposed Hybrid algorithm is significantly lower than that of an optimal placement (more than 65%) as well as that of a proportional mean-based placement, a simple and widely-used scheme [17, 20, 18] wherein the number of servers is proportional to the average demand.

The rest of the paper is organized as follows. Section 2 describes previous work. Section 3 presents our model and the problem. Section 4 deals with sensitivity analysis of dynamic placements. Section 5 describes the lazy placement algorithm. Sections 6 and 7 describes the SCC algorithm for improving placement. Section 8 presents the Hybrid algorithm. Section 9 describes our evaluation methodology. Finally, section 10 concludes this paper.

The reader should note that the framework addressed in this paper is fairly wide and entails, in some cases, many details; for clarity of presentation, we placed some of these details in the technical report [19] and the appendix.

2. RELATED WORK

The problem of resource placement in distributed systems often falls under facility location theory [9]. This area has received significant attention from the viewpoint of both analysis and algorithmic solutions. Our version of the problem differs from traditional facility location problems in that it incorporates stochastic demands and capacity constrains on the locations (servers).

Early works on distributed resource placement primary focused on placing content replicas across a content distribution or a web cache network (see e.g. [16, 15]). However, most of these works have focused on static or deterministic demand profiles, paying little attention to the capacity limits at individual servers and geographical areas.

With the growth of cloud computing and large-scale dynamic services, the problem of dynamic server placement in geo-distributed environments has received increasing attention. Some works studied the problem from a standpoint of a service provider. For example, [24] focused on dynamically optimizing service placement while ensuring performance requirements; and [21] studied algorithms for dynamic scaling of social media applications. Other works looked at the problem from a cloud provider's viewpoint, i.e., assigning workloads to distributed datacenters in a cost-effective manner [22]. These works typically develop an optimization problem with deterministic inputs and apply it periodically. In contrast, we provide dynamic algorithm that inherently accounts for the full distribution of the demand and for repositioning costs, enabling cost-efficient placement with controlled amount of resource repositioning.

A variant of the placement problem which focuses on the static placement problem (see Subsection 3.2) and accounts mainly for service costs (see Subsection 3.1) has been considered in the context of content replication in P2P systems. [20] was perhaps the first to study a network setting similar to ours with an exponentially expanding topology for file sharing systems. It proved the optimality of proportional replication i.e., one based on the mean demand, under the assumption of abundant storage and upload capacity where all possible requests are always be served. In contrast, our model allows for restricting the number of resources (files), resulting in min-cost flow based replication. A variant of the placement problem which accounts mainly for service costs (see basic model in Subsection 3.1) has been considered in the context of content replication in P2P systems. [20] was perhaps the first to study a network setting similar to ours with an exponentially expanding topology for file sharing systems. It proved the optimality of proportional replication i.e., one based on the mean demand, under the assumption of abundant storage and upload capacity where all possible requests are always be served. In contrast, our model allows for restricting the number of resources (files), resulting in min-cost flow based replication.

Other relevant works are [25, 5, 11] that focused on P2P VoD replication systems. [25] proposed an optimal replication algorithm, called RLB, based on the assumption that the number of movies is much smaller than the number of peers. The work focused on small-scale networks, and do not consider geo-distributed topologies. [5] proposed placement framework for large-scale VoD service based on mixed integer program. While their model accounts for arbitrary demand pattern and network structure it assumes deterministic demand, whereas we consider stochastic one. [11] characterized the service efficiency of distributed content platforms as function of servers' storage size by using an asymptotic performance model for dynamic matching algorithms. In this context, our work maps to a single content server storage model, which enables us to solve the combined optimization problem of matching and placement (i.e., static

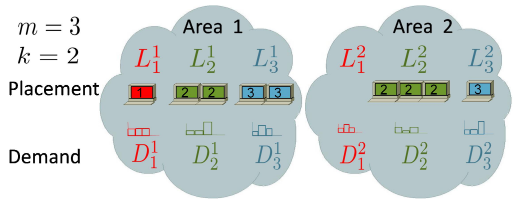


Figure 1: An example of the system topology. Note that the storage is $s^1 = 5$ and $s^2 = 4$ and the placement is $L_1^1 = 1$, $L_1^2 = 0$, $L_2^1 = 2$ etc.

placement) using exact analysis.

This work solves a dynamic setting under a general cloud-oriented cost model; this is in contrast to [18], which formulated and treated the static placement problem under a basic and limited model (reviewed in Subsection 3.1). More differences between [18] and our paper are discussed in Subsection 3.2.

3. THE MODEL AND THE PROBLEM

For the sake of exposition, we start by describing the model used for the dynamic and static placement problem. We then describe theses problems. Although we focus our examples on a SaaS provider that rents servers from a cloud provider, the generality of our model allows solving a variety of resource placement problems spanning from server placement in server farms to personnel placement in call centers.

3.1 The model

We consider a SaaS provider that can rent servers from a cloud provider by placing the servers in k **areas** indexed by $1, 2, \ldots, k$. In each area the provider can place multiple resources (servers) of different **types** indexed by $1, 2, \ldots, m$. Let L_i^j denote the number of type i resources placed in area j. The set of resources (servers) placed in these areas is called a **placement** and denoted by $L = \{L_i^j\}$. We assume that the SaaS provider can place in area j up to s_j resources due to the cloud provider limitations (As Amazon EC2 poses [3]). Placement L is called **feasible** if the number of resources in area j is not larger than s^j, i.e $L^j := \sum_{i=1}^m L_i^j \leq s^j$. We define the **total system storage** as $s := \sum_{j=1}^k s^j$.

We consider a stochastic demand for resources. Let D_i^j be a random variable denoting the number of requests for type-i resource in area j. We do not make any assumption on the distribution of D_i^j, namely it can be of an **arbitrary** (non-negative) **distribution**. We assume that the demand CDF values $\Pr(D_i^j \geq n)$, $n = 0, 1, 2, \ldots$, as well as their mean value $E(D_i^j)$, are calculated in $O(1)$, and are available in an external data base. An example of the topology of a system is depicted in Fig. 1, where every computer represents a resource.

Service cost model. Consider a type i request made in area j and a placement L. If the request is assigned to a resource in L, then the request is called **satisfied**. If the request is satisfied, it is assigned to either of: 1) A resource of L located in area j (and therefore the request is granted **locally**). 2) A resource of L located in a different area (granted **remotely**). If a request is not assigned to any resource in L, then it is called an **unsatisfied** request. The costs of a locally satisfied request, a remotely satisfied request, and an unsatisfied request are denoted by $C_i^{loc}, C_i^{rem}, C_i^{unsat}$, naturally obeying $C_i^{loc} \leq C_i^{rem} \leq C_i^{unsat}$. For example, in cloud-based applications, these costs can represent the revenue loss associated with increasing user response times when a request is granted remotely rather than locally or when its not granted at all.

Given a placement L and a deterministic realization d_i^j of the demand D_i^j, one can derive an optimal assignment of the resources to the demand, yielding minimal assignment cost (see Subsection 4.1). Let $C(L, d_i^j)$ be the optimal assignment cost and denote by g_i^{loc}, g_i^{rem} and g_i^{unsat} the corresponding number of requests granted under the optimal assignment from a local area, granted from a remote area, and unsatisfied ones, respectively. Then, the minimal assignment cost is simply $C(L, d_i^j) = C_i^{loc} \cdot g_i^{loc} + C_i^{rem} \cdot g_i^{rem} + C_i^{unsat} \cdot g_i^{unsat}$. The expected **service cost** $E|_D(C(L))$ is defined as the expected value of the minimal assignment cost over all demand realizations: $E|_D(C(L)) = \sum_{i=1}^m \sum_{j=1}^k C(L, d_i^j) \cdot \Pr(D_i^j = d_i^j)$.

Placement cost model. We associate a **resource cost** with placement L, denoted by $C_r(L)$, which represents the cost of placing and operating the resources of L. We assume that $C_r(L)$ is a semi-separable function. Roughly speaking, this means that $C_r(L)$ consists of the sum of individual functions, each of them is a function either of: 1) The number of resources placed of type i (L_i), 2) the number of resources placed in area j (L^j), 3) The number of type i resources placed in area j (L_i^j). A formal definition of semi-separable functions is given in Section 6.1.

This cost model allows us to capture a variety of server rental plans. For example, the on-demand server pricing of cloud providers such as EC2 [2] can be captured as p_i^j, a fixed price for running type-i server in area-j, yielding a resource cost of $C_r(L) = \sum_{i=1}^m \sum_{j=1}^k L_i^j \cdot p_i^j$. A more complicated example can incorporate software licensing costs, e.g. Google Cloud charges [10], and area-specific charges. Let r_i be the licensing cost associated with type-i server and h^j be an add-on provisioning cost for area j. Then, the (semi-separable) resource cost becomes

$$C_r(L) = \sum_{i=1}^m \sum_{j=1}^k L_i^j \cdot p_i^j + \sum_{j=1}^k L^j \cdot h^j + \sum_{i=1}^m L_i \cdot r_i. \qquad (1)$$

We assume that the resource cost is independent of the demand distribution D_i^j and of the service cost constants. We define the **static placement cost** $C_p(L)$ as the sum of the resource cost and the service cost

$$C_p(L) = C_r(L) + E|_D(C(L)). \qquad (2)$$

We assume that $C_p(L)$ is a convex function. That is, that the marginal profit from adding a resource (see Remark 3.1) decreases as the quantity of resources increases, as is often the case in operating costs of real-world systems.

Resource capacity. A type-i resource can serve at most B_i requests. This parameter can reflect the upload or processing capacity of a server.

3.2 The static placement problem

The static placement problem was addressed in [18]. Our focus here is on the *dynamic problem* (see next subsection).

Also, we are considering a significantly more general cost function than that used in [18], since the model in [18] assumed $C_r(L) = 0$. Another limitation [18] poses is that every server can grant up to one request $B_i = 1$. Therefore the analysis we provide here can also generalize the treatment of the static problem provided in [18]

We define the static placement problem as a minimization of the (total) placement cost under storage constrains. More formally, given the demand distributions $\{D_i^j\}$, $i = 1, \ldots, m$, $j = 1, \ldots, k$, service cost parameters C_i^{loc}, C_i^{rem}, C_i^{unsat}, placement cost parameters, area storage values $s^1, s^2, \ldots s^k$, resource capacities B_i and an optimal matching algorithm, determine the optimal resource placement $L = \{L_i^j\}$, $i = 1, \ldots, m$, $j = 1, \ldots, k$, that minimizes the placement cost $C_r(L) + E|_D(C(L))$ among all feasible placements obeying $L^j = \sum_{i=1}^m L_i^j \le s^j$.

Remark 3.1. *In some applications service cost parameters are alternatively replaced by* ***service revenues parameters*** $R^{loc} \ge R^{rem} \ge R^{unsat}$, *representing the revenue of satisfying the demand. These service revenue parameters represent respectively the Average Revenue Per User (ARPU) of granting a user request locally, remotely, or of not granting the user request. Under this setting we may define an equivalent problem of* ***maximizing*** *placement profit where placement cost is replaced by* placement profit $E|_D(C(L)) - C_r(L)$. *To convert the revenue parameters to cost parameters we multiple the parameters by* -1.

3.3 The dynamic placement problem

In the dynamic placement problem, we assume that the demand is time-dependent where the demand distribution at time slot t is denoted as $D(t) = \{D(t)_i^j\}$. The static placement cost of placement L, which depends on the demand, $D(t)$, is denoted by $C_p^{D(t)}(L)$ and the optimal placement with respect to $D(t)$ is denoted by $L_{opt}(D(t))$. We call algorithm A a **dynamic algorithm** if it computes the placement at time t, $L_A(t)$, only as a function of past information; that is, it cannot use $D(t+1), D(t+2), \ldots$ Note that $L_A(t)$ is not necessarily the optimal placement $L_{opt}(D(t))$.

Two important costs are relevant to a dynamic algorithm: 1) The **static placement cost deviation** (alternatively, the **cost deviation**), and 2) The **resource reposition cost**. The former, denoted $C_{dev}(A, t)$, evaluates the deviation of the static placement cost between the output placement $L_A(t)$ and the optimal placement $L_{opt}(D(t))$ with respect to demand $D(t)$. That is

$$C_{dev}(A, D(t)) \doteq C_p^{D(t)}(L_A(t)) - C_p^{D(t)}(L_{opt}(D(t))). \quad (3)$$

We denote the placement cost deviation of A as the worst-case placement cost deviation over all inputs, i.e $C_{dev}(A) \doteq \max_{D(t)} C_{dev}(A, D(t))$.

The reposition cost evaluates the cost of repositioning the resources, namely of transforming placement $L_A(t)$ to the placement $L_A(t+1)$. The reposition cost, denoted as $r(A)$, counts the number of added and removed resources, and therefore the reposition cost is the L_1 **distance** between $L_A(t)$ and $L_A(t+1)$, i.e

$$r(A, t) \doteq \sum_{i=1}^{m} \sum_{j=1}^{k} |L_A(t)_i^j - L_A(t+1)_i^j|, \quad (4)$$

($L_A(t)_i^j$- number of type-i resources in region j). Finally, we denote the reposition cost of A as the worst-case reposition cost over all inputs, i.e. $r(A) \doteq \max_t r(A, t)$.

The goal of the dynamic placement problem is to develop a dynamic algorithm whose reposition cost and cost deviation are low.

Remark 3.2. *In our model we assume that servers (of the same type) are homogenous with respect to the number of users a server can serve. We show in our technical report [19] that if the servers are heterogeneous, then the static resource placement is NP-hard using the hardness of set-cover. Also, we prove that finding a c-approximation is also NP-hard for* $c < (1 - o(1)) \ln s$. *Therefore, every dynamic algorithm A may derive a solution which suffers from a large placement cost deviation of* $C_{dev}(A) = O(C(L_{opt}) \cdot \ln s)$.

4. SENSITIVITY OF DYNAMIC PLACEMENTS

In a dynamic environment one has to deal with the time varying demands $D(t)$ and with the need to recompute the optimal placement and reposition the resources as the demand changes. Our first major result shows that any algorithm $\mathbf{A}$ that insists on recomputing the optimal placement and reposition the resources accordingly at every time t may be subject to very high reposition cost $r(A)$ (see definition in Subsection 3.3). This holds even if the prior placement computed by A, $L_A(t-1)$, is optimal with respect to $D(t-1)$ (i.e $L_A(t-1) = L_{opt}(D(t-1))$) and the demands sets $D(t-1), D(t)$ are extremely close to each other. A high reposition cost may be quite undesired since it may imply that the system operations must be held due to lengthy repositioning.

In order to carry this analysis we derive a closed-form formula for the static placement cost $C_p(L)$, which can be derived using methods similar to those used in [18].

4.1 Preliminaries: A closed-form formula for the static placement cost

The static placement cost function as presented in Equation (2) is difficult to formulate in a closed-form formula. A key element is a transformation of the (expected) service cost from that equation, $E_{|D}(C(L))$, into "differential revenue", $E_{|D}(R(L))$, which expresses the service cost in relative terms (e.g. cost of being served locally relatively to being served remotely). We use the following definitions:

Definition 1. *The* ***local differential revenue constant*** *is the revenue gained from granting a type-i request locally compared to granting it remotely, i.e* $R_i^{loc} \doteq C_i^{rem} - C_i^{loc} \ge 0$. *The* ***global differential revenue constant*** *is the revenue gain from granting a type-i request remotely compared to not satisfying the request, i.e* $R_i^{glo} \doteq C_i^{unsat} - C_i^{rem} \ge 0$. *We define the* ***service revenue*** *as:*

$$E|_D(R(L)) = \sum_{i=1}^{m} R_i^{glo} \cdot E|_{D_i}(\min(B_i \cdot L_i, D_i)) + \quad (5)$$

$$\sum_{i=1}^{m} R_i^{loc} \cdot [\sum_{j=1}^{k} E|_{D_i^j}(\min(B_i \cdot L_i^j, D_i^j))] \quad (6)$$

To this end, we provide a closed-form formula for the static placement cost using the following theorem:

THEOREM 4.1. *For every placement L, the static placement cost is*

$$C_p(L) = C_r(L) + \sum_{i=1}^{m} E(D_i) \cdot C_i^{unsat} - E|_D(R(L)). \quad (7)$$

.

To show the correctness of theorem we provide an optimal assignment between a placement $L = \{L_i^j\}$ and a demand realization d_i^j, called the Generalized Assignment Algorithm. The Generalized Assignment Algorithm, which is a generalization of the Assignment Algorithm was given in [18], maximizes the number of locally granted requests in every area, and then it grants remotely the other unsatisfied requests. We prove in the technical report [19] that the number of type i requests granted locally is $g_i^{loc} = \min(B_i \cdot L_i^j, d_i^j)$, and the number of type i requests granted globally (i.e, granted locally or remotely) is $g_i^{glo} = \min(B_i \cdot L_i, d_i)$. Then, we prove that $\sum_{i=1}^m E(D_i) \cdot C_i^{unsat} - E|_D(R(L))$ equals to the service cost $E_{|D}(C(L))$ defined in the model section (Section 3).

4.2 Analysis of Placement Sensitivity

To derive the sensitivity of resource placements we start with the definition of strongly ϵ-near.

DEFINITION 2. *Given two vectors $\hat{v} = (v_1, \dots v_n)$ and $\hat{u} = (u_1, \dots u_n)$, the L_1-**distance** between them is $d(\hat{v}, \hat{u}) = \sum_{i=1}^n |v_i - u_i|$. Given two discrete distributions N_1, N_2 defined over the same support set $\{0, 1, 2 \dots\}$, and a parameter k, we define the L_1-**CDF distance** as the L_1-distance between the CDF vectors of N_1 and N_2 over k elements, i.e $d^k(N_1, N_2) = \sum_{n=0}^k |\Pr(N_1 \leq n) - \Pr(N_2 \leq n)|$. The demand sets $D = \{D_i^j\}$, $D' = \{D'^j_i\}$ are called **strongly ϵ-near** iff the following conditions hold: 1) $D_i^j = D'^j_i$ for all $(i,j) \neq (i_0, j_0)$ and 2) $d^\infty(D_{i_0}^{j_0}, D'^{j_0}_{i_0}) < \epsilon$.*

In the following we show that an algorithm that insists on using a static optimal placement at every time t may incur very high reposition cost even if the demand varies only slightly over time. This will result from the following theorem stating that there exist demand sets $D(t-1)$ and $D(t)$ which are strongly ϵ-near while the difference between their optimal placements, $L(d(t))$ and $L(D(t-1))$, implies very high reposition cost.

THEOREM 4.2 (SENSITIVITY IN REPOSITION COST). *For every $\epsilon > 0$ there exist two demand sets $D(t-1), D(t)$ which are strongly ϵ-near while the L_1-distance between the optimal placements $L_{opt}(D(t-1))$ and $L_{opt}(D(t))$, is larger than the maximum storage value i.e $d(L_{opt}(D(t-1)), L_{opt}(D(t))) \geq \max_j s^j$.*

A proof is given in the Appendix.

5. DYNAMIC PLACEMENT USING LAZY ALGORITHM (LA)

To address the issue of a high reposition cost we provide a dynamic algorithm called **Lazy Algorithm** (LA) and denoted as A_{LA}, which avoids repositioning when the placement cost deviation is small. Its operation is based on either computing the static optimal placement at time t, and placing the resources accordingly, or by being lazy and keeping the old placement. We prove that LA satisfies the following conditions: 1) Its placement cost deviation (defined in Subsection 3.3) $C_{dev}(A_{LA})$ is bounded by a given threshold ϵ, and 2) Its reposition cost stays 0, if the demand set $D(t)$ is close to the previous used set (which we called **weakly ϵ-near**).

5.1 Description of the Lazy Algorithm

To introduce LA we present the following definitions:

DEFINITION 3. *Let $D = \{D_i^j\}$, $D' = \{D'^j_i\}$ be two demand sets. The **demand-distance** between D and D', denoted as $d(D, D')$ is*

$$d(D, D') = \sum_{i=1}^{m} \sum_{j=1}^{k} d^s(D_i^j, D'^j_i) R_i^{loc} + \sum_{i=1}^{m} d^s(D_i, D'_i) R_i^{glo}, \quad (8)$$

*where s is the system capacity, R_i^{loc} and R_i^{glo} are respectively the local and global differential revenue constants. This equation resembles Eq (5). Demands sets D and D' are called **weakly ϵ-near** if the distance between them is less than ϵ, i.e $d(D, D') < \epsilon$.*

LA is a simple lazy algorithm with a threshold parameter ϵ. At time t LA holds a reference demand set $D_{ref}(t)$ equaling $D(\tau)$ for some $\tau < t$, where τ is the last time where the algorithm modified its placement. It also holds as reference the optimal placement $L_{opt}(D(\tau))$. The operation of LA at t is simple: It compares $D_{ref}(t)$ with $D(t)$ and checks whether they are weakly ϵ-near; if they are, then the output of LA, $L_A(t)$, is identical to $L_A(t-1)$ and equals to $L_{opt}(D(\tau))$; otherwise LA computes the optimal placement $L_{opt}(D(t))$ and sets this is as its output $L_A(t)$. In this latter case LA also updates the reference demand set $D_{ref}(t)$ to be $D(t)$ and the reference optimal placement to $L_{opt}(D(t))$. The optimal placement $L_{opt}(D(t))$ can be calculated by an optimal placement algorithm We can use for this purpose the BGA algorithm presented in [18].

5.2 Placement cost deviation theorem

We prove that LA has a low placement cost deviation, even in the cases where it does not recompute the optimal placement:

THEOREM 5.1 (PLACEMENT COST DEVIATION THEOREM). *Let ϵ be the threshold parameter of LA. If the demand reference $D_{ref}(t)$ and the demand $D(t)$ are weakly ϵ-near, then the cost deviation with respect to $D(t)$ is bounded by ϵ, i.e,*

$$C_{dev}(A_{LA}, D(t)) = C_p^{D(t)}(L_A(t)) - C_p^{D(t)}(L_{opt}(D(t))) < \epsilon. \quad (9)$$

The theorem is prove in the appendix using the following techniques: 1) We reduce the static placement problem to a min-cost flow problem, as presented in Section 6 below. 2) We use the out-of-kilter algorithm [1] which derives from a non-optimal flow f the min-cost flow f_{opt} (associated with the optimal placement for the reference demand) and showing that it changes the flow weights (and therefore the static placement cost) by at most ϵ.

By Theorem 5.1 we have the following corollary:

COROLLARY 5.2. *LA satisfies the following properties: 1) Its placement cost deviation is less than the threshold parameter ϵ, i.e $C_{dev}(A_{LA}) < \epsilon$, and 2) its reposition cost is 0 if the demand sets are weakly ϵ-near.*

Remark 5.3 (Competitive Ratio). *In the online algorithms literature, which is highly related to the dynamic algorithms literature, the performance of an online algorithm is measured by its* ***competitive-ratio*** *$C_{comp}(L)$, defined as the worst-case ratio between the cost of the online algorithm placement and the cost of the optimal offline placement, i.e,*

$$C_{comp}(L) = \max_{D(t)} \frac{C_p^{D(t)}(L_A(t))}{C_p^{D(t)}(L_{opt}(D(t)))}.$$

Note that this definition can be applied only if the cost function is always non-negative, which is not in our case. However, if we assume that the service revenue parameters are positive (i.e, $C_i^{loc} > 0$) then our cost function $C_p^{D(t)}()$ is non-negative. In this case, the competitive-ratio of LA is $\frac{\epsilon}{\min_{D(t)} C_p^{D(t)}(L_{opt}(D(t)))} + 1$. Note also that the cost of every placement must be larger than the cost of the hypothetical case where every request is grented locally. Thus $C_p^{D(t)}(L_{opt}(D(t))) \geq \sum_{i=1}^{m} E(D_i) \cdot C_i^{loc} > 0$ and therefore LA is bounded from below by a ***constant*** *competitive-ratio of $\frac{\epsilon}{\sum_{i=1}^{m} E(D_i) \cdot C_i^{loc}} + 1$.*

5.3 Computing demand-distance in LA

To compute the demand-distance between $D_{ref}(t) = D$ and $D(t) = D'$, one will need to go over all resource types i and regions j and compute the L_1-distance with parameter s between D_i^j and D'^j_i. Computing the L_1-distance should take $O(s)$, and computing the demand-distance takes $O(skm)$.

In particular distributions one can compute the demand-distance faster than $O(skm)$. For example, if the demand distributions are bounded by some constant, i.e, $\Pr(D'^j_i \geq C) = \Pr(D_i^j \geq C)$ for some constant $c << s$. In this case, computing the demand-distance can be done in $O(ckm)$.

Another interesting property is when one distribution **dominates** the other. Formally, distribution D_1 dominates D_2 if $\Pr(D_1 \geq n) \geq \Pr(D_2 \geq n)$ for every value $n \geq 1$. In the following Claim we show that if the distributions D_1 and D_2 are either Poisson or Binomial, then one dominates the other :

Claim 5.4. *The following holds: 1) If $D_1 \sim Poiss(\lambda_1), D_2 \sim Poiss(\lambda_2)$, where $\lambda_1 > \lambda_2$ then D_1 dominates D_2. 2) If $D_1 \sim Bin(n, p_1), D_2 \sim Bin(n, p_2)$, where $p_1 > p_2$ then D_1 dominates D_2.*

In case where one distribution dominates the other the L_1 distance can be shown to be equal to the expected values of the distributions:

Claim 5.5. *If D_1 dominates D_2 then $d(D_1, D_2) = E(D_1) - E(D_2)$.*

Following Claim 5.5 we may conclude that if some of the distributions pairs, D_i^j and D'^j_i, obey pair-wise dominance, the distance between them can be computed in $O(1)$ operations, and the over all demand distance computation may reduce significantly from $O(skm)$. If all distribution pairs admit pair-wise dominance then the over all demand computation drops to $O(km)$.

5.4 Drawbacks of LA

Although the LA algorithm provides us a tradeoff between the reposition cost and the placement cost deviation, it may encounter high reposition costs and run-time complexity under demand sets that are weakly ϵ-far. First, since the reposition cost is not bounded (see Theorem 4.2), LA may incur very large reposition cost, which is impractical. In our simulations presented in Section 9 we show that LA incurs large reposition cost. Second, LA uses the static placement algorithm which incurs a high computational complexity.

To overcome these difficulties we present the Shortest Cycle-Canceling algorithm (SCC), which reduces the placement cost deviation under a bounded reposition cost. A key building block for SCC is the reduction of the static placement problem into the min-cost flow problem, described next.

6. REDUCTION OF THE STATIC PLACEMENT PROBLEM INTO A MIN-COST FLOW PROBLEM

Here we solve the static placement problem to allow us to prove Theorem 5.1 and is a building block for developing SCC.

We first begin with defining the **loss** function whose properties are a key for solving the static (and dynamic) placement problem. Note the techniques shown here generalize the methods which [18] used the simplified model. However, we used a more generalize approach to solve the static problem.

6.1 The loss function and its properties

The key to show the reduction the loss function we define the loss function simply as the difference between the placement cost function $C_p(L)$ and the constant $\sum_{i=1}^{m} E(D_i) \cdot C_i^{unsat}$, i.e, $loss_p(L) = C_p(L) - \sum_{i=1}^{m} E(D_i) \cdot C_i^{unsat}$. The optimal placement minimizing the static placement cost C_p must minimize the cost.

According to Eq 7 the loss is equal to:

$$loss_p(L) = C_r(L) - E|_D(R(L)). \qquad (10)$$

.

To show the loss function special properties we define the following definitions:

Definition 4. *Let $f : N^{m \cdot k} \to R$ be a discrete function.*

1. *f is called* ***semi-separable*** *iff there exists a set of discrete functions $\{f_i^j\}, \{f_i\}, \{f^j\}$ such that f is expressed by the following formula*

$$f(L_1^1, L_2^1, \ldots) = \sum_{i=1}^{m} \sum_{j=1}^{k} f_i^j(L_i^j) + \sum_{i=1}^{m} f_i(L_i) + \sum_{j=1}^{k} f^j(L^j), \qquad (11)$$

   where $L_i = \sum_{j=1}^{k} L_i^j$, and $L^j = \sum_{i=1}^{m} L_i^j$. The set $M(f) = \{f_i^j\} \bigcup \{f_i\} \bigcup \{f^j\}$ is called the ***marginal functions*** *of f.*

2. *Given a one-dimensional discrete function $g : N \to R$, define its* ***differential function*** *Δg as the difference between its successive values, i.e $\Delta g(n) = g(n) - g(n-1)$ for $n \geq 1$. The function g is called* ***convex*** *if its differential function Δg is monotonically non-decreasing, i.e, $\Delta g(n) \geq \Delta g(n+1)$ for all $n \geq 1$.*

3. *A semi-separable function f is is called **convex** if its marginal functions f_i,f^j,f_i^j are convex.*

In Section 3 we assumed that the resource cost, $C_r()$, is a semi-separable function. This means that there exists a set of marginal functions $M_\zeta(C_r) = \{\zeta_i^j\} \bigcup \{\zeta_i\} \bigcup \{\zeta^j\}$ such that

$$C_r(L) = \sum_{i=1}^{m} \sum_{j=1}^{k} \zeta_i^j(L_i^j) + \sum_{i=1}^{m} \zeta_i(L_i) + \sum_{j=1}^{k} \zeta^j(L^j). \quad (12)$$

In the linear resource cost example presented in Eq (1) the marginal functions are simply $\zeta_i^j(L_i^j) = L_i^j \cdot p_i^j, \zeta^j(L^j) = L^j \cdot h^j$ and $\zeta_i(L_i) = L_i \cdot r_i$.

In the technical report [19] we prove the following claim:

Claim 6.1. *The loss function $loss_p$ is semi-separable and convex.*

Thus, there exists a set of marginal convex functions $M_\gamma(loss_p)$ such that $loss_p(L) = \sum_{i=1}^{m} \sum_{j=1}^{k} \gamma_i^j(L_i^j) + \sum_{i=1}^{m} \gamma_i(L_i) + \sum_{j=1}^{k} \gamma^j(L^j)$.

6.2 Introduction to the min-cost flow problem

We start describing the **min-cost flow problem** [6], which is a generalization of the notable max flow problem. This problem is similar to the **max-flow** problem (see [8]), where we considers a directed graph $G = (V, E)$ where every edge $e \in E$ has integer **capacity** $c(e)$. In addition, every edge is associates with a real-value **weight** $w(e)$ (alternatively, called **cost**). Given a flow f (defined similarly to the max flow problem), we define $f(v) = \sum_{u|(u,v)\in E} f(u, v) = \sum_{u|(v,)\in E} f(v)$ as the **flow in node**. The **flow value** of f as the flow in the source x (and sink y), $|f| = \sum_{(x,v)\in E} f(x, v) = \sum_{(v,y)\in E} f(v, y)$. The **weight (or cost)** of flow f is $w(f) = \sum_{e\in E} f(e)w(e)$.

The classic **min-cost flow** problem with required flow $|f| = k$ is to find a flow $f_{opt}(k)$ of value k that has minimal weight among all flows of value k. This means that for every flow f' such that $|f'| = |f_{opt}(k)| = k$ we have $w(f_{opt}(k)) \leq w(f')$.

6.3 Reduction of the placement problem to a min-cost flow problem

Given the loss function $loss_p$ with its marginal functions $M_\gamma(loss)$, we define the **marginal-differential** functions $\{\Delta\gamma_i^j\} \bigcup \{\Delta\gamma_i\} \bigcup \{\Delta\gamma^j\}$ simply as the differential of the correspond marginal functions $\{\gamma_i^j\} \bigcup \{\gamma_i\} \bigcup \{\gamma^j\}$. The following claim regarding the marginal differential functions is straightforward:

Claim 6.2. *The marginal-differential functions $\Delta\gamma$ can be calculated using the following equations:*

$$\Delta\gamma^j(n) = \zeta^j(n) - \zeta^j(n-1) \quad (13a)$$

$$\Delta\gamma_i^j(n) = \zeta_i^j(n) - \zeta_i^j(n-1) - \sum_{k=1}^{B_i} R_i^{loc} \Pr(D_i^j \geq n \cdot B_i + k) \quad (13b)$$

$$\Delta\gamma_i(n) = \zeta_i(n) - \zeta_i(n-1) - \sum_{k=1}^{B_i} R_i^{glo} \Pr(D_i \geq n \cdot B_i + k). \quad (13c)$$

In a linear resource cost function (Eq (1)) the marginal-differential functions are equal to $\Delta\gamma^j(n) = h^j$, $\Delta\gamma_i^j(n) = p_i^j - R_i^{loc} \Pr(D_i^j \geq n \cdot B_i)$ and $\Delta\gamma_i(n) = r_i - R_i^{glo} \Pr(D_i \geq n \cdot B_i)$.

Remark 6.3. *The marginal-differential $\Delta\gamma(n)$ are 1) monotonically increasing, which stems from the fact that the loss is convex (see Claim 6.1), and 2) are negative if the addition the n^{th} resource benefits the system i.e, has a negative marginal cost.*

We reduce the placement problem to a min-cost problem using the graph given in Fig. 2 as follows: We define a directed 8-layer graph $G^8 = (V^8, E^8)$. On every edge a pair $c(e), w(e)$ so that $c(e)$ is the capacity function (presented in olivegreen color) and $w(e)$ is the weight function. The graph is composed from the following 8 layers: The **source** x, the **(area, #resources)** layer, the **area** layer, the **(area, type)** layer, the **(area, type, #resources)** layer, the **type** layer, the **(type, #resouces)** layer, and finally the **sink** y. We also connect the source x to the sink y.

In the graph we denote area j by a^j, type i by t_i, and #resouces by a number. The (area, #resouces), (area, type, #resources) and (type, #resources) layers, for example, are respectively composed of nodes (a^j, n) (where $1 \leq j \leq k$ and $1 \leq n \leq s^j$), (a^j, t_i, n) (where $1 \leq j \leq k$, $1 \leq i \leq m$,$1 \leq n \leq s^j$) and (t_i, n) (where $1 \leq i \leq m$ and $1 \leq n \leq \sum_{j=1}^{k} s^j$). The entering edges to nodes (a^j, x), (a^j, t_i, x) and (t_i, x) have respectively the marginal-differential weight $\Delta\gamma^j(x)$, $\Delta\gamma_i^j(x)$ and $\Delta\gamma_i(x)$ with capacity of 1. All other edges have weight 0 with capacity ∞.

Finding the min-cost flow on G yields the optimal placement as presented in the following lemma:

Lemma 6.4. *Let f_{opt} be the min-cost flow in G^8 with a required flow of $|f| = \sum_{j=1}^{k} s^j \doteq s$. Let L be a placement with components equal to the flow in nodes (a^j, t_i), i.e $L_i^j = f_{opt}(a^i, t^j)$. Then L solves the placement problem.*

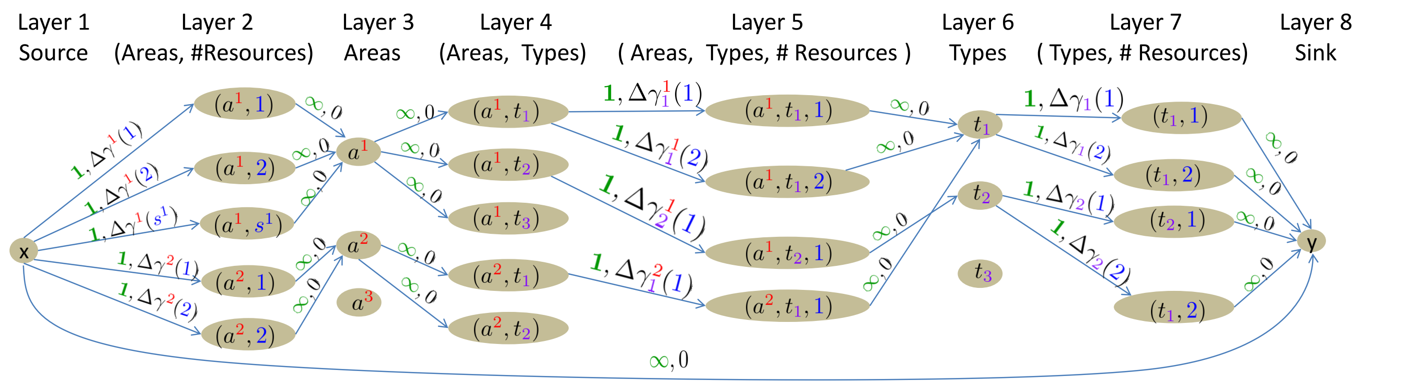
The correctness of the lemma, presented in the technical report [19] stems from the fact that 1) the marginal-differential weights are monotonically increasing 2) every flow has an equivalent placement, as presented in the following claim:

Lemma 6.5. *Let P be a placement. We define a flow $f(P)$ such that 1) The disjoint paths x-(a^j, n)-a^j, (a^j, t_i)-(a^j, t_i, n)-t_i, t_i-(t_i, n)-y are assigned respectively with one unit of flow if $P^j \geq n$, $P_i^j \geq n$ and $P_i \geq n$; Otherwise, the correspond path is assigned with zero unit of flow. 2) The flow between (a^j, t_i), and (a^j, t_i) is P_i^j. 3) The flow between x and y is $s - \sum_{i=1}^{m} \sum_{j=1}^{k} P_i^j$.*

Then the cost of $f(P)$ equals to the cost of P

7. DYNAMIC PLACEMENT UNDER REPOSITION CONSTRAIN- SCC ALGORITHM

The reduction provided in the previous section implies that one can use an optimal min-cost flow algorithm to solve the static placement problem. One particular algorithm for solving the optimal min-cost flow problem, is the well known Cycle-Canceling algorithm, that cancels in every iteration a negative cycle, which reduces the cost of the flow associated with the placement. We will utilize this property to modify



$$\Delta\gamma^j(n) = \zeta^j(n) - \zeta^j(n-1) \quad \Delta\gamma_i^j(n) = \zeta_i^j(n) - \zeta_i^j(n-1) - R_i^{loc}\Pr(D_i^j \ge n \cdot B_i) \quad \Delta\gamma_i(n) = \zeta_i(n) - \zeta_i(n-1) - R_i^{glo}\Pr(D_i \ge n \cdot B_i)$$

Figure 2: The 8-layer graph . Note that the edge weights, which are the marginal-differential functions defined in Claim 6.2, can be negative.

a (previously determined) placement and adapt it to new demand distribution.

The Shortest Cycle-Canceling (SCC) algorithm we propose for the dynamic placement problem is a variation of the well-known Cycle-Canceling algorithm that finds a negative cycle with the minimal number of edges. To implement SCC, we combine the Cycle-Canceling algorithm with the **Shortest negative Cycle (SnC)** algorithm that uses dynamic programming for finding the shortest negative cycle (i.e, with minimal number of edges).

SCC is a dynamic Cycle-Canceling algorithm with a threshold parameter r. SCC ensures that given an initial placement $L(t-1)$ computed at time slot $t-1$, it finds an placement $L(t)$ such that 1) the reposition cost is always less than or equal to r, i.e, $\sum_{i=1}^{m}\sum_{j=1}^{k}|L(t)_i^j - L(t+1)_i^j| < r$. 2) The placement cost deviation of $L(t+1)$ with respect to $D(t+1)$ is smaller than that of $L(t)$.

SCC finds a minimal (in number of edges) cycle, so it can augment many cycles as possible.

7.1 Preliminaries: Cycle-Canceling algorithm

The well-known Cycle-Canceling algorithm solves the min-cost flow problem for general graphs. Given a flow f on a graph $G=(V,E)$, the Cycle-Canceling algorithm uses the **residual graph** $G_f=(V,E_f)$. On the residual graph edges one defines weight w_f and capacity c_f. Then, the residual graph G_f is constructed from graph G and from flow f by the following steps: 1) Add to G_f edges from G, such that edge $(v,v')\in E$ will have weight $w_f(v,v')=w(v,v')$ and capacity of $c_f(v,v')=c(v,v')-f(v,v')$. 2) Add the reverse edges of G. That means, if $(v,v')\in E$, then add edge (v',v) to G_f with weight $w_f(v',v)=-w(v,v')$ and capacity of $c_f(v',v)=f(v,v')$. Note that for every edge e in G_f we have $c(e)\ge 0$. 3) For every edge $e\in E_f$ with capacity $c(e)=0$ set its weight to be $w_f(e)=\infty$. Finally, a cycle C is called negative if the sum of its weights is negative, i.e, $w(C)=\sum e\in Cw(e)<0$.

Detecting a negative cycle is a key building block for Cycle-Canceling. The Cycle-Canceling algorithm can specify which negative cycle detection it will use. For example, it can use the **Bellman-Ford** algorithm that finds some negative cycle, or **Karp's** algorithm that finds a negative cycle with respect to a minimal mean value. The running time of using the Cycle-Canceling algorithm is effected by the negative cycle it finds: if one uses Bellman-Ford then the time complexity of the Cycle-Canceling algorithm might be unbounded, while using Karp's algorithm on a graph $G=(V,E)$ it is at most $O(|V||E|^2\log|V|)$ [1].

The Cycle-Canceling algorithm works iteratively, and in the i^{th} iteration it finds a negative cycle in G_f. It executes the following steps in every iteration: 1) Find a negative cycle C in G_f using some negative cycle detection algorithm. If there is no negative cycle then the algorithm terminates. 2) We augment the flow by $\delta=\min c_f(e)|e\in E>0$ units of flow through C. That means that if $(v,v')=e\in C$ is in the original graph (i.e $e\in E$) then we update $f(e)\leftarrow f(e)+\delta$, and if the reverse edge in G (i.e $(v',v)\in E$) then we update $f(v',v)\leftarrow f(v',v)-\delta$. 3) We update a new residual graph G_f according to the updated flow f.

The following theorem is established in [1]:

THEOREM 7.1. *1) In every iteration of the Cycle-Canceling algorithm the cost of the flow decreases. 2) If there is no negative cycle in G_f then f is a min-cost flow*

7.2 Cycle-Canceling Example

In the following example we demonstrate how canceling a negative cycle improves the placement cost according to the new demand. Consider for instance a placement of time slot $t-1$, $L(t-1)$, with two areas ($k=2$), and two resource types ($m=2$). For every area j the number of type-i resources is 2 i.e, $L_1^1(t-1)=L_1^2(t-1)=L_2^1(t-1)=L_2^2(t-1)=2$. Of course the number of resources in every area is $L^1=L^2=4$. We assume that every server can serve one request ($B_i=1$). We set the resource cost $C_r(L)$ to zero (i.e, $\zeta^j(x)=\zeta_i^j(x)=\zeta_i(x)=0$ for every $x\ge 0$). Due to the new demand $D(t)$, the edges weights in the 8-layer residual graph G_8^f change, and new values are set according to Eq. 13. For instance, the edge weight between the node $(a^1,t_1,3)$ and a type node t_1 are set to $\Delta\gamma_1^1(3)=-\Pr(D(t)_1^1\ge 3)$.As the residual graph G_f^8 adds reverse edges to the 8-layer graph G^8 from Fig. 2, it contains an edge between a type node t_1 and node $(a^2,t_1,2)$. The weight of that edge is set to $-\Delta\gamma_1^2(2)=\Pr(D(t)_1^2\ge 2)$. Note that $\Delta\gamma_1^1(3)$ represents the marginal cost of adding the third type-1 resource to area 1, while $-\Delta\gamma_1^2(2)$ represents

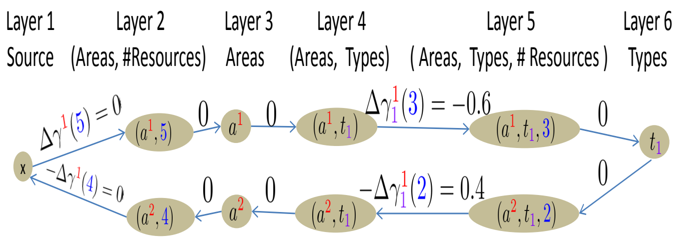


Figure 3: An example demonstrating the cycle-canceling technique in G_f^8

the marginal cost of removing the second resource from area 2.

In our example, we assume the following values for demand distribution $D(t)$: $\Delta\gamma_1^1(3) = -\Pr(D(t)_1^1 \geq 3) = -0.6$ and $-\Delta\gamma_1^2(2) = \Pr(D(t)_1^2 \geq 2) = 0.4$. In Fig. 3 we depict a cycle which represents adding a type-1 resource in area 1 and removing a type-1 resource in area 2. Augmenting flow through the cycle will decrease the cost of the placement by $\Delta\gamma_1^1(3) - \Delta\gamma_1^2(2) = -0.2$. It will add a type-1 resource in area 1 (with marginal cost of $\Delta\gamma_1^1(3)$) and remove a type-1 resource in area 2 (with marginal cost of $-\Delta\gamma_1^2(2)$). After we update the flow, its associated placement (computed by Claim 6.5) will contain three type-1 resources in area 1, and a single type-1 resource in area 2.

As we set the resource cost $C_r(L)$ to be zero, we get that the areal marginal cost of adding the k^{th} resource (from every type) to area 1, which is the edge weight between x and (a^1, k), is $\Delta\gamma^1(k) = 0$ for every $k \geq 1$. Removing the k^{th} resource from area 2, which is the edge weight between (a^2, k) and x, equals to $-\Delta\gamma^2(k) = 0$.

Note that the cycle involves only two edges (one forward and one backward) between layer 4 and 5, implying shifting exactly 1 resource. Augmenting paths that involve shifting k resources will have k forward edges and k backward edges between layer 4 and 5.

7.3 Shortest negative Cycle (SnC)

The Shortest negative Cycle (SnC) algorithm finds the negative cycle with minimal number of edges. Let $G = (V, E)$ be a general directed graph with a weight function w defined over the graph edges E. In the l^{th} iteration it computes a matrix $A(l)$, such that $A_i^j(l)$ is the cost of the shortest path between $i \in V$ and $j \in V$, among all shortest paths with l edges. One can compute the matrix $A(l)$ using the following formulas:

$$A_i^j(1) = \begin{cases} w(i,j) & \text{when } (i,j) \in E. \\ \infty & \text{otherwise.} \end{cases}$$
$$A_i^j(l) = \min_{k|(k,j)\in E} (A_i^k(l-1) + w(k,j)) \text{ for } l \geq 2. \tag{14}$$

Note that computing A_i^j for every pair (i,j) costs $O(d_j)$ operations, where d_j is the outgoing degree of vertex j. Thus, the complexity of one iteration is $O(\sum_{i\in V}\sum_{j\in V} d_j) = O(|V| \cdot |E|))$.

SnC is an iterative algorithm that computes the matrix $A(l)$ in the l^{th} iteration. It finds l_{min} such that there exists a vertex $i \in V$ with a negative shortest distance of length l_{min} , i.e, $A_i^i(l_{min}) < 0$. Next, SnC computes a vertex i_0 with minimal cost, i.e, $i_0 = \arg\min_i A_i^i(l_{min}) < 0$. To return the negative cycle, we compute the parent array $p(j)$, such that $p(1) = p(l_{min}) = i_0$ and $p(l) = \arg\min_k A_i^k(j-1) + w(k, p(l+1))$ for $1 < l < l_{min}$.

Note that if after $|V|$ iterations SnC did not find a negative cycle(i.e, $A_i^i(l) \geq 0$ for every l, i), then the graph G does not contain one.

The following claim, which states the run time complexity of SnC, is proved in a technical report [19]:

Claim 7.2. *SnC determines whether G contains a negative cycle. If such cycle exists- it computes the minimal negative cycle in running time of $O(l_{min}|V||E|)$, where l_{min} is the number of edges in the negative cycle. Otherwise - the running time is $O(|V|^2|E|)$.*

Remark 7.3. *A related problem to the Shortest negative Cycle problem is that of finding a simple negative cycle of minimal cost, which is defined as the sum of its edge costs. This problem was proved to be NP-complete (A proof can be seen in the technical report [19]).*

7.4 The Shortest Cycle-Canceling algorithm (SCC)

SCC is a dynamic Cycle-Canceling algorithm with a threshold parameter r. Initially, the algorithm builds the residual graph of the 8-layer graph G_f^8 with respect to the demand at time t, $D(t)$, and the flow f_{init} representing the placement at $t-1$, $L(t-1)$. In every iteration SCC holds a flow f_{prev} of the previous iteration, where in the first iteration we set $f_{prev} = f_{init}$. SCC uses SnC to find the shortest negative cycle in G_f, denoted as C. If there exists no cycle then f_{prev} is a min-cost flow, and its associated placement $L_i^j = f_{prev}(a^i, t^j)$ must be the optimal-cost placement with respect to demand $D(t)$.

We obtain f_{curr} from augmenting the flow trough cycle C. We check whether the placement associated with f_{curr} violates the reposition cost constraint , i.e, $\sum_{i=1}^m \sum_{j=1}^k |L(t-1)_i^j - f_{curr}(a^i, t^j)| < r$. If so- then the algorithm set the placement for the next time slot $L(t)$ to be the placement associated with the previous flow f_{prev}. Otherwise, the algorithm will set $f_{prev} = f_{curr}$ and will continue until f_{curr} will violate the reposition cost constraint.

The following theorem, which is proved in the technical report [19], establishes the properties of SCC:

Theorem 7.4. *Let r be the threshold parameter of SCC. Then the following conditions hold: 1) SCC decreases the cost of the placement in every iteration. 2) The reposition cost of SCC is at most r. 3) If there exists no negative cycle, then SCC returns the optimal placement.*

SCC finds a minimal (in number of edges) cycle, so it can augment many cycles as possible, without violating the reposition cost constraint .

7.5 Reducing the complexity of SCC

The one major problem of SCC is its time complexity. One iteration of SnC, which is used for finding a negative cycle, costs $O(l_{min}|V||E|)$ (See Claim 7.2). In the case of the 8-layer graph G_f, $|V| = |E| = O(smk)$, where s is the total capacity of the system, m is the number of resource types and k is the number of areas. The time complexity of

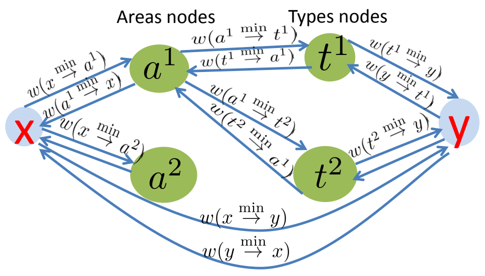

Areas nodes
Types nodes
$w(x \overset{\min}{\to} a^1)$
$w(a^1 \overset{\min}{\to} x)$
$w(x \overset{\min}{\to} a^2)$
$w(a^1 \overset{\min}{\to} t^1)$
$w(t^1 \overset{\min}{\to} a^1)$
$w(a^1 \overset{\min}{\to} t^2)$
$w(t^2 \overset{\min}{\to} a^1)$
$w(t^1 \overset{\min}{\to} y)$
$w(y \overset{\min}{\to} t^1)$
$w(t^2 \overset{\min}{\to} y)$
$w(x \overset{\min}{\to} y)$
$w(y \overset{\min}{\to} x)$
a^1
a^2
t^1
t^2
x
y

Figure 4: The bipartite-like graph G_f^B

one SnC iteration is therefore $O(s^2m^2k^2)$ which practically can be quite large.

To reduce the time complexity we use a reduction of the 8-layer graph G_f^8 to a bipartite graph, which removes unnecessary nodes in the graph. We then imitate the actions of SCC over the bipartite graph. This method is similar to the one used in [18], which imitates the Successive Shortest Path algorithm (an algorithm solving the min-cost flow) over the bipartite graph.

The **bipartite-like graph**, denoted as $G_f^B = (V^B, E_f^B)$, is a sub-graph of the 8-layer residual graph G_f^8 constructed from the 8-layer graph G^8 (i.e $V^B \subseteq V^8$). The graph, depicted in Fig. 4, resembles a bipartite graph, excluding the source x and sink y nodes. The graph contains a layer consists of area nodes a^j, and a layer consists of resource type nodes t^i.

The edge weight between u and v in the bipartite-like graph, which is denoted by $w_f(u \overset{\min}{\to} v)$, equals to the shortest minimal path between u and v in G_f^8 (formal definition given in [18]). Given vertices u, v computing $w_f(u \overset{\min}{\to} v)$ can be done in $O(1)$ (as shown in the technical report [19]).

Imitating the behavior of SCC over the bipartite graph will reduce the time complexity of an SnC iteration to only $O(mk(k+m))$. If we bound the total number of SnC iteration by $c << s$, then the time complexity of SCC is at most $O(cmk(k+m))$. This can be significantly more efficient than running optimal algorithm for the static problem that runs in $O(smk(m+k))$.

7.6 Practical Considerations and a General Algorithm

SCC as designed above aims at bounding the reposition cost incurred in any operation of the algorithm. As such SCC operates regardless of the placement cost deviation, and there might be situations where the application will yield resource reposition whose placement cost deviation is tiny. Such repositions can be avoided using a Hybrid algorithm which combines LA with SCC, presented in the next Section.

8. THE HYBRID ALGORITHM

In this section we provide a general (Hybrid) algorithm which combines the mechanisms of LA and SCC.

The Hybrid algorithm holds four threshold parameters ϵ, r_{min} and r_{max}, where $r_{min} < r_{max}$. At time t LA holds a reference demand set $D_{ref}(t)$ equaling $D(\tau)$ for some $\tau < t$, where τ is the last time where the algorithm modified its placement. It also holds as reference the placement at time $t-1$, $L(t-1)$. Given distributions $D_{ref}(t), D(t)$ and placement $L(t-1)$ the algorithm computes the placement at time slot t, $L(t)$. The algorithm determines whether $D(t)$ and $D_{ref}(t)$ are weakly ϵ-near. If they are - we run SCC over placement $L(t-1)$ (with its associated flow), with the reposition cost threshold set to r_{min}. Otherwise, we run SCC over placement $L(t-1)$ with the reposition cost threshold set to r_{max}.

Note that we can set $r_{min} = 0$, and $r_{max} = \infty$ then which is equivalent for running LA with ϵ. If we set only $\epsilon = 0$ (and r_{max} to any value), then it is equivalent to SCC with r_{max}.

9. PERFORMANCE EVALUATION OF THE DYNAMIC ALGORITHMS

In this section we evaluate the performance of the dynamic algorithms presented in this article; for the sake of providing a scale of reference we compare them with the Proportional Mean placement - A replication strategy proposed in [20]. We simulate a small-size mobile game app (whose audience consists of about at most 9000 users) that uses Amazon EC2 servers. Some of the parameter settings can be found in the appendix.

9.1 Parameter Settings

The mobile app provider places its servers in $k = 3$ regions of Amazon EC2: One in the USA (North Carolina), the other in Europe (Ireland) and the third in Asia (Singapore). The application provider offers two different applications ($m = 2$), each requiring different type of platform from the service provider: One is the a Windows platform, while the other is a Red Hat Enterprise Linux (RHEL) platform. We assume that both a windows and a RHEL server can serve up to $B_{windows} = B_{RHEL} = 500$ users. Due to EC2 limit on on-demand servers, the application provider cannot buy more than 20 servers per region,(see [4]), i.e, we bound the capacity of number of servers to 20 in the USA ($s^1 = 20$), 20 in Europe ($s^2 = 20$) and 20 in Asia ($s^3 = 20$). The provider uses a single instance - the m3.medium VM (see [4]). As explained in Remark 3.2, solving the static resource placement with more than a single instance is NP-hard.

The cost associated with serving a user can be defined as minus the Average Revenue Per User (ARPU) of granting the request (see Remark 3.1). The ARPU is varies between mobile applications as seen in [13]. We assume that the ARPU of the Windows application is 1.5$ (i.e, the cost is $-1.5$$), and the ARPU of RHEL application is 1$, as done in other mobile applications (See [13]). The exact details of the service costs can be found in Table 1 at the appendix.

Finally, we define the resource cost $C_r(p)$ to be the linear function as set in Eq 1. We set the on-demand price of resource type i in area j according to Amazon EC2 price system on the m3.medium VM [4]. In Table 2 in the appendix we define the on-demand costs p_i^j. Note that when using Amazon EC2 servers there is no need to pay licensing costs and area-specific charges i.e,($r_i = h^j = 0$).

We set the total number of requests of resource type i in area j on time t, $D_i^j(t)$, to be a time-dependent Poisson distribution with parameter of $\lambda_i^j(t)$ (as done in [14] and [23]). Our dynamic scheduling uses an hourly based predication, where in every area the average number of requests for Windows and RHEL servers is the same i.e, $\lambda^j_{RHEL}(t) = \lambda^j_{Windows}(t)$. We set the demand parameter to be a periodic function that increases during day time , and decreases during night time (usually, between 4 Pm till 4 Am) as done in many applications (such as [23]). Also, in some web applications (such as [7]) the arrival rate is considered to be unpredictable with a higher variability due to some noise factor. In order to simulate such arrival rates, we add a Additive White Gaussian Noise (AWGN) to the arrival rate formula.

Finally, the arrival rate is

$$\lambda^j_i(t) = 3000 \cdot \sin(\frac{t_j \cdot \pi}{24}) + 300 \cdot \zeta, \qquad (15)$$

where ζ is a white noise, generated by the standard Gaussian (Normal) distribution, and t_j is the local time at area j.

Finally, given a time slot t, we compute the local time in the USA by $t_{USA} = (t \bmod 24)$, the local time in Europe by $t_{Europe} = ((t + 6) \bmod 24)$, and in Asia $t_{Asia} = ((t + 12) \bmod 24)$.

9.2 Performance of the dynamic algorithms over time varying predicted demands

We evaluate the placement cost deviation and the reposition cost with an arrival rate of $\lambda^j_i(t)$ (given in Eq. 15) over a time scale of $t = 0, 1, \ldots, 47$ between the following algorithms: 1) The optimal static placement (BGA). 2) The LA algorithm (Section 5) with threshold $\epsilon = 2000\$$. 3)The SCC (Section 7) with reposition cost $r = 4$. 4) The Hybrid algorithm (Section 8) with thresholds $\epsilon = 2000\$$,$r_{min} = 2$, $r_{max} = 4$. 5) An placement strategy called **Proportional Mean**, as proposed in [20], where the number of type-i servers allocated in every area j is proportional to demand, i.e, there exists a constant $\alpha > 0$ such that $L^j_i(t) = \alpha \cdot E(D^j_i(t))$. For our simulations we set $\alpha = 1.2$. The placement allocated in time $t = 0$ is the optimal static placement respect to demand $D(t) = D(0)$.

In Fig. 5 we depict the **Relative Reposition Cost (RRC)**. The Relative Reposition Cost defined as the reposition cost divided by the number of servers placed. The RCC represents the percentage of servers that were changed. In Fig. 5 we depict relative reposition cost.

We observe the following results: 1) The Hybrid algorithm incurs the lowest reposition cost of at most 10%, and sometimes 0% (i.e, no reposition between successive time slots). The Hybrid has the lowest average reposition cost of the only 5.7%, while the optimal static cost placement and Proportional Mean have respectively 15.8% and 16.8%. That means, the average reposition cost of Hybrid algorithm is 65% lower than the the one the average reposition cost of the optimal placement and Proportional Mean. 2) Proportional Mean and the optimal static cost placement incur a large reposition cost, ranging 10% − 20%. 3) LA incurs highly variable reposition cost: when LA changes the current placement using the optimal static placement algorithm, it incurs a high relative reposition cost (sometimes by 70%). If LA does not changes the placement - the relative reposition cost is 0%. 3) SCC has (almost) a constant reposition cost, which is always around 10%. Its average reposition cost is 9.8%, which is higher than the Hybrid algorithm by a factor of 1.7.

Next we examine whether the fact that our main algorithm (Hybrid) operates under reposition constraints, allows it to achieve close to optimal placements. To this end we depict in Fig. 6 the **relative deviations of a placement cost** for all the algorithms examined; the relative deviation of placement cost is defined as the deviation of placement cost divided by the cost of the static optimal placement. We observe that the cost of every dynamic algorithm placement is higher only by a margin of 1% − 5%. Note that the Hybrid placement algorithm achieves very efficient placements, whose cost deviation is bounded by 1.3%, while obeying strict reposition cost constraints.

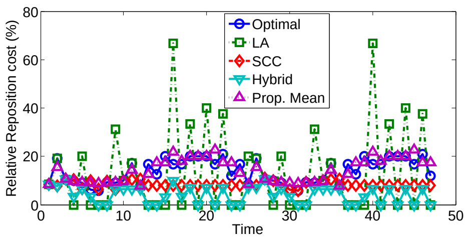


Figure 5: Relative Reposition Cost

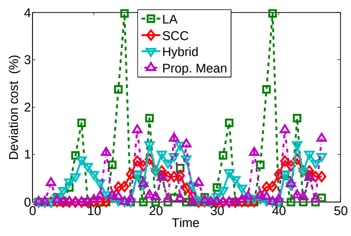


Figure 6: Relative Placement Cost

In Fig. 7 we depict the real cost deviation and observe the following results: 1) The Hybrid algorithm has a cost deviation of at most 200$ per hour (better than Proportional mean and LA), with average daily cost deviation of 2153$ (and 64,610$ per month). 2) LA reposition cost has high variability, and contains large "spikes", which are dependent whether LA changes the placement in time t or not (which is done according to the demand distance between $D(t-1)$ and $D(t)$). If LA changes the placement - then an optimal static placement algorithm was applied, and the cost deviation is 0. However, if the placement was not changed - it can incurs a relatively high maximal cost deviation of 700$ per hour. LA has a high daily cost of 2419$ (and 72,569$ per month) 3) SCC has a minimal cost deviation between the different algorithms across time, and with a low variability. The placement cost of SCC is closer to the optimal static value than the other non-optimal algorithm. Its maximal placement cost deviation is at most 130$ per hour (the lowest among all algorithms), and it has the lowest daily cost deviation of 1471$ per day (and 44,147 per month). 4) The cost deviation of proportional mean replication has high variability. It has maximal cost deviation of 400$ per hour, and an average cost deviation of 2087.95$ per day (62,638$ per month).

9.3 Performance Evaluation - conclusions

We observed that the Hybrid algorithm has the lowest maximal reposition cost and the lowest average reposition cost. The average reposition cost is lower by 65% than the

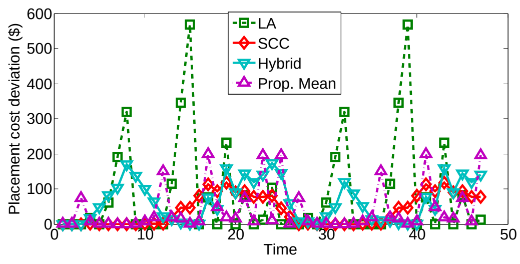


Figure 7: Placement cost deviation

optimal static placement, as well as Proportional Mean. The relative deviation cost of the Hybrid algorithm is at most 1.3%, better than Proportional Mean or LA.

Although SCC has better reposition cost than the Hybrid algorithm, it has a higher reposition cost. LA in every time slot has a high reposition cost (when it changes the placement) or a high placement cost deviation (when it not changes the placement). Proportional mean, as well as optimal static placement, have a high reposition cost.

10. CONCLUDING REMARKS

We dealt with the problem of dynamic resource placement, accounting for dynamically changing stochastic demands with arbitrary distributions as well as for a very rich cost model. We showed that dynamic demand fluctuations may inflict huge reposition costs and therefor a dynamic placement algorithm must account for them. To address this problem we proposed an algorithm that avoids resource reposition when the cost benefits are tiny and conducts a bounded reposition when the cost benefits are substantial. The algorithms proposed are of practical complexity despite the high dimensionality of the problem.

11. REFERENCES

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[2] Amazon EC2 home page. http://aws.amazon.com/ec2, 2013.

[3] Amazon FAQs page. `http://aws.amazon.com/ec2/faqs/#How_many_instances_can_I_run_in_Amazon_EC2`, 2014.

[4] Amazon EC2 pricing page. `http://aws.amazon.com/ec2/pricing/`, 2014.

[5] D. Applegate, A. Archer, V. Gopalakrishnan, S. Lee, and K. K. Ramakrishnan. Optimal content placement for a large-scale vod system. In *ACM CoNEXT*, Philadelphia, USA, Dec 2010.

[6] R. G. Busacker and P. J. Gowen. A procedure for determining a family of minimal cost network flow patterns. Oro technical report 15, Operational Research Office, Johns Hopkins University, Baltimore, MD, September 1961.

[7] V. Cardellini, E. Casalicchio, F. L. Presti, and L. Silvestri. Sla-aware resource management for application service providers in the cloud. In *NCCA*, pages 20–27, 2011.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.

[9] Z. Drezner and H. W. Hamacher. *Facility Location: Applications and Theory*. Springer, 2002.

[10] Google cloud pricing page. https://cloud.google.com/compute/pricing, 2014.

[11] M. Leconte, M. Lelarge, and L. Massoulié. Bipartite graph structures for efficient balancing of heterogeneous loads. In *ACM SIGMETRICS/PERFORMANCE*, pages 41–52, 2012.

[12] Microsoft Azure home page. http://www.windowsazure.com, 2013.

[13] Think Gaming: top paid games. `http://thinkgaming.com/app-sales-data/top-paid-games/`, 2014.

[14] X. Nan, Y. He, and L. Guan. Optimal allocation of virtual machines for cloud-based multimedia applications. In *MMSP*, pages 175–180. IEEE, 2012.

[15] F. L. Presti, C. Petrioli, and C. Vicari. Distributed dynamic replica placement and request redirection in content delivery networks. In *MASCOTS*, pages 366–373, 2007.

[16] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the placement of web server replicas. In *IEEE INFOCOM*, Anchorage, AK, USA, April 2001.

[17] Y. Rochman, H. Levy, and E. Brosh. Max percentile replication for optimal performance in multi-regional p2p vod systems. In *Proceeding of the 9th International Conference on Quantitative Evaluation of SysTems (QEST) 2012*, London, UK, September 2012.

[18] Y. Rochman, H. Levy, and E. Brosh. Resource placement and assignment in distributed network topologies. In *IEEE INFOCOM*, Turin, Italy, April 2013.

[19] Dynamic placement of resources in cloud computing and network applications - technical report. `https://drive.google.com/folderview?id=0B5v_NLuOQ1TAOVBQd3pTXzFXM1U&usp=sharing`, 2014.

[20] S. Tewari and L. Kleinrock. Proportional replication in peer-to-peer networks. In *IEEE INFOCOM*, Barcelona, Spain, April 2006.

[21] Y. Wu, C. Wu, B. Li, L. Zhang, Z. Li, and F. C. Lau. Scaling Social Media Applications into Geo-Distributed Clouds. In *IEEE INFOCOM*, Orlando, Florida, USA, March 2012.

[22] H. Xu and B. Li. Joint Request Mapping and Response Routing for Geo-distributed Cloud Services,. In *IEEE INFOCOM*, Turin, Italy, April 2013.

[23] B. Zhang, G. Kreitz, M. Isaksson, J. Ubillos, G. Urdaneta, J. A. Pouwelse, and D. H. J. Epema. Understanding user behavior in spotify. In *INFOCOM*, pages 220–224. IEEE, 2013.

[24] Q. Zhang, Q. Zhu, M. F. Zhani, R. Boutaba, and J. L. Hellerstein. Dynamic service placement in geographically distributed clouds. *IEEE Journal on Selected Areas in Communications*, 99:pp, 2013.

[25] Y. P. Zhou, T. Z. J. Fu, and D. M. Chiu. Statistical modeling and analysis of p2p replication to support vod service. In *IEEE INFOCOM*, Orlando, FL , USA, July 2011.

APPENDIX

The assignment algorithm

In order to present the assignment algorithm and the correctness of Theorem .1 we define the following definitions:

DEFINITION 5. *Let M be a matching between a placement $L = \{L_i^j\}$ and a demand realization d_i^j. Let $g_i^{loc}(M)$, $g_i^{rem}(M)$ respectively denote the number of type-i requests granted locally and remotely under the matching algorithm M. The number of type-i requests granted* ***globally*** *(i.e, remotely or locally) is denoted by $g_i^{glo}(M) \doteq g_i^{rem}(M) + g_i^{loc}(M)$. We define the* ***local differential revenue constant*** *as the revenue gained from granting a type-i request locally compared to granting it remotely, i.e $R_i^{loc} \doteq C_i^{rem} - C_i^{loc} \geq 0$. The* ***global differential revenue constant*** *is the revenue gain from granting a type-i request remotely compared to not satisfying the request, i.e $R_i^{glo} \doteq C_i^{unsat} - C_i^{rem} \geq 0$. The* matching revenue *of M, denoted by $R(L, d_i^j, M)$, is defined as $R(L, d_i^j, M) = \sum_{i=1}^m (R_i^{glo} \cdot g_i^{glo}(M) + R_i^{loc} \cdot g_i^{loc}(M))$. The* ***maximal revenue value*** *$R(L, d_i^j)$ is the maximum value of the matching revenues $R(L, d_i^j, M)$ over all possible matchings M. We define the* ***service revenue*** *as the expected maximal revenue value, over all demand realizations, i.e $E|_D(R(L)) = \sum R(L, d_i^j) \cdot \Pr(D_i^j = d_i^j)$.*

We first prove that one can convert the service revenue to a service cost by the following claim:

CLAIM .1. *For every placement L we have*

$$E|_D(R(L)) + E|_D(C(L)) = \sum_{i=1}^m C_i^{unsat} D_i. \qquad (16)$$

PROOF PROOF OF CLAIM .1. Consider $\{d_i^j\}$ a demand realization, and M be an assignment between the demand and a placement L. We define the *matching cost* of M simply as

$$C(L, d_i^j, M) = \sum_{i=1}^m C_i^{loc} \cdot g_i^{loc}(M) + C_i^{rem} \cdot g_i^{rem}(M) + C_i^{unsat} \cdot g_i^{unsat}(M) \qquad (17)$$

where $g_i^{loc}(M)$, $g_i^{rem}(M)$ and $g_i^{unsat}(M)$ are respectively the corresponding number of type-i requests granted by M from a local area, granted from a remote area, and unsatisfied ones. Similarly to the revenue case, we define the minimal service cost between placement L and realization $\{d_i^j\}$ as the optimal assignment minimizing the matching cost i.e $C(L, d_i^j) = \arg\min_M C(L, d_i^j, M)$.

We will prove that the *total sum*, which simply as the sum of the cost matching and the revenue matching $R(L, d_i^j, M) + C(L, d_i^j, M)$ is constant. Consider the marginal value of a type-i request req_i in the total sum according to the following cases:

1. If req_i is granted locally, then it is also granted globally. Thus its marginal value of the matching revenue is $R_i^{loc} + R_i^{glo} = C_i^{unsat} - C_i^{loc}$, and the marginal value in the total sum equals to C_i^{unsat}.
2. If req_i is granted remotely, then it is granted globally but not locally. Thus its marginal value of the matching revenue $R_i^{glo} = C_i^{unsat} - C_i^{rem}$ and in the total sum C_i^{unsat}.
3. If req_i is an unsatisfied request, then it nor granted globally or locally. Thus, its marginal value of the matching revenue is 0 and the marginal revenue in the total sum is C_i^{unsat}.

Thus every type-i request has a marginal value of C_i^{unsat} to the total sum. This implies that total sum equals to $C(L, d_i^j, M) + R(L, d_i^j, M) = \sum_{i=1}^m C_i^{unsat} D_i.$, where D_i is the number of type-i requests. Note that $\sum_{i=1}^m C_i^{unsat} D_i$ is independent of the matching M. Thus, a matching M_0 that maximizes matching revenue (i.e $R(L, d_i^j, M_0) = R(L, d_i^j)$) must minimize the matching cost (i.e $C(L, d_i^j, M_0) = C(L, d_i^j)$. Therefore,

$$C(L, d_i^j) + R(L, d_i^j) = C(L, d_i^j, M_0) + R(L, d_i^j, M_0) = \sum_{i=1}^m C_i^{unsat} D_i. \qquad (18)$$

Taking expectation over the demand set D yields the theorem statement. □

Finally, to present the assignment algorithm, we call a type-i resource *partial-loaded* if the number of requests assigned to resource is strictly less than its capability B_i. If the number of requests assigned to resource equals to its capability, then the resource is called *fully-loaded.*

In Algorithm 1 we derive a specific matching algorithm tailored for maximizing the revenue matching.

Algorithm 1 The assignment algorithm

```
Require: An placement $L = \{L_i^j\}$, the demand realization $d_i^j$, and resource capacities $B_i$.
1: for all movie $i$ do
2:   for all region $j$ do
3:     Assign $\min(B_i \cdot L_i^j, d_i^j)$ type-$i$ requests from area $j$ to $\min(L_i^j, \lceil \frac{d_i^j}{B_i} \rceil)$ type-$i$ resources in area-$j$.
4:   end for
5:   while There is an unmatched type-$i$ request and a type-$i$ partial-loaded resource do
6:     Match the request with the resource.
7:   end while
8: end for
```

By the following claim we prove the assignment matching optimality:

CLAIM .2. *Given placement $L = \{L_i^j\}$ and demand realization d_i^j, then the following claims holds:*

1. *For every matching algorithm we have $g_i^{loc}(M) \leq \sum_{j=1}^k \min(B_i \cdot L_i^j, d_i^j)$ and $g_i^{glo}(M) \leq \min(L_i \cdot B_i, d_i)$.*
2. *The assignment algorithm yields a cost as in Eq (??). Moreover, the assignment algorithm maximizes the matching revenue $R(L, d_i^j, M)$.*

PROOF. **Proof of part 1):** Let M be some matching algorithm, and denote $g_{i,j}^{loc}(M)$ as the number of type-i granted locally in region j. Then the number granted locally requests is less than the number of requests, i.e $g_{i,j}^{loc}(M) \leq d_i^j$. Also, the number of requests granted locally is less than the number of resources multiple its capability $g_{i,j}^{glo}(M) \leq L_i^j \cdot B_i$. Therefore, we have $g_{i,j}^{loc}(M) \leq \min(L_i^j \cdot B_i, d_i^j)$, and $g_i^{loc}(M) \leq \sum_{j=1}^k \min(B_i \cdot L_i^j, d_i^j)$ as required. $g_i^{glo}(M) \leq \min(L_i \cdot B_i, d_i)$ is prove similarly.

Proof of part 2): We denote by M_{opt} the optimal assignment algorithm. In the end of Step 2 the number of type-i requests granted locally is $g_i^{loc}(M_{opt}) = \sum_{j=1}^{k} \min(B_i \cdot L_i^j, d_i^j)$. We will prove $g_i^{glo}(M_{opt}) \geq \min(L_i \cdot B_i, d_i)$. Assume by contradiction otherwise, i.e $g_i^{glo}(M_{opt}) < \min(L_i \cdot B_i, d_i)$. Since $g_i^{glo}(M_{opt}) < d_i$ and $g_i^{glo}(M_{opt}) < L_i \cdot B_i$, there exist a unmatched type-i request and a type-i partial-loaded resource. But in Step 5 we match every unmatched type-i request to matched to a type-i partial-loaded resource - a contradiction. By using part 1) of the claim, we yield that the number of type-i requests granted globally is $g_i^{glo}(M_{opt}) = \min(L_i \cdot B_i, d_i)$.

The algorithm optimality is driven immediately by the first part of the claim.

If we use Claim .1, and the definition of the service revenue, we yield Eq. **??**.

□

Proof of claims

Proof Proof of Remark 3.2. To prove the remark, we show and prove even a stronger statement: determining whether there is a placement with non-zero cost is NP-hard, if the servers are heterogenous. Thus, finding a c-approximation is also NP-hard.

For sake of simplicity, assume we have only a single region $k = 1$. The system contains s servers (resources) indexed by $1, 2, \ldots, s$. We denote the resource capacity of placing resource type i in server-l by B_i^l. We denote a placement by $\bar(x) = \{x_i^l\}$, where x_i^l is an indicator such that $x_i^l = 1$ if resource type i is placed on server l; Otherwise, $x_i^l = 0$. Finally, we set the resource cost $C_r(\bar(x))$ to be the linear function

$$C_r(\bar(x)) = \frac{1}{\kappa} \cdot \sum_{i=1}^{m} \sum_{l=1}^{s} x_i^l, \qquad (19)$$

where $\kappa > 0$ is some parameter. Note that $\sum_{i=1}^{m} \sum_{l=1}^{s} x_i^l$ is the number of servers operated. We set the number of requests for type-i resource to be 2, i.e, demand distribution D_i is a constant distribution with $\Pr(D_i = 1) = 2$.

Under these settings, the number of local granted requests for type-i resources is $g_i^{loc} = \min(\sum_{l=1}^{s} x_i^l \cdot B_i^l, 2)$. We set the service costs to be $C_i^{loc} = -\frac{1}{m}$, and $C_i^{rem} = C_i^{ser} = 0$. Thus, the service cost of allocation $\bar(x)$ is

$$E|_D(\bar{x})) = -\frac{1}{s} \cdot \sum_{i=1}^{l} \min(\sum_{l=1}^{s} x_i^l \cdot B_i^l, 2), \qquad (20)$$

and the total placement cost is (according to Eq. 2),

$$C_p(\bar{x})) = \frac{1}{\kappa} \cdot \sum_{i=1}^{m} \sum_{l=1}^{s} x_i^l - \frac{1}{m} \cdot \sum_{i=1}^{m} \min(\sum_{l=1}^{s} x_i^l \cdot B_i^l, 2). \quad (21)$$

Finally, the static placement problem is finding the best placement $\bar(x) = \{x_i^l\}$ that maximizes $C_p(\bar{x}))$ under the assumption that $x_i^l \in \{0, 1\}$, and $\sum_{i=1}^{l} x_i^l \leq 1$.

Given a universal set of elements $U = \{1, 2, \ldots, m'\}$, and a collection of s' sets $\{S_1, S_2, \ldots S_{s'}\}$, where every set S_i is a subset of U ($S_i \subseteq U$), we define a **cover** $C = \{S_{i_1}, S_{i_2}, \ldots, S_{i_k}\}$ such that the union of all sets in C covers U ($U \subseteq \bigcup_{S \in C} S$). The input for the Set Cover Problem (SCP) is a triple (U, S, k), where k is some integer; the question is is whether there is a set-cover C such that $|C| \leq k$. The problem was proved to be NP-hard[].

We will prove that if there is a polynomial algorithm that determines whatever the static placement problem has zero-cost or not, then there is a polynomial algorithm that solves SCP, and thus P=NP. Given an instance (U, S, k) we set the number of resource types to be the size of the universal set ($m = m'$) , and number of servers as the number of sets in the collection ($s = s'$). The resource capacity of placing resource type i in server-l, B_i^l, is defined as follows:

$$B_i^l = \begin{cases} 2 & : i \in S_l \\ 1 & : i \notin S_l \end{cases}$$

We choose to be a number between $\frac{k}{2}$ and $\frac{k+1}{2}$ (for instance, one can choose $\kappa = \frac{\frac{k}{2} + \frac{k+1}{2}}{2}$).

Given these setting, one needs to prove that:

1. If there is a set-cover C such that $|C| \leq k$, then the optimal static placement solution has non-zero cost.
2. If there is no set-cover C such that $|C| \leq k$, then the optimal static placement solution has zero cost.

Proof of 1.: Let C be a set cover s.t, $|C| \leq k$. We can set a placement, where $x_i^l = 1$ iff $S_l \in C$ and $i \in S_l$, and otherwise $x_i^l = 0$. In this case, $\sum_{i=1}^{m} \sum_{l=1}^{s} x_i^l$ equals to size of the set cover C, i.e, $\sum_{i=1}^{m} \sum_{l=1}^{s} x_i^l = |C| \leq k$. Also, since C is a cover, there exists for every i a set $S_{l(i)} \in C$ such that $i \in S_l$. Thus, for every i $\sum_{l=1}^{s} x_i^l \cdot B_i^l \geq x_i^{l(i)} \cdot B_i^{l(i)} \geq 1 \cdot 2 = 2$. Thus, $\frac{1}{m} \cdot \sum_{i=1}^{m} \min(\sum_{l=1}^{s} x_i^l \cdot B_i^l, 2) = 2$ by Eq.21 we have

$$C_p(\bar{x})) \leq \frac{k}{\kappa} - 2. \qquad (22)$$

However, by definition κ, we have $\frac{k}{2} < \kappa$, therefore $C_p(\bar{x})) < 0$. The cost of the optimal static placement problem must be non-zero, as there exists a placement with negative cost.

Proof of 2.:

□

Proof Proof of Claim 4.2. We set $D(t)$ to the instance where the number of requests is zero, i.e. $\Pr(D(t)_i^j = 0) = 1$ for all resource type i and area j. Placing in every region only type-2 resources is optimal (i.e. $L_{opt}(D(t-1))_2^j = s^j$ for every j).

The demand set in the next time slot $D = D(t+1)$ is defined as follows: 1) The probability that the number of requests for resource of type-1 in area 1 is more than n with probability of $\frac{\epsilon}{2^{n+1}}$ (i.e $\Pr(D_1^1(t) \geq n) = \frac{\epsilon}{2^{n+1}}$). 2) The number of requests for type-i resources in area j, such that $(i, j) \neq (1, 1)$ is 0 (i.e $\Pr(D(t)_i^j = 0) = 1$). An optimal placement for $D(t+1)$ is the placement allocating to every region a type-1 resource. Note that $D(t)$ and $D(t+1)$ are strongly ϵ-near, and the distance between $L_{opt}(D(t-1))$ $L_{opt}(D(t))$ is s, as required. □

Proof Proof of theorem 5.1. To prove the theorem, we will use the out-of-kilter algorithm (presented in [1]) that solves the min-cost flow problem. Suppose the demand set $D = D(\tau) = D_{ref}(t)$ was updated to a new demand set $D(t)$. Then the weight function $w()$ which is correspond 8-layer graph G_f^8 of the optimal flow $f = f_{opt}$ and node potentials π, was updated a new weight function w' (see Fig. 2). We use the out-of-kilter algorithm that given the (old) node potentials π and (old) optimal flow f finds a new min-cost optimal flow f' with new node potentials π', with respect to the new demand $D' = D(t)$. We will prove that flow weight is changed by at most ϵ (i.e. $|w'(f) - w'(f')| < \epsilon$), and therefore, by Lemma 6.4, the theorem is be proved.

REMARK .3. *Note that we extend the weight function to be defined also on the reverse edges of G, where $w(u,v) = -w(v,u)$. Using this definition we yield that $w_f(e) = w(e)$ or $w_f(e) = \inf$ for every e in G_f*

Let $e = (u,v)$ be an out-of-kilter edge in the 8-layer graph G^8_f with respect to the initial node potentials reduce weight w'^{π}, i,e., $0 > w'^{\pi}_f(e)$. On the other hand the reduce weight with respect to the previous weight, w^{π}_f, is non-negative and therefore $w^{\pi}_f(e) \geq 0$. By the definition of reduce weights formula of w^{π} and w'^{π} we obtain the following formula

$$w(u,v) - w'(u,v) \geq -w'^{\pi}_f(u,v) = -w'^{\pi}(u,v) > 0, \quad (23)$$

and particularly we have $w(u,v) \neq w'(u,v)$.

For the sake of the proof we call respectively to the edges entering nodes (a^j, n), (a^j, t_i, n), (t_i, n) *area edges*, *area+type edges* and *type edges*. The weight of other edges is zero weight (i.e. $w(e) = w'(e) = 0$), according to Fig. 2. The weight of area edges equals to $w(e) = w'(e) = \Delta\zeta^j(n) = \zeta^j(n) - \zeta^j(n-1)$ (according to Claim 6.2) which is not dependent on the demand distribution D. Thus, out-of-kilter edges must be area+type edges and the type edges.

Denote cyc_i the cycle found in i^{th} of the out-of-kilter Algorithm. Then augmenting the flow along cyc_i increases the flow weight $w'(f)$ by $w'(cyc)$, where $w'(cyc) = \sum_{e \in cyc} w'(e)$. Thus, if the out-of-kilter algorithm runs over t iterations, then

$$w'(f') - w'(f) = \sum_{i=1}^{t} w'(cyc_i) \quad (24)$$

Denote π_i the node potentials in the i^{th} iteration. Then, the weight of every cycle C equals to the cycle weight with respect to node potentials π_i, i.e $w'(C) = w'^{\pi_i}(C) = \sum_{e \in C} w'^{\pi_i}(e)$. Let A_i denote the set of out-of-kilter edges after the i^{th} iteration. Then according to [1] every in-kilter edge $e \in cyc_i \bigcap (A_i)^c$ has reduced weight of $w'^{\pi_i}(e) = 0$. Thus, if we denote by $B_i = cyc_i \bigcap A_i$ the out-of-kilter edges in the i^{th} cycle, then the weight of every cycle cyc_i equals to the sum of out-of-kilter edges i.e. $w'(cyc_i) = \sum_{e \in B_i} w'^{\pi_i}(e)$. Thus, we yield that $w'(f') - w'(f) = \sum_{i=1}^{t} \sum_{e \in B_i} w'^{\pi_i}(e)$.

Let us denote by $B = \bigcup_{i=1}^{t} B_i$ the total set of out-of-kilter edges. The area+type and type edges have capacity of 1. Thus, all out-of-kilter edges have a kilter number, which is the residual capacity $c_f(e)$, of 1. If $e \in B_i$ is an out-of-kilter edge then after augmenting through cyc_i its residual capacity decreases and it becomes an in-kilter edge. Thus, the sets B_i for are disjoint in pairs i.e, $B_i \cap B_j = \emptyset$ for $i \neq j$, and therefore

$$w'(f') - w'(f) = \sum_{e \in B} w'^{\pi_i}(e). \quad (25)$$

Let $(u,v) = e \in B_i$ be an out-of-kilter edge. Then e is an out-of-kilter edge in the $i-1$-iteration with a negative reduce weight i.e. $w'^{\pi_{i-1}}(e) < 0$. The node potentials of i iteration equals to $\pi_i = \pi_{i-1} - d$, and therefore reduce weight of edge (u,v) is $w'^{\pi_i}(e) = w'^{\pi_{i-1}}(e) + d(u) - d(v)$.

But the weight in the i iteration of edge e equals to $\max(0, w'^{\pi_{i-1}}(e))$ 0, and the shortest path to v is not longer than the shortest path to u, i.e. $d(u) \geq d(v)$. Therefore, we imply that $w'^{\pi_i}(e) \geq w'^{\pi_{i-1}}(e)$ for every out-of-kilter edge e, and moreover, one can imply by induction that $w'^{\pi_i}(e) \geq w'^{\pi_0}(e)$, where $\pi_0 = \pi$ is the node potentials in the initial iteration. Since $e \in B_i$ is an out-of-kilter in the initial iteration ($e \in A_0$) by Eq (30) we obtain $w'^{\pi_i}(e) \geq w'^{\pi_0}(e) \geq -|w'(e) - w(e)|$, and by Eq (32) we imply that.

$$w'(f') - w'(f) \geq -\sum_{e \in B} |w'(e) - w(e)|. \quad (26)$$

since f' is the min-cost optimal flow for weight w', then $w'(f') \leq w'(f)$, therefore

$$|w'(f') - w'(f)| = w'(f) - w'(f') \leq \sum_{e \in B} |w'(e) - w(e)|. \quad (27)$$

According to Claim 6.2, if $e \in B$ is an area+type that connects vertex (a^j, t_i, n), then

$$|w(e) - w'(e)| = |R^{loc}_i[\Pr(D^j_i \geq n \cdot B_i) - \Pr(D'^j_i \geq n \cdot B_i)]|. \quad (28)$$

Similarly, if $e \in B$ is a type edge that connects vertex (t_i, n) then

$$|w(e) - w'(e)| = |R^{glo}_i[\Pr(D_i \geq n \cdot B_i) - \Pr(D'_i \geq n \cdot B_i)]| \quad (29)$$

Then combining Eq (34), (35), (33) with the definitions of the demand-distance (Eq (8)) and weakly ϵ-near yields that

$$|w'(f) - w'(f')| \leq d(D, D') < \epsilon$$

As required.

□

PROOF PROOF OF CLAIM 6.1. We assume in Section 3 that the resource cost $C_r(L)$ is semi-separable and by Eq (??) the service revenue $E(R(L))|_D$ is a semi-separable function. Thus, subtracting the resource cost with service revenue, which equals to the loss function (Eq (10)), is also a semi-separable function.

We assume in Section 3 that total placement cost $C_p(L)$ is convex. Also, by claim .1 loss function equals to the sum of the placement cost and a constant $c = \sum_i E(D_i) C^{unsat}_i$, i.e $loss_p(L) = C_p(L) + c$. Thus, the loss function is also convex. □

PROOF PROOF OF CLAIM 6.2. It is sufficient to prove that for every constant C non-negative integer valued random variable X we have $E(\min(X, C)) - E(\min(X, C-1)) = \Pr(X \geq C)$. A proof of it can be found in [17] Claim 6.4. □

PROOF PROOF OF LEMMA 6.4. We will prove that for every placement $P = \{P^j_i\}$ its loss $loss_p(P)$ is not larger than the optimal flow weight f_{opt}, which equals to the loss of its correspond placement L.

Let P be a placement. We define a flow $f(P)$ as follows:

- The disjoint paths x-(a^j, n)-a^j, (a^j, t_i)-(a^j, t_i, n)-t_i, t_i-(t_i, n)-y are assigned respectively with one unit of flow if $P^j \geq n$, $P^j_i \geq n$ and $P_i \geq n$; Otherwise, the correspond path is assigned with zero unit of flow.
- The flow between (a^j, t_i), and (a^j, t_i) is P^j_i

- The flow between x and y is $s - \sum_{i=1}^{m}\sum_{j=1}^{k} P_i^j$

One can verified that 1) f preserves the capacity constraint and conservation of flows 2) the flow of $f(P)$ equals to $|f(P)| = s$ and 3) the weight of $f(P)$ equals to loss cost of P, i.e $w(f(P)) = loss_p(P)$. Since f_{opt} is a min-cost flow of flow value s, then $loss_p(P) = w(f(P)) \le w(f_{opt})$ as required.

As the marginal-differential functions $\Delta\gamma$ are monotonically increasing, WLOG min-cost flow f_{opt} assigns one unit of flow respectively to x-(a^j, n)-a^j, (a^j, t_i)-(a^j, t_i, n)-(t_i),(t_i)-(t_i, n)-y if $L^j \ge n$, $L_i^j \ge n$ and $L_i \ge n$. Therefore the weight of f_{opt} equals to the loss of L. □

Proof Proof of Theorem **??**. Let $p_{opt} = (x, v_1, v_2, \ldots, v_s, y)$ be a shortest path in G_f^8. We denote by $u \to v$ a monotone path that begins in a vertex u and ends in vertex v. We will proved that there exist area nodes $a(1), a(2), \ldots, a(n)$ and a type nodes $t(1), t(2), \ldots, t(n)$ such that $p_{opt} = x \to a(1) \to t(1) \to a(2) \to \ldots \to t(n) \to y$. WLOG, the optimal path p_{opt} is not the edge (x, y).

By Theorem **??** all edges in G_f^8 have non-negative reduced weights. Thus, the graph does not contain negative cycles (a proof can be seen in [1]). WLOG the optimal path p_{opt} does not contain cycles, otherwise the non-negative cycles can be omitted from the path.

The first node in the optimal path v_1 must be a layer-2 node $(a(1), num)$ in G_f^8. The second node, v_2 must be an area node $a(1)$; Otherwise, v_2 will be the source node x and the path p_{opt} will contains a cycle. Thus, the path p_{opt} begins with a forward path between the source node x and an area node $a(1)$ (i.e $v_1 = (a(1), num)$ and $v_2 = a(1)$). In the same way one can show that p_{opt} ends with a forward path between a type node $t(n)$ and the sink node y (i.e. end with the path $t(n)$-$(t(n), num)$-y. Similarly, one can show that by induction that the path between $a(1)$ and $t(n)$ alternately passing between forward paths between area nodes $a(i)$ and type nodes $t(i)$ and backward path between type node $t(i)$ and area node $a(i+1)$. Thus the optimal path is composed from monotone paths.

WLOG, every monotone path in p_{opt} is a shortest monotone path i.e

$$p_{opt} = x \overset{\min}{\to} a^{j_1} \overset{\min}{\to} t^{i_1} \overset{\min}{\to} a^{j_2} \ldots \overset{\min}{\to} a^{j_e} \overset{\min}{\to} t^{i_e} \overset{\min}{\to} y,$$

Therefore, the weight of optimal path in G_f^8, p_{opt}, equals to the weight of a path p^B in G_f^B ($w_f(p_{opt}) = w_f(p_{opt}^B)$). Also, every path p^B in G^B has an equivalent path p in G_f^8 with the same weight (i.e $w_f(p) = w_f(p^B)$). Thus, the weight of p_{opt} equals to the weight of the shortest path in G_f^B as required. □

Proof Proof of Theorem **??**. This simply follows from Lemmas .5, .4 and Theorem **??**. □

Proof Proof of Theorem **??**. This simply follows from Lemmas .5, .4 and Theorem **??**. □

Proof Proof of Corollary **??**. This can be prove by the fact that the flow weight in the ith iteration equals to $w(f_i) = w_{f_{i-1}}(p_{i-1}) + w(f_{i-1})$. □

Proof Proof of Claim 4.2. We set $D(t)$ to the instance where the number of requests is zero, i.e. $\Pr(D(t)_i^j = 0) = 1$ for all resource type i and area j. Placing in every region only type-2 resources is optimal (i.e. $L_{opt}(D(t-1))_2^j = s^j$ for every j).

The demand set in the next time slot $D = D(t+1)$ is defined as follows: 1) The probability that the number of requests for resource of type-1 in area 1 is more than n with probability of $\frac{\epsilon}{2^{n+1}}$ (i.e $\Pr(D_1^1(t) \ge n) = \frac{\epsilon}{2^{n+1}}$). 2) The number of requests for type-i resources in area j, such that $(i, j) \ne (1, 1)$ is 0 (i.e $\Pr(D(t)_i^j = 0) = 1$). An optimal placement for $D(t+1)$ is the placement allocating to every region a type-1 resource. Note that $D(t)$ and $D(t+1)$ are strongly ϵ-near, and the distance between $L_{opt}(D(t-1))$ $L_{opt}(D(t))$ is s, as required. □

Proof Proof of theorem 5.1. To prove the theorem, we will use the out-of-kilter algorithm (presented in Section 10) that solves the min-cost flow problem. Suppose the demand set $D = D(\tau) = D_{ref}(t)$ was updated to a new demand set $D(t)$. Then the weight function $w()$ which is correspond 8-layer graph G_f^8 of the optimal flow $f = f_{opt}$ and node potentials π, was updated a new weight function w' (see Fig. 2). The reduce weights condition correspond to the original optimal flow f (See Section **??**) does not takes place as the weight function changed. Thus, we use the out-of-kilter algorithm that given the (old) node potentials π and (old) optimal flow f finds a new min-cost optimal flow f' with new node potentials π', respect to the new demand $D' = D(t)$. We will prove that flow weight is changed by at most ϵ (i.e. $|w'(f) - w'(f')| < \epsilon$), and therefore, by Lemma 6.4, the theorem is be proved.

Let $e = (u, v)$ be an out-of-kilter edge in the 8-layer graph G_f^8 respect to the initial node potentials reduce weight w'^{π}, i.e., $0 > w'^{\pi}_f(e)$. On the other hand the reduce weight respect to the previous weight, w_f^{π}, is non-negative and therefore $w_f^{\pi}(e) \ge 0$. By the definition of reduce weights formula of w^{π} and w'^{π} we obtain the following formula

$$w(u, v) - w'(u, v) \ge -w'^{\pi}_f(u, v) = -w'^{\pi}(u, v) > 0, \quad (30)$$

and particularly we have $w(u, v) \ne w'(u, v)$.

For the sake of the proof we call respectively to the edges entering nodes (a^j, n), (a^j, t_i, n), (t_i, n) *area edges*, *area+type edges* and *type edges*. The weight of other edges is zero weight (i.e. $w(e) = w'(e) = 0$), according to Fig. 2. The weight of area edges equals to $w(e) = w'(e) = \Delta\zeta^j(n) = \zeta^j(n) - \zeta^j(n-1)$ (according to Claim 6.2) which is not dependent on the demand distribution D. Thus, out-of-kilter edges must be area+type edges and the type edges.

Denote cyc_i the cycle found in ith iteration in State 7 of the out-of-kilter Algorithm. Then augmenting the flow along cyc_i increases (State 9) the flow weight $w'(f)$ by $-w'(cyc)$, where $w'(cyc) = \sum_{e \in cyc} w'(e)$. Thus, if the out-of-kilter algorithm runs over t iterations, then

$$w'(f) - w'(f') = -\sum_{i=1}^{t} w'(cyc_i) \quad (31)$$

Denote π_i the node potentials in the ith iteration. Then, the weight of every cycle C equals to the cycle weight respect to node potentials π_i, i.e $w'(C) = w'^{\pi_i}(C) = \sum_{e \in C} w'^{\pi_i}(e)$. Let A_i denote the set of out-of-kilter edges after the ith iteration. Then according to Lemma .7 every in-kilter edge $e \in cyc_i \bigcap (A_i)^c$ has reduced weight of $w'^{\pi_i}(e) = 0$. Thus, if we denote by $B_i = cyc_i \bigcap A_i$ the out-of-kilter edges in the ith cycle, then the weight of every cycle cyc_i equals to the sum of out-of-kilter edges i.e. $w'(cyc_i) = \sum_{e \in B_i} w'^{\pi_i}(e)$.

Thus, if we denote by $B = \bigcup i = 1^t B_i$ the out-of-kilter edges then we yield that $w'(f) - w'(f') = \sum_{e \in B} w'^{\pi_i}(e)$.

The area+type and type edges have capacity of 1. Thus, all out-of-kilter edges have a kilter number, which is the residual capacity $c_f(e)$, of 1. If $e \in B_i$ is an out-of-kilter edge then after augmenting through cyc_i its residual capacity decreases and it becomes an in-kilter edge. Thus, the sets B_i for are disjoint in pairs i.e, $B_i \cap B_j = \emptyset$ for $i \neq j$, and therefore

$$w'(f) - w'(f') = -\sum_{i=1}^{t} \sum_{e \in B_i} w'^{\pi_i}(e). \tag{32}$$

Let $(u, v) = e \in B_i$ be an out-of-kilter edge. Then e is an out-of-kilter edge in the $i-1$-iteration with a negative reduce weight i.e. $w'^{\pi_{i-1}}(e) < 0$. The node potentials of i iteration equals to $\pi_i = \pi_{i-1} - d$, and therefore reduce weight of edge (u, v) is $w'^{\pi_i}(e) = w'^{\pi_{i-1}}(e) + d(u) - d(v)$.

But the weight in the i iteration of edge e (Step 3) equals to $\max(0, w'^{\pi_{i-1}}(e)) = 0$, and the shortest path to v is not longer than the shortest path to u, i.e. $d(u) \leq d(v)$. Therefore, we imply that $w'^{\pi_i}(e) \geq w'^{\pi_{i-1}}(e)$ for every out-of-kilter edge e, and moreover, one can imply by induction that $w'^{\pi_i}(e) \geq w'^{\pi_0}(e)$, where $\pi_0 = \pi$ is the node potentials in the initial iteration. Since $e \in B_i$ is an out-of-kilter in the initial iteration ($e \in A_0$) by Eq (30) we obtain $w'^{\pi_i}(e) \geq w'^{\pi_0}(e) \geq -|w'(e) - w(e)|$, and by Eq (32) we imply that.

$$w'(f) - w'(f') \leq \sum_{e \in B} |w'(e) - w(e)|. \tag{33}$$

According to Claim 6.2, if $e \in B$ is an area+type edge enters to vertex (a^j, t_i, n), then

$$w(e) - w'(e) = R_i^{loc}[\Pr(D_i^j \geq n \cdot B_i) - \Pr(D'^j_i \geq n \cdot B_i)]. \tag{34}$$

Similarly, if $e \in B$ is a type edge enters to vertex (t_i, n) then

$$w(e) - w'(e) = R_i^{glo}[\Pr(D_i \geq n \cdot B_i) - \Pr(D'_i \geq n \cdot B_i)] \tag{35}$$

Then combining Eq (34), (35), (33) with the definitions of the demand distance (Eq (8)) and weakly ϵ-near yields that

$$|w'(f) - w'(f')| \leq d(D, D') < \epsilon$$

As required.

□

Computation of the weight of edges and node potentials in a bipartite graph

By the following lemmas, we can compute the edges weight and node potentials of the bipartite graph in $O(1)$ (given computing the marginal differential is $O(1)$).

LEMMA .4. *Let f be a flow that SSP calculates in its i^{th} iteration in G_f^8. We denote respectively by f_i^j, f^j, f_i the flow through (a^j, t_i), the flow through a^j and the flow through t_i . Then the minimal monotone paths weights in G_f^8 can be computed in $O(1)$ by the marginal-differential functions (Claim 6.2) as given in the following formulas: 1) $w_f(x \overset{\min}{\rightarrow} a^j) = \Delta\gamma^j(f^j + 1)$. 2) $w_f(a^j \overset{\min}{\rightarrow} t^i) = \Delta\gamma_i^j(f_i^j + 1)$. 3) $w_f(t^j \overset{\min}{\rightarrow} a^i) = -\Delta\gamma_i^j(f_i^j)$ if $f_i^j > 0$ and otherwise ∞. 4) $w_f(t_i \overset{\min}{\rightarrow} y) = \Delta\gamma_i(f_i + 1)$.*

PROOF PROOF OF LEMMA .4. Note that a flow passing through vertex (a^j, n) has a weight of $\Delta\gamma^j(n)$. As the marginal differential functions are monotonically increasing, a min-cost flow must pass through vertices $(a^j, 1), (a^j, 2), \dots, (a^j, f^j)$. Thus, the weight forward paths x-(a^j, n)-a^j is $\Delta\gamma^j(n)$ iff $n > f^j$ and otherwise ∞. Thus, the minimal monotone path between the source x and an area node a^j must pass through vertices $(a^j, f^j + 1)$ with a weight of $\Delta\gamma^j(f^j + 1)$. We have prove part 1), and parts 2), 3) and 4) can be proved by a similar way. □

LEMMA .5. *Let f and π be the node-potentials that SSP calculates in its i^{th} iteration in G_f^8, and let $d^B(v)$ be the shortest path weight between x and v in the **bipartite graph** G_f^B (not the 8-layer residual G_f^8 graph) respect to the node potentials π. Then, SSP updates the node potentials as $\pi(v) = \pi(v) - d^B(v)$ for every vertex v in V^B.*

PROOF PROOF OF LEMMA .5. For this proof we denote $d^8(v) = d(v)$ as the shortest path weight between x and v in the 8-layer residual G_f^8 graph , with respect to π. Then SSP updates the node potentials as $\pi = \pi - d^8 = \pi - d$ (See Section **??**). Thus, it is sufficient to prove that for every vertex v we have $d^B(v) = d^8(v)$.

Let v be a vertex in V^B, and let $p_{opt}^B(v)$ be the shortest path in G^8. Then we have $d^B(v) = w_f^\pi(p_{opt}^B(v))$. In [1] they prove that the reduce weight for every path p between vertices a and b in a general graph G equals to $w^\pi(p) = w(p) - \pi(a) + \pi(b)$ where $w(p)$ is the path weight. Thus, the shortest path respect to the node potentials π equals to $d^B(v) = w_f(p_{opt}^B(v)) - \pi(x) + \pi(v)$

Similarly to the proof Theorem 7.1, there exists a path $p_{opt}^8(v)$ which has the same weight of $p_{opt}^B(v)$ and is the shortest path in G_f^8, namely $w_f(p_{opt}^B(v)) = w_f(p_{opt}^8(v))$. This implies that shortest path respect to π in 8-layer residual equals to $d^8(v) = w_f(p_{opt}^8(v)) - \pi(x) + \pi(v) = w_f(p_{opt}^B(v)) - \pi(x) + \pi(v) = d^B(v)$, as required. □

BGA complexity and the bounded case

First we prove the BGA complexity according to the following claim.

CLAIM .6. *BGA runs in $O(lkm)$ where l is the number of iterations*

PROOF. In every iteration the algorithm finds the shortest paths in G_f^B by Dijkstra algorithm, which costs $O(|E| + |V| \log |V|) = O(km)$. Updating the node potentials and the flow values takes at most $O(|V|) = O(k + m)$ time, and updating the graph weights is at most $O(|E|) = O(km)$ time. □

To solve the bounded case (i.e s^k are finite) we can set the marginal differential functions to $\Delta\gamma^j(n) = \infty$, where $n \geq s^j + 1$ and run BGA according to these new weights. In every iteration, BGA finds a shortest path that can either the edge (x, y) with a zero weight (and therefore by the non-negative stopping rule BGA terminates) or a shortest path that passes through areas node a^j, which must have negative weight.

Assume that in the first s iterations BGA finds only negative shortest paths through areas node a^j. Then according to Lemma .4 every edge (x, a^j) in these shortest paths satisfied $f^j \leq s^j$. Otherwise, the edges (x, a^j) have infinity weight, and the shortest path does not passes them. Since $\sum_{j=1}^{k} f^j$ equals to the number of iterations (which is s), then for every area j we have $f^j = s^j$. Thus, in the next iteration of BGA the shortest path must be the edge (x, y), and BGA terminates.

Thus, after at most s iterations BGA terminates, and its complexity will be at most $O(smk)$ for the bounded case.

out-of-kilter-algorithm

Let f flow be a flow defined on the graph $G =< V, E >$, and assume the reduced weight function w^π is defined on $G_f =< V, E_f >$, such that w^π **does not** satisfies the non-negative reduced weight property.

An edge $e \in E_f$ in the residual graph G_f is called *out-of-kilter* edge if its reduce weight is negative, i.e $w_f^\pi(e) < 0$. For those edges we define its kilter number $k(e)$ to be its residual capacity $k(e) = c_f(e)$. Note that if the kilter number must be different then zero; otherwise, the residual capacity is zero, and the edge weight $w_f(e)$ is set to infinite. For an in-kilter edge the kilter number is set to zero.

The out-of-kilter-algorithm (Algorithm 2) finds in every iteration an out-of-kilter edge and decreases its kilter number. In [1] they proved the following Lemma.

LEMMA .7. *Let f flow be a flow defined on the graph $G =< V, E >$ and let π be an arbitrary node-potentials function. Suppose we run the out-of-kilter algorithm on these parameters, and let $e_i = (u, v)$ and π_i be respectively the out-of-kilter edge chosen in Step2 and node potentials of the i^{th} iteration (defined in step 5). Then the following claims hold:*

1. *If e is an out-of-kilter edge in the cycle w of Step 7 respect to node potentials π_i, then its kilter number is strictly decreases. Moreover, the kilter number of e_i strictly decreases.*
2. *All edges e in the cycle w have non-positive reduce weights (i.e. $w^\pi(e) \leq 0$). Moreover, e is an in-kilter edge in the cycle w, then its reduce weight $w^{\pi_i}(e)$ equals to zero.*
3. *The kilter number of every edge in the residual graph does not increased.*

Of course, combining the previous lemma with the non-negative reduced weight condition proves the optimality of the algorithm.

Service cost (C_i^{xxx})	Locally	Remotely	Unsatisfied
Windows	-1.5$	-1$	1$
RHEL	-1$	-0.9$	1$

Table 1: Service costs used in simulations

Algorithm 2 The out-of-kilter-algorithm

```
Require: A feasible flow $f$ on $G$, a residual graph $G_f$, node
    potentials $\pi$, a source $x$ and a sink $y$ in $G$.
 1: while the network contains an out-of-kilter edge in $G_f$
    do
 2:   Select an out-of-kilter edge $(u, v)$ in $G$.
 3:   Define the length of each arc $e$ in $G_f$ as $\max 0, w^\pi(e)$.
 4:   Let $d()$ denote the shortest path distances from node
      $v$ to all other nodes in $G_f - \{(u, v)\}$ and let $p$ denote
      a shortest path from node $v$ to node $u$
 5:   Update $\pi(v) = \pi(v) - d(v)$ for every vertex $v$
 6:   if $w^\pi(u, v) < 0$ then
 7:     Define the cycle $w = p \bigcup (u, v)$
 8:     Find $\delta = \min\{c(e) | e \in E\}$).
 9:     Augment $\delta$ units of flow through $f$.
10:   end if
11: end while
```

On demand costs (p_i^j)	USA	Europe	Asia
Windows	0.14$	0.133$	0.161$
RHEL	0.137$	0.137$	0.158$

Table 2: Amazon EC2 price system