Dynamic Service Placement in Shared Service Hosting Infrastructures

Qi Zhang, Jin Xiao, Eren Gürses, Martin Karsten, and Raouf Boutaba

David. R. Cheriton School of Computer Science,
University of Waterloo, Waterloo, Canada
{q8zhang,j2xiao,egurses,mkarsten,rboutaba}@cs.uwaterloo.ca

Abstract. Large-scale shared service hosting environments, such as content delivery networks and cloud computing, have gained much popularity in recent years. A key challenge faced by service owners in these environments is to determine the locations where service instances (e.g. virtual machine instances) should be placed such that the hosting cost is minimized while key performance requirements (e.g. response time) are assured. Furthermore, the dynamic nature of service hosting environments favors a distributed and adaptive solution to this problem. In this paper, we present an efficient algorithm for this problem. Our algorithm not only provides a worst-case approximation guarantee, but can also adapt to changes in service demand and infrastructure evolution. The effectiveness of our algorithm is evaluated though realistic simulation studies.

1 Introduction

With the abundance of network bandwidth and computing resources, large-scale service hosting infrastructures (e.g. content-delivery networks, service overlays and cloud computing environments) are gaining popularity in recent years. For instance, PlanetLab, as a shared academic overlay network, is capable of hosting global-scale services including content delivery [1] and file storage [2]. More recently, commercial products such as Amazon Elastic Computing Cloud (EC2) [3], Google AppEngine [4] and Microsoft Azure [5] have emerged as attractive platforms for deploying web-based services. Compared to traditional service hosting infrastructures, a shared hosting infrastructure offers numerous benefits, including (1) eliminating redundant infrastructure components, (2) reducing the cost of service deployment, and (3) making large-scale service deployment viable. Furthermore, the ability to allocate and deallocate resource dynamically affords great flexibility, thus (4) making it possible to perform dynamic scaling of service deployment based on changing customer demands.

In general, three types of roles are present in a shared hosting environment: (1) the *infrastructure provider* (InP) who owns the hosting environment; (2) the *service provider* (SP) who rents resources from the infrastructure provider to run its service, and (3) the *client* who is a service customer. In this context, a service provider's objective is to (a) satisfy its performance requirements specified

M. Crovella et al. (Eds.): NETWORKING 2010, LNCS 6091, pp. 251–264, 2010.

in Service Level Agreements (SLAs) such as response time, and (b) minimize its resource rental cost. For example, response time of requests for CNN.com should be less than 2 seconds [6]; Online game services often have stringent response time requirements [7]. Achieving this delay requirement (objective a) is dependent on where the servers are placed, and typically there is a monetary penalty when this requirement is not met. At the same time, achieving resource cost reduction (objective b) is also server location dependent. Thus, we argue there is a strong and consistent incentive for SPs to choose their servers' placement carefully to minimize the total operating cost. This is the case for cloud computing environment as well. Although the number of data centers used by commercial products is relatively small today, this number will grow (as demand for cloud computing grows), and new architectures such as micro [8] and nano [9] data centers will emerge. A distributed solution to this problem is desirable in such a large-scale and dynamic environment, as frequent re-execution of centralized algorithms can incur significant overhead for collecting global information [10,11]. Hence an adaptive and distributed service placement strategy is strongly favored.

In this paper, we present a solution to the dynamic service placement problem (SPP) that minimizes response time violations and resource rental cost at run time. We formulate SPP mathematically and present a distributed local search algorithm that achieves a theoretical worst-case performance guarantee. The effectiveness of our algorithm is experimentally verified using a set of realistic simulation studies.

The remainder of this paper is organized as follows: We present a generic model of a service hosting infrastructure in Section 2. In Section 3, we present the formulation of SPP and introduce the notations. Our distributed algorithm is presented in Section 4. In Section 5, we show that the algorithm achieves an approximation guarantee of 27. Experimental results are presented in Section 6. We discuss related research in Section 7 and conclude our work in Section 8.

2 System Overview

We consider a hosting platform that consists of servers that are diversely situated across geographical locations. A generic architecture of a service hosting platform is depicted in Figure 1. An instance of a service may be installed on one or many servers. To achieve locality awareness and load-balancing, a client request is redirected to a nearby server with enough CPU and bandwidth capacity to handle the request. The component which is responsible for redirecting request is the *request router*. In practice, a request router can be implemented using a variety of techniques, such as DNS-based request redirection, or direct routing of requests [12]. On the other hand, the status of individual servers, including traffic condition, CPU, memory and bandwidth usage are measured at run-time by a *monitoring module* on each server. It should be pointed out that it is entirely possible that monitoring modules are shared among a group of servers, and our model is fully applicable to such a scenario. The *analysis*

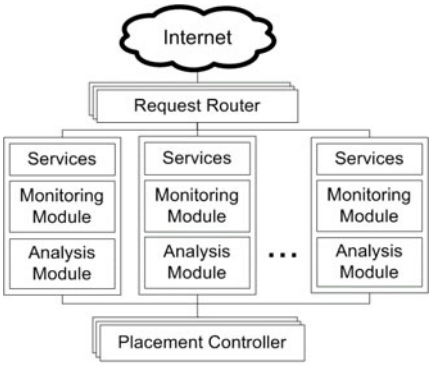


Fig. 1. System Architecture

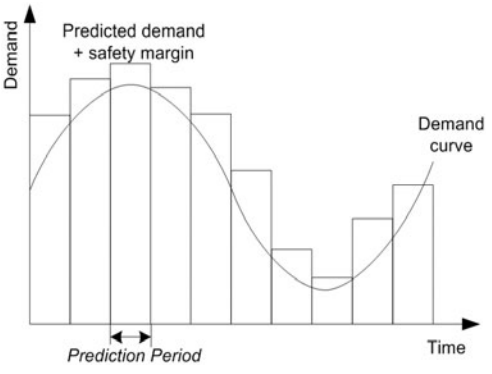


Fig. 2. Dynamic Demand Prediction

modules are responsible for analyzing the current demand and predicting the future demand during the next time interval, as we shall explain below. Finally, the *placement controllers* are responsible for dynamically placing services. This includes copying service software (e.g. virtual machine images) to the server, dynamically starting and stopping service instances. The placement controllers are activated periodically, based on the demand prediction received from the analysis modules. In practice, the placement controllers can be implemented either centralized or distributed. For example, Microsoft Azure uses centralized controllers that are managed by service providers [5]. P2P overlay networks, on the other hand, requires distributed placement controllers that are located on individual peers. In such a scenario, a placement controller can collect relevant information using protocols such as local broadcasting and gossiping [13].

Our main objective is to dynamically adapt placement configurations to changing demands and system conditions. The demand can be modeled as function of time and location. Ideally, we want the placement configuration to be adaptive to individual demand changes. However, this is often impractical since frequent placement reconfiguration can be expensive to implement. Moreover, fine-grained demand fluctuations are usually transient and do not persist for long durations. Therefore, it is necessary to both analyze and periodically predict service fluctuations at coarse-grain levels, as illustrated in figure 2. Furthermore, even when the daily service demand follows some predictable pattern, pre-planned configuration or placement strategies at the infrastructure level are not possible because both the infrastructure and services themselves may change drastically (e.g. unexpected system failures and transient services such as VPN for a sports broadcast). The problem of predicting future service demand has been studied extensively in the past, and many solution techniques (e.g. [14]) have been proposed. It is not our goal to examine better demand prediction methods. Rather, we study how placement configuration should be modified according to a prediction of the future service demand.

3 Problem Definition and Notations

We represent the hosting platform as a bipartite graph $G = (D, F, E)$, where D is the set of clients, F is a set of candidate servers on which service instances can be installed, and E is the set of edges connecting D and F. A service instance may consume two types of resources: (1) *Load-dependent resources*, whose consumption is dependent on the amount of demand served by the service instance. Load-dependent resources typically include CPU, memory and bandwidth. (2) *Load-independent resources*, whose consumption is constant regardless of how much demand the server receives. An example of load-independent resource is storage space of the service software.

We assume every candidate server in F has enough load-independent resources to host the service, since otherwise we can remove it from F. For load-dependent resources, we assume there is a bottleneck capacity cap_s on how many client a service instance $s \in F$ can serve. The bottleneck capacity can be bandwidth, CPU capacity or memory, whichever is more stringent on each server. It should be noted that this bottleneck capacity may change over time. However, as we shall see our local search algorithm can naturally handle this case. In addition, We assume there is a placement cost p_s for renting a service instance on a server s.

Let C_s denote the set of demand assigned to s. For every demand $i \in D$, let s_i represent the service instance who serves i. For every demand $i \in C_{s_i}$, denote by $d(i, s_i)$ the distance between i and s_i that represents the communication latency between i and s_i. Furthermore, let U_{s_i} denote the number of client assigned to a server s_i. Using a simple M/M/1 queuing model [1], We obtain the actual response time $r(i, s_i)$ of a service request i as a function of U_{s_i} and $d(i, s_i)$:

$$r(i, s_i) = d(i, s_i) + \frac{\mu_{s_i}}{1 - U_{s_i}/cap_{s_i}} = d(i, s_i) + \frac{cap_{s_i} \cdot \mu_{s_i}}{cap_{s_i} - U_{s_i}}$$

where μ_s is the service time of a request at the server. The second term essentially models the queuing delay of the request. We construct a cost penalty $cp(i, s_i)$ if the request response time exceeds a threshold value d_{max}. Specifically, we define

$$cp(i, s_i) = a(\frac{r(i, s_i)}{d_{max}})^2$$

where a is a monetary penalty cost. We use a quadratic penalty function here as it reflects the general form of the penalty payout for SLA violation. Higher order polynomial is also possible and does not affect our analysis technique. Our objective is to select a set of servers $S \subseteq F$ to deploy our service as to minimize the sum of hosting cost $c_f(S)$ and service cost $c_s(S)$:

$$c(S) = c_s(S) + c_f(S) = \sum_{i \in D} cp(i, s_i) + \sum_{s \in S} p_s \quad (1)$$

[1] More sophisticated model can be easily incorporated by changing the equations for $r(i, s_i)$ and $cp(i, s_i)$.

Algorithm 1. Local Search Algorithm for SPP

1: **while** $\exists$ an add(s), open(s, T) or close(s, T) that reduces total cost **do**
2: execute this operation

This problem is difficult to solve directly due to the load factor U_s in the equation. We remedy the situation by having a preferred load value l_s for U_s. In this case we define SPP as follows:

Definition 1 (SPP). *Given a demand set D and a server set F , each with a renting cost p_s and a service cost $cp(i, s_i) = a((d(i, s_i) + \frac{\mu_{s_i}}{1-l_{s_i}/cap_{s_i}})/d_{max})^2$, select a set of servers $S \subseteq F$ to host the service and assign the demand to S such that $|C_s| \leq l_s$ for every $s \in S$, minimizing the total cost $c(S) = \sum_{i \in D} cp(i, s_i) + \sum_{s \in S} p_s$.*

It is actually provable that the original problem can be reduced to this formulation (We skip the proof due to space constraints). In fact, setting a preferred load value is more practical for implementation, since typically a service provider may define an expected utilization for each service instance [11].

We observe that SPP is similar to the Capacitated Facility Location Problem (CFLP) [15]. CFLP can be briefly described as follows: Given a graph $G = (V, E)$ and a set of candidate sites $F \subseteq V$ for installing service facilities (each site $s \in F$ has an installation cost f_s and a service capacity U_s), select a subset of sites $S \subseteq F$ to install facilities and assign each client to a facility in S such that the sum of the connection cost $c_s(S) = \sum_{i \in V} d(i, s_i)$ and the installation cost $c_f(S) = \sum_{i \in S} f_s$ is minimized. CFLP is an NP-hard optimization problem [16], for which many centralized approximation algorithms have been proposed. In distributed settings, however, only the uncapacitated case has been studied [17,18].

It can be seen that CFLP is almost identical to SPP except the connection cost is defined as $cp(i, s_i) = d(i, s_i)$. It is easy to show SPP is NP-hard using the same argument for CFLP. Motivated by this observation, we developed an approximation algorithm for SPP based on the local search algorithm for CFLP described in [15]. In our work, we also assume our distance metric satisfies triangular inequality. This is a practical assumption since we can always use the network coordinate services [19] to compute the distance metric. Furthermore, it is also known that Internet triangular inequality is approximately satisfied (i.e. $d(i, j) \leq k(d(i, k) + d(k, j))$), where k is roughly 3 [20]. It is easy to adjust our prove technique to accommodate this effect.

4 A Local Search Approximation Algorithm for SPP

Local search is a well known meta-heuristic for solving optimization problems. A local search algorithm starts with arbitrary solution and incrementally improves the solution using local operations until the solution can no longer be improved.

Local search algorithms can naturally tolerate dynamic changes in the problem input, as they can start from arbitrary solutions. We believe a local search algorithm for SPP is favorable as it not only tolerates system dynamicity, but also provides a provable worst-case performance guarantee.

Based on the theoretical work by Pal et. al. [15], Algorithm 1 is our proposed local search algorithm for SPP. In our work, we use the term server and facility interchangeably. Two particular terminologies used in the paper are opening and closing a facility t, which means installing a service and uninstalling a service at location t. This algorithm starts from any feasible initial solution, and incrementally improves the solution with one of three types of operations: (1) $add(s)$, which opens a facility s and assign a set of clients to s. (2) $open(s, T)$, which opens a facility s, close a set of facilities T, and assign the clients of facilities in T to s. (3) $close(s, T)$, which closes a facility s, open a set of facilities T and assign the client of s to facilities in T. In all three operations, a facility can be opened multiple times but closed only once.

Due to its simplicity and adaptive nature, this algorithm fits very well with our objective. However, this algorithm cannot be directly implemented in a distribute way, since finding an open(s, T) and a close(s, T) requires solving a knapsack problem and a single node capacitated facility location (SNCFL)[21] problem, respectively. Both problems are NP-hard in general, but can be solved in pseudo-polynomial time using dynamic programming [15]. However, this dynamic programming approach does not apply to distributed settings since it is both expensive and requires global knowledge.

Hence, it is our objective to show that both open(s, T) and close(s, T) can be computed locally and distributedly. In this regard, our algorithm is local in the sense that each operation is computed using neighborhood information. Specifically, Each node s in our algorithm maintains a list of neighborhood servers within a fixed radius. The list of neighbors can be obtained using neighborhood discovery protocols or a gossiping protocol [13]. Since the servers are usually static and do not change often overtime, the list of neighboring servers will not require frequent update.

4.1 Implementing Add(s)

The purpose of the add operation is to reduce the total connection cost. In an $add(s)$ operation, if a facility s is not opened, it will become opened and start serving clients. In this case the cost reduction (CR) of the add(s) operation is:

$$\mathrm{CR}(\mathrm{add}(s)) = \sum_{i \in U} cp(i, s_i) - cp(i, s) - p_s$$

where $U \subseteq V$ is a set of clients. On the other hand, if facility s is already opened, it can invite more clients to join its cluster. In this case we set $f_s = 0$ in the above equation. An add(s) operation can be performed when $\mathrm{CR}(\mathrm{add}(s)) > 0$.

However the straightforward implementation, i.e. Letting s contact the other facilities to find the set of potential clients, is cumbersome to implement. This

Algorithm 2. Local Algorithm for computing an admissible open(s, T) move

1: Sort facilities in decreasing order of their cost efficiencies, $c_s \leftarrow 0$
2: **while** $c_s \leq cap_s$ and $\text{costEff}_{open}(t) \geq 0$ for the next t **do**
3: $T \leftarrow T \cup t$, $c_s \leftarrow c_s + |C_t|$
4: **return** either T or the next facility $t \notin T$ in the list, whichever is larger

is because s does not know a-priori which facility has clients with high connection cost. Instead, in our implementation, we allow neighborhood facilities to exchange client information and record potential clients. Once s have recorded enough potential clients, s can become open (if not already opened), and start serving these clients. Observe that a client i must satisfy $cp(i, s) \leq cp(i, s_i)$ in order for the reassignment to be beneficial, hence $d(s, s_i) \leq d(i, s_i) + d(i, s) \leq d(i, s_i) + \sqrt{\frac{cp(i,s)}{a}} \cdot d_{max} \leq d(i, s_i) + \sqrt{\frac{cp(i,s_i)}{a}} \cdot d_{max}$. Therefore s_i only needs to exchange client information with servers within radius $d(i, s_i) + \sqrt{\frac{cp(i,s_i)}{a}} \cdot d_{max}$ to guarantee an admissible add(s) operation will be found by a facility s.

4.2 Implementing Open(s, T)

In an open(s, T) operation, a single facility s is opened and a set of facilities T become closed. All the clients of facilities in T get assigned to s. The total reduction of this operation is computed as:

$$\text{CR}(open(s, T)) = \sum_{t \in T} (p_t - |C_t| cp(t, s)) - p_s$$

Again, notice that if s is already opened before the move, then p_s is set to 0. The key challenge here is to determine the suitable set T. The problem of finding the optimal set T that maximizes $\text{CR}(open(s, T))$ can be formulated as a 0-1 knapsack problem: Let s be a knapsack with capacity cap_s, and each open facility t be an item with size$(t) = |C_t|$ and value$(t) = p_t - |C_t| d(s, t)$. The objective is to select a set of items to be packed in the knapsack to maximize the total value. Although it can be solved in pseudo-polynomial time using dynamic programming, it is not practical to implement because it requires global knowledge. Hence, we replace the dynamic programming procedure by a well-known greedy 0.5 approximation algorithm (Algorithm 2), where the cost efficiency of each facility t is defined as $\frac{value(t)}{size(t)}$, which is $\text{costEff}_{open}(t) = \frac{p_t}{|C_t|} - cp(s, t)$.

To implement Algorithm 2 distributedly, notice that for any facility t, if $cp(s, t) \geq \frac{p_t}{|C_t|}$, then costEff$(t)$ becomes negative and can be safely ignored in the calculation. Hence each facility t only need to contact neighborhood facilities in radius $\sqrt{\frac{p_t}{a \cdot |C_t|}} \cdot d_{max}$. A neighboring facility s can then compute an admissible open(s, T) once it discovers the entire T.

4.3 Implementing close(s, T)

The objective of the close(s, T) is to close a facility s, open a set of facilities T, and assign all of the clients of s to facilities in T. Again, the main difficulty is to find a suitable set T. The cost reduction of close(s, T) can be computed as:

$$\mathrm{CR}(close(s,T)) = p_s - \sum_{t \in T}(p_t + u_t cp(s,t))$$

Where u_t represents the amount of demand assigned to t after closing s. Similar to open(s, T), if a facility $t \in T$ is already opened, then p_t is set to 0 in the above equation. Computing the optimal T can be formulated as a single-demand capacitated facility location problem (SNCFL)[21]. In this problem, we wish to select a set of facilities T with total capacity at least $|C_s|$, assigning clients of s to facilities in T to minimize

$$cost_{SNCFL} = \sum_{t \in T}(p_t + u_t cp(s,t))$$

This problem is also NP-hard. However, a greedy algorithm is known for this problem [21]. In this algorithm, the facilities are sorted by their cost efficiencies, which in our case, is defined as $\mathrm{costEff}_{close}(t) = \frac{p_t}{cap_t} + cp(s,t)$ This algorithm (Algorithm 3) can be described as follows: the output set T is initially empty. we first sort facilities in increasing order of their cost efficiencies, and then try to add facilities T according to the sorting order. In each step, if adding next facility into T will cause $\sum_{t \in T} cap_t \geq |C_s|$ (i.e. facilities in T have enough capacity to handle demands of s), then T is a candidate solution. Now, if $\mathrm{CR}(close(s,T)) \geq 0$, this is an admissible operation and the algorithm stops. Otherwise, we do not add this facility to T. This process repeats until every facilities in U has been examined. The following lemma is a known result for this algorithm:

Lemma 1. *[21] The greedy algorithm outputs a solution S with $cost_{SNCFL} \leq \sum_{t \in T}(2p_t + u_t cp(s,t))$ for any $T \subseteq F$.*

Clearly, a facility in T can not be outside a radius $\sqrt{\frac{p_s}{a}} \cdot d_{max}$, as in this case the reassignment cost will exceed p_s. Hence close(s, T) operation can be computed locally by s within in this radius.

5 Algorithm Analysis

In this section, we show that our algorithm achieves an approximation factor 27. There are two challenges to this problem. First, our distance function is non-linear, as opposed to the linear function used in CFLP. Second, since we have replaced the dynamic programming procedures with greedy algorithms, we need to show our algorithm can still provide a performance guarantee.

Lemma 2. *If an open operation open(s, T) computed using Algorithm 2 is not admissible, then $2p_s + \sum_{t \in T'}(u_t cp(s,t) - p_t) \geq 0$ for any set of $T' \subseteq F \backslash \{s\}$.*

Algorithm 3. Local Algorithm for computing an admissible close(s, T) move

1: $rem \leftarrow |C_s|$, $T \leftarrow \emptyset$, $currentCR \leftarrow 0$, $U \leftarrow$ facilities within radius $\sqrt{\frac{p_s}{a}} \cdot d_{max}$,
2: sort U in increasing order of their cost efficiencies
3: **repeat**
4: $r = \frac{p_s - currentCR}{rem}$
5: **while** not every node in U has been examined **do**
6: $t \leftarrow$ next facility in the sorted list
7: **if** $rem - cap_t \geq 0$ **then**
8: $T \leftarrow T \cup t$, $currentCR \leftarrow \sum_{t \in T}(p_t + cap_t d(s,t))$, $rem \leftarrow rem - cap_t$
9: **else**
10: **if** CR$(close(s, T \cup t)) \geq 0$ **then**
11: **return** $T \cup t$
12: **until** $U == \{\emptyset\}$
13: output there is no admissible close operation

Proof. Since Algorithm 2 is a $\frac{1}{2}$-approximation of the knapsack problem, it is true for any $T' \subseteq F \backslash \{s\}$, $SOL_{ALG2} \geq \frac{1}{2}(\sum_{t \in T'} p_t - |C_t| d(s,t))$. Since it fails to find a admissible operation, $SOL_{ALG2} - p_s \leq 0$ must hold. The claim follows.

Lemma 3. *If a close operation close(s, T) computed using Algorithm 3 is not feasible, then $\sum_{t \in T'}(2p_t + u_t cp(s,t)) - p_s \geq 0$ for any set of $T' \subseteq F \backslash \{s\}$.*

Proof. Since Algorithm 3 is a 2-approximation algorithm for the $SNCFL$, we must have, for any set $T' \subseteq F \backslash \{s\}$, $SOL_{ALG3} \leq \sum_{t \in T'}(2p_t + u_t cp(s,t))$. Since it fails to find an admissible move, we must have $p_s - SOL_{ALG3} \leq 0$. The lemma follows.

Using the above lemmas, we can prove our main result:

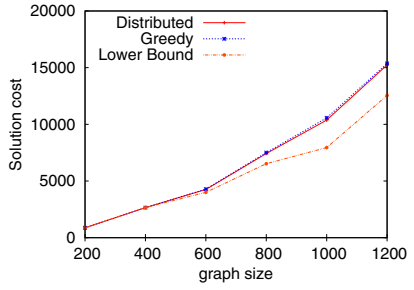
Theorem 1. *The distributed algorithm achieves an approximation factor* 27.

Proof. See appendix.

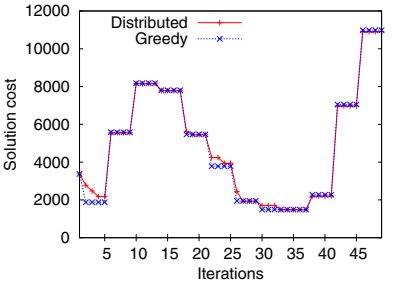
6 Experiments

We have implemented our distributed algorithm in a discrete event simulator and conducted several simulation studies.

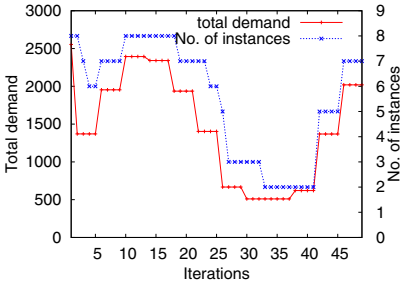
We construct topology graphs using the GTITM generator [22], which can generate transit-stub topologies that simulate latencies between hosts on the Internet. We specify the average communication latency at intra-transit, stub-transit and intra-stub domain links to be 20ms, 5ms and 2ms respectively [23]. We randomly pick a subset (10%) of nodes as candidate locations for placing servers. Each candidate location can host up to 5 instances. In our experiments, we set d_{max} = 400ms, μ_s=20ms, $a = 1$ and $p_s = 4$ for 200 node graph, 8 for



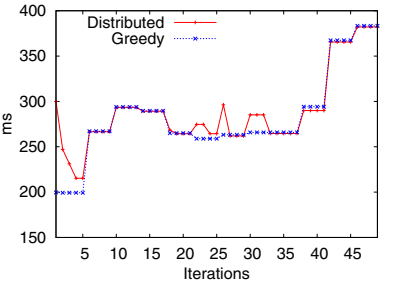
(a) Performance Comparison



(b) Runtime performance



(c) Dynamic server capacity



(d) Average connection cost

Fig. 3. Experimental result

400 node graph and so on. For benchmarking purpose, we have implemented a centralized greedy placement algorithm [24], which has been shown to perform well in practice.

In our first experiment, we evaluate the performance of distributed algorithm in static topologies where there is no demand fluctuation. To estimate the value of optimal solution, we implemented a simple routine that outputs a lower-bound on the optimal solution by ignoring capacity constraints. Our experimental results are shown in Figure 3(a). We observe that although both algorithms perform well compared to the lower-bound values, our algorithm performs better than the greedy algorithm in all cases. This result indicates that, the solution produced by our algorithm is near optimal.

In our next experiment, we evaluate the effectiveness of our algorithm under dynamic conditions, i.e. demand of clients fluctuate overtime and servers may fail randomly. To make experiment realistic, we used a real internet topology graph from the Rocketfuel project [25]. Figure 3(b) shows the cost of the solution at runtime. For comparison purpose, at each time stamp we also show the solution of the greedy heuristic that has access to global topology information. As shown on Figure 3(b), our distributed algorithm is quite adaptive to the dynamic conditions, and in some cases, performs better than the greedy heuristic. Figure 3(c) depicts the change in the number of instances in response to the changes in client population. We observe that the number of instances used, as

suggested by our algorithm, follows the service demand very closely. Figure 3(d) illustrates the average distance between clients and their servers at runtime and it is comparable to the centralized greedy solution. These results suggests our algorithm will perform well in practical settings.

7 Related Work

Placing services in large-scale shared service hosting infrastructures has been studied both theoretically and experimentally. Oppenheimer et. al. [26] studied the problem of placing applications on PlanetLab. They discovered that CPU and network usage can be heterogenous and time-varying among the PlanetLab nodes. Furthermore, a placement decision can become sub-optimal within 30 minutes. They suggested that dynamic service migration can potentially improve the system performance. They also studied a simple load-sensitive service placement strategy and showed it significantly outperforms a random placement strategy. NetFinder [27] is a service discovery system for PlanetLab that employs a greedy heuristic for service selection, based on CPU and bandwidth usage.

Theoretically, Laoutaris et. al. formulated SPP as a uncapacitated facility location problem (UFLP) [10], and presented a local search heuristic for improving the quality of service placement solutions. There are several differences between their work and ours. First, server capacity, such as CPU and bandwidth capacity, are considered in our formulation but not in theirs. Second, their heuristic does not provide a worst-case performance guarantee.

Our problem is also related to the replica placement problem, which has been studied extensively in the literature, primarily in the context of content delivery networks. Most of the early work on this problem focus on centralized cases where the network topology is static [6,24,28,29]. More recent work has also studied dynamic cases [11,30], in which iterative improvement algorithms are proposed. However, none of the above work was able to provide a theoretical worst-case performance guarantee. Our algorithm is as efficient as the above algorithms, and achieves theoretical performance ratio at the same time. A potential research direction is on improving this theoretical performance guarantee.

Another problem related to SPP is the web application placement in data centers. The objective is to achieve load balancing so as to reduce total response time. Several heuristics have been proposed for the problem [31,32], using local algorithms. The main difference between their work and ours is that we consider the distance (which could be latency or hop count) between clients and the servers in our formulation. This is not a concern in their work, since they assume all the servers are located in a single data center.

8 Conclusion

Large-scale shared service hosting infrastructures have emerged as popular platforms for deploying large-scale distributed services. One of the key problems in managing such a service hosting platform is the placement of service instances as

to minimize operating costs, especially in the presence of changing network and system conditions. In this paper, we have presented a distributed algorithm for solving this problem. Our algorithm not only provides a constant approximation guarantee, but is also simple to implement. Furthermore, it can automatically adapt to dynamic network and system conditions.

As part of our future work, we would like to extend our work to handle more complex scenarios such as services with interdependencies. We would also like to improve our approximation guarantee. Furthermore, we are also interested in developing a control-theoretic framework for SPP. Most importantly, we would like to conduct a realistic experimentation on a real service hosting infrastructures such as PlanetLab.

References

1. Wang, L., Park, K., Pang, R., Pai, V., Peterson, L.: Reliability and security in codeen content distribution network. In: Proc. USENIX (2004)
2. Park, K., Pai, V.: Scale and performance in the coblitz large-file distribution service. In: Proc. of NSDI (2006)
3. Amazon elastic computing cloud (amazon ec2), `http://aws.amazon.com/ec2/`
4. Google app engine, `http://code.google.com/appengine/`
5. Azure services platform, `http://www.microsoft.com/azure/default.mspx`
6. Tang, X., Jianliang, X.: On replica placement for qos-aware content distribution. In: Proc. of IEEE INFOCOM (2004)
7. Tobias, F., et al.: The effect of latency and network limitations on mmorpgs: a field study of everquest 2. In: Proc. of ACM SIGCOMM Workshop NetGame (2005)
8. Church, K., Greenberg, A., Hamilton, J.: On delivering embarrassingly distributed cloud services. In: ACM HotNets (2008)
9. Valancius, V., Laoutaris, N., Massoulie, L., Diot, C., Rodriguez, P.: Greening the internet with nano data centers. In: ACM CoNext (2009)
10. Laoutaris, N., et al.: Distributed placement of service facilities in large-scale networks. In: Proc. of IEEE INFOCOM (2007)
11. Vicari, C., Petrioli, C., Presti, F.: Dynamic replica placement and traffic redirection in content delivery networks. In: Proc. of MASCOTS (2007)
12. Pathan, A.-M., Buyya, R.: A taxonomy and survey of content delivery networks. Technical Report, University of Melbourne, Australia (2006)
13. Jelasity, M., Montresor, A., Babaoglu, O.: Gossip-based aggregation in large dynamic networks. ACM Trans. on Computer Systems 23
14. Bodik, P., et al.: Statistical machine learning makes automatic control practical for internet datacenters. In: Proc. of USENIX HotCloud (2009)
15. Pal, M., Tardos, T., Wexler, T.: Facility location with nonuniform hard capacities. In: Proceedings of FOCS (2001)
16. Zhang, J., Chen, B., Ye, Y.: Multi-exchange local search algorithm for capacitated facility location problem. In: Math. of Oper. Research (2004)
17. Frank, C., Romer, K.: Distributed facility location algorithms for flexible configuration of wireless sensor networks. In: DCOSS (2007)
18. Moscibroda, T., Wattenhofer, R.: Facility location: Distributed approximation. In: ACM PODC (2005)
19. Dabek, F., Cox, R., Kaashoek, F., Morris, R.: Vivaldi: A decentralized network coordinate system. In: ACM SIGCOMM 2004 (2004)

20. Francis, P., et al.: An architecture for a global internet host distance estimation service. In: IEEE INFOCOM (1999)
21. Gortz, S., Klose, A.: Analysis of some greedy algorithms for the single-sink fixed-charge transportation problem. Journal of Heuristics (2007)
22. Gtitm homepage, http://www.cc.gatech.edu/projects/gtitm/
23. Ratnasamy, S., Handley, M., Karp, R., Scott, S.: Topologically-aware overlay construction and server selection. In: IEEE INFOCOM (2002)
24. Qiu, L., Padmandabhan, V., Geoffrey, V.: On the placement of web server replicas. In: IEEE INFOCOM (2001)
25. Spring, N., Mahajan, R., Wetherall, D., Anderson, T.: Measuring isp topologies with rocketfuel. IEEE/ACM Transactions on Networking, TON (2009)
26. Openheimer, D., Chun, B., Patterson, D., Snoeren, A., Vahdat, A.: Service placement in a shared wide-area platform. In: Proc. of USENIX (2006)
27. Zhu, Y., Mostafa, A.: Overlay network assignment in planetlab with netfinder. Technical Report (2006)
28. Szymaniak, M., Pierre, G., van Steen, M.: Latency-driven replica placement. In: Proc. of Symoposum on Applications and Internet (2005)
29. Karlsson, M., Karamanolis, C.: Choosing replica placement heuristics for wide-area systems. In: International Conference on Distributed Computing Systems, ICDCS (2004)
30. Presti, F., Bartolini, N., Petrioli, C.: Dynamic replica placement and user request redirection in content delivery networks. In: IEEE International Conference on Communications, ICC (2005)
31. Carrera, D., Steinder, M., Torres, J.: Utility-based placement of dynamic web application with fairness goals. In: IEEE/IFIP Network Operations and Management Symposium, NOMS (2008)
32. Tang, S.M., Chunqiang, Spreitzer, M., Pacifici, G.: A scalable application placement controller for enterprise data centers. In: International World Wide Web Conference (2007)

A Proof of Theorem 1

Let S^* denote the facility set in the optimal solution. We proof our algorithm achieves an approximation ratio of 27.

Lemma 4. *$cp(s,o) \leq 3(cp(i,s)+cp(i,o))$ for any client i and two facilities s,o.*

Proof. By definition, $cp(s,o) = a((d(s,o) + \frac{\mu_o}{1-l_o/cap_o})/d_{max})^2$. Using $d(s,o) \leq d(i,s) + d(i,o)$ to expand this equation, and applying the fact $a^2 + b^2 \geq 2ab$ for any $a, b \in R$, the lemma is proven.

Lemma 4 essentially states that the triangular inequality is 3-satisfied in SPP.

Lemma 5. *$c_s(S) \leq c_f(S^*) + 3c_s(S^*)$ of a local solution S.*

Proof. Similar to lemma 4.1 in [15], except we use the fact the triangular inequality is 3-satisfied.

Now we bound the facility opening cost $c_f(S)$. Similar to [15], we select a set of operations (add, open and close) to convert S to S^*. The key difference between our approach and theirs is that we use greedy algorithms to approximate the optimal open and close operations. Recall that from lemma 2 and 3, $2p_s + \sum_{t \in T'}(u_t cp(s,t) - p_t) \geq 0$ for all feasible open(s,T) operations, and $p_s \leq \sum_{t \in T}(p_t + u_t cp(s,t))$ for all close(s,T) operations. We say an open operation opens s twice, but close each facility in T once. Similarly, an close(s,T) operation close s once and opens each facility in T twice. Hence, using the same set of operations, we can show the following result:

Lemma 6. *With our greedy algorithm 2 and 3, the same set of operations in [15] opens p_t at most 14 times for each $t \in S^*$, and closes p_s exactly once for each $s \in S \backslash S^*$.*

Proof. Follow the same analysis as in Lemma 5.1 of [15], except we use lemma 2 and 3 to bound the cost of open and close operations.

Lemma 7. *The total reassignment cost of all selected operations is at most* $2 \cdot 3(c_s(S^*) + c_s(S))$.

Proof. This proof is identical to Lemma 5.2 in [15], except we use the fact the triangular inequality is 3-satisfied.

Finally, we bound our approximation factor:

Proof (Of Theorem 1). Based on Lemma 6 and 7, the sum of all the inequalities is at most $14c_f(S^*) - c_f(S - S^*) + 2 \cdot 3(c_s(S) + c_s(S^*))$. hence

$$14c_f(S^* \backslash S) - c_f(S \backslash S^*) + 6(c_s(S) + c_s(S^*)) \geq 0$$

Adding $c_s(S) + c_f(S \cap S^*)$ to both sides and using $c_s(S) \leq 3c_s(S^*) + c_f(S^*)$ proved by lemma 5, we obtain $c(S) \leq 23c_f(S^*) + 27c_s(S^*)$.