2011 IEEE 4th International Conference on Cloud Computing

A User Experience-based Cloud Service Redeployment Mechanism

YU KANG*, YANGFAN ZHOU*, ZIBIN ZHENG*, AND MICHAEL R. LYU*,+

*Department of Computer Science and Engineering, The Chinese University of Hong Kong, HongKong, China
+School of Computer Science, National University of Defence Technology, Hunan, China
{ykang, yfzhou, zbzheng, lyu}@cse.cuhk.edu.hk

***Abstract*—Cloud computing has attracted much interest recently from both industry and academic. Nowadays, more and more Internet applications are moving to the cloud environment. Making optimal deployment of cloud applications is critical for providing good performance to attract users. Optimizing user experience is usually required for cloud service deployment. However, it is a challenging task to know the user experience of end users, since there is generally no proactive connection between a user to the machine that will host the service instance. To attack this challenge, in this paper, we first propose a framework to model cloud features and capture user experience. Then based on the collected user connection information, we formulate the redeployment of service instances as *k-median* and *max k-cover* problems. We proposed several approximation algorithms to efficiently solve these problems. Comprehensive experiments are conducted by employing a real-world QoS dataset of service invocation. The experimental results show the effectiveness of our proposed redeployment approaches.**

I. INTRODUCTION

In cloud computing systems, computation, software, and data access can be delivered as services located in data centers distributed over the world [1], [2]. Typically, these services are deployed on instances (*e.g.*, virtual machine instances) in the cloud data centers. Recently, numerous systems have been implemented with the cloud computing paradigm (*e.g.*, Amazon Elastic Compute Cloud (EC2)).

In the emerging cloud computing systems, *auto scaling* and *elastic load balance* are key to host the cloud services. *Auto scaling* enables a dynamic allocation of computing resources to a particular application. In other words, the number of service instances can be dynamically adapted to the request load. For example, EC2 can automatically launch or terminate a virtual machine instance for an EC2 application based on user-defined policies (*e.g.*, CPU usage) [3]. *Elastic load balance* distributes and balances the incoming application traffic (*i.e.*, the user requests) among the service instances (*e.g.*, the virtual machine instances in EC2 [4]).

Auto scaling and elastic load balance directly influence the Internet connections between the end users and the services as they essentially determine the available service instance for an end user. Hence, they are important to the user experience of service performance.

Unfortunately, current auto scaling and elastic load balance techniques are generally not optimized for achieving best service performance. Specifically, typical auto scaling approaches (*e.g.*, that adopted in EC2 [3]) cannot start or terminate a service instance at the data center selected according to the distributions of the end users. For example, when the number of users increases dramatically in an area, a new instance located far away, instead of nearby, may be activated for serving the users. Furthermore, elastic load balance generally redirects user requests to the service instances merely based on loads of the instances. It does not take the user specifics (*e.g.*, user location) into considerations. As a result, a user may be directed to a service instance far away even if there is another available service instance nearby.

In this paper, we model the features of user experience mainly by latency (other features can be extended in the framework) in cloud service. After that we address these issues by proposing a new user experience-based service hosting mechanism. Our mechanism employs a service redeployment method. This method has two advantages:

1) It improves current auto scaling techniques by launching the best set of service instances according to the distributions of end users.
2) It extends elastic load balance. Instead of directing user request to the lightest load service instance, it directs user request to a nearby one.

The prerequisite of such a service hosting mechanism is to know the user experience of a *potential* service instance, before we decide to activate the instance and deliver the user requests to it. This is quite a challenging task, as there is generally no proactive connection between a user to the machine that will host the service instance. Measuring the user experience beforehand is hence impossible. We notice that the user experience of a cloud service depends heavily on the communication delay between the end user and the service instance the user accesses, which is mainly caused by the Internet delay between the user and the data center hosting the instance. We therefore propose a viable method to conveniently measure and predict such an Internet delay.

With the predicted user experiences, the service hosting problem is essentially how to redeploy a set of data centers for hosting the service instances, while guaranteeing the user experience for frequent users. We formulate it as a *k-median* problem and a *max k-cover* problem, which can be efficiently

978-0-7695-4460-1/11 $26.00 © 2011 IEEE
DOI 10.1109/CLOUD.2011.20

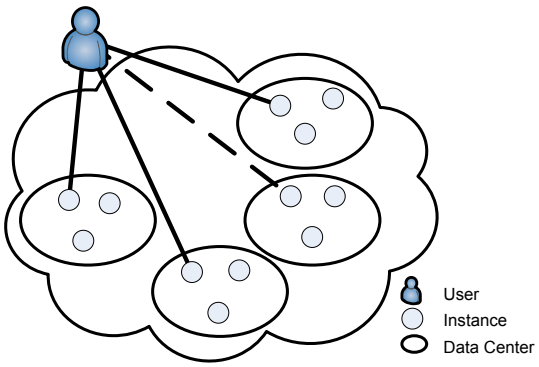

Figure 1. Framework of cloud-based services

solved by several algorithms we proposed in this paper. We evaluate our service hosting mechanism based on a large set of real-world data (roughly 130,000 accesses to Internet-based services). The results show that our mechanism can approach rapid and scalable cloud service hosting.

The rest of this paper is organized as follows. Section II overviews the cloud service hosting mechanism. Section III discusses the method of obtaining user experience. We present two different models and objective functions for modeling instance redeployment problem in Section IV. Section V conducts experiments and discusses the experimental results. Finally, the paper is concluded in Section VII .

II. Overview of Cloud-Based Services

A. Framework of Cloud-Based Services

Figure 1 shows the framework of cloud-based services. A cloud contains several data centers (eclipse in Figure 1). Physical machines are virtualized as instances in the data center. Service providers would deploy service running on these instances. An end user normally connects to the cloud to get data and run applications/services. User requests are directed to the service instances.

A good example is the Chrome OS developed by Google. Via such kind of light client, the end user can access data as well as application logic provided by the cloud as services.

Since the instances are inside the cloud, the connection information especially Round Trip Time (RTT) between a user and an instance can be kept by the cloud provider. The solid lines in Figure 1 represent RTT information being recorded between a user and some instances. Some links are not used thus the related RTT information is missing. We can derive new methods to predict these values. Dash lines in Figure 1 stands for this situation.

Generally user experience contains three elements: the Internet delay between user and cloud data center, the delay inside a data center in the cloud, and the time to process the service request. As machines in a data center are typically connected by gigabit links, delay inside a data center can be ignored. Moreover, the time to process the service requests is only affected by the computing ability of a service instance. As a result, the processing time is the same for two service instances. Hence, the user experience is mainly determined by the Internet delay.

B. Challenges of Hosting the Cloud Services

In order to attract users by low latency, service providers are concerning about where to deploy service instances in the cloud. The challenge of hosting the cloud service in the cloud stems from the difficulties of foreseeing user experience before actually running the service. So normally redeployment is in needed.

After the service running for a period, the Internet delay between users and every cloud data center can either be measured or be predicted. We describe this in Section III. This means in the cloud we can obtain all information regarding the potential positions for deploying service instances, while many existing computing infrastructures such as Internet services do not have such a feature. All the information is organized as a distance matrix. An element in the matrix is the distance value between a user and a data center.

Moreover, we notice the fact that the number of data centers is limited, while there is no bound for the number of services. So there are multiple services deployed in the data centers of a cloud. This fact suggests that we can use this measurement for optimizing any service in the cloud.

Consequently, we employ a distance matrix to formulate the redeployment problem as a *k-median* problem in Section IV-A. However, we have not taken the limiting resource of a single service instance into consideration. Another model of *max k-cover* problem is engaged to deal with this limitation and to make the model more realistic. We discuss this formulation in Section IV-B. Through solving these problems we can redeploy the service instances intelligently.

III. Obtaining User Experience

A. Measure the Internet delay

A user normally requests service in the cloud. A requests is responded by an instance inside the cloud thus the cloud provider is able to record the RTT from the user to the instance. This RTT value is kept as the distance from the user to the data center as the delay inside a data center can be ignored. The user generally calls several services and the related service instances are distributed in different data centers throughout the cloud. As a result we can get plenty of distance values between different users and data centers.

B. Predict the Internet delay

A user may not be able to visit many instances deployed in every data center. So we cannot get some distance data directly. We call it a missing value if the distance between a pair (user u, data center d) is not available. The technique given in [5] can be used to predict the missing values. The

Table I
ALPHABET OF PROBLEM MODEL

Descriptions	Notation
Number of data centers	M
Number of frequent users	N
Number of instances to deploy	k
Distance between user i and data center j	d_{ij}

idea goes as the following. Some users may come from the same place and use the network infrastructure of the same provider, thus their network performance is similar. We can examine the existing values to find these similar users of user u. As these similar users may have visited data center d already, we combine distance values between similar users and data center d intelligently to predict the missing value (user u, data center d). Technical details are in [5].

IV. REDEPLOYING SERVICE INSTANCES

Suppose a service provider p will provide a service s in the cloud, and the service will be distributed on k instances in the cloud due to budget restriction. At first, the cloud service hosting mechanism h can only guess where to place the k instances. But after a period of running service s, h knows a bunch of (totally N) users who use s frequently. The k instances can then be redeployed.

The problem is to redeploy k instances such that the result is optimal for the current N frequent users by considering the distance matrix we obtained in Section III[1]. The notations we used in the following sections are listed in Table I.

A. Minimize Average Cost

Suppose for a specific user u who would like to take the service s, our mechanism would direct the user to the closest of the k instances. We define *cost of user u* as the distance between u and the closest instance. Our objective is to minimize the average cost of N users. Note that N is fixed and the target is equivalent to minimize the total cost.

We formulate this problem as the followings:

Given:

$Z =$ the set of data centers

$C =$ the set of users

$d_{ij} =$ distance between every pair $i, j \in C \times Z$

Minimize:

$$\sum_{i=1}^{N} \min_{j \in Z'} \{d_{ij}\}$$

Subject to:

$$Z' \subset Z, |Z'| = k$$

[1]The network link may vary in different time. Average value of distance can be used in the algorithm. While we are not interested in the exact value of distance, our task is to deploy service instance. As long as the order of distances between a user and different data centers are preserved, our choice of data center is fine.

This is exactly the well known *k-median* problem, which is NP-hard. So we resort to the following fast approximation algorithms.

1) Brute Force: In a small scale (*e.g.*, select 3 instances from M potential data centers), it is possible to list all combinations. We call this *brute force* algorithm in our experiment in Section V. The complexity of this algorithm is $O(M^k \cdot N)$, where M, N and k follow the definition in Table I. If k is small, it can be computed in reasonable time.

2) Greedy Algorithm: Greedy algorithm runs as follows. Suppose we would choose k among M data centers to deploy the instances. In the first iteration, we evaluate each of M data centers individually to determine which one to choose first. We compute average distance from each data center to all users. The one achieving the smallest average cost will be chosen. In the second iteration, we search for another data center. Together with the first data center we have already chosen, the two data centers yield the smallest average cost. We do the iteration until we get k centers. This algorithm runs in $O(k \cdot M \cdot N)$ time.

3) Local Search Algorithm: Local search can provide the current best known bound for approximating *k-median* problem [6].

Algorithm 1 shows the approach. $Cost(S)$ in the algorithm means the average cost for all users. $s \in S, s' \notin S$ are two sets containing the same number of elements. $P(N, M)$ is a polynomial in M and N. The constraint $cost(S - s + s') \leq (1 - \frac{\epsilon}{p(N,M)}) * cost(S)$ is to guarantee that the algorithm terminates in finite steps.

Algorithm 1 Local search algorithm for k-median problem

1: $S \leftarrow$ an arbitrary feasible solution.
2: **while** $\exists S - s + s'$ such that,
$s \in S, s' \notin S$,
$cost(S - s + s') \leq (1 - \frac{\epsilon}{p(N,M)}) * cost(S)$ **do**
3: $S \leftarrow S - s + s'$
4: **end while**
5: **return** S

Assume sets s and s' are of size t. It is easy to verify that the algorithm runs in $O(l \cdot k^t \cdot M^t \cdot N)$ time where l is related to ϵ, and other notations are given in Table I.

On initializing, we use mainly two methods. The first one we utilize is the data centers selected by the greedy algorithm, and the second one is a random vector. As local search would naturally find the local optimum, output of the algorithm is always no worse than that of the original one. So the one initialized by the greedy algorithm would return a better or at least equivalent solution to the greedy result.

4) Random Algorithm: Random algorithm would randomly choose k out of M data centers from a uniform distribution. Every data center has the same possibility to be chosen. This is a simple algorithm. Generally the method to improve the performance of a random algorithm is to run it multiple times. The purpose of this algorithm is to

designate it as a base line of performance. So instead of running random algorithm at fixed times, we determine the times of running dynamically. For example, if we would like to know how good the greedy algorithm is in terms of time complexity and approximation rate, we would use a dynamic random algorithm for comparison. We record the run time of the greedy algorithm T_{greedy} and launch the random algorithm running several times with the total running time roughly equal to T_{greedy}. Consequently we could compare either their relative performance to the optimal choice or their average distance to all users directly. If the result is comparable then we argue that the greedy algorithm is not a good algorithm as its performance is no better than the random algorithm while the approach is more complicated, and vice versa.

B. Maximize Close User Amount

In the previous section, we set our target to minimize average cost of all users. However, in some cases part of the users may be extremely far away from most of the data centers. While considering minimizing the average cost, these users tend to force some service instances deployed in the data center close to them. This kind of users are called outliers in [7]. We could directly put the algorithm of k-median problem with outliers in [7] into practice in our case.

In this paper we employ another model to deal with the outliers. On considering QoS of a service, we believe it is unacceptable if some responses take a very long time. In this model we set a threshold value T for the response time. We try several values of the proper threshold in our experiment. If some $d_{ij} > T$, we drop this link by considering it disconnected. Moreover, to simplify the problem, we assume the user accepts the service as long as the response time is less than the threshold. Overall our target is to find k instances that satisfy as many users as possible. This time our mechanism directs the user to a light load and close enough but not necessary the closest service instance.

To model this situation, consider following problem:

Given Bipartite graph $B(V_1, V_2, E)$ where

$$|V_1| = M,\ |V_2| = N$$
$$i \in V_1, j \in V_2$$
$$\begin{cases} (i,j) \in E, & d_{ij} \le T; \\ (i,j) \notin E, & \text{otherwise.} \end{cases}$$

Maximize:

$$|N_B(V')|$$

Subject to:

$$V' \subset V_1,\ |V'| = k$$

$|N_B(V')|$ is the number of nodes in the neighbor set of V', meaning the target is to choose a subset of V_1 to cover as many vertices in V_2 as possible. Actually it is a *max k-cover* problem.

In this problem we construct M sets by setting i to be the ith vertex in V_1 and the elements in this set is vertices connected to i in V_2. We are trying to find k-sets to cover as many elements as possible.

Max k-cover problem is a classical problem and is well studied. This problem is NP hard hence we do not expect to achieve the exact answer. Again we use approximate algorithms. Greedy algorithm (Algorithm 2) is proven to be one of the best polynomial time algorithm for this problem [8]. It could give a $(1 - 1/e)$ approximation, which means it could cover at least $(1 - 1/e)$ of the maximum elements k-sets could cover.

In the greedy algorithm, every round we find a data center s which, when combined with current selection set S, could cover the maximum users. There may be more than one such data centers. In Algorithm 2 we choose one of them randomly. It can also be done more precisely as we can use a stack to record and try all choices one by one. However, we find that with the growing of k, the amount of branches increases so fast that the reward quickly diminishes. So we do not include this implementation in the paper.

Algorithm 2 Greedy algorithm for max k-cover problem

```
1:  S ← φ.
2:  while Have not covered all users yet & Used less than k
    instances do
3:      maxcover ← 1
4:      clear list l
5:      for all data centers s do
6:          if use s could cover c ≥ maxcover more users than use
            instances in S then
7:              clear list l
8:              add s to l
9:              maxcover ← c
10:         else if use s could cover exactly maxcover more users
            than use instances in S then
11:             add s to l
12:         end if
13:     end for
14:     random select s from l
15:     S ← S + s
16: end while
17: return S
```

Moreover this algorithm can be easily modified to restrict the number of users to whom an instance can be connected. In line 15 of Algorithm 2, instead of $S \leftarrow S + s$ we consider an integer $limit$ if s covers $\ge limit$ more users, we randomly pick up $limit$ number of users to cover. Moreover, we consider s cover $\ge limit$ more users as $= limit$. By these changes we can limit the connections and select a data center to deploy multiple instances if needed. This constraint can make the model more realistic.

Local search method would find local extreme solution thus can be used to help improving the greedy algorithm. Again we use single swap for the original greedy algorithm. However, for the modified algorithm swap is not that easy,

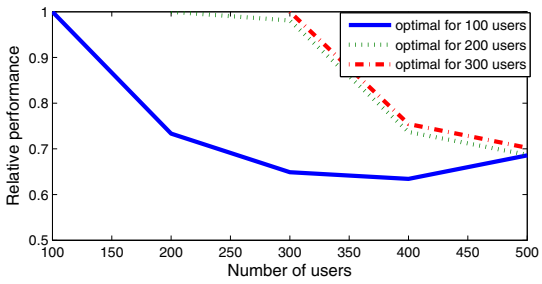

Figure 2. Worst case without redeployment

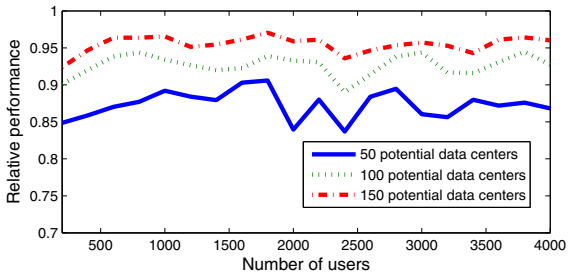

Figure 3. Deploy in limited data centers

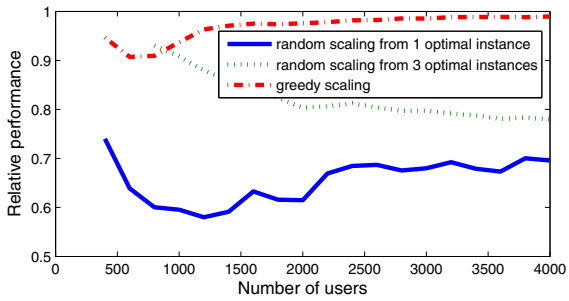

Figure 4. Auto scaling algorithms

so we launch the algorithm for several times to improve its performance.

V. Experiment and Discussion

In this section we conduct experiments to show the necessity of redeployment and to compare different algorithms.

A. Dataset Description

To obtain real-world response time values of different service instances, we implement a WSCrawler and a WSEvaluator using Java. Employing our WSCrawler, addresses of 4,302 openly-accessible services are obtained from the Internet. We deploy our WSEvaluator to 303 distributed computers of PlanetLab, which is a distributed test-bed made up of computers all over the world. In our experiment, each PlanetLab computer invokes all the Internet services for one time and the corresponding response-time value is recorded. By this real-world evaluation, we obtain a 303×4302 matrix containing response-time values.

B. Necessity of Redeployment

The first thing we should do is to demonstrate the necessity of redeploying service instances.

We fix the instance number to 3 and scale the user from 100 to 500. We repeat the experiment for 100 times and find in the worst case the performance is really bad. This is shown in Figure 2. If we deploy the optimal service instances for 100 users, then without redeployment the performance may decrease below 70% of the optimal. The same situation occurs when deploying optimal instances for 200 or 300 users. In worse cases, far away users would cost the main part of the average cost. So the performance of three worse cases tend to be closer. Without redeployment the network performance may discourse the new users. To avoid the worst case from happening, redeployment or at least a performance checking is required.

C. Weakness of Auto Scaling

As discussed before, current cloud service would apply auto scaling in responding to the change of user scale. So it is natural to ask whether it is good enough to simply apply this service without redeployment. We indicate there are two main drawbacks of current auto scaling approaches which can be shown in Figure 3 and Figure 4.

Current auto scaling is limited within one region. We simulate this by selecting out only a part of data centers as the potential data centers to deploy instances. Using the same algorithm to deploy instances, we can see that the average user cost is higher to choose instances from partial data centers than that of choosing from all data centers. Figure 3 shows the result. We pick out 50, 100, 150 data centers from totally 303 data centers and deploy 10 instances in these data centers. We apply the *greedy + single swap* algorithm and the result of using total 303 potential data centers is employed as the baseline. The performance decreases greatly if we choose only 50 potential data centers, where the performance is not quite related to the number of users. Note the distance between users and data centers is better preserved if we choose 100 (33%) or more from the total data centers, and the performance is greatly improved. It is due to the fact that we have some data centers which are close to many users. The average distance would decrease greatly if we could pick up these data centers.

Another disadvantage of current auto scaling approach is that it launches new instances in the data center containing least instances. It is like randomly picking up a new data center to grow. We simulate randomized scaling and the result is in Figure 4. On a small scale (*e.g.*, select no more than 4 instances), we consider all possible combination and find the optimal choice. On a larger case, we use again the *greedy + single swap* algorithm as our best choice. We start from one optimal instance and 200 users. Then we randomly

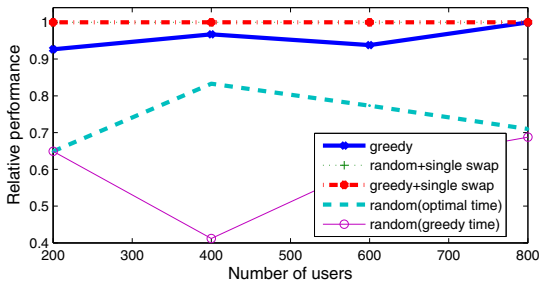

Figure 5. Selecting 3 data centers by redeployment algorithm

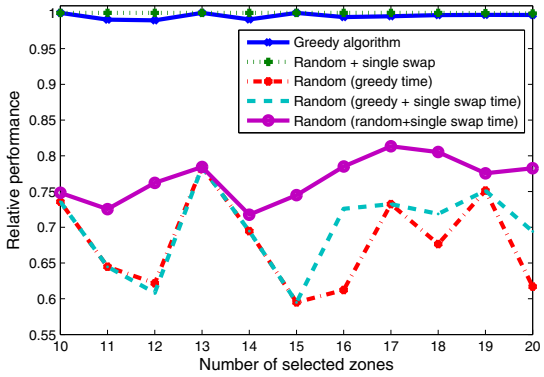

Figure 6. Selecting 10 - 20 data centers for 4000 users

add 200 users at each step and select one more instance. We see the performance is bad in this situation. If we defer the random selecting after we find 3 optimal service instances for 600 users, we would obtain a better choice than doing it in the beginning. As we see in this case, if we could do the scaling a little more intelligently by using a simple greedy algorithm, we can get a very good result. However, there is still some problem by using greedy algorithm, such as some instances would get extremely heavy load. We would discuss this issue later.

D. Comparing the Redeployment Algorithms for k-Median

The previous experiment confirms the need for redeployment. As we discussed in Section IV-A, we formulate the redeployment as *k-median* problem and compare the algorithms we proposed. First we set k to be 3 and M to be 303. We compare the algorithm output with the optimal output generated by the brute force algorithm.

We use the optimal average cost as the baseline and compare the greedy algorithm output. On applying local search (single swap) algorithm we use the output obtained by the greedy algorithm and a random vector as initialization. Notice in Figure 5, the outputs generated by the random algorithm with the time equivalent to the greedy and the local search algorithms are the same. It is because they run in so short a time that the pseudo-random outcomes remain the same if the system time is used as the seed to generate pseudo-random values. From Figure 5 we know in such a small scale, the greedy algorithm is good enough. The local search algorithm would be even better. Moreover, they are better choice in terms of time complexity as they perform better than the random algorithm.

When selecting more than 10 instances the brute force algorithm would run in unacceptably long time, and we use the *greedy + single swap* algorithm as the baseline to observe the relative performance of the greedy and the *random + single swap* algorithms. Once more we compare them with the random algorithm. We select 10 to 20 data centers to deploy instances so as to satisfy 4000 users. The result is shown in Figure 6. The *random + single swap* algorithm is slightly better than the greedy algorithm, but it takes more time to compute. Again, the random algorithm fails to give a satisfiable answer, which means both the greedy and the local search algorithms are well suited.

Table II
TIME COMPLEXITY OF THE ALGORITHMS (UNIT:CPU CLOCK)

	Brute Force	**Greedy**	**Greedy + single swap**	**Random + single swap**
(2,200)	203	0	0	16
(2,400)	375	0	16	31
(2,600)	547	15	15	47
(2,800)	735	0	31	31
(3,600)	78969	0	31	63
(10,4000)	-	94	328	2641
(15,4000)	-	172	500	13109
(20,4000)	-	203	1906	25469

Table II lists some typical computing times for running these algorithms. The pair (a, b) in the table (*e.g*, $(2, 400)$) stand for selecting a service instances to satisfy b users. Values in the table are the counts of the cpu clock. We are interested in their comparative relation. We can see that the brute force algorithm does cost with an exponential time growth. The greedy as well as the *greedy + single swap* algorithms run quite fast comparing to the *random + single swap* algorithm because normally we expect more swaps to find a local optimizer by a random initial vector.

E. Redeployment Algorithms for Max k-Cover

Through experiments we confirm the *greedy + single swap* algorithm is good enough in both result and time complexity to solve *k-median* problem. This makes the redeployment simple; however, there is a limitation of *k-median* model. Recall on formulating redeployment in *k-median* problem we have assumed users are evenly distributed and all users can connect to their nearest service instance. In practice this condition does not always hold.

As an illustration, Figure 7 shows a typical distribution of user connections on the service instances. 20 instances are selected to provide service for 4000 users. We expect all instances to connect to about 200 users each, while the real

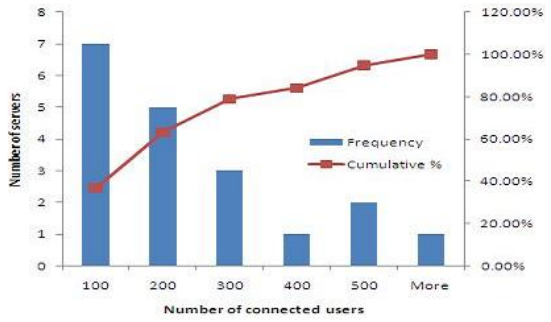

Figure 7. Histogram on number of connected users for each server

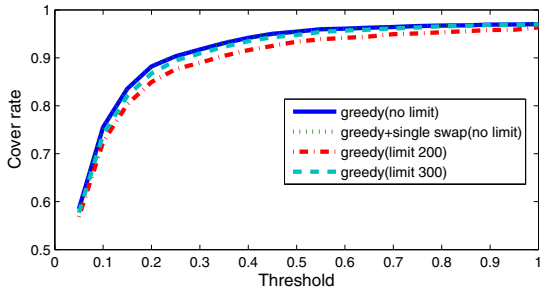

Figure 8. Max k-cover using greedy approach

case is that some instances connect to up to 400 or more users. Therefore these instances get very heavy load and the performance would become unsatisfiable.

The cloud has virtually infinite resource while we have only limited budget. We can only launch limited number of (k) instances in the cloud. If we would like to ease the burden of some instances we are required to put some extra instance nearby and to distribute service instances in k' (where $k' < k$) data centers.

To attack this, we use another model to formulate the problem which is *max k-cover* problem. As discussed in Section IV-B, the threshold to cut edges is a key parameter we should pick up carefully. We employ the entire 303×4302 matrix in our experiment and test the value of the threshold ranging from 0.05 to 1. We select data centers to deploy 20 service instances. Two typical values of limitation are applied. The first one is 215, with $20 * 215 = 4300$, which means we can cover at most 4300 users. The second one is 300 which is a loose limitation. The result is shown in Figure 8. The local search (with single swap) has not much improvement over the greedy algorithm in this problem. A tight limitation (*e.g.*, 200) will affect the performance of coverage; on the other hand loosen the limitation a little, the performance will be close to that of no limitation.

Since we cut the edges with weight less than the threshold, the average cost of the covered users is less than that of the threshold. So the threshold to some extent is related to the average cost of the users. To this end, we compared the average cost of users by the instances we selected using

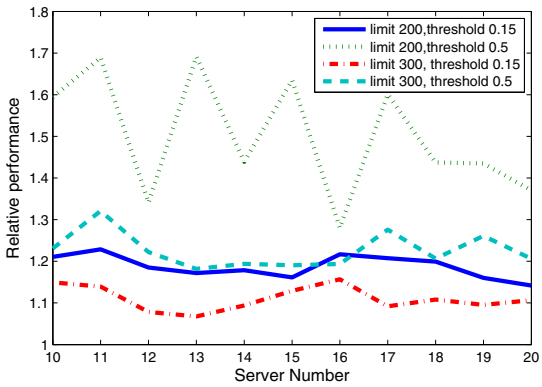

Figure 9. Average cost by max k-cover model

the greedy algorithm for *max k-cover* problem to the cost by instances we selected using the algorithms for *k-median* problem. We use the result generated by the *greedy + single swap* algorithm of *k-median* model as baseline. The result is shown in Figure 9. During the experiment we changed k, the number of instances to choose, and accordingly we set the total number of users to $200 * k$. We pick up two typical values of limitation and threshold. On setting the limitation we use a tight limitation of 200 and a loose one of 300 as done previously. As for the threshold, from previous experiment we get the average cost of above 0.1 using *greedy + single swap* algorithm. So we use 0.15 as the threshold to ensure the cost of the covered users is below 0.15. We note using the threshold of 0.5 can cover most users in *max k-cover* model. So we employ these two values in the experiment. The result shows a lower threshold would do better than a higher one. Although this limitation decreases the performance, the resulting model is more realistic.

VI. Related Work

A. k-Median Problem

Jain and Vazirani [9] provided a 6-approximation for *k-median* problem. Their algorithm used the lagrangian relaxation technique. Their main contribution is that they used the algorithm for facility location problem as a subroutine to solve *k-median* problem. They proved that an lagrangian multiplier preserving α-approximation algorithm for the facility location problem gives rise to a 2α-approximation algorithm for the *k-median* problem. Based on this approach many improvements have been done. Charikar and Guha [10] used a similar idea and achieved a 4-approximation algorithm for *k-median* problem. Jain et al. [11] obtained a new greedy approach for facility location problems. Through improving the subroutine they also got a 4-approximation by the same procedure of the previous algorithm.

Arya et al. [6] first analyzed the local search algorithm for *k-median* and provided a bounded performance guarantee. The algorithm is not complex as we used in Section IV-A.

This algorithm can approximate the optimal solution to the ratio $3+\varepsilon$ and this is the best result known yet.

B. Max k-Cover Problem

Max k-cover problem is related to set cover problem. Many algorithms have been purposed (*e.g.*, [12], [13], [8]). Greedy algorithm [8] is proven to be one of the best polynomial time algorithm for this problem. It gives a $(1-1/e)$ approximation.

C. Web Server Placement

Qiu et al. [14] studied placement of web server replicas by using serval placement algorithms. They provided a good insight to evaluate the algorithm by comparing the output to the result of super-optimal algorithm.

Recently Zhang et al. [15] studied the placement problem in shared service hosting infrastructures. Instead of modeling placement problem as *k-median*, they formulated it as similar to capacitated facility location problem. They defined their own penalty cost instead of using response time directly. However, the previous studies are not specially tailored for cloud computing.

VII. Conclusion and Future Work

In this paper, we highlight the problem of hosting the cloud services. Our work consists two parts. First we propose a framework to address the new features of cloud. In the cloud we can get all possible connection information on potential locations that the service instances could be deployed. The second work is that we formulate the redeployment of service instances as *k-median* and *max k-cover* problems. By doing so, they can be solved by fast approximate algorithms.

We employ a large data set collected via real-world measurement to evaluate the algorithms. The result shows our algorithms work well on the this problem.

In this work, we mainly focus on putting forth the service redeployment problem. Thus we only consider relatively simple cases. The most critical issue we could take into consideration in the future work is to set the cost more consciously on over-connected instances. We only set a limitation of connection in our current work. If an instance is overloaded the response time would not be the same as another instance of that data center. Thus we need to formulate the network capability of service instance carefully with the amount of users.

We believe there exist some patterns of using services. It is possible to figure out potential users. Thus another future work is to optimize initial service instances deployment.

Acknowledgment

The work described in this paper was supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK4154/10E) and sponsored in part by the National Basic Research Program of China (973) under Grant No. 2011CB302600.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *ACM Communication*, vol. 53, no. 4, pp. 50–58, 2010.

[2] L. Qian, Z. Luo, Y. Du, and L. Guo, "Cloud computing: An overview," in *Proc. 1st International Conference on Cloud Computing*, 2009, pp. 626–631.

[3] (2010, Aug.) Amazon auto scaling developer guide. http://aws.amazon.com/documentation/autoscaling/as-dg.pdf.

[4] (2010, Jul.) Elastic load balancing developer guide. http://aws.amazon.com/documentation/elasticloadbalancing/elb-dg.pdf.

[5] H. Ma, I. King, and M. R. Lyu, "Effective missing data prediction for collaborative filtering," in *Proc. 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2007, pp. 39–46.

[6] V. Arya, N. Garg, R. Khandekar, A. Meyerson, K. Munagala, and V. Pandit, "Local search heuristic for k-median and facility location problems," in *Proc. 33rd Annual ACM Symposium on Theory of Computing*, 2001, pp. 21–29.

[7] M. Charikar, S. Khuller, D. M. Mount, and G. Narasimhan, "Algorithms for facility location problems with outliers," in *Proc. 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2001, pp. 642–651.

[8] U. Feige, "A threshold of ln n for approximating set cover," *Journal of the ACM*, vol. 45, pp. 634–652, July 1998.

[9] K. Jain and V. V. Vazirani, "Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and lagrangian relaxation," *Journal of the ACM*, vol. 48, pp. 274–296, March 2001.

[10] M. Charikar and S. Guha, "Improved combinatorial algorithms for the facility location and k-median problems," in *Proc. 40th Annual Symposium on Foundations of Computer Science*, 1999, pp. 378–388.

[11] K. Jain, M. Mahdian, and A. Saberi, "A new greedy approach for facility location problems," in *Proc. 34th Annual ACM Symposium on Theory of Computing*, 2002, pp. 731–740.

[12] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of Operations Research*, vol. 4, no. 3, pp. 233–235, 1979.

[13] L. Lovász, "On the ratio of optimal integral and fractional covers," *Discrete Mathematics*, vol. 13, no. 4, pp. 383–390, 1975.

[14] L. Qiu, V. Padmanabhan, and G. Voelker, "On the placement of web server replicas," in *Proc. of 20th IEEE International Conference on Computer Communications*, vol. 3, pp. 1587–1596.

[15] Q. Zhang, J. Xiao, E. Grses, M. Karsten, and R. Boutaba, "Dynamic service placement in shared service hosting infrastructures," in *NETWORKING 2010*, ser. Lecture Notes in Computer Science, 2010, vol. 6091, pp. 251–264.