Adaptive Service Placement in Dynamic Service Hosting Environments

Qi Zhang Jin Xiao Eren Gürses
Martin Karsten Raouf Boutaba

University of Waterloo

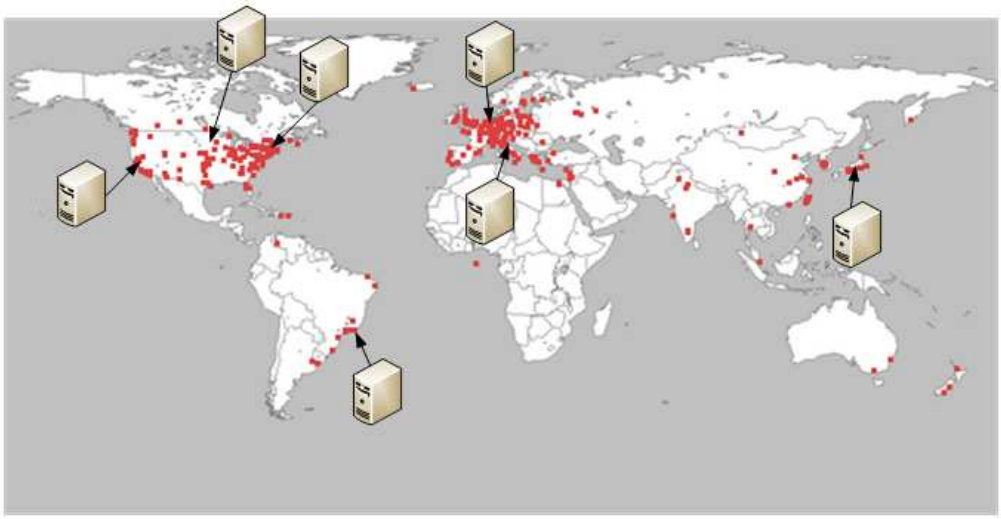
May 5, 2010

Introduction

- Large-scale service hosting environments have gained popularity in recent years
  - Content Delivery Networks (CDNs)
  - P2P networks
  - Service overlays
  - Grid and Cloud computing
- Features and characteristics
  - Decoupling the ownership and the use of resources
    - Infrastructure Provider vs. Service Provider
  - Scaling up and down by adjusting the number of servers
  - A cost is Usually associated with running a server

We are seeing more and more service hosting platforms deployed on Internet these days. Some examples are Content delivery networks (CDNs), P2P networks, service overlays, Grid computing and more recently Cloud computing. All of the above systems share the following similarities: First, the traditional role of service provider has been divided into two: the infrastructure providers who own the resources and service providers who use the resource to run applications. Second, a service provider can dynamically adjust the service capacity by increasing and decreasing the number of servers (a.k.a. service instances) used to serve the demand. Third, there is usually a cost associated with running a server for given period of time.

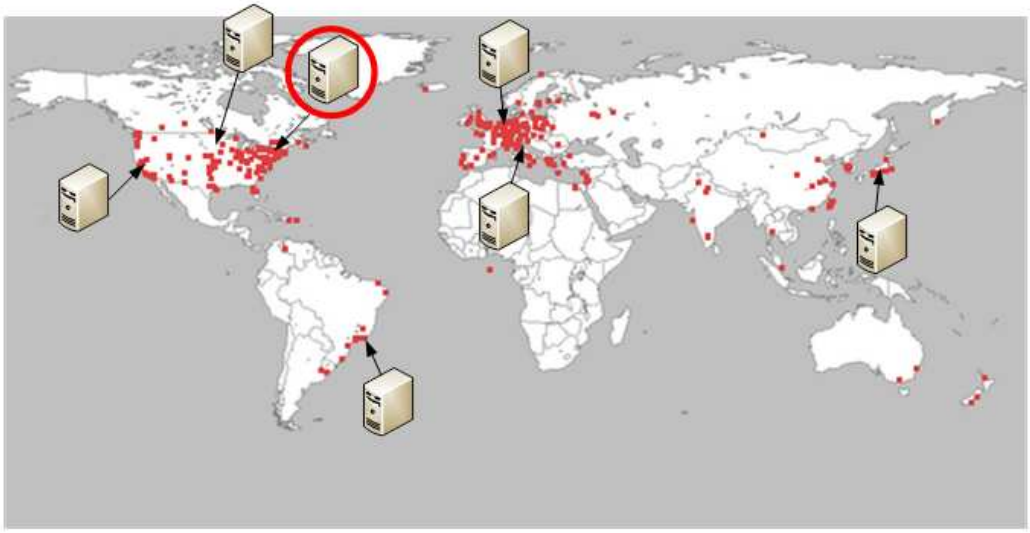
Example: PlanetLab



- Placing service instances ~~are~~ strategic locations can reduce operating cost and improve performance

We use PlanetLab as an example. PlanetLab currently consists of 1000+ nodes located at 500+ global-wide locations. Each node has a finite capacity that limits the number of applications it can host.
Given a large number of candidate locations, a service provider must decide where servers should be placed. A common practice is to place them close to demand so as to minimize the response time. This is illustrated in figure above. Assume the red dots indicate the location of the demands, we want to place the servers as close to the demand as possible, while keeping the number of servers low.

Example Scenario



Now, suppose the demand in US-east region has increased. This may affect the response time in this region since there aren't enough servers to handle the increase in demand.

Example Scenario

Now suppose we launch a new server in the same region. Still we need to decide how we want to divide the demand among the servers.

Example Scenario

Final configuration.

Service Placement Problem

- Mainly deal with latency sensitive services such as content delivery and real-time applications
- Optimizing the placement of services to achieve Service Level Objectives (SLOs) like bounded response time, while minimizing the cost of using resources
- Need to adapt placement configurations to system/network dynamicity
  - Service demand can fluctuate
  - Network/software failure can occur
- A distributed solution is preferred
  - Reduce the overhead of collecting global information and computing global solution

The service provider's objective is to minimize total cost while meeting SLOs. However, as the operating environment evolve, placement configurations can become stale. We want to dynamically adjust the placement configuration according to changes in the environment. To reduce the communication and computation overhead, a distributed algorithm is preferred.

Related Work

- The service placement problem has been studied previously
  - Usually formulated as either a facility location problem (FLP) or a k-median problem
  - NP-hard to solve
- Many heuristics are available
  - Greedy heuristics
  - Fan-out heuristics
  - linear programming based heuristics
- These heuristics are centralized and only ~~works~~ for static graphs
- Recently several distributed heuristics have been proposed
  - Capacity constraint is not considered
  - No performance guarantee

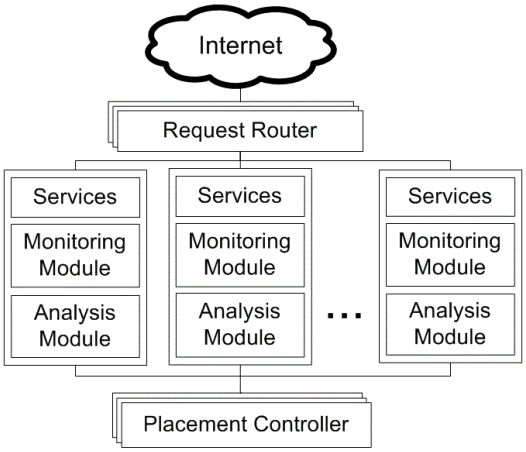
The service placement problem has been studied extensively in the past. So why are we still studying it? The main reason is that most of the existing approaches have limitations. SPP is often formulated as a variant of facility location problem or k-median problem, both problems are NP-hard. There are a lot of heuristics designed for this problem, but there are 3 main drawbacks: (1) Most of heuristics (e.g. greedy heuristic) is centralized. (2) Most of heuristics cannot handle dynamicity. (3) Recently there have been a few papers reporting distributed algorithms. However, they either ignore capacity constraint, or does not provide any performance guarantee.

Our Contribution

- We present a distributed approximation algorithm for this problem
  - Built on the $(9+\epsilon)$ approximation algorithm for capacitated facility location problem (CFLP)
  - Incrementally improve the solution quality through replication and migration
- Suitable for large scale and dynamic environments

Our contribution

System Architecture



- Each server maintains a list of neighborhood servers and candidate locations
- Request routers assign each request to the best available server in a greedy fashion
- Request routers may need to estimate the future demand from each location
  - We can also use the estimated values in our computation

Here is the general system architecture for which our algorithm is designed. Assumptions: the request routers assign demand to service instances in a greedy fashion (by computing the response time as function of distance and server load). Each server maintains a list of neighborhood servers and candidate locations. Notice that there are many possible realizations of the system. For example, CDNs typically use DNS servers as request routers. In P2P systems, peers can act as request routers. To handle rapid increase in demand, sometimes it is necessary to estimate the future demand and plan ahead accordingly (part of our future work). This is also supported by our framework.

Problem Formulation

- ~~Give~~ a bipartite graph $G = (D, F, E)$, where
  - D is the set of demands
  - F is the set of candidate server locations
  - E is a set of edges connecting D and F
- Our goal is to select a subset of servers $S \subseteq F$ and assign D to S, minimizing total operating cost, which is ~~sum~~ of
  - Resource usage cost $C_f(S)$
  - SLO penalty cost $C_s(S)$

Formulation of the problem

Placement Objective

- The response time of a request i served by a server s_i can be computed as:

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

- The penalty cost can be computed as:

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

- The goal of the service placement problem is to minimize

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

The response time of a request consists of network latency and queuing delay
a is a monetary penalty cost
We use a quadratic function because it reflects the general form of the penalty payout for SLA violation.

Local Search Algorithm for CFLP

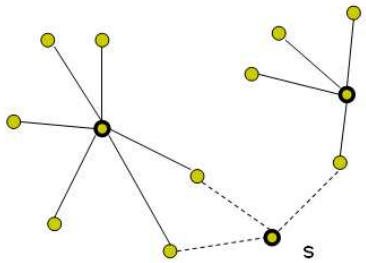
Algorithm 1 $(9+\epsilon)$ Approximtion Algorithm for CFLP

1: **while** $\exists$ an add(s), open(s, T), close(s, T) that reduces the cost by at least ϵ **do**
2: perform the operation
3: **end while**

- Add(s): open s and assign some clients to s
- Open(s, T): open s and close a set of servers T, and assign clients of T to s
- Close(s, T): close s and open a set of servers T, and assign clients of s to T

This algorithm starts from any feasible initial solution, and incrementally improves the solution with one of three types of operations: Add(s), Open(s, T) and Close(s, T).
Open s means installing a server at location s, Close s means uninstalling a service at location s. Notice that in our algorithm, a server can be be opened multiple times (subsequent openings do not perform installation), but closed only once.
This algorithm fits our objective very well, since it is both simple and adaptive to changes. However, this algorithm cannot be directly implemented in a distribute way, because finding an open(s, T) and a close(s, T) requires solving NP-hard optimization problems. The authors of the paper solved both problems using dynamic programming. However dynamic programming is both expensive to compute and requires global knowledge. Our contribution is to show that we can replace the dynamic programming procedures by simples approximation algorithms, such that the resulting algorithm is distributed and still providing an approximation guarantee.

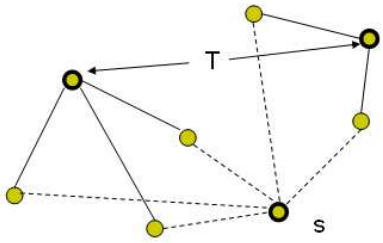
Implementing add(s)



- Open s and assign some clients to s
- It can be shown that the distance between s and s_i is at most $d(i, s_i) + \sqrt{\frac{cp(i,s_i)}{a}} \cdot d_{max}$

This slide explains how add(s) is implemented. s_i only need to talk to servers within radius $d(i, s_i) + \sqrt{\frac{cp(i,s_i)}{a}} \cdot d_{max}$.
(An interesting fact is that request router is actually performing add(s) as request arrives. In this case, the demand to be assigned is a single request, and s is the server that the demand is assigned to.)

Implementing open(s, T)



- Open s and close a set of servers T, and assign clients of T to s
- Reduces to a 0-1 knapsack problem: Select a set of servers T to maximize total cost reduction

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

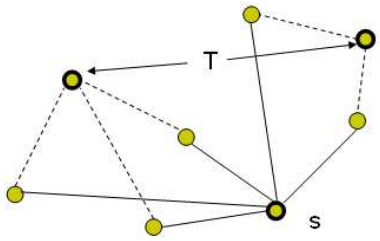
This slide explains how open(s, T) is performed

Implementing open(s, T)

- Replace dynamic programming procedure by a simple greedy algorithm:
  - The cost efficiency of a server t is defined as
    $$\text{costEff}_{open}(t) = \frac{p_t}{|C_t|} - cp(s, t)$$
  - Simply add servers in increasing order of their cost efficiencies until s is full.
- each server t only ~~need~~ to contact neighborhood servers with distance less than $\sqrt{\frac{p_t}{a \cdot |C_t|}} \cdot d_{max}$. A neighborhood server s can then compute an admissible open(s, T) once it discovers the entire T.

We replace the dynamic programming procedure by a simple greedy algorithm with approximation ratio $\frac{1}{2}$.

Implementing close(s, T)



- close s and open a set of servers T and assign demand u_t of s to T to maximize

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

- Can be rewritten as a single node capacitated facility location (SNCFL) problem:

$$cost_{SNCFL} = \min_{T \subseteq F \setminus S} \sum_{t \in T}(p_t + u_t cp(s, t))$$

SNCFL is also NP-hard, and can be solve optimally in pseudo-polynomial time using dynamic programming.

u_t denote the number of requests that will be assigned to t.

Implementing close(s, T)

- Again, we replace the dynamic programming procedure by a simple greedy algorithm
  - The cost efficiency of a server t is defined as
    $$\text{costEff}_{close}(t) = \frac{p_t}{cap_t} + cp(s, t)$$
  - Sort the servers in increasing order of their cost efficiencies
  - Greedily add servers into the set T until there are enough servers to handle all the demands of s. Then we have a candidate solution
  - For each candidate solution, we can remove the last server and then continue adding servers after the last server to get the next candidate solution
  - The best among all candidate solutions is our final solution
- s only needs to contact servers within radius $\sqrt{\frac{p_s}{a}} \cdot d_{max}$.

This greedy algorithm has an approximation ratio 2.

Analyzing the approximation ratio

- Need to show that the new algorithm still achieves an approximation guarantee
- It can be proved that the greedy algorithms achieve a weaker performance guarantee
  - Compared to the original algorithm, our algorithm can miss some open and close moves
- ~~Follow~~ a similar analysis, we can show that our algorithm achieves an constant approximation factor

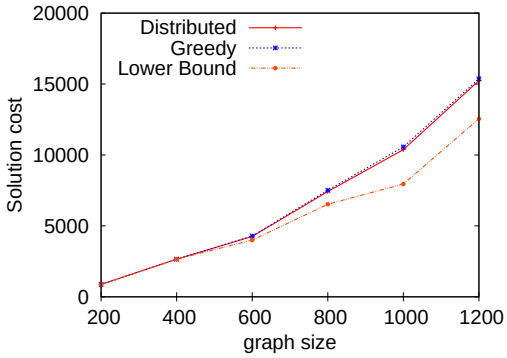
More details can be found in the paper.

Experiments

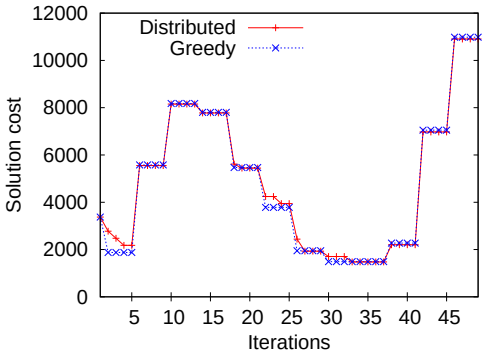
- We conducted several experiments to evaluate the performance of our algorithm
  - Static topologies generated using GTITM topology generator
  - Rocketfuel topology with demand fluctuation
- For benchmarking purpose, we also implemented a well-known greedy placement algorithm
  - has been shown to perform well in practice

The greedy algorithm is centralized, and only works on static graphs

Experiments



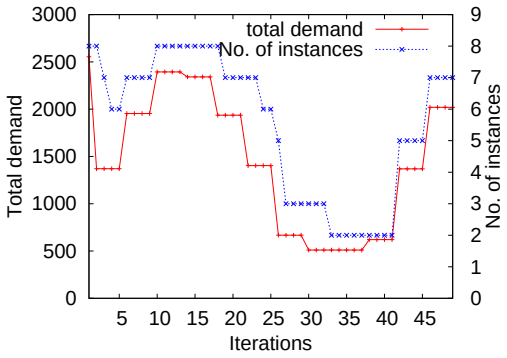
Performance on static topologies



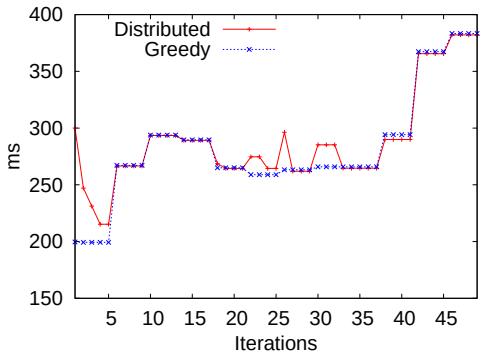
Performance on Rocketfuel topology

Parameters: $d_{max} = 400ms, \mu_s = 20ms, a = 1, p_s = 4$.
The lower bound is computed by exhaustive search. We also ignored capacity constraint to speedup the search.
We first evaluated our algorithm using topologies generated from GTITM. The topologies are static (i.e. no demand fluctuation). it can be observed that our algorithm performs as well as the centralized greedy algorithm in most of the cases, and sometimes better the greedy algorithm
We then evaluated out algorithm using Rocketfuel topology. The demands are configured to fluctuate in a (roughly) periodic pattern. We also implemented the greedy algorithm that has access to global knowledge and can solve the problem instantaneously. The figure on the right depicts the outcome of a typical run. Our algorithm performs almost as good as the greedy algorithm

Experiments



Runtime service capacity



Average connection cost

More diagrams for the same test run. We observe that our algorithm can indeed add and remove instances according to demand fluctuation. The connection cost (SLO penalty) is also comparable to the solution produced by the greedy algorithm.

Conclusion

- As service hosting environments gain their popularity, ~~optimally~~ resource allocation becomes a key problem
- We presented a distributed algorithm for this problem
  - Computing improvement moves using local knowledge
  - Practical for implementation
- Future work
  - Implement our algorithm in PlanetLab or real P2P networks
  - Study techniques for demand prediction
  - Analyze the impact of dynamic reconfiguration
  - Extend our work to more generalized settings

Our algorithm is also applicable to other domains such as gateway placement in wireless networks