Probabilistic Scalable Application Placement in Distributed Systems

Roy Friedman        Svetlana Rubanovich
Department of Computer Science
The Technion
Haifa 32000
Israel

December 28, 2000

Abstract

This paper investigates a scalable probabilistic scheme for placing applications in large distributed systems like the Internet. The scheme proposed here assumes that for each application there is a set of good and bad potential locations. This is used to derive a randomized protocol that places an application in a good location with high probability in a constant number of steps. The protocol is then extended and simulated for the case where there are multiple applications. It is also shown how to adapt these results to the problem of placing a virtual server near its clients. Finally, the competitive ratio of the basic algorithm is computed, and is shown to be less than \( \min(1 + \frac{y}{d}, 2) \), where \( y \) is the cost of moving from one location to another, and \( d \) is the cost paid for each time unit spent in the initial location.

1 Introduction

The Internet offers a potential for global services that can be accessed remotely from anywhere in the world. However, to fully realize this potential, such services must always provide fast and reliable connection to their clients. A virtual server, that is, a server that can transparently migrate and replicate itself in the Internet is an appealing concept in that respect [7, 20]. The ability of a virtual server to migrate in order to be close to its clients is useful in order to minimize the latency experienced by clients, and improve the reliability of their connections to the service.

Given that the Internet operates around the clock, the locale of clients accessing a particular service might change depending on the time of day. In particular, news services are

*This research is partially supported by the Israeli Ministry of Science, Basic Infrastructure Fund, Project #9762.
likely to busy during business hours, while entertainment services are likely to busy during the evening. Thus, the part of the world in which it is business hours, or evening hours, changes as the day advances, and so does the locale from which clients try to access the corresponding sites. However, special events might trigger an influx of clients to a service in a non-deterministic manner. Hence, it is preferable to build mechanisms that control the locations of global Internet services in a way that can adapt to dynamic and unpredictable changes. Moreover, due to the enormous size of the Internet, such mechanisms should be highly scalable, distributed, and asynchronous.

In this paper we report the results of investigating one such mechanism for placing services in a large scale distributed systems. In fact, in our work we assume a somewhat generalized formulation of the above problem, in which we try to place applications in good locations, whose size is some percentage of all possible locations. That is, we assume that for each application there is a set of good locations, which can correspond to areas in the Internet that are close to clients in the special case of Internet services. The goal of our mechanism is to find one of the good locations and place the application there.

In our work, we start by looking at a single application. We propose two versions of a protocol in which the application randomly picks a candidate location and moves to it if it is better than the current location. We analyze this protocol, and show that with high probability, the application will be placed in a good location after a finite number of steps. We then extend this idea in three directions. First, we explain how this protocol can be used to solve the global service placement problem based on sampling access log files. Second, we examine by simulation the behavior of this protocol when there are multiple partially competing applications. Finally, we compute the competitive ratio of the proposed algorithm against an off-line algorithm that knows the sequence of locations probed.

The rest of this paper is organized as follows: Section 2 details related work in this area. Section 3 considers the problem of placing a single application in a good location with high probability, while Section 4 expands the discussion to multiple applications. Section 5 proposes a two-phase scalable protocol for migrating virtual servers, based on the results of the previous sections. Section 6 computes the competitive ratio of the basic protocol introduced in Section 3. We conclude with a discussion in Section 7.

2 Related Work

The application placement problem is a special case of the general task scheduling problem, which has been studied at great length. One of the principle results of scheduling theory research is that optimally solving general forms of scheduling is NP-hard [9]. In practice, this means that optimally solving most scheduling problems (or in our case placement problem) is intractable. Fortunately, in most practical situations the optimal solution for application placement is not needed and some approximate solution is considered good enough.

There are different approaches for solving the task scheduling problem. The diffusive approach, also called nearest-neighbor approach [6], is based on the following idea. First, distribute the tasks among computers organized in continuous blocks. If in the end there are too many tasks in a processor, move some tasks to a neighboring computer that has fewer tasks. A similar idea is used by [8]; in their approach each process equalizes its load with
its neighbors. The advantage of these approaches is that they work for arbitrary number of processors. However, in each step a computer communicates only with its neighbors. Thus, it takes too much time to distribute the load within a large system, and therefore not applicable for a network containing a large number of LANs.

To expand load balancing schemes to interconnected LANs, a two level algorithm for systems that support group communication was proposed in [11]. At the LAN level, all computers in a LAN maintain loading information in the entire LAN using a globally ordered group channel. The load in different LANs is balanced by moving applications from overloaded LANs to underloaded ones by exchanging information with neighboring LANs. This approach decreases the efforts of maintaining the load information within the LAN, but still requires point-to-point connection of LANs for exchanging their loads, which is not applicable to large networks of interconnected LANs.

A more flexible and scalable approach is suggested by Kremien et al. in [13]. They proposed a load sharing algorithm that partitions a system into sets of nodes, called domains. Each node dynamically and adaptively selects other nodes to be included in its domain from the whole system. In each domain load sharing is applied independently of how it is applied within other domains.

There are a number of solutions that try to support the needs of large wide area heterogeneous computer systems. Legion [12, 17, 21] is an object based software project designed for a system of millions of hosts and objects tied together with high speed links. Users working on their home machines see the illusion of a single computer with access to all kinds of data and physical resources. Groups of users can construct shared virtual work spaces to collaborate research and exchange information. Legion’s users require wide range of services in many different dimensions. To support this, Legion provides such system level services as naming, binding and migration. The goal of migration service is to determine how to place applications onto processors when they are needed, so that they can perform their tasks. As part of the Legion project, a general framework for object/task placement has been developed. This framework requires certain interface to all Legion objects, but implementation of the interface is left up to the object, so users can implement their own migration policy.

Another project that exploits heterogeneous supercomputing resources is Globe [16, 20], which provides an object based framework for developing wide area distributed applications. Globe allows construction of worldwide scalable objects that a vast number of processes can share. These objects are physically distributed and each object fully encapsulates its own policy for replication, migration, and so on. One of the examples of using such distributed objects is implementation of worldwide scalable Web documents.

The Symphony project [7] is a management infrastructure for executing virtual servers in the Internet. Virtual servers are distributed servers that can grow and shrink in the number of machines they are running on, and migrate from one place to the other in the Internet. Symphony allows organizations to share their computing resources at a global level, in order to maximize their utilization. One of the important concerns in Symphony, which influences a virtual server’s performance, is determining how to move or replicate an application when it is needed. The work reported in this paper is part of the Symphony effort.

Another related work is the problem of placing balls into bins. Suppose that $n$ balls are thrown into $n$ bins, such that each ball chooses a bin independently and uniformly at
random, and the bins are empty initially. Let the load of a bin be the number of balls in that bin after all balls have been thrown. It was shown [1, 2] that with high probability (by which we mean $1 - O(1/n)$), the maximal load of any bin is $\Theta(\frac{\log n}{\log \log n})$. An important extension of this result was proved by Azar et al. [2]. Suppose that the balls are placed sequentially, so that for each ball we choose two bins independently and uniformly at random and place the ball into the emptier bin (breaking ties arbitrarily). In this case, the maximal load drops to $\Theta(\frac{\log \log n}{\log 2})$ with high probability. Thus, having two choices instead of a single choice leads to an exponential improvement in the maximal load. Having more than two choices further improves the maximal load by only a constant factor. That is, if each ball has $d$ choices, then the maximal load is $\Theta(\frac{\log \log n}{\log d})$ with high probability. The method of Azar et al. [2] requires the resting place of the balls to be determine sequentially. This limits its applicability in parallel and distributed settings, a major drawback when compared to the simple randomized approach. Adler et al. [1, 14, 15] have expanded the approach of Azar et al. to parallel and distributed settings and proposed a parallel version of Azar’s et al. method. They have provided a lower bound on the maximum load of $\Omega(\sqrt{\frac{\log n}{\log \log n}})$ using $r$ rounds of communication and provided an algorithm that matches this lower bound. A disadvantage of this approach is that it requires synchronous rounds.

Competitive analysis is often used to evaluate the performance of an online algorithm. The performance of an online algorithm is compared with the performance of an optimal offline algorithm that knows the future requests in advance. The maximum ratio between their respective performances, taken over all possible input sequences, is called the competitive ratio [3].

Consider the following assignment/scheduling problem. There are $n$ servers that are ready to provide service to a set of customers. Each customer’s job can be handled by any of the subset of the servers. Customers arrive one-by-one and the problem is to assign each customer to an appropriate server in order to balance the load on the servers. The reassignments of tasks are not permitted. Azar et al. [3] proved that the competitive ratio of any deterministic online algorithm for this problem is $\lceil \log_2 n \rceil$ (up to an additive 1) and the competitive ratio of any randomized algorithm is $\ln(n)$. For the case where each job can be handled by all the servers, Graham [10] proved that the greedy algorithm achieves a competitive ratio of $2 - 1/n$.

Black [4] considered the following shared memory management problem. There are multiprocessors that physically distribute their memory among a number of memories local to each processor or cluster of processors. The memory access is non-uniform. The particular issue of concern is deciding which local memories should contain which data pages. In the migration problem a page must be kept in exactly one local memory. In the replication problem a page may be kept in any subset of the local memories, but once a page is placed in some local memory, it cannot be dropped from there. Black considered a network of processors as a graph in which each edge has a length representing the cost of accessing a remote memory page. Black obtained 3-competitive algorithms for the migration problem on some types of network topologies and 2-competitive algorithms for the replication problem on restricted network topologies.
3 Probabilistic Application Placement

Assume an application and \( n \) possible locations to place the application. A location can typically be a computer, but might also be a LAN or a cluster, as discussed in Section 5. Each location is associated with a \( \textit{goodness} \) number. A location with smaller goodness number is better than a location with greater goodness number. We define a \( \textit{good location} \) to be a computer that belongs to the group of \( r\% \) best computers for some \( r \), and \( \textit{bad location} \) as a computer that is not in this group. Our goal is to find a good location for an application with minimal effort. The notion of good and bad location depends on the application.

Note that the sets of good and bad locations can be different for different applications. For example, if we would like to place applications in computers that are lightly loaded, then the goodness number indicates load. In this case the good locations are the \( r\% \) least loaded computers and they are the same for all applications. However, if we want to place applications near their clients, then the goodness number indicates proximity of an application to its clients. In other words, a location is good for an application if it is one of the \( r\% \) nearest locations to its clients. Since different applications may have different sets of clients, different applications may have different sets of good locations.

We assume that the number of locations \( n \) is relatively large, thus, it is not scalable, and in many cases infeasible, to check all possible locations simultaneously and decide which is the best. We propose a distributed probabilistic method for evaluation locations. We assume that each location can provide information about its goodness number through an \( \textit{information service} \). In order to determine the goodness number of some location, an application should send request to the information service at this location. Since these interactions with information services take some time and load the network, our goal is to minimize the number of locations we probe. We achieve this goal by checking a randomly chosen set of locations and making probabilistic guarantees. In this section we consider a couple of algorithms for placing one application.

3.1 First Algorithm

We assume that the application is initially placed in an arbitrary location. At each step, the algorithm randomly chooses one location and checks its goodness number. If the chosen location has a better goodness number than the current location, the application moves to it. Otherwise, the application stays in the current location. One can see that in each step the algorithm can only move the application to a better location, but never to a worse one, so the question is how many steps we need in order to arrive to a good location.

The probability that after \( k \) steps the algorithm will still not find a good location is \( (1 - r)^k \). Thus, the probability \( p \) that the application will end up in a good location equals to \( p = 1 - (1 - r)^k \). For example, if we would like the application to end up in one of the 10% best locations, then it will happen after 21 steps with 90% probability. Notice that the probability \( p \) does not depend on the number of possible locations, so for a large number of locations we can promise that after a small constant number of steps the algorithm will choose a good location with high probability.
3.2 Second Algorithm

The second algorithm is similar to the first, with the one exception. The second algorithm maintains a list of checked locations to avoid revisiting locations. That is, at each step the algorithm randomly chooses a location that is not already in the checked list, and adds it to the checked list. The algorithm then compares the goodness number of the new location with the current one, and if it is better, it moves the application to the new location. Note that the analysis of this protocol is different from before, since here the probability of not finding a good location decreases in each step.

In order to determine how many steps are needed to reach a good location with high probability, we will use a Markov Chain. We define $n(n+1)/2$ states as follows: Each state $I_i = (x_i, y_i)$ represents a situation in which $x_i$ locations are left unchecked and there are no more than $y_i$ better locations than the current one. Thus, the states are:

$1, 0 \quad (1 \text{ location, 0 better})$
$2, 0 2, 1$
$3, 0 3, 1 3, 2$
$\ldots$

$n-1, 0 \quad n-1, 1 \ldots n-1, n-2$
$n, 0 \quad n, 1 \ldots n, n-1.$

Each state $I_i, 1 \leq i \leq n(n+1)/2,$ is a pair $(x_i, y_i),$ where $1 \leq x_i \leq n$, $0 \leq y_i \leq x_i - 1.$ We denote the set of all possible states by $S$: $S = \{I_0, I_1, \ldots, I_{n(n+1)/2}\}.$ The probability $p_{ij}$ for moving from a state $I_i$ to a state $I_j$ depends on the following possible cases:

- $y_i = y_j = 0, x_j = x_i - 1.$
  In this case the application is in the best location ($y_i = 0$). Since the application moves only to a better location and there is no such location in this state, there is only one possible move from the current state $(x_i, 0)$ to the state $(x_i - 1, 0).$ Hence, $p_{ij}$ here is equal to 1.

- $y_i > 0, x_j = x_i - 1, y_j < y_i.$
  There are $y_i$ locations which are better than the current one in this case. The probability $p_{ij}$ for finding one of the better locations ($y_j < y_i$) from a total of $x_j$ locations equals to $1/x_j.$

- $y_i > 0, x_j = x_i - 1, y_j = y_i.$
  In this case we are looking for the probability of not finding any better location ($y_j = y_i$). This probability equals to $1 - \text{probability of finding some better location.}$ Since there are $y_i$ better locations, the probability $p_{ij} = 1 - (1/x_j + \cdots + 1/x_j) = 1 - y_i/x_j.$

Since the application moves from the current state only to a better one, all other cases are impossible. Thus, the probability $p_{ij}$ for moving from a state $I_i$ to a state $I_j$ equals to

$$p_{ij} = \begin{cases} 
1 & \text{if } y_i = y_j = 0, x_j = x_i - 1; \\
1/x_j & \text{if } y_i > 0, x_j = x_i - 1, y_j < y_i; \\
1 - y_i/x_j & \text{if } y_i > 0, x_j = x_i - 1, y_j = y_i; \\
0 & \text{otherwise}.
\end{cases}$$
Therefore, the transition matrix $\Pi$ is a square matrix of size $\frac{n(n+1)}{2} \times \frac{n(n+1)}{2}$ that looks as follows:

$$
\Pi = [p_{ij}] = \\
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 1/2 & 1/2 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 1/2 & 1/2 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 1/3 & 2/3 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 1/3 & 1/3 & 1/3 & 0 & \cdots & 0 \\
0 & 0 & 0 & 1/3 & 1/3 & 1/3 & 0 & \cdots & 0 \\
& \vdots & & & & & & & \\
0 & \cdots & 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & \cdots & \frac{1}{n-1} & \frac{n-2}{n-1} & 0 & \cdots & \cdots & \cdots & \cdots \\
0 & \cdots & \frac{1}{n-1} & \frac{n-3}{n-1} & 0 & \cdots & \cdots & \cdots & \cdots \\
0 & \cdots & \frac{1}{n-1} & \frac{1}{n-1} & \cdots & \frac{1}{n-1} & 0 & \cdots & \cdots
\end{pmatrix}
$$

All elements of $\Pi$ are non-negative and the row sums $\sum_j p_{ij}$ equal to 1 for all $i$. Let $p_i^{(k)}$ be a probability that after $k$ steps we are in state $I_i$. If $v^{(k)} = (p_i^{(k)}, i \in S)$ is the state distribution or state probability vector after $k$ steps, then

$$v^{(k)} = v^{(k-1)}\Pi = v^{(0)}\Pi^k.$$

$v^{(0)}$ is an initial state probability vector. We assume that initially we do not know how many locations are better, so there is equal probability to be in each state. Thus $v^{(0)}$ has $\frac{1}{n(n+1)}$ in each coordinate.

The space complexity of matrix $\Pi$ is $O(n^4)$, thus, $\Pi$ is too large to compute $v^{(k)}$s for large $n$s. In order to overcome this we reduce the problem to the following one. We define $n$ states. Each state $I_i$, $0 \leq i \leq n-1$ represents that there are $i$ better locations. Since the application moves only to a better location, the only possible cases for moving from a state $I_i$ to a state $I_j$ during step $k$ are:

- $j < i$
  In this case the application moves from the current location to a better one. The probability to find one better location from the total number of $n - k$ locations remaining after $k$ steps is equal to $\frac{1}{n-k}$.

- $i = j$
  The probability that no better location was found is equal to $1 - \text{probability of finding some better location}$. Therefore, $p_{ij}^{(k)} = 1 - (\frac{1}{n-k} + \cdots + \frac{1}{n-k}) = 1 - \frac{i}{n-k}$. 

In summary, the probability \( p'_{ij} \) for moving from a state \( I_i' \) to a state \( I_j' \) during step \( k \) equals to

\[
p'_{ij} = \begin{cases} 
\frac{1}{n-k} & \text{if } j < i \\
1 - \frac{j-i}{n-k} & \text{if } i = j \\
0 & \text{otherwise}
\end{cases}
\]

There is no single transition matrix in this case, but for each step we obtain a different one. The transition matrix \( \Pi'_k \) for step \( k \) looks as follows:

\[
\Pi'_k = \frac{1}{n-k} \begin{pmatrix}
{n-1} & 0 & \cdots & 0 \\
1 & n-2 & \cdots & 0 \\
1 & 1 & n-3 & 0 \\
& & \vdots & \ddots \\
1 & 1 & \cdots & 1 & 0 \\
{k-1} & 0 & \cdots & 0 \\
0 & k-1 & \cdots & 0 \\
0 & 0 & k-1 & \cdots \\
0 & 0 & \cdots & 0 & k-1
\end{pmatrix}
\]

As noted earlier, \( v^{(k)} = v^{(k-1)}\Pi'_k \), and initially we can be in each state with equal probability, i.e., \( v^{(0)} \) has \( \frac{1}{n} \) in each coordinate. Matrix \( \Pi'_k \) is easily computed for each step \( k \) and its space complexity is \( O(n^2) \) instead of \( O(n^4) \) as it was in the previous case.

The numerical values of probabilities to be in a good location on a certain step are shown in Figure 1. The \( x \)-axis represents the number of steps, where the \( y \)-axis corresponds to the probability of reaching a good location. The graphs are computed for 2000 locations when 10\% of them are good (see Figure 1(a)) or 5\% are good (see Figure 1(b)). One can see that in this case in order to be in the 10\% best locations with probability 90\% we need only 21 steps and in order to be in the 5\% best locations we need 44 steps.

Computing the probability according to the first model serves as a good approximation for the second one. But in the second model it is guaranteed that in the worst case we still need a finite number of steps in order to move to the \( r \%) best locations as opposed to the first model where this number of steps is infinite in the worst case. Table 1 presents the dependence of the number of steps needed to reach a good location with probability 90\% on the number of locations \( n \) and the percentage of good locations \( r \).
Figure 1: The probability to be in a good location depending on the number of steps for 2000 locations: (a) 10% of locations are good; (b) 5% of locations are good.

Table 1: The number of steps needed to reach a good location with probability 90% for different parameters n and r.

<table>
<thead>
<tr>
<th>n \ r</th>
<th>100</th>
<th>300</th>
<th>500</th>
<th>1000</th>
<th>3000</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>19</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>5%</td>
<td>36</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
</tr>
<tr>
<td>2%</td>
<td>68</td>
<td>94</td>
<td>101</td>
<td>107</td>
<td>111</td>
</tr>
</tbody>
</table>

3.2.1 Load Balancing

Consider a case when the goodness number indicates load. Then the application placing problem becomes the problem of load balancing. Thus, we are faced with the following problem. Assume an application and n possible computers with different loads. The goal is to place the application on a computer belonging to the group of r% least loaded computers with high probability when a number of reassignments are permitted. We would like to note that when the number of reassignment is not limited, it is trivial to find the computer with lowest load. But the process of reassignment is expensive. Thus, we are interested in minimizing the number of reassignments. In order to solve the problem we can use our probabilistic algorithm from the previous subsection. In the first step we place the application on a random location. In each next step we choose a random computer from the set of unchecked computers, check its load, and if the location is less loaded than the current one, we move the application to the new computer. Otherwise we stay in the current location. As we have seen above, using this algorithm, after a small number of steps we end up in good location with high probability.
4 Multiple Applications

4.1 Load Balancing

We now expand our model to incorporate load balancing of multiple applications. Here we assume a set of \( n \) locations and \( m \) applications. Each location has an initial goodness number which is the same for all applications. Also, each application has some weight which it adds to the goodness of its current location. This weight can be thought of as the amount of resources needed for the application. Our goal is to place applications on computers that have the lowest goodness number, or in other words, minimize the maximal goodness number. Formally, as before, we define a good location for an application \( A \) to be a location with one of the \( r\% \) lowest goodness number for \( A \) while other locations are bad for application \( A \). Of course, if there are not enough resources in good locations for placing all applications, we might be forced to place some applications in bad locations, but we strive to minimize this.

We consider the following algorithm: At each step, the algorithm randomly and independently chooses a new location for each application. If it has a smaller goodness number, the application moves to the new place. Otherwise, the application remains in the current location. There are two ways for checking locations and moving the applications: simultaneously and sequentially. One of the problems with simultaneous checking is that it is not guaranteed that a location that is good for some application when it is probed remains good for that application by the time it moves there. This is because some other application might also move there in the meantime.

In order to solve this problem we permit only one application to address each location at the same time. That is, an application that wants to probe a location for its goodness number locks this location until it decides whether to move there or not. Thus, each location cannot be probed by more than one application simultaneously. We claim that this is feasible even in large systems. In such systems the probability that a large number of applications will address the same location is small, and thus, only a few applications might be blocked due to this, and even this blocking is for relatively short time. Due to the considerable cost associated with moving an application, we prefer to block applications addressing the same location rather than moving them to this location simultaneously just to find out afterwards that some of them must move again due to the increase in the goodness number. We would like to note that under this blocking mechanism the algorithm remains asynchronous and synchronization is only needed when several applications try to address the same location simultaneously.

As opposed to the previous model, where at each step the location of applications can only improve, here this condition is not fulfilled. It is possible that several applications simultaneously move to the same location, which makes this location bad for some of them. In particular, a location might change from being good to being bad and vice versa during the execution of the algorithm.

It is difficult to give an analytical solution for the problem of multiple applications. Thus, we conducted a number of simulations as follows. We instantiate the set of \( n \) locations with uniformly randomly distributed goodness numbers from the interval \( (0, 1) \), and place each of the \( m \) applications in a randomly chosen location. Also, the weight of each application is set
to $w$. That is, moving an application to a given location adds $w$ to its goodness number. For each application there is a transition vector that consists of ids of locations to check during the execution of the algorithm. This is the vector of uniformly distributed random numbers from the interval $[1 : n]$. Each entry $i$ in the transition vector contains the id of a location to probe at step $i$. I.e., the algorithm starts to check locations from the beginning of this vector and continue to the end of it. During step $i$ of the simulation, applications check the $i$th entry in their transition vector, but they do this sequentially, one after the other. If the checked location is better than the current one for a given application, the application moves to it. Otherwise, it stays in the current location.

We expect that when the number of good locations is significantly greater than the number of applications, the behavior of the system will be similar to the model of one application. We assume that in this case applications hardly ever conflict with each other, and therefore each application behaves as if it is alone in the system. Thus, we expect that in this case all applications will find good locations during a small number of steps. On the other hand, when the number of applications is significantly greater than the number of good locations, we expect that during a number of steps applications will occupy all good locations, but some will end up in bad locations. Another question is whether the system will stabilize, and how fast it will happen. That is, whether each application will settle in one location, or whether applications will continuously move forever.

Simulations were carried for different parameters $n$ and $m$. Parameter $w$ was set to 0.1 for all applications. The typical results of simulations are shown in Figures 2-4. In Figures 2-4(a) the transfers of applications between locations with different goodness numbers are shown. The $x$-axis shows the number of steps while the goodness number of each location on the $y$-axis. In Figures 2-4(b) the transfers of applications between different locations are shown. The $x$-axis represents the step number and the $y$-axis corresponds to the location id in that step. In our experiments the system always stabilized after a small number of steps. That is, every application settles in some location and did not move from it again. In the case when there is enough place in good locations, the applications settled in them. In the opposite case, some applications capture good locations, but others ended up in bad locations. As was assumed, in the case when the number of good locations is greater than the number of applications, the observed behavior is similar to the case of only one application. This also validates the formal analysis of Section 3. In particular, after a small number of steps, each application ends up in a good location.

We would like to note that when applications have different weights, it is not guaranteed that our protocol achieves the same result. It may be possible that the system reaches a stable state but this state is not optimal. This means that all applications can be in such locations that there are no better locations for them, but the system is not balanced, a phenomena also known as local minimum in hill climbing algorithms [18]. That is, it is possible that by simultaneously changing the location of several applications the system will be more balanced.

To illustrate this, consider the example in Figure 5. Assume that we have two locations $i_1, i_2$ and 4 applications $m_1, m_2, m_3, \text{ and } m_4$. Both locations have the same initial goodness number and the weights of the applications are 1, 5, 2, and 6 respectively. Assume that $m_1$ and $m_2$ are in location $i_1$ and $m_3$ and $m_4$ are in $i_2$, as shown in Figure 5(a). One can
Figure 2: The results of simulation for \( n = 1000 \) and \( m = 20 \): (a) the goodness number of locations in which applications are settled depending on the number of steps; (b) the ids of locations on which applications are settled in a given step. Here there are many more potential locations than applications, so applications end up in good locations very quickly (left graph), and after a short initial phase, most application settle in one location (right graph).

see that there is no better location for any application but the state of the system is not optimal. If we have simultaneously changed the locations of both \( m_1 \) and \( m_3 \) as shown in Figure 5(b), we would achieve a balanced state and the goodness number in both locations would be the same. In the case when all applications have equal weight this scenario is impossible, because simultaneously changing the locations of several applications does not change locations goodness number.

### 4.2 General Applications’ Placement

We now turn the discussion to a more common case when different applications have different sets of good locations. This model is applicable, e.g., for the problem of placing applications close to their clients, as discussed also in Section 5. As different applications may have different clients, the set of good locations for each application might be different than for other applications.

We formalize the problem in the following way. Assume \( m \) applications and \( n \) possible locations to place these applications. Each location is associated with a goodness vector of size \( m \) of goodness numbers. Entry \( i \) of the goodness vector corresponding to location \( L \) represents the goodness number of location \( L \) w.r.t. application \( i \). For example, when goodness number represents proximity to clients, then entry \( i \) of the goodness vector for location \( L \) expresses the proximity of location \( L \) to the clients of application \( i \).

In addition, each location has a maximal allowed load, corresponding to the maximum number of resources it has. As in the previous section, each application imposes a load \( w \) on the location where it resides, such that the total load of all applications cannot surpass the maximal allowed load for that location. As before, we define a good location for application
i as a location which belongs to $r\%$ best locations for this application. Our goal is to find a good location for each application with high probability.

Note that the main difference from the previous subsection is that now each application has its own set of good locations, which may differ from the set of good locations of some other application. Thus, the algorithm used in the previous section can be used as is.

As before, due to the difficulty in analyzing the system analytically, we conducted a number of simulations. We assume that locations load are uniformly randomly distributed numbers from interval $(0; 1)$. All applications have the same weight 0.1. In order to understand the system behavior in different situations, the simulations were carried for the following cases:

1. The applications have distinct sets of good locations;
2. The sets of good locations for different applications overlap;
3. The applications have the same sets of good locations.

We mark the number of common good locations by $s$ and the total number of good locations by $r$. If $s = 0$, we have Case 1, when all applications have different sets of good locations. If $s = r$, all applications have the same sets of good locations and we receive Case 3. If $0 < s < r$, we receive Case 2, when the sets of good locations for different applications overlap.

We assume that applications are independent. Thus, the system behavior in Case 1 is expected to be similar to the system with one application from Section 3. That is, after a small number of steps all applications will find a good locations. In Case 3, we have the same system as in the previous subsection. Thus, we expect to finish with a stable system.
Figure 4: The results of simulation for \( n = 100 \) and \( m = 200 \): (a) the goodness number of locations in which applications are settled depending on the number of steps; (b) the ids of location on which applications are settled in a given step. Here there are fewer potential locations than application, so some applications end up in bad locations (left graph). Surprisingly, most applications settle down quite rapidly (right graph).

Figure 5: An example of incorrect work of the algorithm: (a) unbalanced system received after execution of the algorithm; (b) possible balanced system.

The needed number of steps will depend on the relation between the number of applications and the number of good locations.

The simulations were carried for different values of parameters \( n, m \) and \( s \). In the presentation of the simulations’ results, for each application we have ranked the locations according to their goodness number. That is, the \( i \)th location for an application \( A \) is the location whose goodness number is the \( i \)th best for \( A \).

The typical results of simulations are shown in Figures 6 - 8. The number of steps is represented by the \( x \)-axis and the ranks of locations chosen by applications is represented by the \( y \)-axis. That is, if an application \( A \) is placed on location \( L \) ranked \( i \) in some step, then location \( L \) is the \( i \)th best location for application \( A \).

The graphs in each figure show applications moving from their initial locations to better ones. One can see that in all cases the system stabilized after a small number of steps. Naturally, a system with no common good locations stabilized faster than the system in
which there are common good locations. As the share of common good locations increases, a larger number of steps is needed in order to end up in a good location. As expected, the algorithm achieves better results as the number of locations increases. Since the number of steps needed to end up in a good location is almost constant, the larger the number of possible locations we have, the smaller the percentage of locations that must be visited before a good location is found. When there are too many applications in the system, i.e., there are not enough resources for placing all of them in good locations, it takes more time for the system to stabilize. In this case some applications end up in bad locations.

![Figure 6](image6.png)

**Figure 6:** The typical results of simulations in the case $s = 0$: (a) 1000 locations, 500 applications; (b) 500 locations, 300 applications. Here there are no common good locations, so the system behaves as if there is only one application.

![Figure 7](image7.png)

**Figure 7:** The typical results of simulations in the case $s = 10\%k$: (a) 1000 locations, 500 applications; (b) 500 locations, 300 applications. Here there are a few common good locations, so it takes longer for applications to find good locations.
5 Placing Virtual Servers Near Clients

In this section we consider the problem of placing virtual servers near their clients. A virtual server is a server whose location in the Internet is virtual, i.e., it may move from one physical site to another and the virtual server identity is not bounded to a fixed physical computer [7]. Each server has its own clients located somewhere in the Internet. Our goal is to place each server closest to its clients. We measure the distance between a server and its clients by average round trip communication time (RTT) from the server to the clients. We assume that RTT between two computers belonging to the same LAN is insignificant. Thus, we can measure RTT between LANs instead of measuring RTT between computers. We propose a two-step protocol for determining the server placement. In the first step we choose a LAN from the Internet that is closest to the virtual servers clients and in the second step we find the best computer within this LAN on which the server will be placed.

We would like to emphasize that we assume that virtual servers need not move very frequently between LANs. This is based on the assumption that the locale of “the average client” does not change very quickly.

5.1 The Protocol

In order to solve the problem we use the following model. Each LAN is associated with a goodness number that expresses the proximity of the LAN to the average client, as detailed below. Since the distance of one application to its clients does not depend on other applications, this is the same problem as in Section 3. I.e., there are $n$ locations (LANs) with different goodness numbers and an application. Since there is a huge number of LANs in the Internet, we cannot check all of them. Thus, the goal is to place the application in one of the $r\%$ closest LANs to its clients with high probability. In order to do so we use the probabilistic distributed algorithm from Section 3 for evaluation of LANs. In order to compare different locations and determine goodness numbers we use agents. Each agent
computes the distance to the clients from the LAN it visits (location goodness number) and returns the answer to the server.

The protocol works as follows. In each step it randomly chooses $k$ LANs and sends agents to these LANs. One additional agent runs in the server’s LAN. In order to determine the distance to the clients, each agent executes the first step of the protocol described below and sends the calculated distance to the server. The server chooses the minimal distance $\text{min\_dist}$ returned from agents and compares it with the distance $\text{curr\_dist}$ received from the agent on the server’s LAN. If $\text{min\_dist}$ is less than $\text{curr\_dist}$, the server executes the second step of the protocol in order to choose a computer from the corresponding LAN, and moves to the chosen computer. Otherwise, the virtual server remains in its current place.

We assume here that if an agent runs in a LAN without enough resources for server placement, it terminates and returns to the server a very large goodness number. Thus, the server receives relevant distances only from agents that are running on LANs with enough resources. As shown in Section 3, in order to move to one of the $r\%$ best LANs with probability $p$ we need $k$ agents where $k$ should satisfy the condition $p \leq 1 - (1 - r)^k$. As mentioned earlier in Section 3, $k$ does not depend on the total number of LANs but only on the desired probability and the required percentage of good LANs.

### 5.1.1 The First Step of the Protocol

In order to determine the LAN that has minimal distance to clients, we assume that each virtual server has a log file. This file consists of IP addresses of clients that addressed the virtual server during some period of time. Each agent has access to this file and the goal of the agent is to compute the distance from the LAN it explores to the “average” client from this log file. We measure the distance to a client by executing ping to the IP address of the client. Ping works by sending an ICMP package to the client and returns (among some other data) RTT of this package [19]. In order to obtain a reasonable estimate for RTT we execute several pings to each client. We use as a distance to a client the average RTT of these pings without the worst $10\%$ of them. We decided to drop the worst results since these are often caused by short lived overloads in the network, or even by swapping of the pinging process, etc. These extremely slow pings can significantly decrease the average computed RTT even though in practice they do not represent the distance to the client.

Log files can become very large, especially for heavily loaded servers. Thus, pinging each entry in the log file takes too much time and loads the network. Since in the first step of the protocol we are looking for a LAN and not for a specific computer, we are interested in RTT time to a LAN. Therefore we can significantly decrease the number of IP addresses to ping by determining IP addresses of LANs and pinging only one computer from each LAN. For example, we have checked the log files of our department’s HTTP server and found that the number of distinct LANs was only about $2\%$ of the total number of entries in that file. Thus, in our protocol, each agent generates a LAN-file file that consists of IP addresses of one computer from each LAN presented in the log file. The following summarizes the protocol executed by each agent:

- Ping $\text{num}$ times to each entry in $\text{LAN-file}$.

17
- Compute average RTT of these pings for each such entry; do not take into account the worst 10% measured times.
- Send computed average RTT to the server.

### 5.1.2 The Second Step of the Protocol

The goal of the second step is to choose a computer with highest performance from the LAN identified during the first step. We assume that each server has a list of architectures and a list of operating systems on which it can be placed. In order to find a computer for a virtual server placement we should find all computers in the LAN that have an architecture and operating system that the server can run on (we assume that this group of computers is relatively small) and choose the one with highest performance.

### 6 Competitive Ratio

#### 6.1 Terminology and Definitions

An online algorithm is an algorithm that receives its input online and must produce in real time a sequence of decisions based on past events without knowing the future. Competitive analysis measures the quality of an online algorithm by comparing its performance to that of an optimal off-line algorithm, which is (for the online problem) an unrealizable algorithm that has full knowledge of all past and future inputs [5]. In order to compare the performance of these algorithms a cost function is defined. We denote by \( F(I) \) the set of all feasible outputs associated with input \( I \). For each possible input \( I \), the cost function represents the cost of the output \( O \in F(I) \) with respect to an input \( I \). That is, given any legal input \( I \), an algorithm \( ALG \) computes a feasible output \( ALG[I] \in F(I) \). The cost associated with this feasible output is denoted by \( ALG(I) \) or \( C(I, ALG[I]) \).

Similarly to [5], we define an optimal algorithm as an algorithm for which for all legal inputs \( I \)

\[
OPT(I) = \min_{O \in F(I)} C(I, O).
\]

An online algorithm \( ALG \) is \( c \)-competitive if there is a constant \( \alpha \) such that for all finite input sequence \( I \),

\[
ALG(I) \leq c \cdot OPT(I) + \alpha.
\]

Thus, for each input \( I \), a \( c \)-competitive algorithm guarantees to incur a cost within a factor \( c \) of the optimal off-line cost (up to the additive constant \( \alpha \)). If algorithm \( ALG \) is \( c \)-competitive, we say that \( ALG \) attains a competitive ratio \( c \). Although \( c \) may be a function of the problem parameters, it must be independent of the input \( I \). The competitive ratio is always at least 1, and the smaller it is, the better \( ALG \) performs with respect to \( OPT \) [5].
6.2 Competitive Ratio of Our Placement Algorithm

We compute the competitive ratio of our probabilistic algorithm for placing one application from Section 3. With each execution of the algorithm we associate a cost function and compute the competitive ratio of the online algorithm in two different ways.

6.2.1 Absolute Cost

With each execution of the algorithm we associate the following cost function. For each move from one location to another the algorithm pays \( y \), and for staying \( k \) steps in a location with goodness number \( d \) it pays \( d \cdot k \). Thus, being in a good place costs less than in a bad place. In order to move the application from one location to another, it needs to save its current state and resume execution of the process from the saved state in the new location. Thus, the act of moving the application from one location to another is complex and expensive, and we associate with it a cost \( y \). In general, \( y \) may depend on the locations that the application moves between and therefore can change from step to step. However, in order to simplify the discussion, we assume that \( y \) is the same on each step.

Consider the following situation. Assume that the application is in some location and the online algorithm finds a better location with smaller goodness number. The application moves to the new location and the algorithm pays \( y \) for this move. But, it may be possible that after a number of steps we will find an even better place than the previous one. Thus, it may be cheaper to remain in the original place until the second, better, place is found than move twice. We consider the optimal off-line algorithm that knows in advance which locations will be probed in the future, and therefore, can compute when to move the application to a new location and when to remain in the old one in order to pay a cheaper total cost.

We compute the difference between the online and off-line algorithms. Consider the scenario shown in Figure 9. Assume that in some step the application is in a location with goodness number \( d \). After \( k_i \) steps the online algorithm finds a better location with goodness number \( d - \Delta_{i+1} \) and the application moves to it. After \( k_{i+1} \) more steps the algorithm finds the next better location with goodness number \( d - \Delta_{i+1} - \Delta_{i+2} \).

![Figure 9: Computing the difference between the online and optimal off-line algorithms.](image)

The cost that the online algorithm pays for this sequence of steps is

\[
C(\text{online}) = k_i d + y + k_{i+1} (d - \Delta_{i+1}) + y
\]

The optimal off-line algorithm for the same sequence of steps pays

\[
C(\text{offline}) = \min \left\{ \begin{array}{l} C(\text{online}) \\ (k_i + k_{i+1})d + y \end{array} \right. 
\]

There is a difference between the two algorithms only when the off-line algorithm does not move to a new place. Thus, if a better location is found \( j \) times, the difference in the cost paid by the two algorithms is
\[ C(online) - C(offline) = k=id - k_{i+1}\Delta_{i+1} - k_{i+2}(\Delta_{i+1} + \Delta_{i+2}) - \cdots \]
\[ -k_{i+j}(\Delta_{i+1} + \Delta_{i+2} + \cdots + \Delta_{i+j}) \]  

(1)

Recall that the online algorithm moves the application each time it finds a better location, while the off-line algorithm does not. Therefore, with the online algorithm, the application is always in the location that is not worse than the location in which the optimal off-line algorithm is. Hence, if in some step we did not find a better location, the application remains in its current state in both executions. Since the current location in the online algorithm can only be better than the one chosen by the off-line algorithm, the cost that the online algorithm pays for such a step is not greater than the cost of the off-line algorithm. Therefore, the difference in the costs of the online and off-line algorithms is maximized if in each step we find a better location so that the online algorithm moves the application while the off-line does not. Thus, in order to find the maximal difference in cost, we assume that all \( k_i \)s in (1) are 1. In order to simplify the calculations we assume that all \( \Delta_i \)s in (1) are the same and equal to \( \Delta \). Thus, we receive

\[ C(online) - C(offline) = jy - (\Delta + 2\Delta + \cdots + j\Delta) \]
\[ = (y - \Delta) + (y - 2\Delta) + \cdots + (y - j\Delta) \]  

(2)

From Equation (2), one can see that in order to maximize the difference in the costs of the online and the off-line execution, the off-line algorithm should not move the application until \( y > j\Delta \). Therefore, \( j \) should satisfy the following inequality

\[ \frac{y}{\Delta} - 1 \leq j < \frac{y}{\Delta} \]  

(3)

For the defined above \( j \) steps, the optimal off-line algorithm pays

\[ C(offline) = (j+1)d + y \]

Using Equation (3) we receive that for these \( j \) steps the online algorithm pays

\[ C(online) = d + y + y + \Delta + y + \cdots + y + d - j\Delta + y \]
\[ = (j+1)d + (j+1)y - \frac{j+1}{2}j\Delta \]
\[ \leq (j+1)d + (j+1)y - \frac{j}{2}y \]
\[ = (j+1)d + \frac{j+2}{2}y \]

Thus, the online algorithm has the following competitive ratio \( c \):

\[ c = \frac{C(online)}{C(offline)} \leq 1 + \frac{jy/2}{(j+1)d + y} \leq 1 + \frac{y}{2d} \]  

(4)
The last step in Equation (4) follows from the following arithmetic:

$$\frac{2(j+1)d+2y}{jy} \geq \frac{2(j+1)d+2y}{(j+1)y} \geq \frac{2d}{y}$$  \hspace{1cm} (5)$$

One can see from Equation (4) that the online and off-line algorithms are close when $y$ is small or/and $d$ is large. In the first case, when $y$ is small, it is not expensive to move to a new place and therefore, the optimal off-line algorithm moves more, which makes it closer to the online algorithm. In the second case, when $d$ is large, it is expensive to remain in the same location. Thus, here too the off-line algorithm moves more and is therefore closer to the online algorithm.

### 6.2.2 Relative Cost

In order to further refine the result of Section 6.2.1, we consider here a slightly different cost function, and compute the competitive ratio of our algorithm according to it.

Assume some unrealizable ideal algorithm $\mathcal{A}$ that moves the application each time it finds a better location but does not pay for it. Thus, the only thing that $\mathcal{A}$ pays is the cost of being in a given location. We define a cost that an algorithm pays during an execution relatively to $\mathcal{A}$’s cost. That is, for some algorithm $ALG$, the relative cost $C'$ is defined as

$$C'(ALG) = C(ALG) - C(\mathcal{A})$$

![Figure 10: The cost of the online and the optimal off-line algorithms relatively to the cost of the ideal algorithm $\mathcal{A}$.

As we have seen in the previous subsection, the maximal difference in the cost that the online and the optimal off-line algorithms pay is in the case when a better location is found in each step. The corresponding situation is shown in Figure 10. $\mathcal{A}$ and the online algorithm move to a new location in each step and the optimal off-line algorithm stays in the same place during $j$ steps, where $j$ satisfies Equation (3), and moves after that. The relative cost
that the online and the optimal off-line algorithms pay for this sequence of $j$ steps is

$$
C'(\text{online}) = (j+1)y \\
C'(\text{offline}) = \Delta(0 + 1 + \cdots + j) + y = \frac{j(j+1)}{2}\Delta + y
$$

From (3) we obtain

$$
C'(\text{offline}) > \frac{j(j+1)}{2}\Delta + j\Delta = \frac{j+3}{2}j\Delta
$$

Using (3), the competitive ratio of the online algorithm according to the relative cost $C'$ is

$$
c' = \frac{C'(\text{online})}{C'(\text{offline})} < \frac{y}{\Delta} \frac{j+1}{j(j+3)} \leq 2 \frac{(j+1)^2}{j(j+3)}
$$

(6)

Note that if $j = 0$, then the online and optimal offline algorithms behave the same. Thus, to make things worse for the online algorithm, we assume that $j \geq 1$. Thus, $\frac{(j+1)^2}{j(j+3)} \leq 1$ and we obtain from (6)

$$
c' < 2
$$

(7)

Now interpret this result in terms of the absolute cost. Denote by

$$
C_{\text{on}} = \frac{C(\text{online})}{C(\mathcal{A})} \\
C_{\text{off}} = \frac{C(\text{offline})}{C(\mathcal{A})}
$$

(8)

Thus, from (7) we receive that

$$
c' = \frac{C'(\text{online})}{C'(\text{offline})} = \frac{C(\text{online}) - C(\mathcal{A})}{C(\text{offline}) - C(\mathcal{A})} = \frac{C_{\text{on}} - 1}{C_{\text{off}} - 1} < 2
$$

Since the cost that the off-line algorithm pays is always greater than the cost of the ideal algorithm $\mathcal{A}$, we get that $C_{\text{off}} - 1 > 0$ and it follows that

$$
C_{\text{on}} < 2C_{\text{off}} - 1 \\
\frac{C_{\text{on}}}{2C_{\text{off}} - 1} < 1
$$

On the other hand,

$$
\frac{C_{\text{on}}}{2C_{\text{off}} - 1} > \frac{C_{\text{on}}}{2C_{\text{off}}}
$$

Therefore,

$$
\frac{C_{\text{on}}}{2C_{\text{off}}} < \frac{C_{\text{on}}}{2C_{\text{off}} - 1} < 1 \\
\frac{C_{\text{on}}}{C_{\text{off}}} < 2
$$
Thus, using (8) we receive that the competitive ratio $c$ according to the absolute cost $C$ is

$$c = \frac{C_{\text{online}}}{C_{\text{offline}}} = \frac{C_{\text{on}}}{C_{\text{off}}} < 2$$

(9)

Therefore, from (4) and (9) we receive that the competitive ratio of the online algorithm is

$$c = \min(1 + \frac{y}{2d}, 2)$$

7 Discussion

In this paper we have analyzed the problem of placing an application in a very large distributed system in a good location with high probability, using a small number of steps. We have shown an algorithm that achieves this goal, proved its behavior for a single application and simulated it for multiple applications, and showed that it exhibits a good competitive ratio. Also, we have presented a scalable two-step protocol for placing virtual servers near their clients, which is based on the application placement algorithms we discussed.

Yet, there are some other aspects of this problem that are not addressed by this work. For example, a heuristic for making smart choices regarding when a process should move and when it should stay even if it found a better place might improve the average overall cost of an application. That is, even though we have shown a good competitive ratio, this measure only states that in the worst case our protocol is at most twice as bad as an off-line optimal algorithm. However, it says nothing about the average or common behavior.

Another issue is analyzing or simulating a situation in which multiple applications are allowed to simultaneously check the same location for its goodness number. Also, simulating the case when there are different weights to different applications, and providing a solution to the problem associated with it as presented in Section 4.1 is desirable. This includes combining our protocol with methods that can exchange two or more applications simultaneously, even if neither one would benefit if it were to move by itself. The challenge is to do so in a scalable way. Presumably, it will also have to rely on probabilistic methods. Finally, it would be highly valuable to integrate our protocol from Section 5 into symphony [7], and measure how well it works in a full scale system.

Acknowledgements: We would like to thank Hadas Shachnai for helping us with the final competitive analysis results.
References


