Andrei Ivanov
John Mofor
Tsotne Tabidze

Placement System for Running Jobs over a Data Center Network

Design Report

May 9th 2014

1 Overview

By exposing a vast network of virtual machines to users, Data Centers offer a tremendous amount of resources. These resources however, come at an expensive cost, hence bringing up the need to utilize the Data Center as efficiently as possible. Our placement system's goal is to minimize the total Data Center usage time for our user, given only the expected traffic between the user's virtual machines.

To achieve our goal, we need to gather more information about the state of the Data Center's Network, then using this state, perform some computations to solve for an optimal placement configuration for the given virtual machines. Consequently, our system is divided into two main components: the Measurement component, and the Placement component. The Measurement component periodically records the throughput over groups in the Data Center network. Using this continuous inflow of measurements from the Measurement Component, the Placement component does all the computations required to find optimal placement configurations at each time steps - about every five seconds. Coordinating these two components is the Master Virtual machine - the first virtual machine our system reserves in the Data Center.

The subsequent sections will breakdown every components and mechanisms mentioned above, giving further details and illustrations.

2 Design

The Data Center is divided into 4 clusters of 6 groups each. Each group is in-turn made up of 48 machines, each machine being able to support up to 4 virtual machines. We define *path capacity* as the capacity of the link with the lowest capacity along the path between those two machines. Our design rests on two very important observations about this Data Center network:

- The theoretical *path capacity* between virtual machines increase as the path endpoints move down into deeper partitions.As an illustration, the *path capacity* between two machines in different clusters is 10Gb/s, while that between two virtual machines in the same group is about 100Gb/s.
- The longer the path, with high probability, the slower it becomes to traffic between the two virtual machines linked by that path. This is because, as the path elongates, the end-points become more and more prone to share a link with one or more other users of the Data Center - sharing a link will reduce the user's effective bandwidth, hence the overall path speed.

Based on the aforementioned observations, our placement strategy strives to minimize the total usage time by maximizing the utilization of smaller partitions first, before expanding to larger partitions, i.e. it places as many virtual machines as possible first within a single physical machine, then to machines within a group, then to machines within a cluster, then finally to machines within different clusters. This way, we both shorten the overall path lengths, thereby reducing our dependence on other users' traffic, while efficiently exploiting the huge capacities of the smaller partitions.

In an ideal case where N (the number of virtual machines required by the user) can be contained within a single group, there is no need for gathering measurements across all groups. This is because the ideal configuration will simply assign all N virtual machines to the group with enough space in accordance to observation 1 above (fill in increasing order of partition size). The non-ideal case brings with it the need for measurements that will be used to find a placement configuration across groups.

With the big idea laid down, the subsequent subsections will now describe in-depth each of the major components brought together to realize this system.

2.1 Measurement component

The Measurement component is made up of two parts, the Master virtual machine, and the Probing virtual machines (probe, for short). The probes are in charge of gathering measurements and making them available for the Master virtual machine. The measurement in question refers to the throughput between the probe, and all other probes in the same cluster.

2.1.1 Master virtual machine

The very first step done by the Placement system is to allocate a special (Master) virtual machine for the user, using the *(machine_id, IP_addr) random_place(v)* function of the Data Center's API. The Master virtual machine is in charge of using all streams of measurement data from the Probing virtual machines to periodically solve for optimal placement configuration of all N virtual machines at each time step.

Before attempting any measurement operations however, using the Data Center's API *machine occupancy* function, the Master verifies that no group can contain all N virtual machines. Should a group satisfy $N \leq (48 - \mathit{group\_occupancy})$, then it simply assigns all N virtual machines to that group, by filling in order of lowest $\mathit{machine\_occupancy}(0/4)$ to highest $(3/4)$. We define *group_occupancy* as the sum of *machine_occupancy*'s for all 48 machines in that group. Should no such group exist however, then the Measurement component becomes non-trivial, thereby requiring the Master virtual machine for configuration computation. See Fig (1).

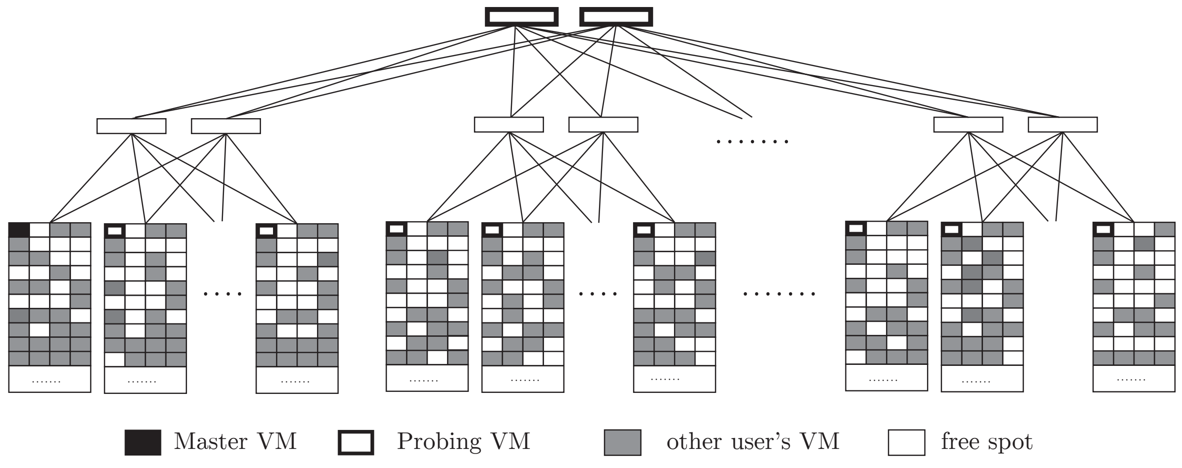

Figure 1: Structure of Data Center with both Master VM and Probing VM's

2.1.2 Probing virtual machines

In a non-trivial measurement scenario, the Master virtual machine will attempt to place a Probing virtual machine in each group. The Master virtual machine would accomplish this using the *place(v,m)* function while iterating over m. The pseudo-code in Listing (1) illustrates this mechanism.

Listing 1: Pseudo-code for establishing probes.

```
1  IP_addr reserve(Mach_ID m)
2      return place(dummyVirtualMachine,m);
3
4  IP_addr swap(VM v, Mach_ID m)
5      free(dummyVirtualMachine,m);
6      return place(v,m)
7
8  List<IP_addr> placeProbes(VM probe)
9      List< Pair<IP_addr,Mach_ID> > allProbes;
10     // Reserve first available spot in each group.
11     for groupStart in 1:1104:48, // 1104 + 48 = 1152
12         for machineIndex in 1:48:1,
13             Mach_ID m = groupStart + machineIndex;
14             IP_addr addr = reserve(m);
15             if addr!=null,
16                 allProbes.add(Pair(addr,m));
17                 break;
18
19     // Initialize the probes, informing them of their neighbors.
20     for ip,m in allProbes,
21         VM probeCopy = probe.copy
22         List<IP_addr> n=[ip of all probes in same cluster as m]);
23         probeCopy.setNeighbors(n);
24         ip = swap(probeCopy,m); //update IP in allProbes
25     return allProbes;
```

As illustrated from line 19 to 24 in the pseudo-code, every probe knows the IP of it's neighbors (probes in the same cluster). With that information, the probes can then start gathering measurements thus making the information available for the Master virtual machine.

2.1.3 Measurement mechanism

It is very likely that the user will have multiple virtual machines running per group, each with traffic flowing in and out. To this case, each probe will send as much data as it can as fast as it is allowed to, to its neighbors, thereby mimicking traffic from multiple different virtual machines - See Fig (2).

The probes keep sharing traffic for approximately 100ms after every time step. On the other hand, at each time step, the Master VM will pause all N user VM's, and after 100ms pass the time step, it will use the Data center's API function *tcp_throughput(probe)* to record the throughput for every probe, then will

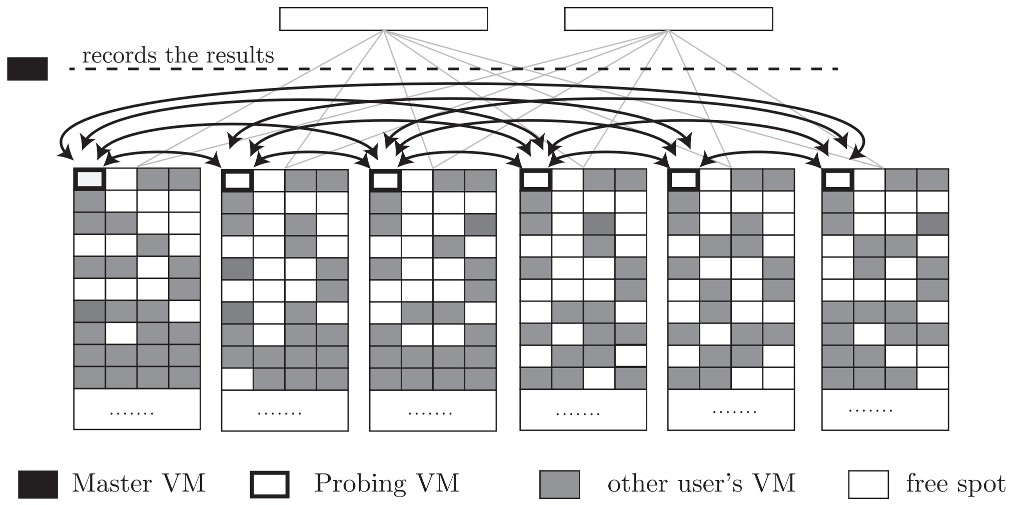

Figure 2: Illustration of Probing Strategy

resume all user VM's. See time-line in Fig (3) for illustration.

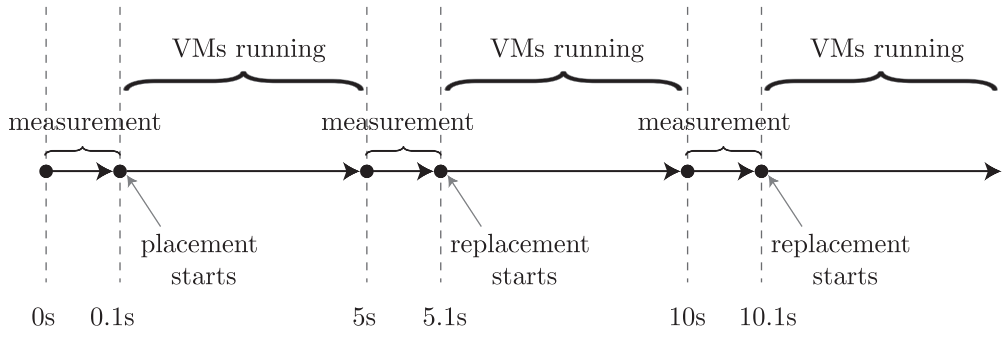

Figure 3: Time-line illustrating the measurement/placement cycle.

The user VMs are paused during the measurement, because the system seeks to measure throughput due to other users' jobs, i.e. not influenced by our user's job. With these TCP throughput results, the Master VM will then be ready to generate the set of results required by the Placement component.

2.1.4 Measurement results

The placement component requires two main results to function in the case of a non-trivial measuresument scenario. These are:

- *Average internal cluster throughput:* This could be defined as the average of all recorded tcp_throughputs of probes within a cluster. The master VM computes this value by averaging tcp_throughputs of probes withing the same cluster, at 100ms pass every time step.
- *Cluster occupancy:* This could be defined as the sum of machine occupancies for all machines within a cluster. The master computes this by executing the *machine_occupancy* function of the Data center's API, while summing through, for all the machines in the cluster.

With these two sets of results, the Placement algorithm proceeds as described in subsequent sections.

2.2 Placement component

Using the data (TCP throughput) from the probes, the Placement component determines a cluster that would maximize the *average internal cluster throughput.* After choosing this cluster, it selects the best group in accordance with observation 1 (i.e. group with the largest number of free spots), and finally uses a greedy algorithm to choose which user VMs to place inside each group. This procedure is periodically rerunned in order to adjust to traffic changes as well as the progress of our work - See time-line Fig(3).

2.2.1 Placement Metric

To maximize the internal throughput of a cluster the Placement component chooses a cluster with minimum occupancy in order to maximize the use of high capacity links within groups. In addition, it also chooses the cluster with the best throughput between groups to have a high transfer rate in between VMs in different groups. Because it is not always possible to choose a cluster with both minimum occupancy and best throughput, we define a following metric

$$M(\text{cluster}) = \frac{\alpha}{\text{occupancy}} + \beta \cdot \text{throughput}$$

We first can use α and β to normalize between occupancy and throughput, because these measures have very different units. We then can use α and β to prioritize one of the quantities. Occupancy is more valued because the links inside groups are much better than those in between them. Thus the Placement component prefers bigger values of α. The best cluster can thus be defined as that which maximizes M.

To choose between the groups within a cluster, we rely on the occupancy and we prefer the groups with more spaces available for VMs.

2.2.2 Initial placement

Now, given the way to choose between clusters and groups, the Placement component uses the following strategy to place user VMs:

1. Choose the best cluster
2. Choose the best group
3. Select the pair of VMs with the biggest workload, i.e. the pair (VM$_i$, VM$_j$)) with the largest value of $B_{ij} + B_{ji}$ and place it in the first available machines in the selected group.
4. While there are still places available in the group from the VMs that have not yet been placed anywhere, select the VM that maximizes the workload with the VMs already placed in the group. That is, given the inserted VMs $S = \{s_1, s_2, ..., s_k\}$ we choose the VM$_j$ so that

   $$j = \text{argmax}_{i \notin S} \left(\sum_{\ell=1}^{k} B_{is_\ell} + B_{s_\ell i} \right)$$

   This greedy approach is shown in Figure 4.
5. After the group is filled we repeat steps 2-4 until all the groups in the cluster are filled. Repeat steps 1-5 until we place all the VMs.

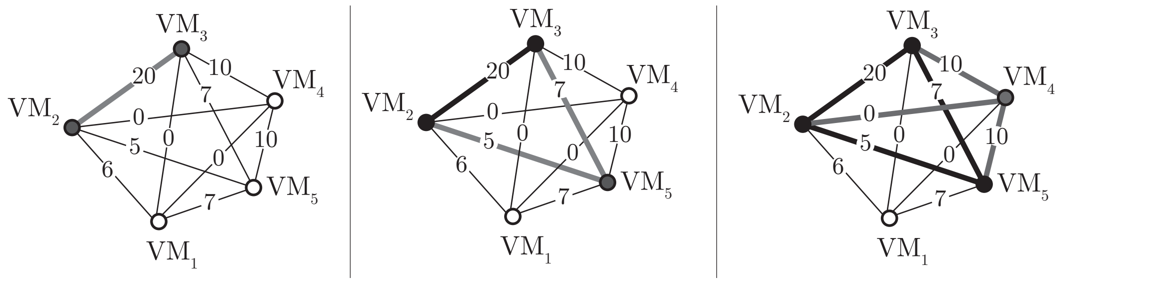

Figure 4: An example of greedy algorithm procedure. Each edge represents the pairwise workload between two VMs, i.e. $B_{ij} + B_{ji}$. The algorithm tries to find 4 VMs out of the given 5 to place in a group. It first chooses VM$_2$ and VM$_3$ with the biggest workload equal to 20, then it chooses VM$_5$ because is maximizes the workload with VM$_2$ and VM$_3$, etc.

2.2.3 Dynamic replacement

The initial placement described above will provide us a good performance while our assumptions of maximum load are true. However, if we leave the VMs in the same place for a long time we may start under-using the system for a few reasons. The VMs inside the group may have finished their biggest transfers and may not maximize the internal workload anymore. Some other cluster we have not used may have gotten freed up in terms of both occupancy and average throughput. Therefore we include a strategy of periodical replacement of VMs that would again maximize the internal throughput inside groups.

5 seconds after we placed or replaced our VMs we pause the execution of VMs, and rerun the measurements and computations as described in 2.2.1. to get the new ranking of clusters and groups inside clusters. We also update the matrix B to reflect the amount of transfer work that is left to be done. We can find these numbers by calling `progress(u,v)` for all pairs of VMs.

We then repeat the procedure from 2.2.2. with slight changes. First, we imagine that all the VMs are unplaced and as before find the best cluster, then the best group. At every step in the chosen groups we can have both free spots and the old VMs placed 5 seconds before. We consider both of these to be available for new placement. We run our greedy algorithm and either place the newly chosen VM in a free spot or swap it with an old VM. We repeat this process until all the VMs are replaced. This is illustrated in Figure 5.

3 Analysis

In this section we analyze the greedy algorithm for choosing VM placements, the behavior of our strategy in three different scenarios, and the extra cost caused by the master VM and the VMs temporarily allocated by measurement component.

3.1 The greedy algorithm

One of the key points of our system design is the algorithm that tries to group the VMs so that the workload inside each group is maximized. We notice that maximizing the internal workload is equivalent to minimizing the workload between VMs in different groups because $W_{\text{internal}} + W_{\text{external}} = \sum B_{ij}$. Therefore if we imagine the VMs being vertices of a graph with edge weights defined by B, then our problem is similar to the *minimum k-cut* problem which asks to find the set of edges of minimum weight that divides the graph into k connected components. However, there are several limitations related to the *minimum k-cut*

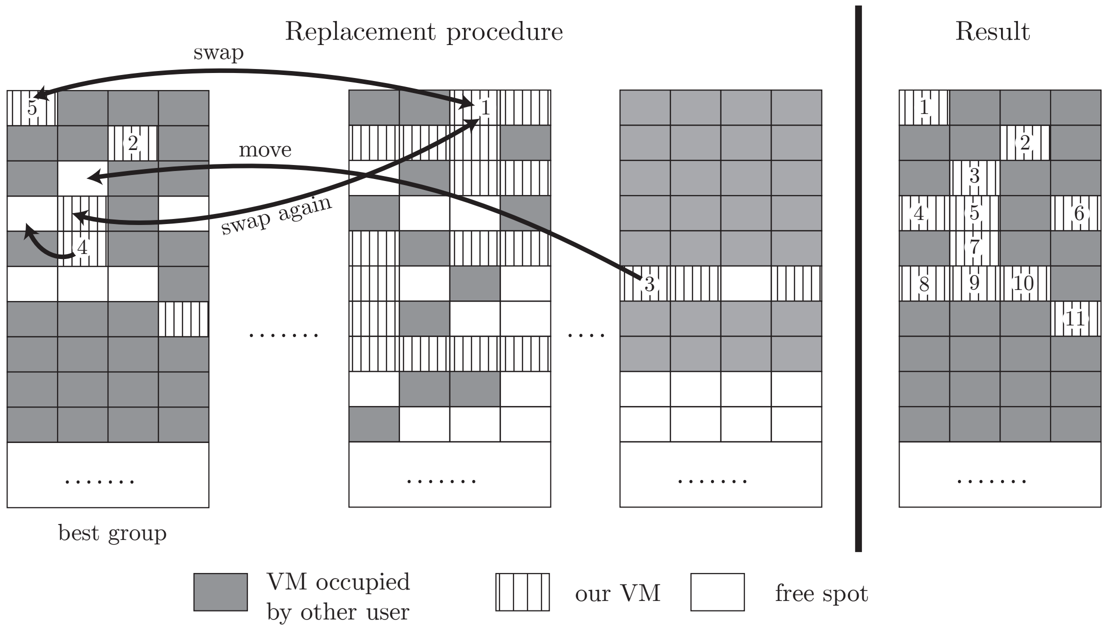

Figure 5: An example of the replacement procedure and its result. The numbers represent the order in which our greedy algorithm picks the new VMs that all have to be placed in one group. On the left are the examples of some moves that our master performs in order to place all the VMs in the best group. On the right we have the state of the group after the replacement is finished.

problem for our situation:

- the *min-cut* problem is NP-complete if k is given as a part of the input [1]. This basically means that it's impossible to find an optimal solution within a reasonable amount of time.
- the existing approximation algorithms [2] are still unusable for our application. The algorithms have complexities of $O(n^3)$ and $O(n^4)$ which for $n = 1000$ will take around 100ms time on modern machines. This runtime is unacceptable given our fast changing environment and the need to rerun the algorithm to adapt to these changes.
- the approximation algorithms don't account for the available sizes of the groups. For example, the algorithm may return a group with a size of 30, however we may not have a group with 30 spots available.
- the approximation algorithms are static, i.e. they can't adapt to fast changes, such as a group getting filled, that can occur while the algorithms are running.

We found that our greedy algorithm is a good approximation of the optimal solution. First of all, it easily adapts to dynamically changing availability of spots in each group. Secondly, it's complexity is only $O(n^2)$, which should take a few milliseconds for the worst case of $n \approx 1000$.

We tested how well the algorithm performs compared to the optimal solution in the following way: we generated "group size" and let the greedy algorithm fill these groups in. To find the optimal solution we generated all possible partitions of the VMs in less than the given number of groups, selected only the ones that would fit in and found the one with the biggest internal workload among the ones selected. We were only able to run the tests for $n \leq 15$ because the number of partitions gets too big for bigger input. We generated 118 random tests and calculated the ratio between the greedy and optimal solutions. You can find the results in Figure 6. For this input the greedy algorithm performed extremely well with the mean approximation ratio 0.9516 (std=0.041). We expect this ratio to decrease for a bigger input, however we

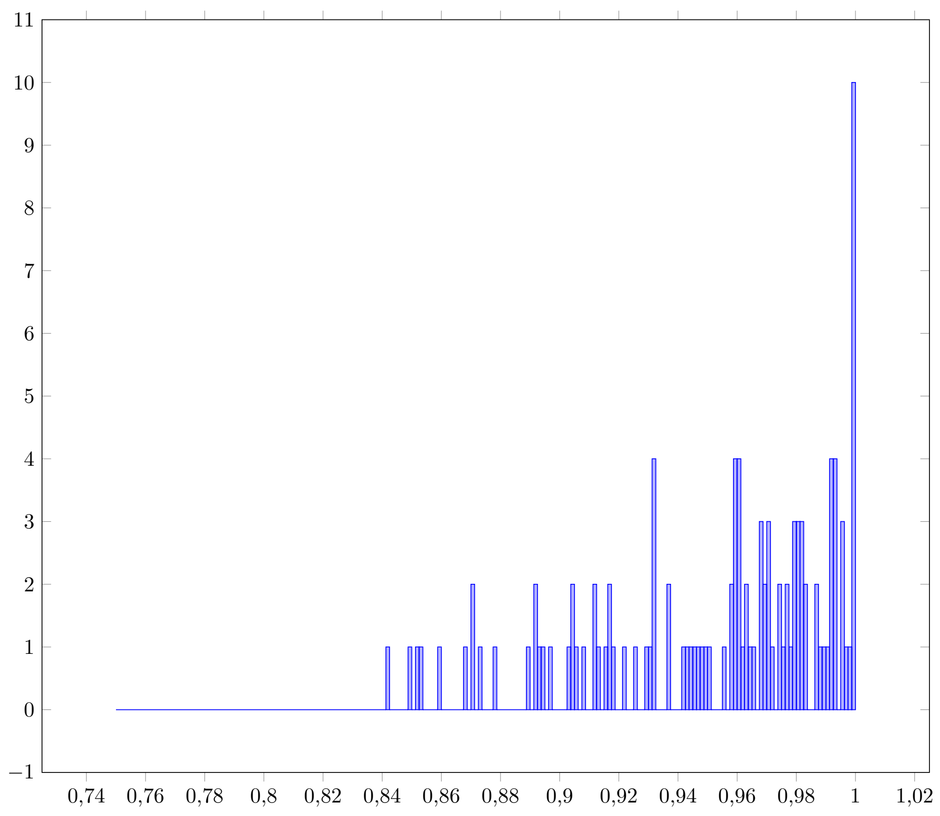

Figure 6: The histogram of the results of the test of the greedy vs optimal solution. The x axis corresponds the the approximation ratio and the y axis corresponds to the number of times this ratio has occurred.

will still get a reasonable approximation even for $n = 1000$.

Therefore, the greedy algorithm serves well our purpose of maximizing the internal workload that relies on high bandwidth links, but it also minimizes the external workload that relies on links with lower performance.

3.2 Performance Analysis in Different Scenarios

3.2.1 First Scenario

Consider a case when each physical machine already hosts 3 VMs, each pair of VMs have the same network throughput, the user has 100 VMs, and each pair of VMs have to send the same amount of data to each other.

In such case, the initial placement of our algorithm will place the first 48 VMs in the first group, the following 48 VMs in the second group, and the last 4 VMs in the third group. We assume that the data transfer between VMs in the same group is very fast and takes a very small amount of time. Every five seconds we'll pause all VMs and carry out measurements for 100 milliseconds. Right after the measurements we'll do replacements. Our algorithm will again take the first 48 VMs in the first group, the second 48 VMs in the second group, and last 4 VMs in the third group. However, the VMs will be grouped in a different order. This is caused by the fact that matrix B also gets updated in the measurement section. Since within a group the connection is much faster than the connection between VMs across groups, matrix B will no longer be balanced. Then, our algorithm will group together those VMs which have more data to transfer. These will be the VMs which were in different groups. For instance, if in the initial placement VMs with indices 1...48 are in the first group, 49...96 are in the second group, and 97...100 are in the third group, then after the replacement, the placements will be as follows: First group will contain VMs 1, 2, 3, ..., 21, 49, 50, 51, ..., 70, 97, 98, 99, 100. The second group will contain VMs 22, 23, 24, ..., 46, 71, 72, 73, ..., 94. And the third group will contain VMs 47, 48, 95, 96. As you can see, the VMs get spread from the group, and those who have more data to transfer will be grouped together. With this process, relocations will happen al long as there is any data to be sent, balancing the matrix B (data to be sent between VMs - which gets updated continuously). In most cases, it is safe to assume that the time it takes to finish the data transfer between two VMs in the same group is less than 5 seconds (because the throughput is very high). In this case, notice that during the first 5 seconds, nearly half of the total data transfer is finished. During the next 5 seconds, again nearly half of the remaining data transfer is finished, and so on. Therefore, the (Number of relocations) will be $O(\log n) =$ 7. Therefore, the expected runtime will be close to 7*(5+0.1)=35.7 seconds. The overhead for the measurement part is 0.1*(Number of relocations), which, in the best case, is 0.7.

3.2.2 Second Scenario

Consider a case where the user's 10 VMs are distributed in groups 1..6, which are highly congested. Then suddenly many machines in Group 19..24 finish working and the links become capable of delivering a high throughput.

According to our system design, the measurement component will activate in the following 5 seconds. It will find out that the cluster 4, which contains groups 19..24, has a potentially much better average throughput than cluster 1, which contains groups 1..6 and therefore the user's 10 VMs. After the measurement process, which takes 100ms, the VMs will be relocated in a better cluster, which is cluster 4 in our case. Moreover, our algorithm will try to put the VMs in the same group if possible, so that we achieve maximum data transfer speeds. However, if this is not possible the master will distribute the VMs in the different groups of the 4th cluster, in which case we'll also get very high throughput. Therefore, in the worst case the total spent time will be t+5+0.1+T, where t is the time when groups 19..24 become available, 5 seconds is the maximum time till the next measurement, 0.1 seconds is the measurement time, and T is the time needed to transfer data between VMs after they're placed in the groups 19..24. Notice, that the relocation can happen many times (if $t > 5$), however the VMs will always be in the least busy cluster. In our case such a cluster is the one containing groups 19..24.

3.2.3 Third Scenario

Consider a case when each user of the data center uses the strategy described in this paper to place VMs. There will be one and only huge problem - the measurements will happen at the same time.

If two or more different users perform measurements at the same time, all of them will place one VM in each group, and try to measure average throughput in each cluster by sending as much data as possible to each other. It's clear that they will get in each other's way, and none of them will receive the real throughput. Since the relocation process depends on the accuracy of these measurements, it means the VM relocations will not be as optimal as before. However, it's not a complete disaster, since in most cases we can assume that the measured throughputs will be close to $\frac{\text{Actual Throughput}}{\text{Amount of measurers}}$, and therefore the order of clusters which are used for the relocation will stay the same.

It is clear that, in order to achieve the best performance, we need a way to prevent concurrent measurements. The way to achieve this is through coordination. First of all, the master VMs of each user need to have a dedicated port, where they'll send their internal messages. Every VM should be sending "Hello" messages to all other machines every 100ms. If the other machine happen to host another user's master VM, then it replies back. Thus, a connection is made and every master VM has a list of every other master VM. The first thing the new master should do, before starting the measurement, is getting a list of other masters. Then, the master VMs should exchange messages to figure out the times each user should run the measurements. The requirement is that the measurement times shouldn't cross. This way up to 50 different user can schedule their 100ms long measurements, so that all the measurements happen consecutively, and every 5 seconds for each user. If the amount of users is $n > 50$, the masters will increase the time between two measurements (0.1*n). Therefore, this addition completely eliminates the concurrent measurements. So, the users won't affect each other negatively anymore and the system as a whole won't have any major problems.

We didn't include this in the previous sections, because it would make the design very complicated to understand.

3.2.4 The Best Scenario

There are two conditions, for which our strategy works particularly well. The first one is when the matrix B is initially naturally clustered. This means that VMs can be divided in some sets $S_1, S_2, \ldots, S_k$ in such a way, that the total information sent between VMs in different sets is much less than the information sent between VMs in the same set:

$$\sum_{\substack{i \in S_k \\ j \in S_p \\ k \neq p}} B[i][j] \ll \sum_{\substack{i \in S_k \\ j \in S_k}} B[i][j]$$

In this case, our greedy algorithm is able to figure out these sets of VMs and puts them in the same group, or cluster. The second condition is when the groups have many available VM slots. This is necessary, because if the size of any S_i is large, then we should be able to put them in the same group. If these conditions are met, then the VMs which have huge amounts of data to send to each other will end up in the same group, and there will be almost no data to be transferred between groups, where the throughput is slow. Therefore, the data transfer will finish as fast as possible. In a simplified case, if the pairs of VMs in the same set have to send D amount of data, the VMs in the different sets doe have to send any information, and the VM slots in the groups of the data center are mostly available, then by this strategy the master will put VMs in the same set physically in the same groups. Therefore, we can assume that the data transfer will finish in $\frac{D}{100Gb/sec}$.

3.2.5 The Worst Scenario

When designing our algorithm, we made several assumptions. The most important one is that the throughput between VMs inside the group is better than the throughput between VMs in different groups, and the throughput between VMs inside a cluster is better than the throughput between VMs in different clusters. Depending on these assumptions, our greedy algorithm tries to put VMs in the same group, or if not possible then in the same cluster. The assumptions are true with very high probability. However, in some cases there can be a very high congestion inside a cluster, in such case it's better to spread out VMs in different clusters compared to clustering them together.

3.3 Extra Cost

As we discussed in section 2.1, the measurement component temporarily allocates additional VMs, in order to understand the current load in different clusters. To be precise, we allocate exactly one VM in each group, if there exists a free VM slot. Therefore, we periodically allocate a maximum of 24 different VMs for 100ms every 5 second. Also, we have the master VM turned on all the time. Suppose the user has n VMs to allocate, the task is finished in t seconds, and the user would pay d dollars without extra cost. Since n VMs are running all the time, the cost of running one VM for one second is $c = \frac{d}{nt}$. Therefore, the cost of running the master VM is $c_{master} = \frac{d}{n}$. The measurement VMs run in total $\frac{0.1}{5}t = 0.02t$ seconds. Therefore, its cost is $c_{measurement} = \frac{d}{nt} * 24 * 0.02t = \frac{0.48d}{n}$. The extra cost is the sum of costs of the master VM and the measurement VMs and is therefore, $c_{extra} = \frac{1.48d}{n}$. The important question is, how much is the extra cost compared to the original cost. The answer is $\frac{c_{extra}}{d} = \frac{1.48}{n}$. This table represents the additional cost, by percentage, added by master VM and the measurement component for different amounts of user's VMs:

Number of User's VMs	10	20	50	100	200	500	1000
Additional Cost	14.8%	7.4%	2.96%	1.48%	0.74%	0.296%	0.148%

The additional cost is inversely proportional with the amount of user's VMs. Therefore, this design is more efficient when the user has many VMs to execute.

4 Conclusion

Our strategy serves well our main goal of maximizing the work within the groups and minimizing the load outside of them. Thus, we mainly rely on the high performance links inside the groups which allowed us to keep our design simple without the need to worry about the links between clusters that will, in most cases, be very congested.

5 References

[1] Garey, M. R.; Johnson, D. S. (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, W.H. Freeman, ISBN 0-7167-1044-7

[2] Saran, H.; Vazirani, V. (1991), *"Finding k-cuts within twice the optimal"*, Proc. 32nd Ann. IEEE Symp. on Foundations of Comput. Sci, IEEE Computer Society, pp. 743-751