Stable Marriage Matching for Homogenizing Load Distribution in Cloud Data Center

Disha Sangar, Ramesh Upreti, Hårek Haugerud, Kyrre Begnum, and Anis Yazidi

Autonomous Systems and Networks Research Group,
Department of Computer Science,
Oslo Metropolitan University, Oslo, Norway

Abstract. Running a sheer virtualized data center with the help of *Virtual Machines* (VM) is the de facto-standard in modern data centers. Live migration offers immense flexibility opportunities as it endows the system administrators with tools to seamlessly move VMs across physical machines. Several studies have shown that the resource utilization within a data center is not homogeneous across the physical servers. Load imbalance situations are observed where a significant portion of servers are either in overloaded or underloaded states. Apart from leading to inefficient usage of energy by underloaded servers, this might lead to serious QoS degradation issues in the overloaded servers.
In this paper, we propose a lightweight decentralized solution for homogenizing the load across different machines in a data center by mapping the problem to a Stable Marriage matching problem. The algorithm judiciously chooses pairs of overloaded and underloaded servers for matching and subsequently VM migrations are performed to reduce load imbalance. For the purpose of comparisons, three different greedy matching algorithms are also introduced. In order to verify the feasibility of our approach in real-life scenarios, we implement our solution on a small test-bed. For the larger scale scenarios, we provide simulation results that demonstrate the efficiency of the algorithm and its ability to yield a near-optimal solution compared to other algorithms. The results are promising, given the low computational footprint of the algorithm.

Keywords: Self-Organization · Cloud Computing · Stable Marriage · Distributed Load Balancing

1 Introduction

Major systems and Internet based services have grown to such a scale that we now use the term "hyper scale" to describe them. Furthermore, hyper scale architectures are often deployed in cloud based environments, which offers a flexible pay-as-you-go model.

From a system administrator's perspective, optimizing a hyper scale solution implies introducing system behaviour that can yield automated reactions to changes in configurations and fault occurrences. For instance, auto scaling is a desired behaviour model for websites to optimize cost and performance in accordance to usage patterns.

There are two different perspectives on how an automated behaviour can be implemented within the field of cloud computing. One of the perspectives is to implement the behaviour in the infrastructure, which is the paradigm embraced by the industry. The other alternative is to introduce behaviour as a part of the Virtual Machine (VM), which opens up a possibility for cloud independent models.

Several studies have shown that the resource utilization within a data center varies drastically across the physical servers [12, 28, 4]. Load imbalance situations are observed where a significant portion of servers are either in overloaded or underloaded states. Apart from leading to inefficient usage of energy by the presence of underloaded servers, this might lead to serious QoS degradation issues in the overloaded servers. The aim of this paper is to present an efficient and yet simple solution for homogenizing the load in data centers. The potential gain with this research is to find an efficient and less complex way of operating a data center. Stable Marriage is a an intriguing theory emanating from the field of economy and holds many promises in the field of computer science and more particularly in the field of cloud management. In this paper, we apply the theory of Stable Marriage matching in order to devise a load homogenizing scheme within a data center. We also modify the original algorithm in order to support distributed execution.

Various studies on *s*elf-organizing approaches have been emerging in the recent years to efficiently solve computationally hard problems where centralized solutions might not scale or might also create a single point a failure.

The aim of this paper is to provide a distributed solution for achieving distributed load balancing in a data center which is inspired by the the Stable Marriage algorithm [16]. The algorithm implements message exchange between pairs of servers. It is worth emphasizing that modern distributed systems often use gossip protocols to solve problems that might be difficult to solve in other ways, either because the underlying network has an inconvenient structure, is extremely large, or because gossip solutions are the most efficient ones available [17] in terms of communication. We shall adopt the Stable Marriage algorithm and study its behavior under different scales.

2 Stable Matching

According to Shapley et al. [15] an allocation where no individual perceives any gain from any further trade is called *Stable*. Stability is a central theory in the field of cooperative game theory that emanates from mathematical economics which seeks to know how any constellation of rational individuals might cooperate to choose an allocation.

Shapley [14] introduced the concept of *pairwise matching*. Pairwise matching studies how individuals can be paired up when they all have different preferences regarding who are their best matches. The matching was analyzed at an abstract level where the idea of marriage was used as an illustrative example.

For this experiment Shapley et al. tested how ten women and ten men should be matched, while respecting their individual preferences. The main challenge was to find a simple method that would lead to stable matching, where no couples would break up and form new matches which would make them better off. The solution was *deferred acceptance*, a simple set of rules that always led straight to the stable matching.

Deferred acceptance can be set up in two different ways, either men propose to women or women propose to men. If women propose to men the process begins with each woman proposing to the man she likes the best. Each man then looks at the different proposals he has received, if any, and regards the best proposal and rejects the others. The women who were rejected in the first round, then move along to propose to their second best choice. This will continue in a loop until no women wants to make any further proposals. Shapley et al. [15] proved this algorithm mathematically and showed that this algorithm always leads to stable matching.

The specific way the algorithm was set up turned out to have an important distributional consequence. The outcome of the algorithm might differ significantly depending on whether the right to propose was given to the women or to the men. If men proposed this lead to the worst outcome from the women's perspective. This is because if women proposed, no woman would be worse off than if the men had been given the right to propose [15].

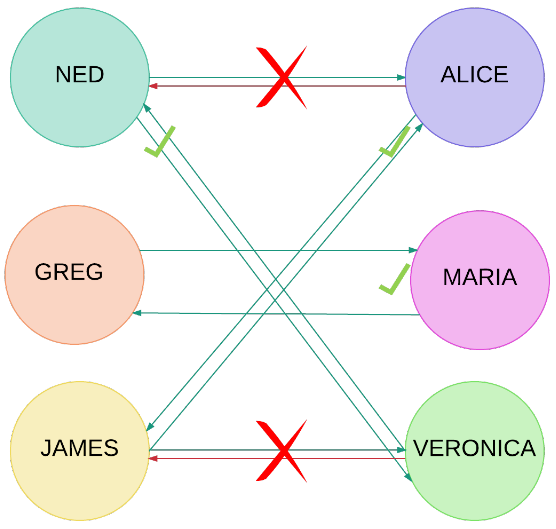


Fig. 1. Stable Matching

The model depicted in Figure 1 presents the selection process for Stable Matching. On the right side we find the women with their preferences and to the left the men with their respective preferences.

3 Related work

In this section, we shall review some prominent works on distributed approaches for homogenizing the load in a data center. It is worth mentioning that the related work in this particular area is rather sparse.

Marzolla et al. [17] propose a decentralized gossip-based algorithm, called *V-MAN* to address to the issues regarding consolidation. V-MAN is invoked periodically to create consolidate VMs into fewer servers. They assume that the cloud system has a communication layer, so that any pair of servers can exchange messages between them. The work of Marzolla et al. [17] yields very promising results which show that using V-MAN converges fast – after less than 5 rounds of message exchanging between the servers.

In [3], the authors use scouts which are allowed to move from one PM (physical machine) to another – to be able to recognize which compute node might be a suitable migration destination for a VM. This is completely opposite of what V-MAN does. It does not rely on any subset like scouts, instead each server can individually cooperate to identify a new VM location, which makes V-MAN scalable. It is also to be noted that any server can leave or join the cloud at any time.

Sedaghat et al [25] use a Peer-to-Peer protocol to achieve energy efficiency and increase the resource utilization. The Peer-to-Peer protocol provides mechanisms for nodes to join, leave, publish or search for a resource-object in the overlay or network. This solution also considers multi-dimensionality – because the algorithm needs to be specified to be dimension aware, each PMs proportionality should be considered. Each node is a peer where a peer sampling service, known as newscast, provides each peer with a list of peers whom are to be considered neighbours. Each peer only know k random neighbours which map its local view. The aim is to improve a common value which is defined as the total imbalance of each pair at the time of decision-making by redistributing the VMs.The work uses a modified dimension aware algorithm to tackle the multi-dimensional problem. The algorithm is iterative and starts from an arbitrary VM placement. When the algorithm converges, a reconfiguration plan is set so the migration of the VMs can start.

A survey by Hummaida et al. [10] is focused on adaptation of computing resources. In [29], a peer-to-peer distributed and decentralized approach is proposed that enables servers to communicate with each others without a centralized controller. It uses a node discovery service which is run periodically to find new neighbouring servers to communicate with. The algorithm decides whether two servers should exchange a VM based on the predefined objectives. Similarly, in [20], the authors propose a decentralized approach for user-centric elasticity management to achieve conflict free solutions between customer satisfaction and business objectives. Dynamical allocation of CPU resources to servers is performed in [13] which integrates the Kalman filter into feedback controllers to track the CPU utilizations and update the allocations accordingly.

Siebenhaar et al. [26] use a decentralized approach to achieve better resource management using a two phase negotiation protocol to conduct negotiations with multiple providers across multiple tiers. Moreover, in [8] the authors present an architecture for dynamic resource provisioning via distributed decisions where each server makes its own utilization decision based on its own current capacity and workload characteristics. The authors also introduce a light-weight provisioning optimizer with a replaceable routing algorithm for resource provisioning and scaling. The authors claim that with this solution the resource provisioning system will be more scalable, reliable, traceable, and simple to manage.

Calcavecchia et al. [6] start by criticizing centralized solutions and states that it is not suitable for managing large-scale systems. The authors introduce a Decentralized Probabilistic Auto-Scaling Algorithm (DEPAS) which is integrated into a P2P architecture and allows simultaneous auto-scaling of services over multiple cloud infrastructures. They conduct simulations and real platform based experiments and claims that their approach is capable of handling (i): keeping track of: overall utilization of all Instant Cloud resources within the target range, (ii): maintaining service response times close to those achieved through optimal centralized auto-scale approaches.

Another interesting decentralized approach is presented by [5] where an emphasis is given to low price based deployment and dynamic scaling of component based applications to meet SLA performance and availability goals. The work gives priority to a low price model instead of normalizing loads to all servers, meaning the dynamic economic fitness of the servers will decide whether resources are replicated, migrated to another server, or deleted. Each server stores the table of complete mapping between instances and servers and a gossip protocol is used for mapping instances.

In another survey [21] the authors aim to classify and provide a concise summary of the several proposals for cloud resource elasticity today. They present a taxonomy covering a wide range of aspects, and discuss details for each of the aspects, and the main research challenges. Finally, they propose fields that require further research. A more recent article on cloud computing elasticity [1] reviews both classical and recent elasticity solutions and provides an overview of containerization. It also discusses major issues and research challenges related to elasticity in cloud computing. The authors comprehensively review and analyze the proposals developed in this field.

In a survey on elasticity management in PaaS systems [19] the authors claim that ideally, that elasticity management should be done by specialised companies: the platform as a service (PaaS) providers. The PaaS layer is placed on top of an IaaS layer in a regular cloud computing architecture. The authors provide a tutorial on the requirements to be considered and the current solutions to the challenges being present in elastic PaaS systems and conclude that elasticity management in the PaaS service model is an active research area amenable to improvement.

Most of the papers mentioned above concern scaling of resources and load distribution by developers. However, there has been some recent development in the field of cloud computing which might entirely shift the burden of scaling resources from developers and system designer to cloud providers. This new cloud computing paradigm is called Serverless Computing. The term Serverless Computing is a platform (Function-as-a-service) that hides the server usage from the developer and runs code on-demand, scaling accordingly and only billing for the time the code is running [7].

The term Serverless can however be a bit misleading, serverless doesn't equal to no backend service, it just means that all of the server space and infrastructure issues are handled by the vendor [27]. Many larger companies (e.g Amazon AWS, Azure, etc) now offer these type of solutions, where you can deploy your code to the cloud, and only pay for the amount of time the code is used, while all the administration of the servers are handled by the vendor [2].

AWS was one of the first vendors to introduce the concept of serverless computing in 2014 [11]. According to Castro et al. [7] serverless seems to be the natural progression in the advancement and adoption of VM and container technology, where each layer of the abstraction leads to more lightweight units of computation, saving resources, being more cost effective and increasing the speed of development. Castro et al. conclude that serverless computing lowers the bar for developers by delegating to the platform provider much of the operational complexity of monitoring and scaling large-scale applications. However, the developer now needs to work around limitations on the stateless nature of their functions, and understand how to map their application's SLAs to those of the serverless platform and other dependent services [7].

4 Solution

4.1 Overview of a functioning framework

As the algorithm implemented will be based on a real life inspiration, it is important to understand that the outcome can end in two different cases. Just as each relationship does not end in marriage neither will the decision of the PMs. Each PM can be viewed as individuals making their own "life choices".

Figure 2 and 3 enhances the different outcomes the algorithm can opt for and how the framework is set up to work around the execution of the algorithm. Note that the environment later implemented is not in an actual data center. Our main goal is to achieve load balance in a distributed cloud data center, but our solution is restricted to load-balancing of CPU-intensive applications as we have not considered the impact of other resources such as memory, network and disk. The intention is to create a framework that can handle any given scenario or setup for CPU-intensive applications.

The basic framework for both scenarios are the same, it is a data center consisting of PMs with different weight. However, as explained in the section

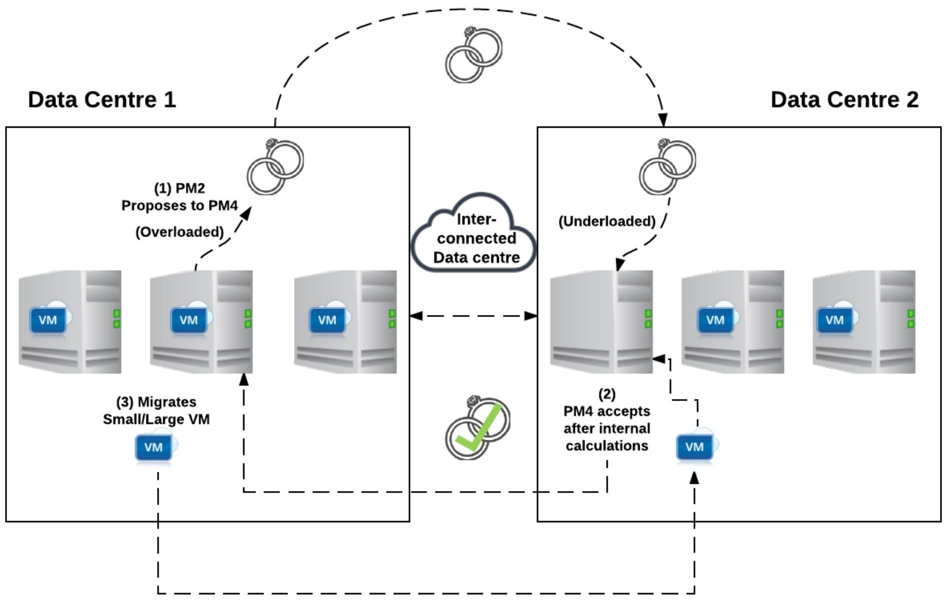


Fig. 2. Proposal accepted

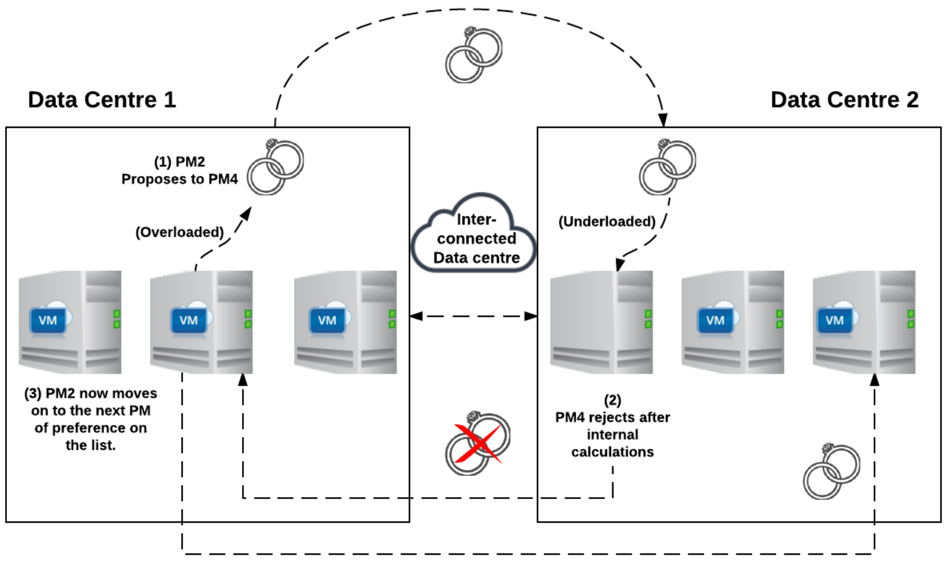


Fig. 3. Proposal Rejected

above, based on the calculations of the underloaded server in the second scenario the proposal is rejected and the PM moves on to the next best on their list. This process is supposed to be a continuous process, unless the target load for each PM is achieved, then the process stops entirely.

4.2 Bin Packing with Stable Marriage

The bin packing problem is the challenge of packing an amount n of objects in to as few bins as possible. In this case, the servers are the bins and the VMs are the objects. This terminology fits the Stable Marriage algorithm well, as the bins are the humans who are in quest of a partner (bin) which represents a good match while satisfying some constraints in terms of capacity. A constraint can be defined in many ways, but for a bin some common constraints would be the height of the box, its width and depth. Our algorithm will focus on Virtual CPUs (VCPUs) and memory as constraints. The aim of the algorithm is to even and equalize the load of the data center by evenly distribute the weight of the VMs across the bins in such a way that the bins should neither be overfilled nor underfilled.

4.3 Stable Marriage Animation

A known set of servers is divided into two groups of overutilized (men) and underutilized (women). The goal is to find a perfect match for the overutilized servers. The matchmaking is based on three values, the average CPU load, the imbalance before and after migration (calculated before the eventual migration) and the profit of such a marriage.

The figures below demonstrate the expected outcome of implementing the Stable Marriage algorithm. This approach is mainly centralized and the PMs know the allocated values of each other. This means that each PM, both over and underutilized, has a list of preferred men and women they want to propose to or receive a proposal from.

Figure 4 shows that PM1 has reached full capacity as marked by the red line. The red line represents the average capacity that each PM can handle. Assume that each group of men can only handle four or six full servers, in this case PM1 has then reached its full capacity and so has PM2. They need to migrate the load to a underloaded PM of preference, so that they can balance the load equally. Hence, the overloaded server PM1 proposes to his first choice, PM3.

The female set of servers have their own method to calculate the advantage/disadvantage of such a marriage. If the underutilized PM calculates a higher imbalance than before the marriage, she sees this as a disadvantage and rejects the proposal. This method also avoids that the proposing party becomes underutilized in the future.

After being rejected, PM1 proceeds to his next best choice which is PM4. PM4 then calculates the imbalance before and after the proposed marriage to check if it improves after a potential migration. In this case the imbalance factor improves, and PM4 accepts the proposal. The migration can now take place.

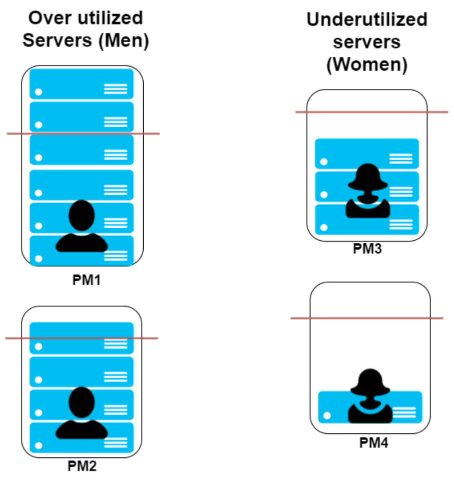


Fig. 4. Set of over/under utilized servers

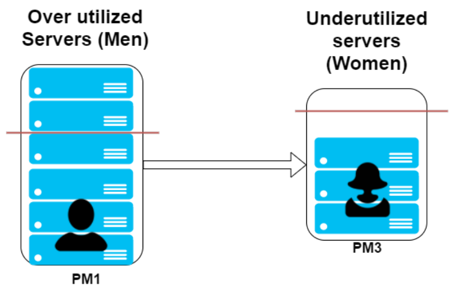


Fig. 5. PM1 proposes to PM3

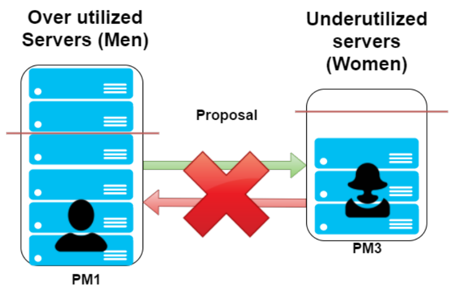


Fig. 6. PM3 rejects PM1 seeing no benefit to this marriage.

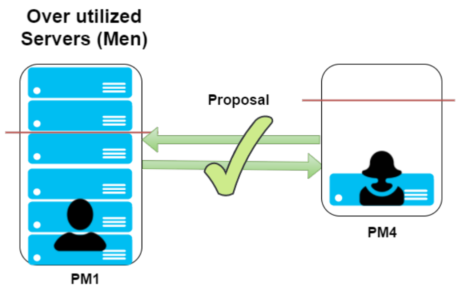


Fig. 7. PM4 accepts PM1's proposal

Since PM4 has the same amount of capacity to accept load, the server is not over-utilized and the load has been balanced between the married PMs.

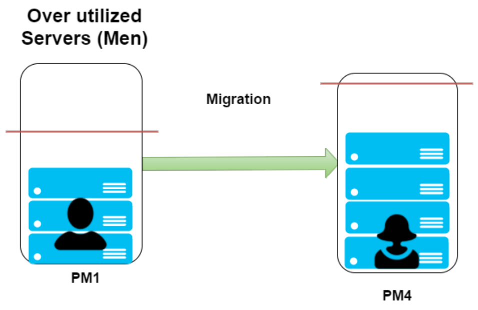


Fig. 8. Migration successful

This particular animation doesn't have any scheme implemented, it just gives an idea of how the algorithm is supposed to work. The schemes will only make a difference in terms of the size of the VMs that is migrated. The figure below shows how the VM sizes may differ on each PM and how the migration process may look inside each server.

4.4 The Proposed Stable Marriage Algorithm

In our preliminary version of this work [24], we proposed two implementations of the Stable Marriage algorithm called *Migrate Smallest-Greedy Matching* (MS-GM) and *Migrate Largest-Greedy Matching* (ML-GM). We observed that both of the methods have some pitfalls. The *Migrate Smallest* algorithm usually results in a better overall result, but at the cost of a higher number of migrations. On the other hand, the *Migrate Largest* algorithm requires a smaller number of

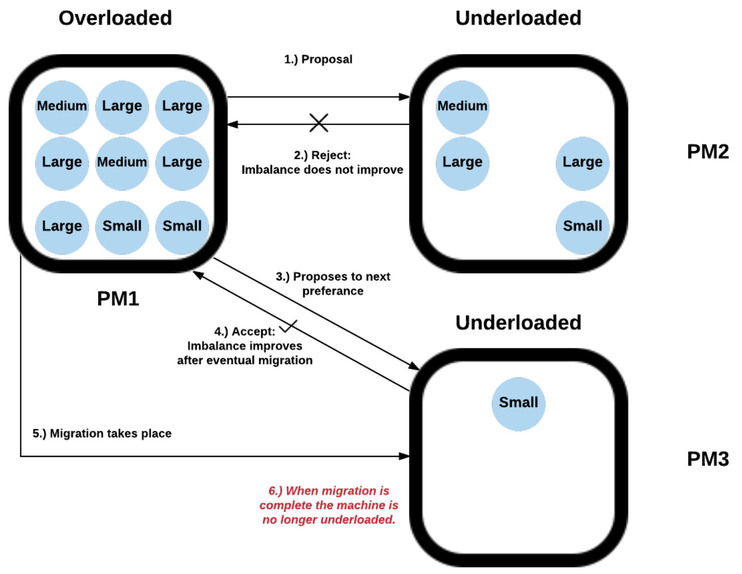

Overloaded
Underloaded
1.) Proposal
Medium
Large
Large
Large
Medium
Large
Large
Small
Small
2.) Reject:
Imbalance does not improve
Medium
Large
Large
Small
PM2
PM1
3.) Proposes to next preferance
4.) Accept:
Imbalance improves after eventual migration
Underloaded
Small
5.) Migration takes place
PM3
6.) When migration is complete the machine is no longer underloaded.

Fig. 9. PMs with various VM sizes

migrations in order to reach a final state at the cost of a higher final imbalance. In order to improve on these two algorithms, we propose a Stable Marriage based approach which will provide a minimum imbalance result while still yielding a low number of migrations. In addition to this, in order to compare with the efficiency of new the Stable Marriage (SM) algorithm, three different kinds of greedy matching algorithms are introduced.

In the example shown in Figure 9 PM1 is overloaded while PM2 and PM3 are underloaded. As can be seen, PM1 is highly overloaded and PM3 is highly underloaded, therefore the SM algorithm pairs PM1 and PM3 and migrates the largest VM of PM1 to PM3. Then the algorithm will calculate the imbalance of each PM again and compute the overload and underload. Based on the amount of overload and underload, SM will determine a new pair of PMs and move the optimal VM based on size. This process will continue until a minimum imbalance is achieved. Figure 10 illustrates that the resources allocated to the VMs are different. The resources of the three VMs correspond to those of the three smallest E2 high-CPU machine types of Google Cloud Engine. We restrict our study to these three VM sizes as it simplifies the solution while this design choice still is complex enough to illustrate the usefulness of our solution. It is straightforward to generalize this by simply including VMs of different sizes. It should be noted that in the experiments only the number of CPUs is taken into account when determining whether a server is underloaded or overloaded. In a more extensive solution other resources and features, like memory, network bandwidth, disk usage, runtime monitoring, message overheads and migration time should be taken into account. However, for CPU intensive applications our solution includes the most important feature.

The complexity per time instance of the SM algorithm is $O(n_o n_u)$ where n_o and n_u are the numbers of overloaded, respectively underloaded servers at each time instant. Thus, the number of messages exchanged is in the order of $O(n_o n_u)$ at each iteration of the algorithm. The complexity of the greedy ones is $O(n_u)$ since the overloaded servers are those that propose. The upper bound on the running time (and the number of migrations) is the total number of VMs in all servers.

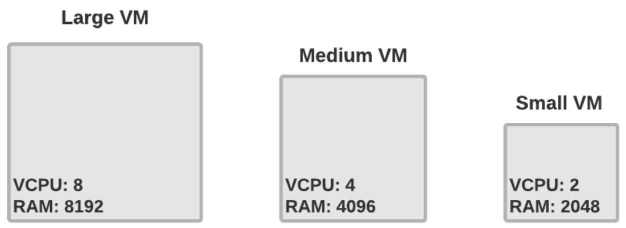


Fig. 10. VMs with their allocated resources.

4.5 The Greedy Matching Approach

The greedy matching approach is inspired by the greedy algorithm. In this approach we first take the one most overloaded server and compare the imbalance with every underloaded server with a predefined movement method and choose the one that will reduce the imbalance most. If there are multiple cases which have the same imbalance result, one of them will be picked at random. Once this process is done, the algorithm again calculates the imbalance of each server and continues the above process until the minimum imbalance result is gained. The three predefined movement methods are denoted as Migrate Smallest (MS), Migrate Medium (MF) and Migrate Largest (ML). For each of the methods, a VM of the size corresponding to the name of the method is migrated from the most overloaded PM to the most underloaded PM.

5 Implementation of Stable Marriage

In this section we give details about the implementation of the Stable Marriage algorithm which aims to find the perfect match for gaining load balance. Each node is considered an individual with preferences and demands. These are taken into consideration to be able to find the perfect balance for each individual node.

The flow diagram in Figure 11 gives an insight into how the Stable Marriage Algorithm operates and the different procedures involved.

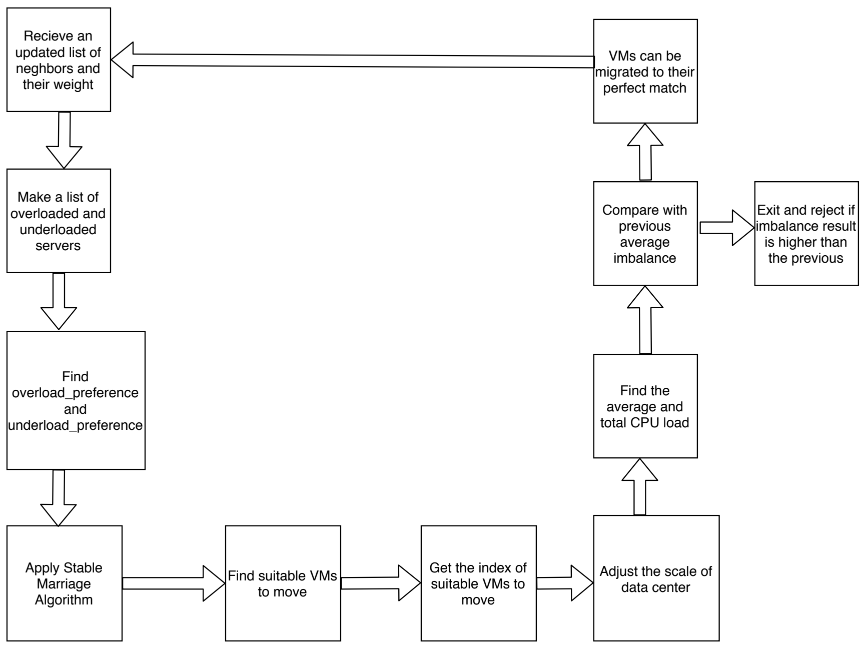

Recieve an updated list of neghbors and their weight
Make a list of overloaded and underloaded servers
Find overload_preference and underload_preference
Apply Stable Marriage Algorithm
Find suitable VMs to move
Get the index of suitable VMs to move
Adjust the scale of data center
Find the average and total CPU load
Compare with previous average imbalance
Exit and reject if imbalance result is higher than the previous
VMs can be migrated to their perfect match

Fig. 11. Flow Diagram of the Stable Marriage Implementation

The following describes the most important parameters of the algorithm inspired by Rao et al. [23]. We define the CPU load to simply be the number of VCPUs of a VM. Let C_i be the total CPU load of server i contained in its VMs and let c_j be the number of VCPUs assigned to VM_j:

$$C_i = \sum_{VM_j} c_j$$

Each server i has a maximum CPU capacity C_i^{max} which is its number of physical CPUs and we define this to be the maximum number of VCPUs a server can contain:

$$C_i \leq C_i^{max}$$

When it comes to consolidation, most algorithms take into account the bottleneck resource as a sole criterion for achieving better consolidation decisions. Similarly, when it comes to load balancing, one can base the algorithm on the imbalanced resource whether it is CPU or memory. For the sake of simplicity, we assume that the CPU is the most imbalanced resource in our data center, which in real life is often the case.

Average load $\overline{C}$ is defined as the average CPU load of the N Physical Machines:

$$\overline{C} = \sum_{PM_i} C_i/N = \sum_{i=1}^{N} C_i/N$$

If the system was perfectly balanced and all servers of same size, each server would have this number of VCPUs. Furthermore, we define the average capacity of a server as

$$\overline{C^{max}} = \sum_{PM_i} C_i{}^{max}/N$$

We define the target load to be the result when evenly distributing the load according to the capacity of the servers. Let T_i be the the target load at PM_i when there is no imbalance:

$$T_i = \frac{C_i{}^{max}}{\overline{C^{max}}}\overline{C}$$

If all the machines have the same capacity, this would reduce to equal load on each server:

$$T_i = \overline{C} = \sum_{PM_i} C_i/N$$

We define the imbalance I_i of a server or physical machine PM_i in terms of CPU load as the deviation of the load of machine PM_i from the target CPU load:

The following pseudo code shows how the possible gain of a migration between an overloaded server and an underloaded server is calculated:

```
Gain_of_Migration_Couple
# Calculate imbalance before an eventual migration
<calculate imbalance of overloaded server>
<calculate imbalance of underloaded server>
overloaded_pref = [make preferences based on overloaded size ]
underloaded_pref = [make preferences based of underloaded size]
SM = stablemarriage(overloaded_pref, underloaded_pref)
for each pair on SM
    check difference and move suitable VMs
continue the process until minimum imbalance is achieved
```

The Stable Marriage algorithm operates in rounds and it stops when no more “gain” in terms of reducing imbalance can be achieved. The algorithm will then exit. In other terms, if there is no beneficial proposal that reduces the imbalance or the proposals will increase the imbalance, the algorithm will stop whenever there are no possibilities to reduce further imbalance. It also restricts overloaded servers to become underloaded, which means that PMs may also decline a proposal if the overloaded server becomes underloaded. This means that a node can never become overbalanced again or underbalanced to take more VMs on board. This is an important part of the implementation, as the point of the Stable Marriage algorithm is to *stabilize* the system, this algorithm contributes to the stability factor.

6 Experiments

6.1 Experimental Set-up

Figure 12 is a model which gives an overview of the structure in which the project will be implemented. This is a figure which shows how the different components from entirely different worlds are paired together. The bottom layer is the physical hardware consisting of PM1-PM3 or Lab01-Lab03 which are the assigned name on the OS. This layer is controlled by the hypervisor KVM, which is in control of the virtual environment, also the network of VMs which are later spawned in layer 3.

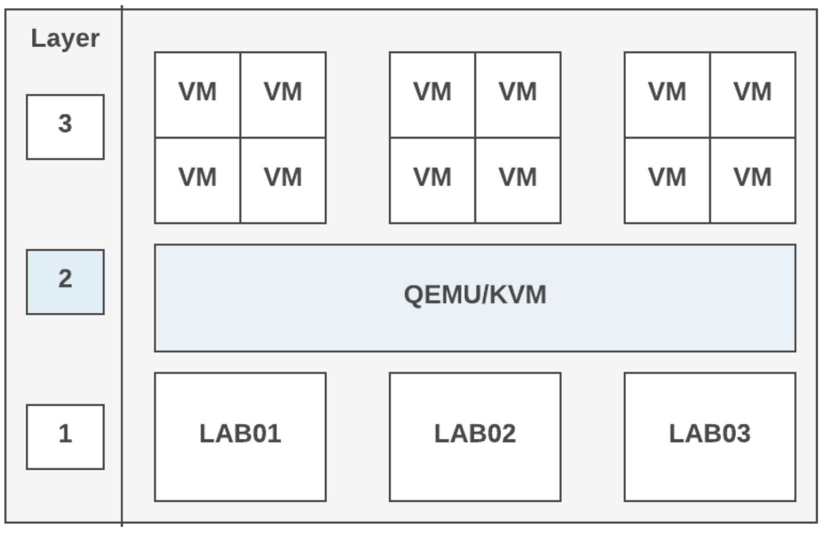


Fig. 12. Design

6.2 Environment Configuration

Evidently, a framework is built with several services and components, which are necessary for an environment to work. To set up a virtual environment for this project several physical and virtual technologies were necessary.

The physical servers in this project are stored in a server room at Oslo Metropolitan University. There are three dedicated servers for this project, as seen in figure 13. The setup consists of a dedicated gateway to connect to the outside. All of the PMs are inter-connected through a dedicated switch.

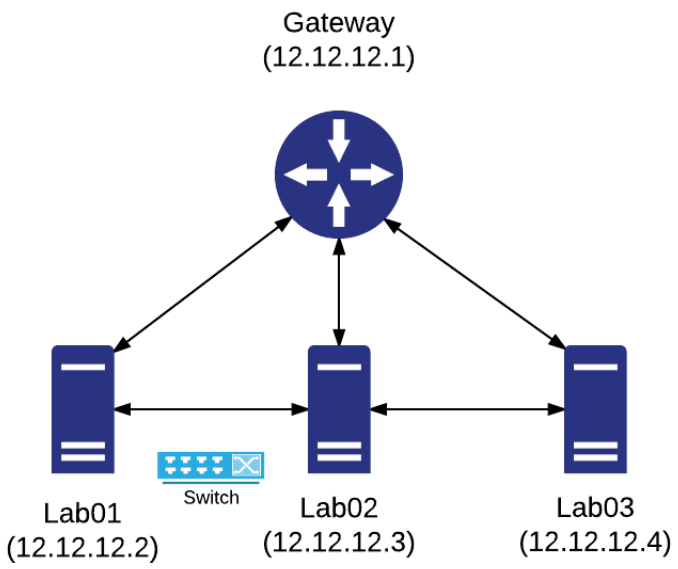
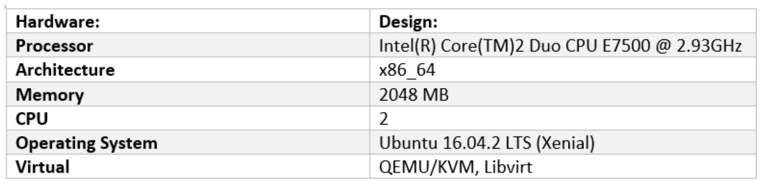


Fig. 13. Overview of the Physical lab structure

Each server is allocated with same specifications:



Hardware:	**Design:**
Processor	Intel(R) Core(TM)2 Duo CPU E7500 @ 2.93GHz
Architecture	x86_64
Memory	2048 MB
CPU	2
Operating System	Ubuntu 16.04.2 LTS (Xenial)
Virtual	QEMU/KVM, Libvirt

Fig. 14. Physical attributes

These are the details for the physical hardware which are dedicated for the virtual implementation. The PMs run Ubuntu which is easier to work with especially with QEMU and KVM for virtualization of the environment.

6.3 Virtual Configuration

The next step is to configure the virtual network. This network will also ensure that when migrating VMs from one host to another, this happens within the same virtualized network. Figure 15 below shows how the PMs are connected and how the VMs reside inside the PMs. The VMs are attached to a virtual bridge by birth. This is a actually a virtual switch, however it is called a bridge and used with KVM/QEMU hypervisors to be able to use live migration for instance.

To connect the PMs together, a physical switch is used.

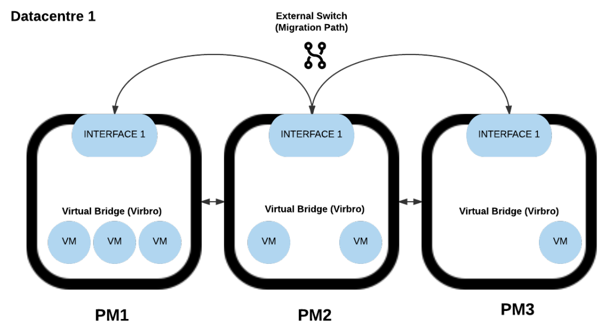


Fig. 15. Physical Lab details

The different VM flavors that will be used in the experiments are given in Figure 16. Each PM will be given a combination of VMs of different flavors.

6.4 Comparison of Greedy Matching and Stable Marriage

The aim of these experiments is to show and compare the impact of different types of greedy matching approaches with the Stable Marriage migration approach in terms of imbalance and the number of migrations taking place in a virtualized environment with different flavours of VMs. For the sake of clarity, all the abbreviations used in the algorithms are summarized in the Table 1. To test the effectiveness of the migration algorithms, three different test scenarios have been created, TEST-10, TEST-50 and TEST-100. The TEST-10 experiment refers to a small scale system with a bin size of 10 and where each bin or PM contains a number of VMs between 5 and 10. Similarly, TEST-50 refers to

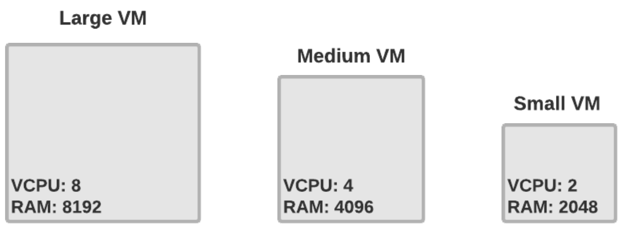


Fig. 16. VM flavors

medium system with bins size of 50 where each bin contains a random number of VMs in the range 40 to 50. Lastly, TEST-100 refers to a large scale system, with 100 bins and each bin has a random number of VMs between 80 and 100. The results presented below are the average result of 100 experiments. In order to find the most effective approach each experiment is performed using all the four algorithms.

Abbreviation	Definition
GM	Greedy Matching
SM	Stable Marriage
MS-GM	Migrate Smallest-Greedy Matching
MM-GM	Migrate Medium-Greedy Matching
ML-GM	Migrate Largest-Greedy Matching

Table 1. List of abbreviations used for algorithms

Migrate Smallest-Greedy Matching vs Stable Marriage Figure 17 and 18 depict the results for the three different test scenarios and shows the comparison between Migrate Smallest-Greedy Matching (MS-GM) and Stable Marriage (SM) migration approaches in terms of imbalance and migrations. In TEST-10, the initial average imbalance was 8.65 which is reduced to 1.18 by the Migrate Smallest-Greedy Matching (MS-GM) approach while the Stable Marriage approach minimized the initial imbalance to 0.83. Interestingly in Figure 18, we can see the Stable Marriage approach takes less than half the number of VM migrations compared to the MS-GM approach for the two largest experiments.

In TEST-50, the initial average imbalance was 18.06. The MS-GM approach brought down this imbalance to 0.81 after 399 VM migrations while SM reduced the initial average imbalance to 0.69 after only 145 VM migrations. In TEST-100, the initial average imbalance was 29.82. The SM approach required 223

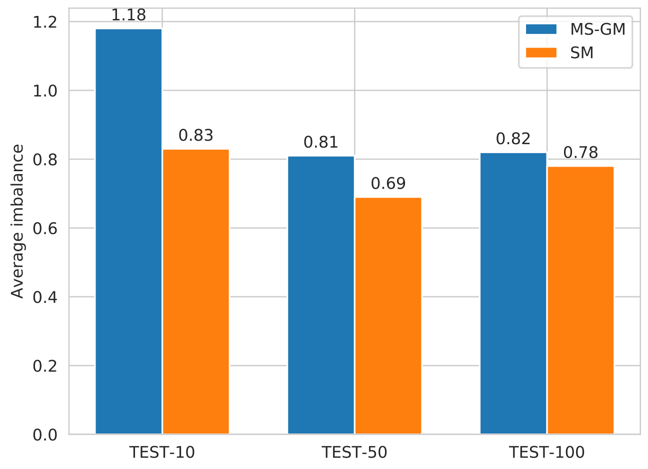


Fig. 17. Migrate Smallest-Greedy Matching (MS-GM) vs Stable Matching (SM)

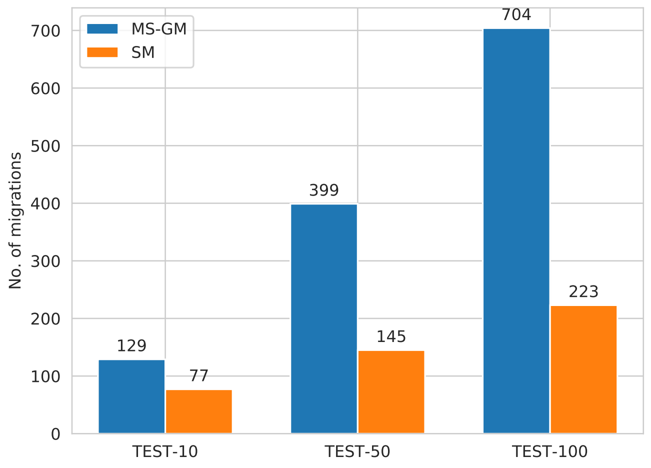


Fig. 18. MS-GM migrations vs SM migrations

migrations to reduce the imbalance to 0.78 while MS-GM required more than three times this number of migrations, 704, and minimized the imbalance to 0.82 which is higher than the result of SM.

Migrate Medium-Greedy Matching vs Stable Marriage Figure 19 shows a comparison of the results of Migrate Medium-Greedy Matching and Stable Marriage approach obtained from three different tests. In TEST-10, the MM-GM approach reduced the initial average imbalance to 2.59 while the SM approach yielded a more than three times smaller imbalance result. From Figure 20 we see that SM not only gave better imbalance results but it also needed a smaller number of migrations. Also for TEST-50 and TEST-100, SM provided better imbalance results compared to MM and needed fewer migrations in order to reach these results.

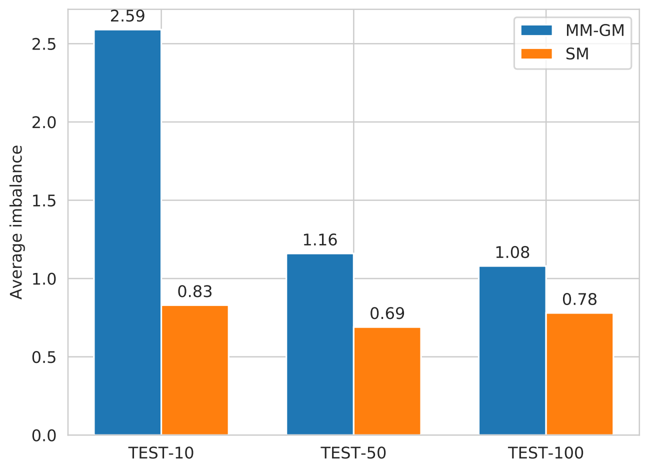


Fig. 19. Migrate Medium-Greedy Matching (MM-GM) vs Stable Matching (SM)

Migrate Largest-Greedy Matching vs Stable Marriage Figure 21 shows the results obtained from the three different tests for Migrate Largest-Greedy Matching (ML-GM) and Stable Marriage (SM). In this comparison, we can observe that in all the three tests, ML was not able to reduce the initial average imbalance to less than 2 while SM minimized the imbalance to less than 0.83 all cases. On the other hand, in Figure 22 we see that ML-GM required slightly

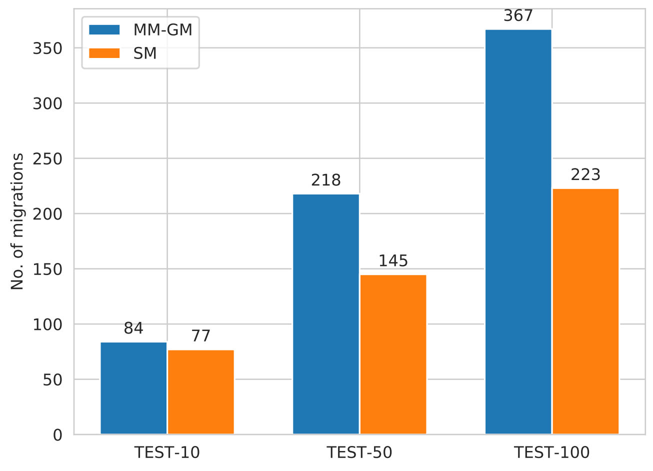


Fig. 20. MM-GM migrations vs SM migrations

fewer VM migrations than SM. This is because when ML-GM migrates VMs it always moves the largest VMs and then the total number of migrations will be smaller than for other methods.

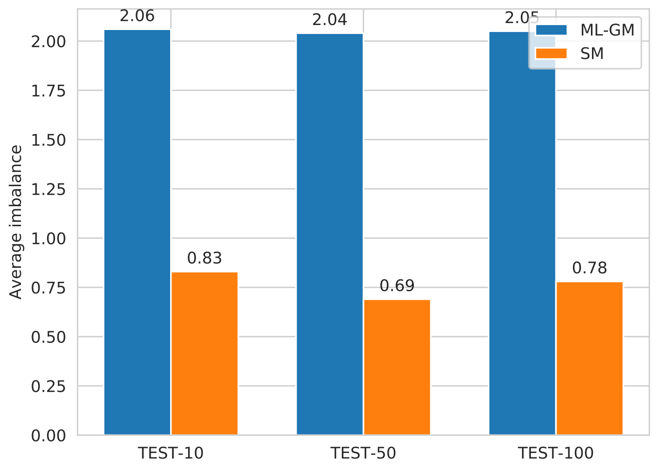


Fig. 21. Migrate Largest-Greedy Matching (ML-GM) vs Stable Matching (SM)

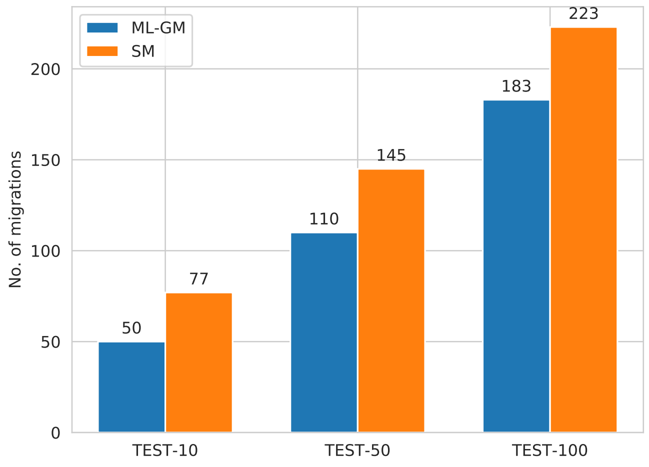


Fig. 22. ML-GM migrations vs SM migrations

In TEST-100, the ML-GM approach reduced the initial average imbalance to 2.05 from 29.82 whereas SM minimized it to 0.78 which is a nearly three times better result. However, ML-GM needed just 183 migrations and SM needed 41 more migrations to reach its lowest imbalance state.

Overall imbalance reduction Figure 23 illustrates how the average imbalance was reduced as function of the number of migrations for the four algorithms tested for the TEST-100 experiment. For the greedy matching algorithms, as the size of the VMs moved decreases an increasing number of iterations are needed to reach the final state . This is not surprising as a larger number of small VMs must be moved when reducing an imbalance of the same magnitude compared to when moving large VMs. However, migrating using small VMs leads to a final state with smaller imbalance. The final state of the SM algorithm has on average the smallest imbalance and at the same time it uses just a few more migrations to reach the final state compared to the ML-GM algorithm.

Figure 24 shows the reduction of average imbalance as function of the number of iterations needed to reach the final state of smallest possible imbalance. It is apparent that the SM algorithm needs an order of magnitude fewer iterations to find imbalance compared to the greedy algorithms. This is for a large part due to the fact that the SM algorithm for each iteration migrates VMs between all the bins in parallel while the GM algorithms migrates just a single VM for each iteration. Nevertheless, this means that the SM algorithm in a real data center will reach a balanced state much faster than the other algorithms.

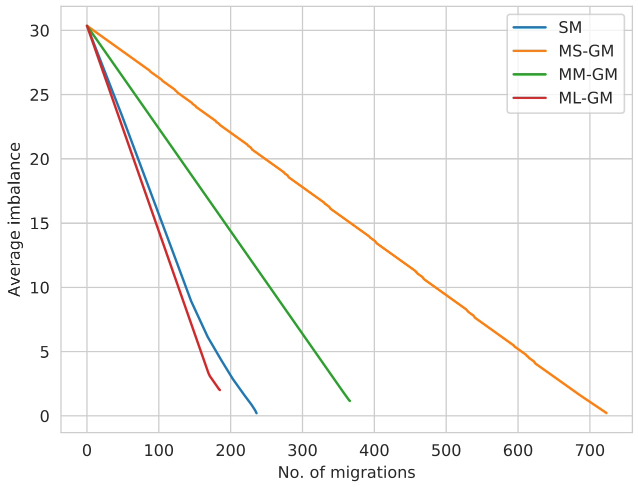


Fig. 23. Imbalance reduction as function of number of migrations in the TEST-100 experiment.

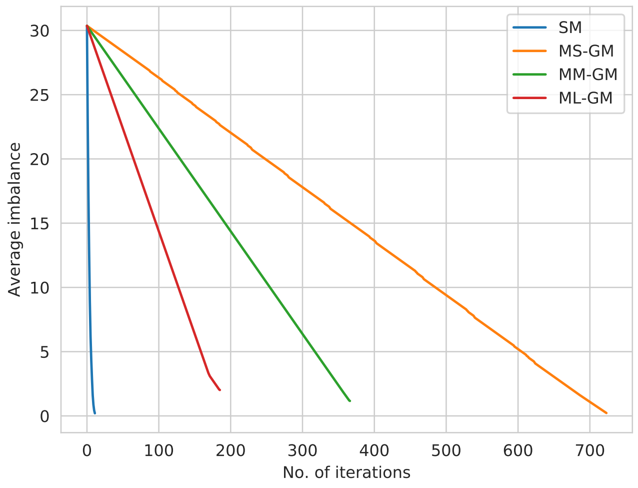


Fig. 24. Imbalance reduction as function of number of iterations in the TEST-100 experiment.

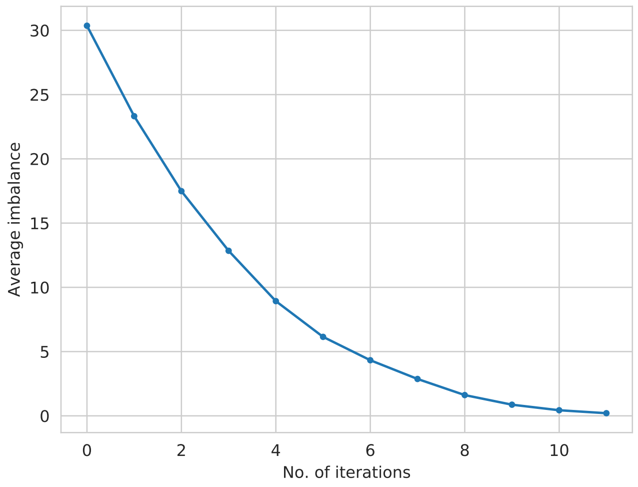


Fig. 25. Reduction of imbalance as function of iterations for the Stable Marriage algorithm in the TEST-100 experiment.

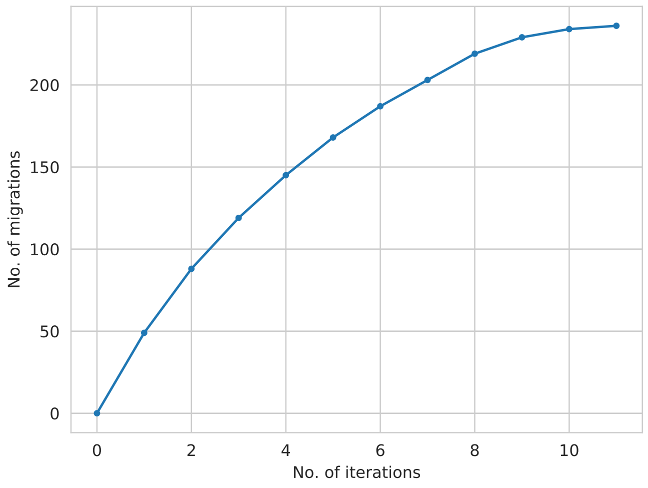


Fig. 26. Number of migrations as function of number of iterations for the Stable Marriage algorithm in the TEST-100 experiment.

Figure 25 shows the average imbalance reduction of our proposed SM algorithm as function of number of iterations. In the first few iterations the reduction ratio is very high while for the last few iterations the reduction in average imbalance is very small as the optimal balance is reached.

Figure 26 illustrates how the number of migrations varies as the iterations of our proposed SM algorithm is performed. In the first iteration, 50 VMs are migrated from 50 overloaded servers to 50 underloaded servers at the same time, meaning that all of the 100 bins of the TEST-100 experiment take part. In the second iteration 40 parallel migrations take place, meaning that 20 of the bins have already reach a state close to be balanced. The SM algorithm continues in the same manner efficiently reducing the imbalance and reaches a balanced state after only 11 iterations. On the other hand, the greedy matching algorithms migrates just a single VM for each iteration and thus spend much more time reaching a balanced situation.

Performance Optimization In the case of load management, the optimal solution is to have a perfect load balance between servers, i.e., to reduce the imbalance to 0. The load distribution problem is known to be NP hard and hence intractable for large systems [18, 9]. Our goal is to design a system that yields near-optimal solutions.

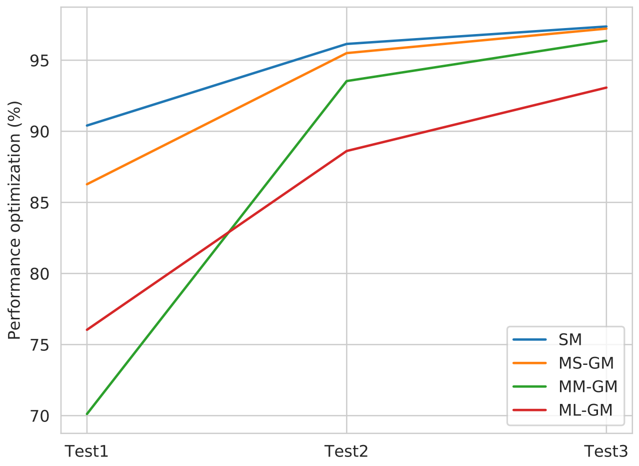


Fig. 27. Performance optimization of the proposed Stable Marriage and greedy matching algorithms.

Figure 27 illustrates the performance optimization of the proposed stable marriage and three different greedy matching algorithms in terms of imbalance reduction in three different test cases. In all three test scenarios the proposed stable marriage algorithm is clearly best, while ML-GM and MM-GM have the worst optimization percentage. However, in the case of test3, the MS-GM performance is close to the one of the SM algorithm. But as we already presented in Figure 23 and Figure 24, SM-GM requires a larger number of migrations and iterations which leads to extra overhead in terms of processing and time.

In a recent paper dealing with load balanced task scheduling for cloud computing, Panda et al. [22] propose a new algorithm reckoned as Probabilistic approach for Load Balancing (PLB). They run extensive simulations and compare their results with several algorithms which have been used in this context namely, Random, Cloud List Scheduling (CLS), Greedy and Round Robin (RR) [30]. The problem of load balancing tasks on VMs is comparable to load balancing VMs on physical servers and one of their Key Performance Indicators (KPI) is the dispersion of a set of loads from its average load, which is comparable to the load imbalance metric reported above [23]. In the simulations of Panda et al. the results of the Greedy algorithm are roughly equal to the performance of the other classical algorithms. We see this as an indication of how well our SM algorithm performs compared to other algorithms traditionally used for load balancing in cloud computing.

7 Conclusion

In our preliminary work [24], we addressed the problem of homogenizing the load in a cloud data center using the concept of the Stable Marriage algorithm. The results were promising and demonstrated the ability of the proposed algorithms to efficiently distribute the load across different physical servers. Although the proposed algorithms in [24] are based on the principles of Stable Marriage, they have a greedy matching principle and fail to replicate and incarnate the ideas of Stable Marriage theory in a proper way. Furthermore, in the analysis of results we found that the Migrate Smallest method gives better imbalance results but requires a large number of migrations while Migrate Smallest requires smaller number of migrations but at the cost of a larger final imbalance. To overcome this problem we propose in this paper a more conform Stable Marriage algorithm and compare its results to three different types of greedy matching algorithms. Based on the results presented in section 6.4, we can conclude that the Stable Marriage algorithm yields the lowest imbalance result combined with the lowest number of migrations. A single exception is that the Migrate Largest algorithm needs slightly fewer migrations, but its final imbalance is then larger. The Stable Marriage algorithm requires three times fewer migrations and provides the lowest imbalance compared to the other two greedy algorithms. Our results also show that the number of iterations needed to reach a balanced state is an order of magnitude smaller than the number of iterations needed by the three greedy matching algorithms.

References

1. Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., Merle, P.: Elasticity in cloud computing: state of the art and research challenges. IEEE Transactions on Services Computing **11**(2), 430–447 (2017)
2. Amazon: Serverless computing (2020 (accessed June 17, 2020)), url-https://aws.amazon.com/serverless/
3. Barbagallo, D., Di Nitto, E., Dubois, D.J., Mirandola, R.: A bio-inspired algorithm for energy optimization in a self-organizing data center. In: Self-Organizing Architectures, pp. 127–151. Springer (2010)
4. Barroso, L.A., Hölzle, U., Ranganathan, P.: The datacenter as a computer: Designing warehouse-scale machines. Synthesis Lectures on Computer Architecture **13**(3), i–189 (2018)
5. Bonvin, N., Papaioannou, T.G., Aberer, K.: Autonomic sla-driven provisioning for cloud applications. In: 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. pp. 434–443. IEEE (2011)
6. Calcavecchia, N.M., Caprarescu, B.A., Di Nitto, E., Dubois, D.J., Petcu, D.: Depas: a decentralized probabilistic algorithm for auto-scaling. Computing **94**(8-10), 701–730 (2012)
7. Castro, P., Ishakian, V., Muthusamy, V., Slominski, A.: The rise of serverless computing. Communications of the ACM **62**(12), 44–54 (2019)
8. Chieu, T.C., Chan, H.: Dynamic resource allocation via distributed decisions in cloud environment. In: 2011 IEEE 8th International Conference on e-Business Engineering. pp. 125–130. IEEE (2011)
9. Garey, M.R., Johnson, D.S.: Computers and intractability, vol. 174. freeman San Francisco (1979)
10. Hummaida, A.R., Paton, N.W., Sakellariou, R.: Adaptation in cloud resource configuration: a survey. Journal of Cloud Computing **5**(1), 7 (2016)
11. Jangda, A., Pinckney, D., Brun, Y., Guha, A.: Formal foundations of serverless computing. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 1–26 (2019)
12. Jin, C., Bai, X., Yang, C., Mao, W., Xu, X.: A review of power consumption models of servers in data centers. Applied Energy **265**, 114806 (2020)
13. Kalyvianaki, E., Charalambous, T., Hand, S.: Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In: Proceedings of the 6th international conference on Autonomic computing. pp. 117–126 (2009)
14. Levine, D.K.: Introduction to the special issue in honor of lloyd shapley: Eight topics in game theory. Games and Economic Behavior **108**, 1 – 12 (2018). https://doi.org/https://doi.org/10.1016/j.geb.2018.05.001, http://www.sciencedirect.com/science/article/pii/S089982561830068X, special Issue in Honor of Lloyd Shapley: Seven Topics in Game Theory
15. Lloyd Shapley, A.R.: "stable matching: Theory, evidence, and practical design", "https://www.nobelprize.org/uploads/2018/06/popular-economicsciences2012.pdf"
16. Manlove, D.F.: Algorithmics of matching under preferences, vol. 2. World Scientific (2013)
17. Marzolla, M., Babaoglu, O., Panzieri, F.: Server consolidation in clouds through gossiping. In: 2011 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM). pp. 1–6. IEEE (2011)

18. Mishra, S.K., Sahoo, B., Parida, P.P.: Load balancing in cloud computing: a big picture. Journal of King Saud University-Computer and Information Sciences **32**(2), 149–158 (2020)
19. Muñoz-Escoí, F.D., Bernabéu-Aubán, J.M.: A survey on elasticity management in paas systems. Computing **99**(7), 617–656 (2017)
20. Najjar, A., Serpaggi, X., Gravier, C., Boissier, O.: Multi-agent negotiation for user-centric elasticity management in the cloud. In: 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing. pp. 357–362. IEEE (2013)
21. Naskos, A., Gounaris, A., Sioutas, S.: Cloud elasticity: a survey. In: International Workshop on Algorithmic Aspects of Cloud Computing. pp. 151–167. Springer (2015)
22. Panda, S.K., Jana, P.K.: Load balanced task scheduling for cloud computing: A probabilistic approach. Knowledge and Information Systems **61**(3), 1607–1631 (2019)
23. Rao, A., Lakshminarayanan, K., Surana, S., Karp, R., Stoica, I.: Load balancing in structured p2p systems. In: International Workshop on Peer-to-Peer Systems. pp. 68–79. Springer (2003)
24. Sangar, D., Haugerud, H., Yazidi, A., Begnum, K.: A decentralized approach for homogenizing load distribution: In cloud data center based on stable marriage matching. In: Proceedings of the 11th International Conference on Management of Digital EcoSystems. pp. 292–299 (2019)
25. Sedaghat, M., Hernández-Rodriguez, F., Elmroth, E., Girdzijauskas, S.: Divide the task, multiply the outcome: Cooperative vm consolidation. In: 2014 IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom). pp. 300–305. IEEE (2014)
26. Siebenhaar, M., Lampe, U., Schuller, D., Steinmetz, R., et al.: Concurrent negotiations in cloud-based systems. In: International Workshop on Grid Economics and Business Models. pp. 17–31. Springer (2011)
27. Taibi, D., El Ioini, N., Pahl, C., Niederkofler, J.R.S.: Serverless cloud computing (function-as-a-service) patterns: A multivocal literature review. In: Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER'20) (2020)
28. Vasques, T.L., Moura, P., de Almeida, A.: A review on energy efficiency and demand response with focus on small and medium data centers. Energy Efficiency pp. 1–30 (2019)
29. Wuhib, F., Stadler, R., Lindgren, H.: Dynamic resource allocation with management objectives—implementation for an openstack cloud. In: 2012 8th international conference on network and service management (cnsm) and 2012 workshop on systems virtualiztion management (svm). pp. 309–315. IEEE (2012)
30. Xu, M., Tian, W., Buyya, R.: A survey on load balancing algorithms for virtual machines placement in cloud computing. Concurrency and Computation: Practice and Experience **29**(12), e4123 (2017)