ClusterFlux: a Dynamic System for Placing Virtual Machines on a Cloud Computing Network

Kevin Baptista, Bennett Cyphers, Raluca Ifrim, Jordan Ugalde

Design Project 2
May 9th, 2014
Tiffany Chen

Section 1: Introduction

Cloud computing platforms, such as the Amazon Elastic Compute Cloud (EC2) and Google Compute Engine, are increasingly popular solutions for large, flexible applications. When these systems are used for complex, multi-nodal applications with intensive network requirements, network transfer speeds may become the limiting factor on application completion time. Clusterflux is a system for placing and dynamically moving virtual machines (VMs) around a physical data center in order to minimize the network time necessary to complete a client's tasks.

The Clusterflux algorithm attempts to keep virtual machines with strong interdependency (and therefore heavy intercommunication requirements) as close as possible to each other within the data center. In this paper, "close" refers to placement such that the two VMs may communicate via as few intermediary links as possible. Clusterflux maximizes this across all tasks by polling VMs for their interconnectivity requirements and continually moving one machine at a time to the group with which it shares the most dependencies, while also determining when it is appropriate to move a cluster of VMs to a different group.

Clusterflux makes several notable assumptions in regards to the behavior of the cloud network: that VMs can be moved within the network near instantaneously, that the physical machines of the network can be quickly scanned to determine exactly which physical machines have room for exactly how many VMs, and, crucially, that the throughput rates between VMs contained within a single group are significantly greater than the throughput between VMs in different groups. Given these assumptions, Clusterflux leads to faster application running time in a variety of situations than any other VM placement scheme we considered.

Section 2: Problem

Clusterflux is designed for a data center network like the shown in figure 1 below. Each physical machine in the center can run up to four virtual machines simultaneously. There are 1152 physical machines in total, organized into 24 groups of 48. Machines in a group are all connected to a *group router*, which links them to the rest of the data center. Each group router is connected to two *aggregate routers*, of which there are eight in total. Finally, each aggregate router is also connected to two *core routers*, which bridge the gaps between aggregate sections of the network.

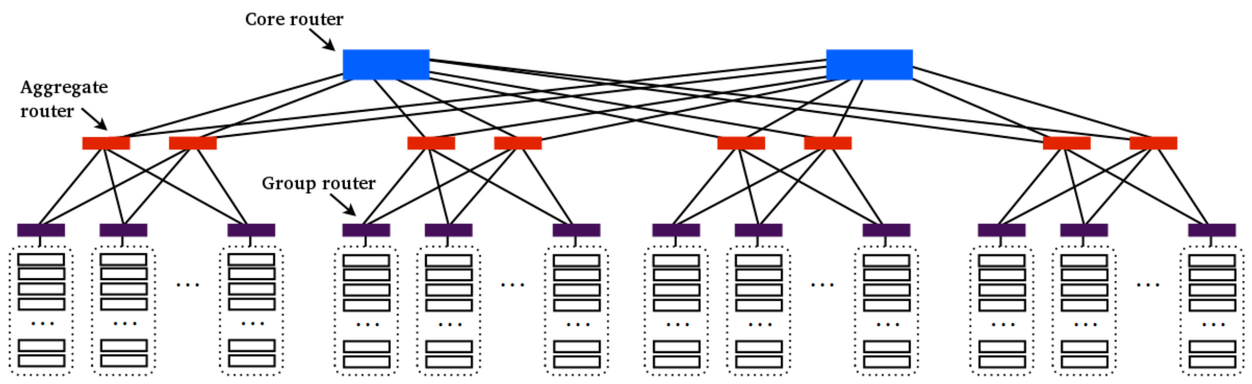

Figure 1: *the data center architecture*

All machines within a group have very fast connections to one another. Since the connections between group routers and their aggregate routers are utilized by dozens or possibly hundreds of machines at once, traffic along these "aggregate links" may be slow. Furthermore, the links between aggregate routers and the core routers ("core links") serve even more connections, so they will tend to be slower still. Instances of Clusterflux are run on an external server owned by Clusterflux, Inc. Each instance is assigned a user with a set of n VMs to place around the network, and an n x n matrix, B, indicating the amount of data which each VM has to transfer to each other one. Aside from this, the server knows nothing about the network or other users (even those also using Clusterflux) except what data it can gather.

Section 3: Design

3.1: Tradeoffs

The guiding principle behind our algorithm is that data connections within groups are extremely fast, and intergroup connections are unreliable at best, and impassable at worst. In line with this, our solution doesn't take into consideration the network connection rates between different groups. Therefore the connection rates between VMs on different groups are often suboptimal using Clusterflux. However, because the connection rates within a group are reliable and constant, and Clusterflux dynamically packs as many VMs into the same group as possible, the end result is an algorithm with intermittent periods of slow connection between intergroup VMs that minimizes overall runtime. The tradeoff associated with not knowing the traffic within the network is covered in the next section regarding measurement.

3.2: Measurement

We focus Clusterflux's measurements largely on the distribution of VMs around the data center rather than on network throughput. Measuring the amount of spaces available in a group is fast, accurate, and can be performed for each group in the network several times per minute. In contrast, passive TCP throughput measurements tend to significantly underestimate the actual capacity of links, and active throughput measurements, while more accurate, require large, otherwise useless packets. To gather useful data, active TCP measurements require the user to pay for additional VMs to send probe packets and create significant additional traffic, either slowing down or interrupting communication between existing VMs. Since link throughput can change quite rapidly, keeping an up-to-date index of intergroup connectivity would require frequent such interruptions to normal network traffic.

Therefore, we chose to eschew TCP throughput measurements entirely; instead, we that estimate the speed of a group's outgoing links is roughly proportional to its VM density. This relationship does not always hold: a group with just a few VMs sending large amounts of data across the network may still suffer from slow intergroup communication speeds, and groups with lots of VMs do not necessarily experience congestion. However, we believe that on average, this assumption provides an effective and time-efficient heuristic.

3.2.1: Group space table

At any given time, our server keeps an up-to-date table listing the number of available spaces in each group in the data center. To maintain the table, a process on the server cycles through all 1152 physical machines every thirty seconds, and makes a **`machine_occupancy()`** query on each one. `machine_occupancy(machine)` is a function provided by the data center's public API, which returns the amount of free VM slots a physical machine has available. We assume that these calls are both computationally efficient and low data volume, so performing thousands of them per minute should not put too much strain on either the server or the network's connections. The server stores the latest results of its query in a hash table, *machines_occupied*, mapping the ID number of each group (1 through 24) to a list of machines with available VM slots. If a machine has more than one available space, it appears once for each VM it can accommodate; this lets the server assess the total available spaces in a group with a query to the length of *machines_occupied*[group]. The server abstracts the previous operation to a function call, *available_space*(group).

3.2.2: Group desirability

After initialization, the server use the availability table to compute desirability values for each group. Since our algorithm will try to place as many VMs as possible in groups together, a group's desirability is directly proportional to the amount of open space it has. Furthermore, we estimate that groups which share an aggregate router should have faster throughput between each other than those which must communicate through a core router. Therefore, aggregate sections of the datacenter become more desirable overall the more open spaces their groups have. Using this, we define the following function:

```
int desirability(group) {
        int sum = 0;
        // Loop over the other groups on this aggregate router
        foreach (g in group.aggregate_neighbors)
              sum += available_space(g);
        int avg = sum / length(group.aggregate_neighbors);
        return avg + available_space(group);
  }
```

Figure 2: *a function to calculate the desirability of a group*

Clusterflux uses this to determine which groups receive preference when placing VMs for the first time, and when moving clusters during upkeep.

3.3: Placement

Clusterflux's VM placement scheme relies on a few assumptions about network and application behavior. First, we assume that tasks among VMs are independent: any task requiring network connectivity does not need to wait for any other tasks to complete first, so the order in which network connections are completed does not matter. Second, we presume that data transfer speeds among machines within a group will always be at least as fast as intergroup communication speeds, and will usually be orders of magnitude faster. The data center specifications state that all links in the network have a capacity of 10 GB/s, and that all machines within a group are fully connected, so VMs communicating within a group will always enjoy the maximum possible throughput.

We know that VMs are charged ten cents a minute, rounded up to the next minute. Since we are given that moving a VM is essentially instantaneous, we can assume that relocations are essentially free if done at the end of a minute. Clusterflux's design relies on a central server outside of the data center that manages the placement of VMs and various measurements. We assume that our VMs are able to communicate with the central server quickly and with minimal impact on overall performance. Finally, we assume that once a VM has finished transmitting all of its data, it can be removed from the network and the other VMs will continue to operate as normal. Removing VMs that have finished early saves money for the user and frees up space.

Based on these assumptions, our system's strategy is to place virtual machines together in groups as often as possible, and to maximize the size of these groups whenever possible. Accordingly, we first place VMs greedily into the most desirable groups, and then regularly update locations to cluster together VMs that are transmitting large amounts of data to each other.

3.3.1: Preprocessing

Before our system begins placing VMs, the server queries each machine in the data center, and updates its occupancy table with the amount of free space in each group. Clusterflux then calculates `desirability(group)` for each group, as described above. The groups are sorted in descending order by desirability, and stored in a queue.

3.3.2: Setup

The server uses a greedy strategy to initially place VMs, executed as follows.

1. Place the pair of VMs that have the largest amount of data to transfer to one another in the group with the highest weight (which we'll call *best_group*). It doesn't matter which machines within the group they are placed on.
2. While there are open slots in *best_group*:
    a. Find the VM that has the most data to exchange with the VMs already in *best_group*, and place it there.
    b. Repeat step 2a.
3. Once *best_group* has filled up, set *best_group* to the next one in the queue calculated above.
4. Repeat steps 2-3 either until all VMs have been placed or there is no more room in the data center.

Our initial placement strategy optimizes well for any datacenter state, since VMs that need to transmit data to each other are more likely to have a fast link.

3.3.3: Maintenance

After the VMs have been initially placed throughout the network, it is important that their positioning respond quickly and dynamically to the needs of our user. This section details the considerations our system makes in moving the virtual machines within the network in an attempt to achieve high performance at all times.

3.3.3.1: Clusters

Here we define a useful abstraction that is use in our placement algorithm: the cluster. A cluster is defined as a set of virtual machines that are all contained within a single group of machines in the data center. Although each cluster is permanently tied to a specific group, clusters can be easily merged and split, allowing quick movement of sets of VMs from one group to another.

```
object Cluster {
      int group;
      DisjointSet VMs;

      // Move all VMs from this cluster to another cluster
      Cluster merge(Cluster cluster) {
            int g = cluster.group;
            foreach (vm in this.VMs) {
```

```
                        move(vm, g);
                    }
                    cluster.VMs.union(this.VMs);
                    this.VMs = DisjointSet(); // reset this cluster's VMs
                    return cluster;
                }
            }
```

Figure 3: Implementation of the Cluster object.

Every VM belongs to a cluster. Clusters are tracked on our external server in a hashtable, *clusters*, which maps each of the group IDs, 1-24, to a cluster object. Each cluster stores a group identifier and set of the ip addresses of the VMs in the group. The set can be implemented as a Disjoint Set data structure. Clusters support a single operation: merge, which moves all of a cluster's VMs to another cluster. An example of the Cluster API is shown in Figure 3.

3.3.3.2: Dynamic VM movement

Each individual virtual machine will maintain a key: value dictionary, *group_scores*, which maps the integer ID of each group in the network to a value indicating how beneficial a move to that group would be. The value for each group is calculated with the following algorithm:

```
float get_score(group) {
    value = 0;
    foreach (VM in group) {
        value += VM.data_to_transfer(self);
        value += self.data_to_transfer(VM);
    }
    value -= get_score(self.group);
    return value;
}
```

Figure 4: The algorithm for determining the group score for a particular VM.

The value for *group_scores*[group] is therefore equal to the amount of data which has yet to be transferred between the virtual machine and every VM in the target group, minus the score for the VM's current group. We say a move to a group would be "beneficial" if the group's score is greater than zero.

Using this information, our server dictates two kinds of operations on VMs: *moves* and *swaps*. Every few seconds, each VM caches its old *group_scores* table (to prevent race conditions), recomputes the table, and sends a request to our server asking to be moved to the group with the highest score, including a group ID number and the corresponding group score. The server then determines which VM request contains the highest group score, and checks whether it is possible to move that VM to its requested group. If there is space for the VM in the desired group, the server performs a *move*: it sends a request to the data center to the remove the VM, followed by a call to the data center's API, `place(vm, machine)`, for some available machine in the desired group. `place()` is a function built in to the data center which initializes the VM on the requested machine if possible, or returns false if the machine is full.

If the desired group is full, the server performs a *swap*, executed as follows:

1. Find the VM in the requested group that will benefit most from moving to the group of the original VM. This is accomplished by requesting the *group_scores* table from each VM in the destination group, and finding the VM with the maximum value for *group_scores*[origin group].
2. Remove both VMs from their positions in the data center, saving their old machine IDs.
3. Replace them on each others' old machines with `place()`.

3.3.3.3: Cluster movement

In addition to moving VMs around, the Clusterflux server runs a modified version of the `desirability()` function for each group in the data center every few seconds; this version counts machines occupied by the server's client as "empty." Similar to the procedure for VMs described above, our measurement algorithm looks for clusters for which the most desirable group is different from the group the cluster is currently in. On every loop, the server looks all clusters with positive such differentials. The server sorts the clusters in ascending order of cluster size, and iterates over the sorted list. For each cluster, if there is enough empty room in the desired group, `G`, to fit the entire cluster, it is moved as whole to the new group with `cluster.merge(clusters[G])`.

This strategy ensures that the smallest clusters are always merged first, with the intention of achieving more efficient clustering. By moving the smallest clusters first, the likelihood that the multiple smaller clusters merge into larger clusters increases. Even if this cluster movement does not lead to immediate cluster merging, more room for VMs opens up in the groups that the clusters move to, which will eventually result in larger clusters due to the movement of individual VMs.

3.3.3.4: Termination

Whenever a VM has no more data to transmit to any of the other VMs, we remove it from the network. This frees up space for other VMs and prevents wasteful spending.

Section 4: Analysis

4.1 General Analysis

In addition to running the client's application, each VM runs a thread which recalculates the VM's group_scores table every five seconds. This thread then sends a request to the server asking to be moved to the highest-scored group. To update the table, the VMs request the full occupancy table from the server before each recalculation. The size of the table is bounded by the number of machines in the system (1152), so the amount of data transmitted from the server (approximately 1 KB) should have a minimal impact on the connection speed of the VM.

Scores are calculated with constant time operations for all of the VMs in the current group as well as the VMs in the potential new group. The runtime of this calculation is therefore O(n), where n is bounded by 4608 (the maximum number of virtual machines), which is also minimally obstructive to the VM's normal calculations.

In order to ensure that the central server is not overwhelmed, we scale back how often VMs recalculate group scores and send requests depending on how many VMs the user is running. We have found that a base value of once every 10 seconds balances score accuracy with additional VM workload well for average cases. The requests themselves are fairly small (containing mostly numbers such as the ID of the group to which the VM would like to move and its group score) so we expect these computations to be feasible. If each request/response packet is bounded by 2 KB (based on our analysis above), and there are < 5000 VMs in the system, the never needs to exchange more than 1 MB/s of data over the network, which is manageable on modern hardware.

To keep track of VM density, the server polls all 1152 machines for occupancy every ten seconds, with each poll running in roughly constant time. Each ping transfers very small amounts of data in very short amounts of time, so even with thousands of pings, we expect that the constant factor will result in little additional overhead.

4.2: Testing Infrastructure

4.2.1 The Simulation

Using a simulator we constructed in Python, we tested the performance of our solutions in various scenarios as well as various straw-man placement strategies. The simulation makes the following simplifications to the problem:

- All active links are fully utilized at all times. This means that if VM *A* has data to transfer to VM *B*, and both are in the network, *A* will transmit data to *B* as fast as possible until the connection is finished or one of them is removed.
- Throughput across a link is divided evenly among all the connections using the link. The throughput for any link is calculated as (total throughput) / (number of connections). While this does not provide an accurate simulation of TCP behavior, we believe it gives a decent, worst-case estimate of traffic speed between two points.

- Redundant routers in the network are treated as one object. There are 4 aggregate routers and 1 core router. Each aggregate and core link is assigned a default value of 20 GB/s. This ignores some of the nuances of a real data center but preserves the essential behavior of the network as the number of machines scales.

- Although the actual network contains 1152 total machines, simulating this many VMs on one laptop, in real time, in Python caused some performance issues. To compensate, we lowered the total number of machines in the network to 288 for most of the tests, maintaining the router architecture as described above but reducing the number of machines per group to 12. We found that this still provided useful data about our scheme's performance without too much strain on the authors' processors.

4.2.2 Testing procedure

To test different strategies, we created two generic classes, Server and VirtualMachine, which handle the logic of placing and updating VMs in the data center. These were subclassed to implement the logic for the Clusterflux scheme (in *SmartServer* and *SmartVM*), as well as for a variety of straw-man placement schemes for comparison (in *RandomServer*, *GreedyServer*, *StragglerServer*, and *PairwiseServer*). The full source code is available in a public repository on GitHub [1].

We tested using the following procedure:

> First, a fresh data center instance is initialized with 288 machines, and populated with 20 "dumb" users with 10 VMs each. These "dumb" users have totally connected VM networks (wherein each VM transmits data to each other VM) with very large, essentially infinite connections; their purpose is simply to create network congestion for the real users. The dummies place their VMs around the network randomly, and leave them there permanently.
>
> Next, we initialize ten instances of the server type being tested, labeled *user 0* through *user 9*. Each server is given 20 VMs, with a random, partially-connected B matrix such that each VM has outgoing connections to exactly five other VMs, and each connection has a random amount of data between 0 and 100 GB to transfer. One at a time, each server is started by calling its *start()* function, which places its VMs around the network according to whatever logic it uses.
>
> Finally, each server is updated once per second by calling its *loop()* function, which gathers data about the network and executes on it, and the state of the data center is output to the command line. This continues until every server has completed all of its tasks, and the simulation finishes.

4.2.3 Results

We performed the above test with four different types of server.

- ***SmartServer*** implements the Clusterflux algorithm as described in this paper. However, in order to test the recovery capabilities of Clusterflux, it initially places each VM in the first available physical machine it finds, disregarding group scores. This means that some of the users end up with VMs distributed across different groups and must move them around to achieve maximum throughput.
- ***GreedyServer*** performs a simple greedy placement on startup, attempting to place all VMs in the first available group, then moving on to the next open one; it does not move any VMs after the initial placement.
- ***StragglerServer*** places all VMs randomly at first. Then, during *loop()*, it chooses the VM pair which has made the least progress in its transfer (as a percentage of the total data the pair has to transfer) and places one of the VMs in a new random location.
- ***PairwiseServer*** is a modification of *GreedyServer*: after the initial greedy placement, in *loop()*, the server chooses a random pair which has not completed its transfer, and moves them onto the same machine.

We measured the total amount of computation time, in seconds, used by each scheme. The results of one round of testing are shown in Table 1:

	SmartServer	**GreedyServer**	**StragglerServer**	**PairwiseServer**
user 0	186	181	75946	181
user 1	185	186	68261	677
user 2	236	4916	72357	176
user 3	185	181	73875	785
user 4	177	13163	68429	182
user 5	230	167	74761	1623
user 6	217	24674	67953	182
user 7	182	173	71727	1603
user 8	265	15609	70950	182
user 9	187	178	78561	1720
average (secs)	**205**	**5943**	**72282**	**731**

Table 1: *Comparison of total computation time used, by placement strategy. Each user was given 20 VMs with random amounts of data to exchange; all values are in seconds.*

For all users, Clusterflux performed as well or better than the straw-man schemes, and it performed several times better than any of the other schemes on average. While this is no great feat considering the other servers were never designed to be performant, we believe it demonstrates the efficacy of the scheme in a plausible general case. Furthermore, every *SmartServer* user was within a factor of 1.5 of the best time achieved by *any* user, which indicates consistent performance even with significantly different starting distributions. We will examine performance in more exotic network configurations below.

4.3: Use Cases

In order to effectively analyze the performance of Clusterflux, we look in-depth at three use cases that test various aspects of the system.

4.3.1: High VM Occupancy

We first consider a case where every machine in the data center already hosts three other VMs. The user has 100 VMs to place that will all need to send the same amount of data to each other.

Clusterflux starts by using the machine_occupancy(m) function on all machines on the network to detect that every machine is initially hosting 3 VMs. The algorithm then places one VM on every machine in the first group, one VM on every machine in the second group, and one VM on four machines in the third group. From here, this use case can break into two significant sub cases.

The first sub case is that where every machine persistently hosts 3 VMs that don't belong to the user. In this situation, VMs will be frequently swapping between groups 1 and 2 since the VMs need to transfer the same amount of data between all VMs. Since nearly the all of the user's VMs are evenly split between groups 1 and 2, at any given time almost every VM is connected to about half of all of the VMs using the extremely fast network connection afforded within a group. Given this fact, the total estimated runtime is approximately twice the time it takes for the required data to transfer within a group.

In the second sub case, the machines start opening up over time. In this case, open spaces appear either in groups with the user's VMs or other groups. In the second situation the algorithm will detect the openings and move a cluster to whichever group has the most openings. After that, the following step for both situations is the same; individual VMs will start joining clusters with more openings and the clusters will increase in size. This sub case has as its upper bound the same approximate time as the first sub case, but in this situation it is possible that all 100 VMs become clustered in the same group, in which case the expected completion time will approach that of the case in which all VMs share a group.

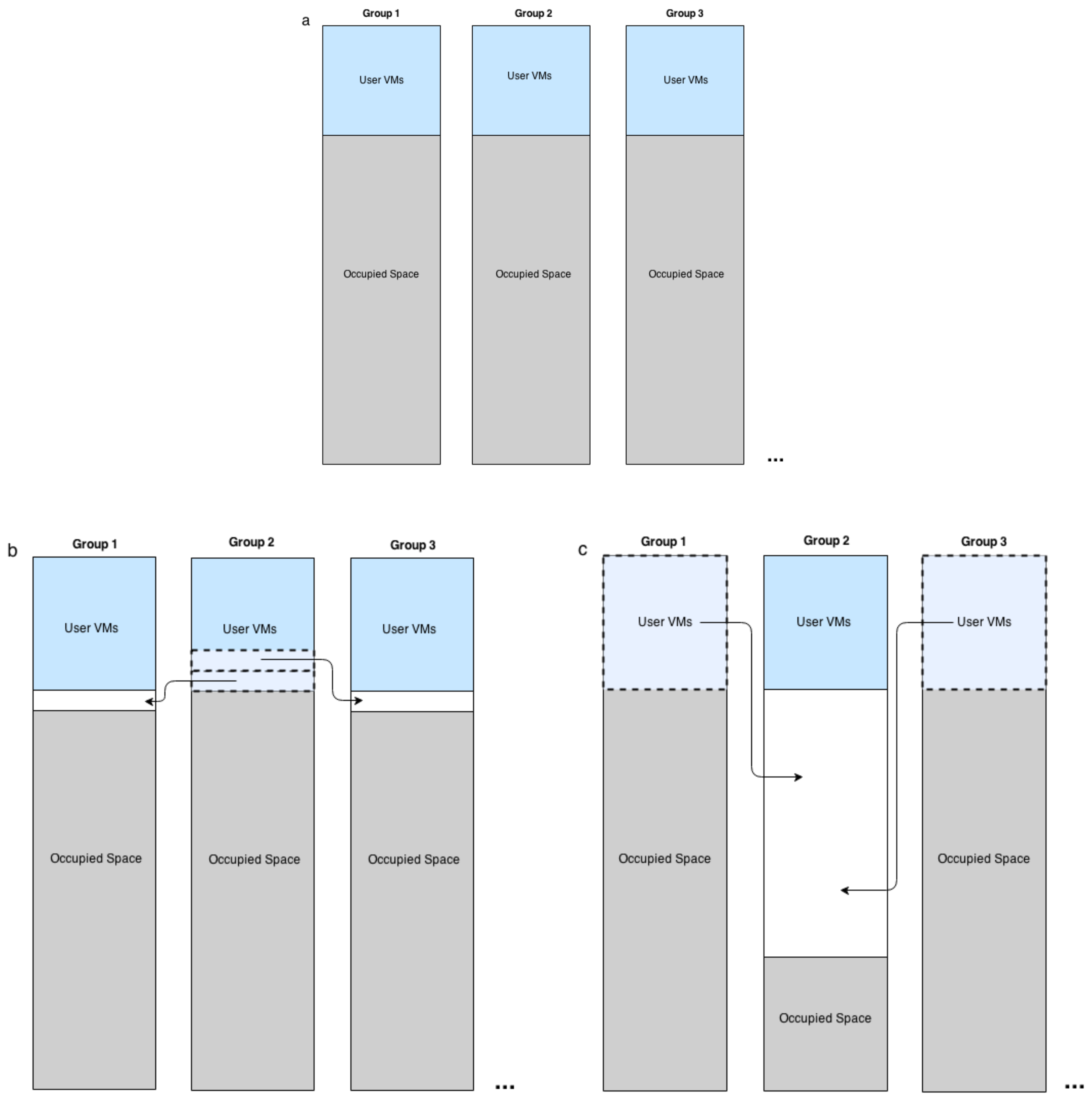

Figure 5: Behavior of VMs in "shallow" groups with the Clusterflux algorithm.

a. *Simplified original setup for 4.2.1.*

b. *Individual VM movement. As soon as other users' VMs finish, individual VMs move to clusters where they will transfer the most data.*

c. *Cluster movement. If large amounts of space open up, clusters may move together and merge.*

Since the fastest network connections occur within a group, it is reasonable to consider sole usage of intragroup connection to be optimal. As such, in the first subcase the presented solution is on average only a factor of two worse than the optimal solution and the solution in the second subcase is bounded by the runtime of the first subcase and the optimal solution.

4.3.2: Newly Freed Space

Next, we consider a user who only has 10 VMs, which have been placed throughout the first six groups and have low throughput due to congestion. In this scenario, a large application in a different set of groups (group IDs 19-24, on the other side of the data center) terminates, leaving many open spaces and underutilized links.

Our central server will detect the newly open spaces from its regular measurements, within ten seconds using the default refresh time. As described earlier, the server will try to move smaller clusters into open groups whenever the new groups offer more space. In this case, since the user has few VMs to begin with, it is highly probable that all 10 VMs will fit in one of the groups that opened up and will merge into a single cluster. This will minimize time to completion, as all VMs are communicating via the high-throughput group links.

4.3.3: Multiple Clusterflux Users on the Same Network

Our scheme will tend to perform better when dominating the logic of the network as a whole. With every user in the data center running our algorithm, it will perform optimally in the best case. In the worst case - depending on how the network is accessed - it will perform similarly to the first example described.

4.3.3.1: Worst Case

One potential issue arises when two users running our scheme join the network at the exact same time. Consider a general case in which two users (A and B) with the same number of VMs populate two open groups simultaneously, with half of each user's VMs in group 1 and half in group 2. Since we are considering the worst case, we assume this is accomplished in such a way that the groups 1 and 2 become fully filled with all of both users VMs and insufficient space is ever open in any other group for our algorithm to move the clusters. Then this problem because approximately equivalent to the scenario described in section **4.2.1.** The only movement of the VMs after the initial placement will be swapping, until one of the users' VMs completes all of its tasks or space opens up elsewhere in the network. However, in practice, it is highly unlikely that two users' initial VM placement routines will ever interleave in such a way.

4.3.3.2: Typical Case

In a typical case, every user will try and pack all of their VMs greedily within a single group. If a server accesses the network at a time when a group is empty enough to host all of its VMs, then the algorithm will place all VMs in the same group and Clusterflux will behave optimally. If there is not enough room to fit all VMs in any one group, since every user is trying to pack their VMs as densely as possible, every group will eventually tend to be dominated by a single user. This is due to the positive feedback loop created by the VM moving mechanism: As one user's VMs move towards a single group, the group becomes less desirable to the VMs of other users, which move away from it. The migration of other users away from the group frees up more spaces, making it even more desirable for the first user, and causing the clustering to continue. As "like attracts like," the network approaches a state where all users are organized in discrete sections of the data center, and each user's tasks can complete as quickly as possible.

Section 5: Conclusion

Clusterflux is an efficient virtual machine management system that performs well under a variety of circumstances. By clustering VMs in groups and ensuring fast links between VMs that have large amounts of data to transmit, Clusterflux effectively minimizes transmission time and cost to the client in most use cases. Clusterflux tracks the locations of all other virtual machines in the data center, and uses this information to rearrange the client's VMs into the best configuration possible. The flexibility of our system, and its ability to adapt quickly to changes in the data center, is due in large part to this single-metric approach to network diagnostics. It also means that Clusterflux relies in part on assumptions and heuristics that may not always be accurate, and that the system may be vulnerable to slowdown from unforeseen link congestion or failure. This can be addressed in future work on Clusterflux, and considerations should be made regarding how best to integrate the measurement of TCP throughput into the system in a cost-effective way. However, since the core strategy behind the scheme - bringing connected VMs close together to communicate at high speed, and seeking the largest chunks of real estate in the network to maximize grouping - is simple and resilient, we believe Clusterflux is a fast, effective, reliable choice for network placement and management.

References

1. Cyphers, Bennett, Raluca Ifrim, Jordan Ugalde, and Kevin Baptista (2014). *DP2_test*. https://github.com/bcyphers/DP2_test.

2. Dias, Daniel S., and Luis H. Costa. "Online Traffic-aware Virtual Machine Placement in Data Center Networks." *IEEE* (2012). *http://ieeexplore.ieee.org/Xplore/home.jsp*.

3. Meng, Xiaoqiao, Vasileios Pappas, and Li Zhang. "Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement." *IEEE* (2010). *http://ieeexplore.ieee.org/Xplore/home.jsp*.