# VrooM

Defne Gurel, Mycal Tucker, Zack Drach
DP2 Report
Section R11 & R12
May 9, 2014

## Abstract

These days, it is common to run distributed applications in public data centers. In this setting, network congestion might slow down the application, costing extra time and money. Vroom is an application that optimizes distributed computations by reacting to changing network conditions. Its key innovations include the use of a *probabilistic generative model* for making inferences about network conditions, and the use of *simulated annealing* for solving the optimization problem.

# Introduction

Distributed computations are executed on a number of *virtual machines* (VMs) that are hosted on physical machines. The physical machines are connected by a hierarchy of routers. Because congested routers are detrimental to the application, it is important to place the VMs in uncongested parts of the network.

The problem of VM placement is difficult because network conditions are constantly changing. Vroom addresses this problem by continuously measuring the network conditions and adjusting the placement of VMs.

## Design Overview

The system architecture consists of a single master VM, multiple worker VMs, and some additional exploration VMs to assist with measurements.

In order to reduce the overhead of measurement, Vroom leverages our knowledge of the network topology to detect router congestion and infer route capacities between VMs.

In order determine an optimal configuration of VMs, Vroom uses a random-walk algorithm which can find a nearly-optimal configuration within a finite amount of time.

## Tradeoffs

Since measurements cost bandwidth, time, and money, Vroom determines the measurement frequency based on application size and current performance. Preliminary analysis shows that the Vroom measurements will consume no more than 1.4% of total network bandwidth, yet still allow a 10VM instance discover a large block of underutilized racks in 20 to 40 seconds.

# System Architecture

The system has three main components.

- The master VM schedules measurements, launches worker VMs, and optimizes their placement.
- Worker VMs run the user application and measure network conditions.
- Exploration VMs measure network conditions.

Here, we describe the programs and data structures on these three components.

## Master VM

The master VM runs two daemon processes. One of them schedules and collects measurements about network conditions. The other one manages the worker placement.

Both processes share data via an on-disk database, such as *Berkeley DB*, and they can send events to each other via a Unix pipe. Figure 1 illustrates the master VM.

### User Interface

To start a new master VM, users provide a configuration file describing the application they want to run. Once the master node is launched, no further input is required from the users. However, the user may choose to receive periodic updates about the status of their application.

### Rationale

Using a single master VM simplifies our system significantly. Distributing coordination among the workers would increase fault-tolerance, but it would also raise additional problems.

- the inference an placement algorithms are not easily parallelizable
- frequent movement of the worker VMs would interfere with coordination
- replication would require more total bandwidth and CPU time

### Master Database

The following data is stored the master database.

- **Measurement Table**: measurements and inferences about router capacities
- **Data Matrix**: recent histories of the network usage between worker VMs, current amount of data left to transfer
- **Location Table**: current location (`machine_id/ip_address`) of the worker and exploration VMs and a list of empty VM slots
- **Freshness Queue**: list of network racks that have stale measurement data

Figure 1 also shows the access patterns of the database tables. The contents of each table are described later in this report.

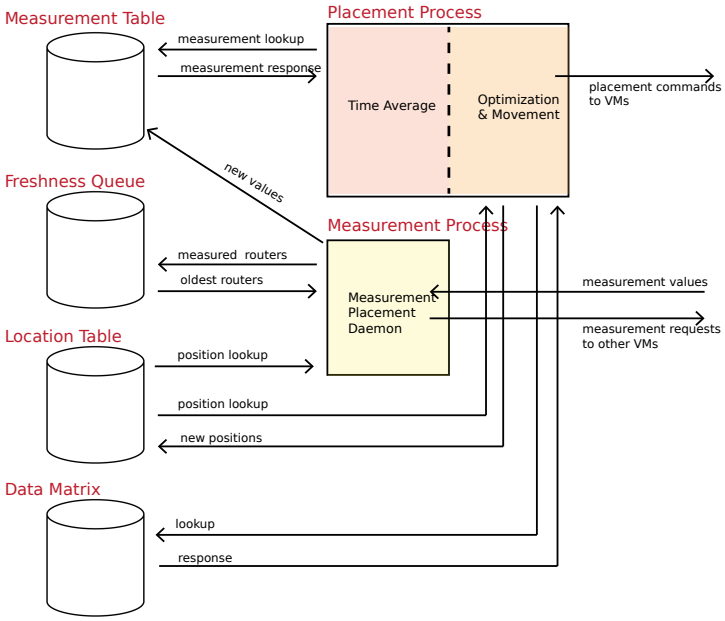


Figure 1: Access patterns in the master VM

## Worker VM and Exploration VM

Worker VMs also run two daemon processes. The main daemon executes the user application. The second daemon measures network conditions when instructed by the master VM.

Exploration VMs assist with measurements, so they only need to run the measurement daemon.

All VMs communicate with each other using RPC calls. Many operations require coordination between multiple VMs, but the exact protocol will not be described in this report. Figure 2 illustrates some of the RPC messages that get sent between VMs.

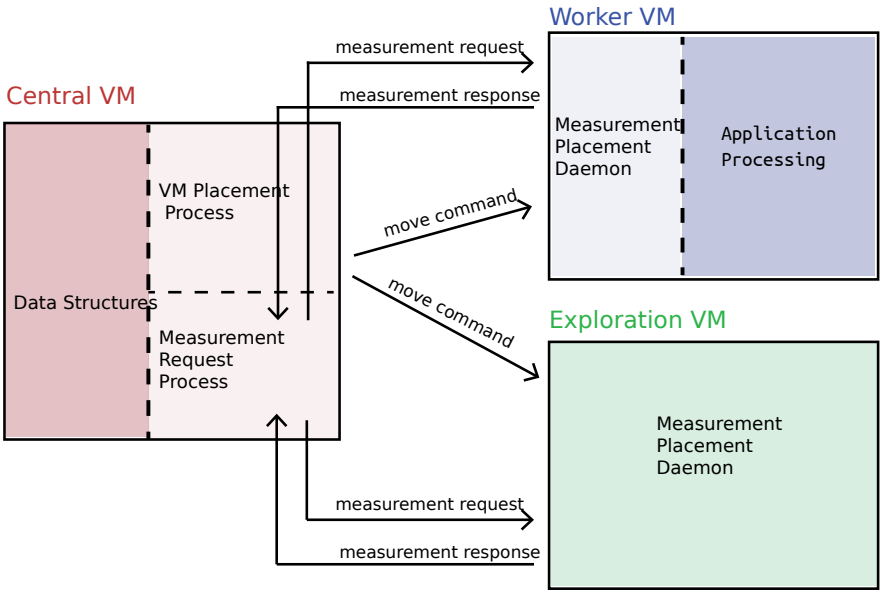


Figure 2: Example RPC calls between VMs

# Measurement Subsystem

The measurement subsystem encompasses the measurement daemons on the master, worker, and exploration VMs.

## Application Metrics

The measurement subsystem keeps track of **throughput** and **data left** on all worker VMs.

### Throughput

The system call `tcp_throughput(v)` returns the throughput of the TCP connection between a worker VM and another VM. This function is called periodically on each worker, and the results are stored in the *Data Matrix*.

Because of the bursty nature of traffic in a datacenter, the throughput data is sent through a lowpass filter before being stored in the database. Without the lowpass filter, the placement subsystem might be falsely triggered by the noise.

### Data Left

The system call `progress(u, v)` returns the amount of data left to transfer between two VMs. This function is called periodically, and the results are store in the *Data Matrix*.

## Path Capacity

The placement subsystem requires the **path capacity** all pairs of VMs. The path capacity is defined to be the maximum achievable throughput between two VMs at a particular point in time. This can be measured by flooding the connection between two VMs until the TCP window stops increasing.

Path capacity measurements are fed into a machine learning algorithm. The output of the algorithm is persisted in the *Measurement Table*.

Figure 3 shows a network where several flood measurements are occurring.

### Rejected Approach

A simple implementation could measure the path capacity between each pair of racks. This is appealing because the path capacity between a pair of VMs is usually equal to the path capacity between the two containing racks.

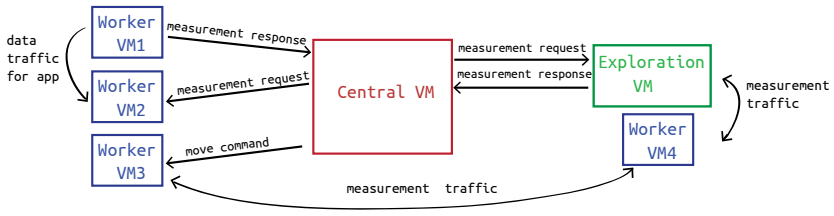


Figure 3: Example of several measurements in progress.

However, this strategy too costly. Assuming that each measurement lasts 1 second, measuring all $24^2$ pairs of racks will 19.2 minutes of VM time, and up to 5.8 terabytes of data transfer (assuming a 10 Gb/s maximum bandwidth).

By using a machine learning algorithm, we can drastically reduce the number of measurements while still detecting major network events in 20 to 40 seconds.

### Flood Length

Capacity measurements will flood the network with packets for one second before using the `tcp_throughput()` system call to measure the maximum throughput. By analyzing the TCP protocol, we conclude that it should require less than one second to reach the maximum TCP window size

If the path is not congested, we can assume that the network round trip time is 1ms and that the maximum link capacity is 10 GB/s. In this case, the maximum TCP window size is:

$$\begin{aligned} W_{max} &= (10GB/s) * (1ms) \\ &= 10MB \end{aligned}$$

TCP slow-start doubles the window size per round trip. If the initial window size is 1500 bytes, it should take 13 round trips, or 13 milliseconds, to saturate an empty link.

However, if the path is congested, a packet loss might cause TCP to switch to AIMD early on. A congested path might have a maximum capacity of 100 MB/s and a latency of 3 ms. This corresponds to a 314 KB maximum window size, which would take 210 round trips, or 630 ms to fill using pure AIMD and no further packet losses.

### Flooding Frequency

The measurement daemon continuously schedules path capacity measurement by sending RPC messages to worker and exploration VMs. The exact timing of the measurements is determined by a probability distribution such that the expected number of measurements taken during a one minute interval is proportional to a constant $P$.

The constant $P$ determines the tradeoff between cost and accuracy. Further analysis of this tradeoff is explored in the performance analysis.

### Exploration

To ensure that the entire network gets explored uniformly, the measurement daemon will temporarily provision exploration VMs in un-explored parts of the network. The rate of provisioning is also determined by the constant $P$.

The measurement daemon uses a simple rule to determine the placement of exploration VMs. The **Freshness Queue** orders each rack by the last time it was explored. When it is time to provision some new exploration VMs, the measurement daemon chooses the rack at the front of the queue. Then, it temporarily provisions exploration VMs to measure:

- The path capacity between two VMs in that rack
- The path capacity between a VM in the chosen rack and a VM in a nearby rack
- The path capacitiy between a VM in the chosen rack and a VM in a random rack

The above strategy prioritizes measurements between nearby racks because our application is most interested in finding network regions with high local capacity.

### Interference

Traffic generated by application may interfere with the accuracy of a capacity measurement. To solve this problem, the application daemon on a worker is temporarily paused if that worker is involved in a capacity measurement. This is okay, because measurements are rare and only last one second apiece.

## Machine Learning

The machine learning algorithm is used to infer the path capacities for VM pairs that were not explicitly meaured.

### Summary

The algorithm models the congestion in the routers in order to predict the route capacity between virtual machines.

Specifically we define a *probabilistic generative model* which predicts the expected path capacities conditioned on the hidden state of the routers. The state of each router is modeled during small, discrete time intervals, and it evolves through time according to a *hidden Markov model* (HMM).

During each time interval, a the most recent path capacity measurements are considered, and the router states are inferred by sampling their probability with the *Metropolis Hastings* algorithm.

After each round of inference, the new router states are stored in the *Measurement Table*.

### Generative Model

Each route between two VMs passes through one or more routers. It is assumed that at any point in time, a router $i$ can contribute a fixed amount of capacity $R_i$ to a the route. For example, if the network is empty, each router can dedicate 100% of its 10 GB/s link capacity to a connection. However, due to competing network connections, a router's $R_i$ may be much lower.

### Assumptions

For simplicity, router pairs (such as the core routers) are treated as single units. Additionally, we choose not to model many of the complexities of individual routers (such as buffer pressure and queue length).

### Route Capacity

The expected capacity $E[V_{jk}]$ of a VM to VM connection is determined by the capacity of the bottleneck router. In mathematical terms, $E[V_{jk}] = min(R_1, R_2, ..., R_n)$ for all router capacities $R_i$ on the path between VMs $j$ and $k$. Again, we are making several simplifying assumptions about the nature of the network.

This concept is illustrated in Figure 4.

### Measurement Error

The capacity measurements $V_{jk}$ are sampled over 1-second intervals. To model the measurement uncertainty, $V_{jk}$ is drawn from a normal distribution with variance $\sigma^2$.

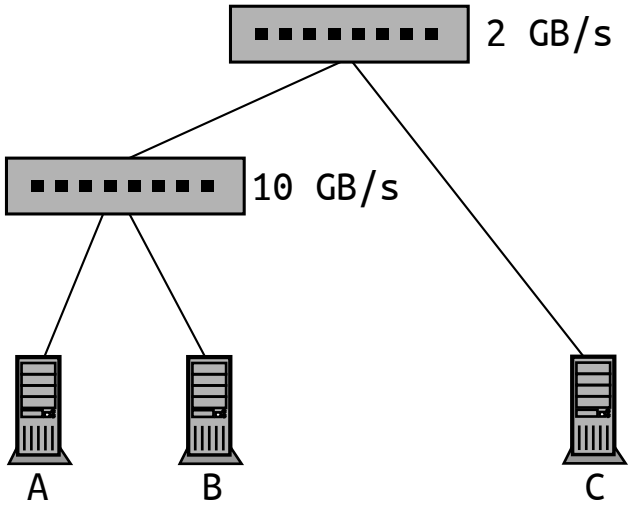


Figure 4: Here, VMs A and B are expected to have a 10 GB/s path capacity. VMs A and C are expected to have a 2 GB/s route capacity, due to the bottleneck router at the top of the figure.

### Hidden Markov Model

The $R_i$ for each router is assumed to be stable for a short intervals of time (such as 20 seconds). Over the long term, these values of $R_i$ should track the state of the routers.

The value of $R_i$ during interval $t$ is denoted as $R_i^t$. The probability distribution of $R_i^t$ is a beta distribution centered around its value in the previous time interval $R_i^{t-1}$. This captures the intuition that a struggling router during one period is likely to still be struggling during the next one. Statisticians would recognize this pattern as a *Hidden Markov Model*.

### Mathematical Representation

The full generative model is written below.

$$R_i^t \sim \text{Beta}(c \cdot R_i^{t-1}, c \cdot (1 - R_i^{t-1}))$$
$$V_{jk} \sim \text{Gaussian}(min(R_1, R_2, ..., R_n), \sigma^2)$$

where $c$ and $\sigma^2$ are user-defined parameters.

### Metropolis Hastings

During each time interval, new capacity measurements are taken, and the new distribution each $R_i^t$ is sampled using the Metropolis Hastings algorithm. Using this algorithm, we can compute aggregate statistics for each $R_i^t$ (eg. mean and covariance) as well as an approximation of the maximum likelihood estimator.

The measurement subsystem will send the new statistics of $R_i^t$ to the placement subsystem. Given these statistics, the optimization algorithm can easily calculate the mean and and variance of all $V_{ij}$ on demand.

### Performance

Based on the author's experience with a similarly sized model, it should take around 40,000 iterations of Metropolis Hastings to find an optimal solution. A machine-learning prototyping platform such as *Venture* would require about 6 ms per iteration, for a total of 240 seconds. However, by writing the sampler in C++, we expect a 10x to 100x speedup, fitting within our 20 second time budget.

# Placement Subsystem

The placement subsystem places and moves VM in the datacenter in order to decrease an objective cost function. The optimization is performed by an algorithm called *simulated annealing*, which is a semi-greedy, semi-random algorithm that finds a nearly-optimal system arrangement in bounded time.

## Cost Function

Given a configuration of VMs, we can calulate the cost of that configuration. We can compare two different arrangements and decide which one we prefer by choosing the one with the lower cost.

The choice of cost function defines the behavior of placement algorithm. The user of our system is given a choice of cost functions, each with its own pros and cons.

A cost function consists of two components:

- **Path Cost** determines the cost of a path between two VMs
- **System Score** is a function that combines all of the link costs in a configuration

The following cost functions will be described in terms of these two primitives.

### Minimize Time Until Finish

The simplest cost function is the following.

- **Path cost**: `data_left / path_capacity`
- **System score**: `max(link_cost)`

In this case, the link cost is the time until a link is finished transmitting its data. The system score is the amount of time until the straggler link finishes transmitting.

In a perfect world, minimizing this cost function should minimize the time until a job's completion.

### Minimize Total VM Execution Time

An alternative cost could be the following.

- **Path cost**: `data_left / path_capacity`

- **System score**: `sum(max(link_cost per VM))`

The link cost is the same as before, but the system score is different. Since a VM must be turned on until its last link is finished transmitting, `max(link_cost per VM)` equals the time each VM will remained turned on. Summing over all VMs gives the total VM time spent.

This cost function is appealing, because minimizing the VM time is equivalent to minimizing the money spent to complete the computation.

### Linear Combination

We also considered considered using a linear combination of variables.

- **Path cost**: `-a*path_capacity + b*data_left`
- **System score**: `sum(link_cost)`

Although a linear function is easier to optimize, it was not clear how this cost function would translate into our high-level goal of reducing computation time.

### Problems

All of the above cost functions assume that:

1. application is always transmitting the maximum possible amount of data
2. the network is unchanging

Both of these assumptions are unfounded, so we apply some additional modifications to these cost functions.

### Utilization

To address the first assumption, we considered an example where an application has lots of data left to transfer, but is not currently sending a lot of data. Figure 5 shows such an example.

To address this example, we defined the variable `utilization` as follows

```
utilization = data_left/path_capacity
```

If a path has `utilization = 0`, then there is no need to optimize that path. On the other hand, if `utilization = 1`, that path is being bottlenecked, and the placement algorithm should try to alleviate that bottleneck.

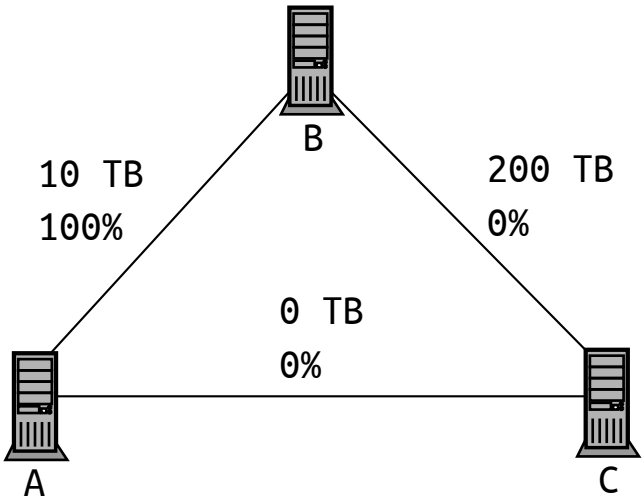


Figure 5: Three VMs, along with the `data_left` and `utilization` for each link. Although VMs B and C have lots of data left to transfer, they are currently not sending any data. In this case, it is more important to optimize the connection between A and B, because they are currently sending a lot of data.

### Better Metric

We invented a better metric to take `utilization` into account.

- **Path cost**: `data_left / path_capacity * (utilization ^ a)`
- **System score**: `sum(max(link_cost per VM))`

Where `a` is a user-defined constant. This score has the nice characteristic of being zero if there is either no data left to transfer or if utilization is zero (which means that the VM isn't currently sending any data). In addition, if utilization of the link is 100%, the link cost equals the expected time to transfer all the data.

## Execution

Whenever the measurement daemon notifies the placement daemon that new measurements are available, the placement daemon uses an optimization algorithm to find an arrangement of VM's that minimizes the system cost. If the new configuration improves the cost by at least 5%, then the actual VMs on the network are moved to the new configuration.

Periodically, the placement updates the *Location Table* with the information returned by the `machine_occupancy(m)` function call.

### Movement failures

Sometimes, the application will fail to move the VMs to the new configuration. This occurs when a different user of the datacenter steals an empty VM slot that our application is trying to allocate.

Our application deals with this case gracefully. When it detects a placement error, it takes inventory of the empty VMs in the network (via the `machine_occupancy` API call) and re-executes the placement algorithm after a randomized delay. The purpose of the delay is explained in the third case analysis.

### Stability

Because the network is constantly changing (assumption 2 from the previous section), movements are expected to occur quite often.

A movement failure might temporarily leave a configuration in a sub-optimal state while the placement algorithm is re-executed. Because of the risk associated with VM movement, the placement algorithm will only move to a new configuration if the cost improvement is at least 5%.

## Optimization

The optimization algorithm is used to find the configuration of VMs that minimizes the cost function. We considered the following two algorithms before choosing to use an algorithm called *simulated annealing* (SA).

### Exhaustive Enumeration

The simple solution to this optimization problem is exhaustive enumeration. However, for the moderate case of about 50 VMs and 1000 available slots, it would take $\binom{1000}{50} = 9.4 \times 10^{84}$ iterations, which is greater than the number of atoms in the observable universe.

### Greedy Approach

A greedy solution to this optimization problem would likely get stuck in a deep local minimum, such as the situation described later in Analysis Case 2.

### Simulate Annealing

SA lies somewhere between the previous two approaches. It performs random walk of the space of all possible configurations, with a bias towards lower-cost configurations. After a fixed amount of computation time, SA returns the lowest-cost configuration that it found.

Although SA does not guarentee a globally optimal configuration, it has a probabilistic guarentee to find a reasonably optimal configuration within a fixed period of time.

## Transition Functions

In order the perform random walk over configurations, SA requires a set of transition functions. The main two transition functions will be:

1. swap two application VMs
2. move an application VM to a free slot

### Avoiding Local Minima

For performance-related reasons, we need to deter the random walk from getting stuck in a deep local minimum. For example, there might be a situation where moving any single VM would produce a higher cost, but moving multiple VMs

in unison produces a lower cost. To address this case, we use an additional transition function:

3. move a random subset of VMs into a set of free slots, placing them as close together as possible

#### Search Space Diameter

SA requires that the search space has a relatively small diameter, or that the number of transitions between any two configurations is small. This condition is satisfied. Moving between any two configurations simply requires $n$ moves or swaps where $n$ is the number of VMs.

# Performance Analysis

First, we analysize the space, bandwidth, and money used by an instance of our system. Next we describe how our application responds to some example workloads.

## Cost Analysis

#### Space Analysis

The data structures in the master VM are expected to take no more than a few hundred megabytes of space.

- **Measurement Table** contains on the order of 60 router entries.
- **Data Matrix** contains up to $4608^2$ or 21 million entries. If each entry is 10 bytes, the total is 210 MB.
- **Location Table** contains up to 4608 entries
- **Freshness Queue** contains 48 entries

#### RPC Bandwidth

The calls to `tcp_throughput` and `progress` are stored in the Data Matrix. Thus, in the worst case (when the user owns all 4609 VMs in the data center), these calls will require 210MB of bandwidth per sample. By sampling these values every 10 seconds, they use up to 20 MB/s of bandwidth on the master node. However, the sampling rate may easily be reduced or increased depending on the situation.

### Measurement Frequency

The central VM schedules periodic path capacity measurements that flood the network. The measurements are scheduled by a Poisson process such that the average number of measurements in a one minute interval is proportional to a constant $P$.

$P$ is defined to be the average number of capacity measurements that occur during a one minute interval if *all* virtual machines in the data center were using our measurement algorithm.

If an instance only uses 1/8th of the total VMs in the data center, it will spawn $P/8$ measurements per minute.

The following performance analysis shows that the value $P = 16500$ will consume only 1.2% of the total inter-rack network capacity, and will still allow a 10 VM instance discover a large block of underutilized racks in about 20 seconds.

### Flooding Bandwidth

The path capacity measurements require a lot more bandwidth. If the user owns all 4609 VMs in the data center, there will be approximately $P$ path capacity measurements occuring every minute.

Each rack has two 10 GB/s router connections. If that bandwidth is shared fairly among the 192 VMs on each rack, the average path capacity between VMs in different racks is 10MB/s.

The total inter-rack network capacity is the sum of all of the first-tier router connections divided by two (since each connection must have a source and a destination). The number comes out to 240 GB/s.

Using $P = 16500$ means that the system will flood $16500 * 10\text{MB} = 165\text{GB}$ of measurement traffic every minute, or 2.8 GB/s. This is equal to 1.2% of the total inter-rack network capacity. This should have minimal effect on user applications.

### Measurement Cost

Assume the user is running a 10VM application. Using the value $P = 16500$, we expect the system to take 36 path capacity measurements per minute. Assuming that each measurement takes 2 second of VM time (either on worker or exploration nodes), the measurements require a total of 72 seconds of VM time per minute. This means that measurements will increase the execution cost by $72/600 = 12\%$.

Users of larger instances may choose to use a smaller $P$ value to decrease the cost of measurement, while still delivering reasonable results.

Additionally, one must factor in the cost of running the master VM.

## Workload Analysis

Our system performs well when the application requires high throughput, even if some of the transfers depend on each other, or if the workload is uneven. However, if the application is CPU bound, or if all routers are equally congested, then the benefits are not realized.

### High Throughput Application

Suppose there is a 10 VM application where all nodes have an equal amount of data to send, and they are always able to send the maximum amount of data possible.

In this case, the overhead of running the measurement system is 12%, and the overhead of the master node is $1/10 = 10\%$. If VrooM is successful if it is able to improve the placement of the VMs by at least 22%

If VrooM is able to place all machines in the same rack, the throughput will increase by several orders of magnitude (MB/s to GB/s). In this case, the 22% cost overhead is clearly justified.

### Uneven Data Matrix

Because the placement algorithm minimizes `max(data_left/path_capacity)` per VM, worker VMs that have the most data to transfer will be placed closer together. Thus, it will produce a good result for all possible data matrices.

### Dependant Transfers

Suppose one half of the network transfers must be completed before the second half may begin.

Because the cost function incorporates the utilization of the links between nodes, VrooM will not bother optimizing the inactive connections. This allow the system to optimize more heavily for the first half of network transfers until they are finished.

### CPU Bound Jobs

VrooM optimizes for network bandwidth, and will not help applications that have additional bottlenecks, such as CPU or disk IO. In these cases, the cost of measurements will outweigh the benefits of increased network bandwidth.

#### Uneven Congestion

If the congestion between routers is highly uneven, then there is a very high probability that VrooM will pay for itself by bypassing the slow routers. Moving from a congested part of the network to an uncongested part of the network could increase bandwidth from around 5MB/s to more than 100MB/s, resulting cost savings of 2000%.

On the other hand, if the network congestion is uniform, VrooM has less room to optimize. In this case, the measurement costs will outweigh the benefits.

# Case Analysis

## Analysis Case 1

First, we consider a case where each machine in the data center already hosts 3 VMs and our application needs to place 100 identical VMs.

#### Exploration Phase

Since our system has no information about the current state of the network, it starts by using exploration nodes for taking measurements at a rate of 360 measurements per minute (based on $P=16500). After 20 seconds, there should be enough measurements to identify any underperforming racks or routers.

#### Placement Phase

Then, our system then runs the placement algorithm to find an optimal initial placement of the 100 VMs. Assuming that the measurement algorithm deduced that the higher-tier routers are congested, all of the machines would be placed as close together as possible in the least-congested groups, since this would minimize the configuration cost function.

## Analysis Case 2

Next, we consider a case where the user has 10 VMs in racks 1-6, a highly congested part of the network. Suddenly, a large number of high-throughput machines become available in racks 19-24. Our system is able to detect the underutilized machines and relocate the 10 VMs in a timely manner.

### Measurement Phase

The 10 VM cluster is permitted to explore 36 racks per minute. Since each rack exploration takes 3 measurements, the cluster explores 12 new racks per minute. With $1 - (3/4)^4 = 70\%$ probability, the cluster will explore the a high performance rack within 20 seconds, and with 90% probability, the cluster will explore one within 40 seconds. We assume that the machine learning algorithm correctly identifies this rack as having high-performance.

### Placement Phase

Immediately after the new measurements arrive, the placement algorithm is executed. Because of the addition of transition function 3 described in the simulated annealing section, the SA algorithm will move all 10 VMs to the new group with extremely high probability.

## Analysis Case 3

Finally, we consider the case where multiple users of the datacenter are using our placement algorithm. In this context, there are two main concerns: measurement overhead and contention.

### Measurement Overhead

Since our bandwidth measurements involve flooding network links with dummy packets, the network could become saturated if too many measurements are taken. This would result in poor performance for everybody.

Because the total amount of measurements per minute is capped by the constant $P = 16500$, measurement overhead is guarenteed to not exceed 1.4% of the router capacity.

### Contention

In Analysis Case 2, an instance of our system discovered a high-performance group and attempted to move its VMs into that group. If multiple instances of our system are executing simultaneously, they will both try to move their VMs into the same group at roughly the same time. As a result, some of the VMs will fail to move.

This is problematic for two reasons. First, the failed VM movements could potentially leave both instances worse off than before. Secondly, the instances could repeatedly conflict with each other as they attempt to fix the situation.

Our system deals with contention by introducing a random delay whenever an instance fails to move a VM. This way, if multiple instances conflict during a round of VM placements, the random delay will prevent them from conflicting on the next round of placements with high probability.

## Conclusion

VrooM is capable of increasing application performance by at least 2000% in highly dynamic, congested datacenter network. By using a machine learning algorithm to minimize measurement overhead, VrooM will consume no more than 1.4% of network bandwidth and no more than 12% of VM time. Since these parameters are configurable, power users may optimize their clusters to reduce the system overhead.

Since VrooM is optimized for a particular domain, it makes several assumptions which might not be valid in other domains. However, the ideas of *probabilistic generative models* and *simulated annealing* could be applied to many other network optimization problems.

Word Count: 4930