# Beyond the Ring: Quantized Heterogeneous Consistent Hashing

Yoav Levi, Isaac Keslassy
*Technion*

***Abstract*—Consistent hashing (CH) is a crucial building block for load-balancers. It enables packets of the same flow to keep being mapped to the same server whenever possible. Load-balancers implement heterogeneous CH to deal with the varying server speeds. However, they mostly rely on the old Ring algorithm, suffering from practical scalability and stability issues, as well as from a lack of theoretical stability guarantees.**

**This paper presents a new framework for heterogeneous CH. The framework relies on quantization using virtual servers, and on a min-max fairness-based mapping algorithm denoted M3. The paper establishes necessary and sufficient conditions to guarantee stability for any arbitrary heterogeneous server service rate. We also explain why M3 presents better scalability properties than all heterogeneous CH alternatives, including a faster key lookup rate and a lower memory footprint. Finally, evaluations show that M3 also offers a significantly increased stability region.**

## I. Introduction

**Heterogeneous CH.** Consistent hashing (CH) maps the keys (identifiers) of incoming objects into a set of servers, while attempting to achieve (1) *minimal disruption*, i.e., minimum mapping changes as servers are arbitrarily removed or added, and (2) *balance*, i.e., even load-balancing of the keys across servers such that no server is overloaded [1]–[4]. For example, assume a hash function $h$ that uniformly maps any key to a number in $[1, 1024]$. If we have 8 servers, a naive load-balancing method would map any key $x$ to server number $s = h(x) \bmod 8$. This method achieves *balance* because $h$ is uniform. Assume now that some server $s' \neq s$ fails and we are left with 7 servers. This method would attempt to remap key $x$ to server $(h(x) \bmod 7)$, and therefore would most likely terminate the connection associated with the key $x$ even though the server $s$ of this connection did not fail. Therefore, this naive method does not achieve *minimal disruption* and is not CH.

Today, CH is a crucial building block in datacenter load-balancing [5]–[8]. L4 load-balancers use CH to evenly forward incoming packets to servers, while maintaining the affinity of TCP connections as servers are removed or added. CH is also used in many other applications, from distributed storage to content delivery networks, and from equal-cost multi-path (ECMP) routing to distributed hash tables [9]–[15].

Datacenter load-balancers need *heterogeneous CH*, as they rely on servers with vastly different service rates. These service-rate differences may be permanent: e.g., if the servers belong to different server generations; rely on different hardware to process packets (CPU, GPU, SmartNIC, FPGA, ASIC); or have different memory access speeds (hard disk drives vs. flash memory). The differences may also be temporary: e.g., if the servers suffer from varying levels of network congestion; host additional virtual machines for other applications, arbitrarily affecting their effective service rate; or are rebooting and gradually increasing their service rate while checking that everything runs as expected (a common feature called *slow start* but unrelated to TCP's slow start) [8]. The load-balancer is expected to be able to periodically update each of the expected server rates, then run an heterogeneous CH algorithm that seamlessly fits these updated rates.

**Related work.** Table I summarizes the main existing CH algorithms. Unfortunately, most of the literature focuses on *homogeneous CH*, as shown in the first four lines with Ring, Dynamo (strategy 3, an extension of Ring), MaglevHash, and AnchorHash [1], [4], [6], [13], [16]–[19].

To implement *heterogeneous CH*, papers in the literature [2], [3], [13], [20] and industry load-balancers [7], [8], [21], [22] mostly use a heuristic *weighted Ring* hashing algorithm, which relies on *virtual servers* to fall back on the homogeneous-CH Ring algorithm [16]. For instance, assume we want to load-balance between two servers 1 and 2 with respective service rates $\mu_1 = 1/3$ and $\mu_2 = 2/3$. Because of the well-known poor balance properties of the Ring algorithm [21], a common approach is to assign a large proportional number of virtual servers to each (e.g., 100 virtual servers to server 1 and 200 to server 2, using a total of $q = 300$ virtual servers), and then run homogeneous CH on the $q = 300$ virtual servers. Any packet mapped to a virtual server of server $i$ will then be forwarded to server $i$. Unfortunately, weighted-Ring solutions share many issues. First, the number of virtual servers is always chosen arbitrarily, without stability guarantee. Second, the weighted-Ring structure constrains the lookup rate: in our example with 300 virtual servers, organizing them as an ordered binary tree would yield an expected number of at least $\log_2(300)$ lookups per key. Finally, we show in evaluations that the poor balance in weighted-Ring solutions leads to instability.

There are many weighted-Ring solutions. As shown in the fifth line of the table, ACH [2] is an algorithm for storage-related load-balancing. It assigns each server a number of virtual servers that is equal to the ratio of its storage capacity by an arbitrary fixed constant ($C_{throttle}$).

The weighted Ring is also implemented in Ketama [22],

TABLE I. Performance comparison between known CH algorithms. Ring [16], Dynamo (strategy 3) [13], MaglevHash [6] and AnchorHash [1] are designed for homogeneous systems only. Weighted-Ring approaches like ACH [2] support heterogeneity, but suffer from sub-optimal lookup speed, memory footprint, and especially balance, affecting their stability. HRW [17] supports both heterogeneity and consistency but has a prohibitive key-lookup rate. Our algorithm M3 attempts to provide both scalability and stability, at the cost of a non-ideal balance, since it only guarantees stability up to some load $\rho$ that can be set arbitrarily close to 1 (Eq. (1)).

| | **Hetero.** | **Scalability** | | **Consistency** | |
|---|---|---|---|---|---|
| | | **Lookup rate** | **Memory** | **Min. disrupt.** | **Balance** |
| Ring | × | ✓✗ | ✓✗ | ✓ | × |
| Dynamo (strategy 3) | × | ✓ | ✓ | ✓ | ✓✗ |
| MaglevHash | × | ✓ | ✓✗ | × | ✓✗ |
| AnchorHash | × | ✓ | ✓ | ✓ | ✓ |
| ACH (weighted Ring) | ✓ | ✓✗ | ✓✗ | ✓ | × |
| HRW | ✓ | × | ✓ | ✓ | ✓ |
| **M3** | ✓ | ✓ | ✓ | ✓ | ✓✗ |

[23], the CH library in the NGINX platform [7], [21], and apparently also in the HAProxy platform [8], although details are lacking.

Hierarchical CH [3] uses two different hash rings to treat two types of servers (SSDs vs. HDDs), however this setup is very limited and not suitable for most heterogeneous systems. Weighted DxHash [24] suggests to assign a different number of virtual nodes to each physical server, but does not mention how many to assign to each physical node. Beyond the weighted-Ring approach, the only other published heterogeneous CH algorithm is HRW [17] (sixth line). HRW achieves an excellent balance, but with a prohibitive lookup rate that makes it impractical for real-world systems. Additional homogeneous CH papers like Maglev mention the possibility of heterogeneity support but keep any heterogeneous version undisclosed [6], [13].

Beyond heterogeneous CH, optimally load-balancing balls into non-uniform bins is a well-known problem that can be tackled using several algorithms [25]. However, these algorithms do not address the online variant with bin additions and deletions, which is crucial for CH. Furthermore, there has been a large recent literature on load-balancing to heterogeneous servers without the CH constraint, although it tackles different fundamental questions [26]–[36]. Finally, both homogeneous and heterogeneous CH can add connection tracking to maximize per-connection consistency when the server set changes [37].

**Contributions.** In this paper, we argue that heterogeneous CH is a crucial building block in the load-balancer industry component, yet it currently lacks any sound fundamental basis and achieves poor results. Our goal is to solve two significant issues in the predominant weighted-Ring approach. First, on the fundamental side, given an arbitrary set of server rates $\{\mu_i\}$, there is no known stability guarantee linking the number $q$ of virtual servers, the number $n$ of servers, and the maximum stable load $\rho$. Second, on the practical side, the weighted-Ring approach displays poor stability and scalability properties that hinder the adoption of large-scale heterogeneous CH.

We start by defining a new formal framework for heterogeneous CH with virtual servers, denoted as the *Hash&Map* (*H&M*) framework. This framework consists of two simple stages: a hashing stage that maps an incoming key to a virtual server, then a mapping stage that forwards any key of this virtual server to a given physical server.

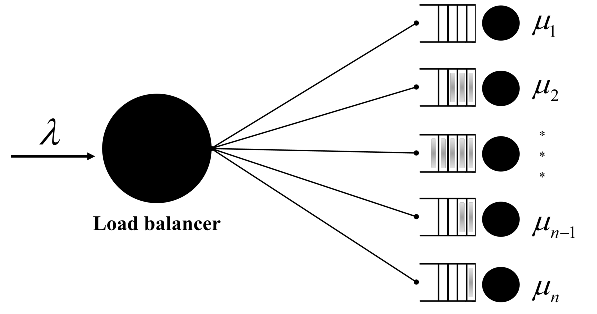


Fig. 1. Classical heterogeneous load-balancing system, including a load-balancer of normalized arrival rate $\lambda$, and $n$ physical servers of normalized service rates $\mu_i$.

We introduce the *Min-Max Mapping* (*M3*) algorithm for allocating virtual servers in the H&M framework. We prove that M3 is optimal, i.e., no other algorithm can achieve a better stability region in the H&M framework. We further show that since M3 operates within the H&M framework and the load is split between an integer number $q$ of virtual servers, then M3 needs to deal with quantization constraints that derive from this integer number. Given a goal of guaranteeing a large stable load $\rho$, we prove the main fundamental result (Thm. 3), providing a simple sufficient and necessary condition for M3 to achieve stability over the entire set of all heterogeneous systems. Formally, Thm. 3 establishes that the M3 algorithm is guaranteed to be stable at load $\rho$ given *any* heterogeneous server service rates $\{\mu_i\}_{i=1}^n$, iff:

$$q > (n-1) \cdot \frac{\rho}{1-\rho}. \tag{1}$$

For instance, for $n = 100$ servers and $\rho < 0.99$, M3 guarantees stability given any arbitrary service rates as long as it can use $q = (100-1) \cdot \frac{0.99}{1-0.99} + 1 = 9,802$ virtual servers (well below typical industry numbers of 65K to 650K [6]).

Furthermore, we compare our suggested M3 algorithm against two algorithms from the literature: NGINX's Ketama library [22] and ACH [2]. Both algorithms rely on a weighted Ring approach. We run evaluations in load-balancer and distributed-storage configurations. We find that M3 can achieve high loads, while both other algorithms suffer from intrinsic stability issues, especially with a high number of servers. In addition, M3 is more scalable, with a higher lookup rate ($O(1)$ lookup cost vs. $O(\log_2 q)$) and a lower memory footprint (by up to two orders of magnitude).

The full code for this paper is available online [38].

## II. Model

**System.** As Fig. 1 illustrates, we consider a heterogeneous load-balancing system where a load-balancer spreads the incoming packets among $n$ servers. Each server has its own queue and follows a first-come-first-served discipline. Let $\mu_i$ denote the service rate of server $i$. For simplicity, we use a normalized notation such that the total service rate is 1, i.e., $\mu \equiv \sum_{i=1}^{n} \mu_i = 1$. We denote by $\bar{\mu} \equiv \{\mu_i\}_{i=1}^{n}$ the service rate vector. Likewise, we denote by $\lambda$ the packet arrival rate at the system, and by $\rho \equiv \frac{\lambda}{\mu}$ the load. Due to the normalization, $\rho = \lambda$ since $\mu = 1$. In the remainder, we use either $\lambda$ or $\rho$, depending on whichever makes formulas more intuitive.

**Load-balancer.** Each packet has a key. We assume that the Simple Uniform Hashing Assumption (SUHA) holds [1], [39], i.e., that the output of the hashing function of the packet keys is distributed uniformly across its codomain. We further assume that the load-balancer assigns each server $i$ a constant fraction of the incoming traffic, such that at each server $i$ the (normalized) incoming rate is $\lambda_i$ with $\sum_{i=1}^{n} \lambda_i = \lambda$, and the server load is $\rho_i = \frac{\lambda_i}{\mu_i}$.

**Stability.** To keep the queueing model general, we assume that the queueing model for each server $i$ satisfies the following property: server $i$ is *stable* iff its arrival rate is strictly smaller than its service rate, i.e., $\mu_i - \lambda_i > 0$. This is satisfied for instance in the M/M/1 and M/D/1 queueing models. Likewise, a system with a given load-balancing algorithm is *stable* (or, in short, its algorithm is stable) iff all its servers are stable. A system is *admissible* iff its arrival rate is smaller than its total service rate, i.e., $\sum_{i=1}^{n} \mu_i - \lambda = 1 - \lambda > 0$. In the paper, we always assume admissibility.

**Homogeneous CH.** We define an homogeneous load-balancing algorithm as *CH* iff it satisfies:
(1) *Minimal Disruption:* Any change in the number of servers leads to a minimal amount of key re-mappings. Namely, when a server is removed, only the keys that were assigned to that server should be re-mapped; and when a server is added, only keys that are newly assigned to the new server should be re-mapped.
(2) *Balance:* Any key chosen u.a.r. (uniformly at random) in the set of keys has an equal probability of being assigned to each physical server.

In heterogeneous systems, the minimal-disruption property is desirable too. However, since heterogeneous systems have non-identical servers, uniformly balancing the load among them can lead to instability.

## III. H&M FRAMEWORK

**Design goals: scalability and heterogeneity.** In this section, our goal is to design a scalable CH for heterogeneous systems. We want to achieve scalability along two important metrics: (1) a high *key-lookup rate*, by minimizing the number of hash operations per key, and (2) a low *memory footprint*, even when there is a large number of servers. At the same time, we want a system that is flexible enough to fit a large degree of heterogeneity in the service rates of the servers.

**H&M framework.** As Fig. 2 illustrates, to achieve both scalability and heterogeneity, we intuitively leverage the idea of using *indirection*, and introduce a middle-layer set of $q \in \mathbf{N}^*$ virtual servers. Thus, we split our system into two

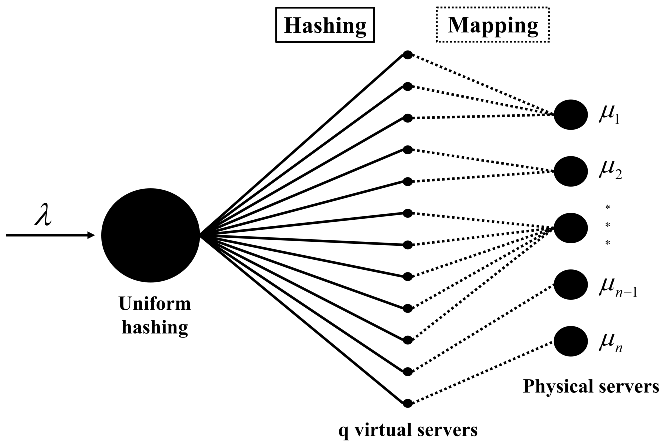


Fig. 2. H&M framework, consisting of two stages: first, the load-balancer hashes uniformly to $q$ identical virtual servers. Next, each virtual server is mapped to one of the physical servers by the algorithm described in Sec. IV-B.

stages: first, a hashing stage; then, a mapping stage. We denote this general architecture as a *Hash&Map* (*H&M*) framework.

**Hashing stage.** Specifically, using standard uniform hashing, the load-balancer first hashes the keys of incoming packets into the set of $q$ virtual servers, so a given key is always mapped to the same virtual server. Thus, each virtual server receives the same average load of $\frac{\lambda}{q}$ (according to the SUHA principle in Sec. II).

**Mapping stage.** In the second stage, we map each virtual server into a unique physical server, so a key that is always mapped to the same virtual server is now also always mapped to the same physical server. As discussed in the next section, the mapping algorithm attempts to match more virtual servers to servers with a larger service rate, so as to provide more traffic to stronger servers. Specifically, let $q_i \in \mathbb{N}$ denote the number of virtual servers mapped to physical server $i$, with $\sum_{i=1}^{n} q_i = q$. Then stronger servers should get a higher $q_i$. Note that H&M can store the second-stage mapping as a single array of $q$ values in $[1, n]$.

**Key lookups**. When packets arrive at the H&M system, they are assigned to their physical server using two single operations:

1) The load-balancer hashes their key to one of the virtual servers using a single uniform hashing operation.
2) The load-balancer accesses once the array of $q$ elements and finds the server $i$ that the virtual server is mapped to. It then sends the packet to server $i$.

**Scalability.** The H&M framework fits our two scalability design goals: (1) *High lookup rate*: For each packet, it only needs one hashing operation and one memory-read access. Thus it achieves an $O(1)$ complexity, vs. $O(\log q)$ in weighted Ring for example. (2) *Low memory footprint*: beyond the above array, which needs one memory word per virtual server, the memory footprint is negligible.

## IV. M3 ALGORITHM

We have found that the H&M framework fits our scalability goals. In this section, we focus on the second-stage mapping algorithm in the H&M framework and show how it can help adapt to the heterogeneous service rates.

### A. Challenges

The use of indirection through the middle-layer of virtual servers introduces *quantization*. We saw that each virtual server contributes a rate quantum of $\frac{\lambda}{q}$. Therefore, if $q_i$ virtual servers are mapped to a given physical server, its incoming rate is $\lambda_i = \lambda \cdot \frac{q_i}{q}$.

**Quantization challenge 1: irrational service rates.** Consider the case of two servers with irrational service rates $\bar{\mu} = \left(\frac{1}{e}, 1 - \frac{1}{e}\right)$. No finite quantization with a finite $q$ can provide full balance, because a rational number $\frac{q_i}{q}$ with a finite denominator $q$ cannot approximate an irrational number arbitrarily close. Specifically, the load of each server $i$ is $\rho_i \equiv \frac{\lambda_i}{\mu_i} = \frac{\lambda \cdot \frac{q_i}{q}}{\mu_i}$. At full heterogeneous balance, we would want $\rho_1 = \rho_2$, which can be rewritten as $\frac{q_1}{q_2} = \frac{\mu_1}{\mu_2}$. The left side is rational, and the right side is irrational, thus a perfect load balancing is impossible.

**Quantization challenge 2: non-monotonicity.** Fig. 3 illustrates the stability challenges in the H&M framework. First, Fig. 3(a) shows an unstable system at load $\lambda = \rho = 0.8$. Each of the $q = 10$ virtual servers distributes a quantum of exactly 10% of the load, i.e., $\frac{\lambda}{q} = 0.08$. We can see that no matter the mapping of the last virtual server (as shown by the dashed red lines), all choices lead to instability. For example, server 1 is stable iff $\lambda_1 = \lambda \cdot \frac{q_1}{q} < \mu_1 = 0.15$. Since $\lambda \cdot \frac{2}{q} = 0.16 > 0.15$, server 1 cannot be assigned a second virtual server. Likewise, server 2 cannot be assigned a third virtual server ($3 \cdot 0.08 > 0.23$), and servers 3 and 4 cannot be assigned a fourth ($4 \cdot 0.08 > 0.31$). Due to quantization, there is no feasible mapping of the ten virtual servers that can stabilize all four physical servers.

Fig. 3(b) illustrates the same system but with $q = 20$ virtual servers. Now the additional virtual servers provide smaller quanta that enable stability.

Fig. 3(c) shows the same system with $q = 6$ virtual servers. Even though $q$ has decreased when compared to Fig. 3(a) and therefore the quantization is coarser, the system is stable. Therefore, the stability property is *not* monotonic with $q$: the relationship is more complex than appears at first.

In fact, Table II illustrates the values of $q$ for which the system reaches stability, and shows how the result is not obvious to the eye.

**Definition.** The above examples show that providing system stability is not always possible within the quantized H&M framework. As a result, we redefine the goal of the H&M mapping algorithm: within the quantization constraint established by a given $q$, it should provide system stability to the heterogeneous servers *whenever possible*, i.e., whenever at least one other algorithm can provide it under the same constraints. This is reflected in the following definition:

TABLE II. Stability of the system in Fig. 3 as a function of the number $q$ of virtual servers

| $q$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Stability | × | × | × | × | × | ✓ | ✓ | ✓ | ✓ | × | ✓ | ✓ | ✓ |

**Definition 1 (quantized consistent hashing).** *Within the H&M framework, a mapping algorithm satisfies the quantized CH property iff it achieves:*

1) Minimal disruption: *As in the homogeneous CH definition, a change in the number of servers causes a minimum number of mapping changes.*
2) Maximal stability: *Whenever possible given the quantization parameter $q$, the load at each server $i$ must be admissible, i.e., $\rho_i < 1$.*

We later introduce an algorithm that satisfies this definition, and prove that in the homogeneous case where all server rates are equal, if we assume that $q$ is a multiple of $n$, then it also satisfies the traditional definition for homogeneous CH based on balance. Thus, the above definition can intuitively be seen as generalizing the classical homogeneous definition to take into account both heterogeneity and quantization.

**Stability challenges.** The above *maximal-stability* condition in Def. 1 states that we want to achieve stability whenever possible. Since we have stability at server $i$ when $\rho_i < 1$, it makes sense to roughly equalize the server loads $\rho_i$ across the different servers so that none will exceed 1. To do so, we would intuitively want to minimize the maximal server load and make sure it is under 1. We formally define this max-load minimization policy as follows:

**Definition 2 (Min-max fairness).** *A server load vector $\bar{\rho}^m$ is min-max fair iff*

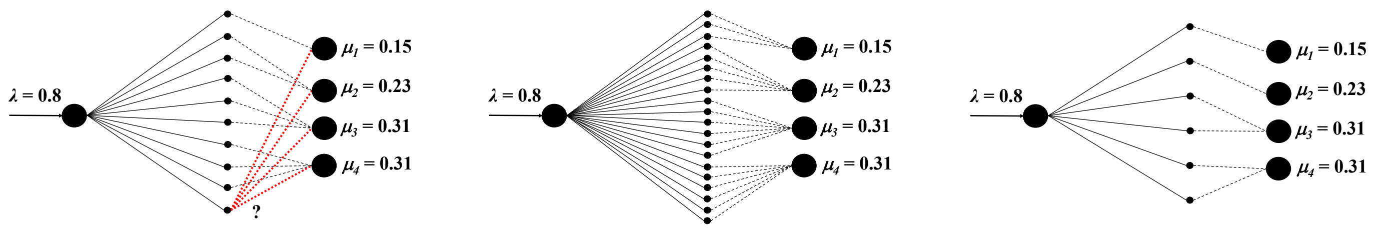
$$\max \bar{\rho}^m \leq \max \bar{\rho}$$

*for any feasible load vector $\bar{\rho}$, i.e., any $\bar{\rho}$ that results from a feasible second-stage mapping in the H&M framework.*

### B. The M3 Algorithm

We now describe the Min-Max Mapping (M3) algorithm, which is designed to satisfy the min-max fairness definition (Def. 2). We later prove that it also satisfies the two properties of the quantized-CH definition (Def. 1). M3 consists of two steps: first, computing the number $q_i$ of virtual servers that should be mapped to each physical server; then, obtaining an exact mapping from the set of virtual servers to the set of physical servers that indeed maps $q_i$ virtual servers to server $i$.

**Step 1: computing $q_i$.** The M3 algorithm starts by computing the number $q_i$ of virtual servers that should be mapped to each physical server. M3 follows a greedy min-max fairness algorithm, similar to a weighted water-filling approach. It does so in an iterative manner, from the first virtual server to the last: for each considered virtual server, it compares the result of mapping it to each of the $n$ servers, and picks the server that greedily minimizes the resulting maximum server load.



(a) $q = 10$: unstable system (b) $q = 20$: stable system (c) $q = 6$: stable system

Fig. 3. Non-monotonicity of the stability condition: Illustration of the same system with three different values for the number $q$ of virtual servers. With (a) $q = 10$, no mapping can achieve stability. However, with (b) $q = 20$ and (c) $q = 6$, the same system does achieve stability.

Specifically, for each physical server $i$, it assumes that it is assigned this additional virtual server, thus getting $q_i+1$ virtual servers and a resulting load of $\rho_i \equiv \frac{\lambda_i}{\mu_i} = \frac{\lambda \cdot \frac{q_i+1}{q}}{\mu_i}$. Then, since $\lambda$ and $q$ do not depend on $i$, to find the physical server that would least suffer from the additional load of an extra virtual server, it finds the $i$ that minimizes $\frac{q_i+1}{\mu_i}$ among the various $i$'s, and assigns it this virtual server. (In case of tie-break, M3 picks the first index that achieves this minimum.)

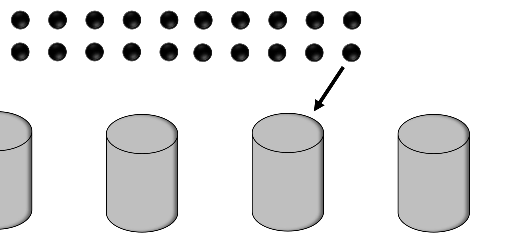
**Example for step 1.** Fig. 4 illustrates how M3 operates in the setting of Fig. 3(b), i.e., how it greedily assigns the $q = 20$ virtual servers to the four physical servers. Fig. 4(a) first shows how it assigns the initial virtual server. M3 computes the ratio $\frac{q_i+1}{\mu_i} = \frac{1}{\mu_i}$ for each physical server $i$, looking at the consequence of assigning it this first virtual server. Servers 3 and 4 have the smallest ratio, meaning that they can best handle an additional load, since they have the largest service rate $\mu_i$. Therefore, M3 assigns the first virtual server to server 3.

Next, Fig. 4(b) shows how M3 assigns the last virtual server in the same manner. This time the second server gets the smallest ratio. The final number of virtual servers for each physical server is $\{3, 5, 6, 6\}$, respectively, summing up to $q = 20$ and corresponding indeed to the allocation in Fig. 3(b).

**Step 2: obtaining a mapping.** Once M3 has computed $q_i$ for each physical server, it uses these values to compute a mapping (denoted $\mathcal{M}$) between the virtual and physical servers so that each physical server $i$ is indeed assigned $q_i$ virtual servers. This is a simple step: for instance, the first server is simply assigned the set of virtual servers $\{1, \ldots, q_1\}$, the second server is assigned $\{q_1 + 1, \ldots, q_1 + q_2\}$, and so on. As a general formula, each server $i$ is assigned virtual servers $\left\{ \left( \sum_{j=1}^{j=i-1} q_j \right) + 1, \ldots, \sum_{j=1}^{j=i} q_j \right\}$.

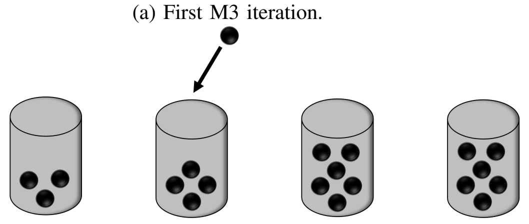
**Mapping implementation.** As previously mentioned, H&M can store the second-stage mapping $\mathcal{M}$ as a single array of $q$ values in $[1, n]$. Thus, note that the regularity or contiguity of $\mathcal{M}$ is not an issue. In addition, to speed up operations during server changes, we also store the reverse mapping, i.e., for each physical server $i$, we remember its mapped virtual servers using a small stack data structure. The update operation of this stack is done effectively by pushing (respectively popping) the added (resp. deleted) virtual servers.

**Server failures and additions.** An important feature of consistent hashing is the ability to quickly react to unexpected changes of the server set (e.g., server failure or addition). In addition, we want to react rapidly to changes in the server service rates. Hence, upon any change of either the number of servers $n$ or their service rate vector $\bar{\mu}$, the M3 algorithm first updates their values, then updates the values of $\{q_i\}$, and finally updates the mapping $\mathcal{M}$ of the virtual to physical servers. Specifically, to update the mapping, we first consider the servers $i$ for which $q_i$ has strictly decreased. For each such server $i$, we pop from the virtual-server stack a number of virtual servers corresponding to the $q_i$ decrease. We push all these popped orphan virtual servers into a temporary stack. Finally, we consider the servers $i$ for which $q_i$ has strictly increased. Using this temporary stack, we greedily allocate



| | $\mu_1 = 0.15$ | $\mu_2 = 0.23$ | $\mu_3 = 0.31$ | $\mu_4 = 0.31$ |
|---|---|---|---|---|
| $q_i$ | 0 | 0 | 0 | 0 |
| $\frac{q_i + 1}{\mu_i}$ | 6.6 | 4.3 | 3.2 min | 3.2 |

(a) First M3 iteration.



| | $\mu_1 = 0.15$ | $\mu_2 = 0.23$ | $\mu_3 = 0.31$ | $\mu_4 = 0.31$ |
|---|---|---|---|---|
| $q_i$ | 3 | 4 | 6 | 6 |
| $\frac{q_i + 1}{\mu_i}$ | 26.6 | 21.7 min | 22.5 | 22.5 |

(b) Last M3 iteration.

Fig. 4. Illustration of how M3 iteratively assigns $q = 20$ virtual servers to four physical servers, using the settings of Fig. 3(b). (a) In the first iteration, it picks the third server. (b) In the last iteration, it picks the second server.

each such server $i$ a number of virtual servers corresponding to the $q_i$ increase. Since $\sum q_i = q$ is constant, all virtual servers are exactly allocated in the mapping.

**Min-max acceleration.** M3 can run an improved greedy fairness sub-routine that substantially accelerates its run-time. For instance, in the example above where it needs to split $q = 20$ virtual servers between four servers with service rates $\bar{\mu} = (0.15, 0.23, 0.31, 0.31)$, it can reduce the 20 needed steps to a single one. To do so, it starts by pre-allocating $q_i = \lfloor \mu_i \cdot q \rfloor$ for each server $i$, obtaining $\lfloor \mu_1 \cdot q \rfloor = 3$ virtual servers to the first server, $\lfloor \mu_2 \cdot q \rfloor = 4$ to the second, and $\lfloor \mu_3 \cdot q \rfloor = 6$ for the third and the fourth, with a total of $3+4+6+6 = 19$ pre-allocated virtual servers, and thus only needs the last step of Fig. 4(b) to determine how to assign the 20th virtual server.

The intuition for this pre-allocation is that ideally, we would like to assign the real number $\mu_i \cdot q$ of virtual servers to each physical server to achieve the perfect balance. If we assign any lesser integer number, there is no risk of exceeding the min-max server load achieved by M3, and further iterations converge to the same final allocation. For space reasons, we do not include the full proof of the correctness of this accelerated sub-routine.

**M3 is quantized CH.** The following theorem states that M3 satisfies the quantized-CH definition (Def. 1):

**Theorem 1 (quantized CH).** *The M3 algorithm is quantized CH.*

*Proof Outline.* We need to prove that M3 satisfies both the minimal-disruption and maximal-stability properties (Def. 1). We proceed in three steps:

*Step 1: minimal-disruption.* We start by proving the minimal-disruption property. We first assume that any change in the cluster is atomic (i.e., consists of a change of at most one server, which can either join or depart the cluster), or can be represented by cascading atomic changes (i.e., adding three physical servers is equivalent to three consecutive atomic server additions). We then show that upon any atomic change, once we distribute the virtual servers according to the M3 algorithm, the following statements are invariant and always stay true:

- Upon adding a physical server and re-mapping the virtual servers across the new cluster, the number of virtual servers mapped to each of the old (existing) physical servers cannot increase. Intuitively, this is because their share of the total service rate cannot increase, so M3 will not assign them more virtual servers.
- Upon removing a physical server and re-mapping the virtual servers across the new cluster, the number of virtual servers mapped to each of the remaining physical servers cannot decrease. This follows the same intuition.

Then, we use induction to prove that the previous statements are true also for non-atomic changes using consecutive atomic changes. Finally, we conclude that in each case of adding (respectively removing) servers, the keys that need to be remapped to different physical servers are only the ones that are now (resp. were previously) mapped to the newly added (resp. removed) servers.

*Step 2: min-max fairness.* We then prove that M3 is min-max fair (Def. 2). We first consider the initial greedy approach. We mathematically prove that the M3 assignment of virtual servers is min-max fair after the first step. Then, using induction, we prove that each succeeding assignment is also min-max fair, including the $q^{th}$ (last) assignment. Finally, we show that server additions and deletions do not affect the min-max fairness property.

*Step 3: maximal-stability.* We complete the proof by showing that any algorithm that is min-max fair is also maximally stable (Def. 1). To do so, we show that the maximal load of a min-max fair algorithm cannot exceed that of any other algorithm. Therefore, if another algorithm is stable, i.e., its maximal load is strictly under 1, then a min-max fair algorithm is also necessarily stable, concluding the proof. □

In addition, the following theorem proves that in the homogeneous case, when $q$ is a multiple of $n$, M3 also satisfies the traditional homogeneous-CH definition.

**Theorem 2 (homogeneous CH).** *In the homogeneous case, when $q$ is a multiple of $n$, min-max fairness is equivalent to balance and M3 also achieves homogeneous CH.*

*Proof.* We show that for a homogeneous set, a min-max fair relative load vector ($\bar{\rho}_{fair}$) is always equal to the balanced relative load vector ($\bar{\rho}_{balanced}$). A balanced relative load vector is computed by mapping the same number of virtual servers to all physical servers:

$$\rho_{i,balanced} = \frac{\lambda_{i,balanced}}{\mu_i} = \frac{\frac{\left(\frac{q}{n}\right)}{q} \cdot \rho}{\mu_i} = \frac{\frac{\rho}{n}}{\frac{1}{n}} = \rho, \quad \forall i$$

Now we assume by contradiction that the load from the min-max fair mapping is not the same as that from the balanced mapping, i.e., $\bar{\rho}_{fair} \neq \bar{\rho}_{balanced}$. Hence $\exists i : \rho_{i,fair} \neq \rho_{i,balanced}$ and $\exists i : \lambda_{i,fair} \neq \lambda_{i,balanced}$. This inequality implies that there exists an index $j \in (1 \ldots n)$ s.t. $\lambda_{j,fair} > \lambda_{j,balanced}$ and $\rho_{j,fair} > \rho_{j,balanced} = \rho$. So the following holds:

$$\max \bar{\rho}_{fair} \geq \rho_{j,fair} > \rho_{j,balanced} = \rho = \max \bar{\rho}_{balanced}.$$

This stands in contradiction to the definition of min-max fairness. Hence the correctness of the theorem. □

**Complexity.** The initial pre-allocation step of the accelerated algorithm consists of $n$ assignments $q_i = \lfloor \mu_i \cdot q \rfloor$, each in $O(1)$ time. We then have

$$q - \sum_{i=1}^{n} \lfloor \mu_i \cdot q \rfloor = \sum_{i=1}^{n} (\mu_i \cdot q - \lfloor \mu_i \cdot q \rfloor) \leq \sum_{i=1}^{n} 1 = n$$

steps. At each step, as in Fig. 4, we determine an $i'$ with the minimum $\frac{q_i+1}{\mu_i}$ value, and then increase $q_{i'}$ by one. Thus, we can build a binary search tree in $O(n \log n)$ complexity, then use it to determine and update this $i'$ in $O(\log n)$ complexity at each step. Overall, the total run-time complexity is in $O(n \log n)$. The total space complexity needed for the mapping implementation is in $O(q)$.

## V. Stability guarantees

Throughout this section, we analyze the fundamental properties of the M3 algorithm. We start with our main result, which provides stability guarantees when we allow for *any* arbitrary heterogeneous service rates (Sec. V-A). We also look at the implications for (1) system loads that achieve system stability, (2) a variable number of servers and variable load, and (3) the overprovision factor (Sec. V-B). Finally, we focus on a single heterogeneous service rate vector

### *A. Main result: guaranteed stability for any service rates*

We start with the main fundamental result of this paper. We want the H&M architecture to be able to deal with any arbitrary heterogeneous service rate vector. For instance, we could envision that each server periodically measures and sends the exponentially-weighted average of its service rate to the load-balancer. The load-balancer will then run the M3 algorithm based on the new normalized service rate vector and on a pre-determined total number $q$ of virtual servers, and reassign virtual servers to each physical server.

The parameter $q$ can vary by several orders of magnitude. Intuitively, a larger $q$ provides a finer-grained quantization, but also hinders scalability. Today, there exists no stability guarantee to help the network designer when setting $q$. The following theorem provides a necessary and sufficient condition for M3 to be able to achieve stability given *any* arbitrary heterogeneous service rate vector in the H&M framework. The condition links the load $\rho$, the number of virtual servers $n$ and the number of physical servers $q$. The proof also shows that for any service vector, no other algorithm could do better than M3.

**Theorem 3 (stability for any vector).** *The M3 algorithm is guaranteed to be stable for any service rate vector $\bar{\mu}$ iff:*

$$q > (n-1) \cdot \frac{\rho}{1-\rho}. \tag{2}$$

**Example.** Consider again the four-server example of Fig. 3. Applying Thm. 3 to that specific case with $\rho = 0.8$ and $n = 4$, we get that the value of $q$ needed to maintain stability for any $\bar{\mu}$ would be $q > (n-1) \cdot \frac{\rho}{1-\rho} = 3 \cdot \frac{0.8}{1-0.8} = 12$, i.e., the system is guaranteed to be stable for any $\bar{\mu}$ *iff* $q \geq 13$.

Clearly, from Table II, we know that the system can be stable with a lower $q$ for this specific $\bar{\mu}$. However, Thm. 3 states that for $q \in \{6, 7, 8, 9, 11, 12\}$, even though the system is stable in this specific case, there are other $\bar{\mu}$s that provide counter-examples and will cause instability. There will not be a counter-example iff $q \geq 13$.

*Proof Outline.* We first show that the condition in Eq. (2) is sufficient for the M3 algorithm to reach stability, then establish that it is also necessary for stability.

*First direction: sufficiency.* We introduce a new algorithm, denoted as *baseline policy*, for which Eq. (2) suffices to always reach stability. The *baseline policy* essentially attemps to assign the maximal number $\hat{q}_i$ of virtual servers that can be mapped to a physical server without overloading it. Using simple arithmetic, we prove that $\hat{q}_i = \left\lceil \frac{\mu_i \cdot q}{\lambda} \right\rceil - 1$, i.e., a server $i$ is stable iff

$$q_i \leq \hat{q}_i \equiv \left\lceil \frac{\mu_i \cdot q}{\lambda} \right\rceil - 1. \tag{3}$$

In addition, we are careful to define $q_i$ such that the sum of all $q_i$'s never exceeds the total number $q$ of available virtual servers, i.e., $\sum_{j=1}^{i} q_j$ should never exceed $q$. Thus, the baseline policy's $q_i$ assignments are formally defined as follows:

$$\begin{cases} q_i = \min\left( \left\lceil \frac{\mu_i \cdot q}{\lambda} \right\rceil - 1, q - \sum_{j=1}^{i-1} q_j \right) & \text{if } i < n, \\ q_n = q - \sum_{j=1}^{n-1} q_j. \end{cases} \tag{4}$$

We show that the baseline policy is always stable whenever Eq. (2) is satisfied.

Since we already know that M3 is maximally stable (Thm. 1) and that the baseline policy stabilizes the system in H&M, then M3 necessarily stabilizes that system too.

*Second direction: necessity.* In order to prove that Eq. (2) is also necessary to achieve stability, we assume by contradiction that Eq. (2) does not hold:

$$q \leq (n-1) \cdot \frac{\lambda}{1-\lambda},$$

which we can rephrase by defining a non-negative $\varepsilon$:

$$\varepsilon \equiv (n-1) \cdot \lambda - q \cdot (1-\lambda) \geq 0.$$

We now provide a constructive counter-example in the form of an admissible system that cannot be stabilized given *any* number $\tilde{q}$ of virtual servers that satisfies $\tilde{q} \leq q$. First, we set the first $n-1$ servers to be identical:

$$\mu_i = \mu_0 = \frac{\lambda}{\tilde{q}} \cdot \left(1 - \frac{\varepsilon}{n-1}\right) \quad \forall i \in \{1 \dots n-1\},$$

and after assuming that each of the first $n-1$ servers are stable, we arithmetically show that the $n^{th}$ server cannot be stable. In such a case, no algorithm can stabilize the provided system under our initial assumption, including the M3 algorithm. □

### *B. Corollary Results*

We now turn to emphasizing different aspects resulting from Thm. 3, presenting first two direct corollaries, then a more general theorem on the overprovision factor.

**Maximum stable load.** We can rearrange the formula of Thm. 3 and provide a clearer condition on the maximum stable system load $\rho$, showing why $q$ needs to be quite larger than $n-1$ to reach a high stable load. For instance, a ratio of $\frac{q}{n-1} = 100$ will result in a stable load whenever $\rho < 1 - \frac{1}{101} \approx 0.99$.

**Corollary 1 (maximum stable load).** *The M3 algorithm is guaranteed to be stable for any service rate vector $\bar{\mu}$ iff:*

$$\rho < \frac{q}{q+n-1} = 1 - \frac{1}{\frac{q}{n-1}+1}$$

*Proof Outline.* By rearranging the terms of Eq. (2) in Thm. 3. □

**Variable parameters.** Since the H&M framework is used for quantized CH, it needs to be flexible to changes in the server set (Def. 1), therefore the number of servers $n$ is variable. In addition, in a real-world system, the load may be varying as well. The following corollary ensures that this has no impact on the validity of Thm. 3.

**Corollary 2 (variable $(n, \rho)$).** *Assume the system's parameters $(n, \rho)$ are allowed to vary arbitrarily within $[1, n_{max}] \times [0, \rho_{max}]$, with $\rho_{max} < 1$. The M3 algorithm is guaranteed to be stable for any $n, \rho$ and any service rate vector $\bar{\mu}$ iff*

$$q > (n_{max} - 1) \cdot \frac{\rho_{max}}{1 - \rho_{max}}.$$

*Proof.* We denote $n_s, \rho_s$ as the current system's parameters satisfying $n_s \in [1, n_{max}], \rho_s \in [0, \rho_{max}]$. Since the function $f(n, \rho) = (n-1) \cdot \frac{\rho}{1-\rho}$ is monotonic in both of its parameters,

$$q > (n_{max} - 1) \cdot \frac{\rho_{max}}{1 - \rho_{max}} \geq (n_s - 1) \cdot \frac{\rho_s}{1 - \rho_s}.$$

According to Thm. 3, the algorithm is thus guaranteed to be stable, proving the sufficiency. The necessity is directly proved by assigning $(n, \rho) = (n_{max}, \rho_{max})$. □

**Overprovision.** In the load-balancing industry, a metric for the quality of a load-balancer is its *overprovision* factor, defined as $\max_i \left(\frac{\rho_i}{\rho}\right)$ [6]. A perfectly balanced system achieves an overprovision of 1. The following theorem quantifies the additional degree of overprovision that is due to quantization (ignoring additional effects that are independent of our model, such as an uneven key distribution and a poor hash function). It shows again why $q$ needs to be quite larger than $n-1$. For instance, a ratio of 100 will result in a 1.01 guaranteed overprovision upper-bound.

**Theorem 4 (overprovision).** *M3 satisfies:*

$$\max_i \left(\frac{\rho_i}{\rho}\right) \leq 1 + \frac{n-1}{q}.$$

*Proof Outline.* In its iterative greedy search, M3 always assigns the next virtual server to the physical server that will be least loaded after the addition. We use the notation $q_i^k$ for the number of virtual servers mapped to physical server $i$ after iteration $k$ where $i \in \{1, \ldots, n\}$ and $k \in \{1, \ldots, q\}$. We also assume $q_i^0 = 0, \quad \forall i$.

Since the M3 algorithm assigns the virtual servers one-by-one using $q$ iterations, it is known by definition that physical server $i$ is selected at iteration $k$ if:

$$\frac{q_i^{k-1}+1}{\mu_i} = \min_{j \in \{1,\ldots,n\}} \frac{q_j^{k-1}+1}{\mu_j}, \tag{5}$$

and after iteration $k$, we have:

$$q_i^k = \begin{cases} q_i^{k-1}+1, & \text{server } i \text{ was selected at iteration } k \\ q_i^{k-1}, & else. \end{cases} \tag{6}$$

By definition:

$$\sum_i q_i^k = k. \tag{7}$$

Let's denote a sequence of indices $i_{1,\ldots,q}$ s.t. $i_k$ is the physical server selected at iteration $k$. Now we use the fact $\sum_i \mu_i = 1$, together with Eq. (5), Eq. (6) and Eq. (7), to obtain:

$$\begin{aligned} \frac{q_{i_k}^k}{\mu_i} = \frac{q_{i_k}^{k-1}+1}{\mu_i} &\leq \sum_j \mu_j \cdot \left(\frac{q_j^{k-1}+1}{\mu_j}\right) \\ &= \sum_j \left(q_j^{k-1}+1\right) \\ &= n + \sum_j q_j^{k-1} = n+k-1. \end{aligned} \tag{8}$$

We assign $k = q$ in Eq. (8) to get:

$$\frac{q_{i_q}^q}{\mu_{i_q}} \leq q+n-1. \tag{9}$$

In addition, we prove by induction the interesting result that the latest assigned server $i_k$ has the *highest* load amongst all servers:

$$i_k \in \operatorname*{argmax}_j \left\{\frac{q_j^k}{\mu_j}\right\}$$

Combining this result to Eq. (9), the following result also holds:

$$\max_i \rho_i = \frac{\lambda_{i_q}}{\mu_{i_q}} = \frac{\frac{q_{i_q}^q}{q} \cdot \rho}{\mu_{i_q}} \leq \rho \cdot \left(\frac{q+n-1}{q}\right),$$

yielding the result. □

## C. Guaranteed stability for a given service rate vector

After considering sets of possible heterogeneous service rate vectors, we now turn to characterizing stability when tackling a single given service rates vector $\bar{\mu}$. In such a case, as illustrated in Table II, there is no critical $q$ value below which the system is unstable and above which it is stable. Instead, given some vector $\bar{\mu}$, we provide a fixed-point equation that characterizes a stable $q$.

**Theorem 5.** *Given an arbitrary heterogeneous service rates vector $\bar{\mu}$, the M3 algorithm is stable iff $q$ satisfies the following fixed-point equation:*

$$q \leq \sum_{i=1}^n \hat{q}_i \equiv \sum_{i=1}^n \left(\left\lceil \frac{\mu_i \cdot q}{\rho} \right\rceil - 1\right). \tag{10}$$

*Proof. First direction: sufficiency.* Assume Eq. (10) holds. We first choose a mapping policy that maintains the following property, and conclude that this mapping policy is stable:

$$q_i \leq \hat{q}_i = \left\lceil \frac{\mu_i \cdot q}{\rho} \right\rceil - 1, \quad \forall i \in \{1, \ldots, n\}. \tag{11}$$

Specifically, we use the baseline policy from the proof of Thm. 3. By definition of the baseline policy (Eq. (4)), servers $\{1 \ldots n-1\}$ always satisfy Eq. (11). As for the $n^{th}$ server, we divide into two cases:

1) The case where all of the $n-1$ first servers were assigned a number $q_i = \hat{q}_i$ of virtual servers, i.e., Eq. (11) is always an equality. Then
$$\sum_{i=1}^{n-1} q_i = \sum_{i=1}^{n-1} \left( \left\lceil \frac{\mu_i \cdot q}{\rho} \right\rceil - 1 \right).$$
In this case, we know that $q_n = q - \sum_{i=1}^{n-1} \left( \left\lceil \frac{\mu_i \cdot q}{\rho} \right\rceil - 1 \right)$ by definition of the baseline policy. Therefore, by Eq. (10),
$$q_n \leq \left\lceil \frac{\mu_n \cdot q}{\rho} \right\rceil - 1 = \hat{q}_n.$$
2) Eq. (11) is sometimes a strict inequality. Let $j$ denote the smallest index for which it is not an equality. In this case, by definition of the baseline policy (Eq. (4)), all servers $\tilde{k} > j$ are assigned exactly zero virtual servers, including the $n^{th}$ server, thus the $n^{th}$ server also satisfies Eq. (10).

Therefore the baseline policy is stable whenever Eq. (10) is satisfied. Since M3 is maximally stable (as showed in the proof of Thm. 1), we deduce that M3 is also stable.

*Second direction: necessity.* Assume $q > \sum_{i=1}^{n} \hat{q}_i$. By definition, any mapping policy must retain the property $\sum_{i=1}^{n} q_i = q$. Therefore there must exist an $i \in \{1, \ldots, n\}$ such that:
$$q_i > \hat{q}_i.$$
In this case, according to the previous results establishing that $\hat{q}_i$ is a strict threshold for stability, server $i$ is unstable. Hence, the whole system is unstable. □

## VI. Evaluations

The following evaluations help us confirm the soundness of the theoretical results, as well as compare the performance of the M3 algorithm against practical heterogeneous algorithms from the literature and industry.

### A. Maximal load measured across the servers

**Settings.** Thm. 3 states that for *any* service rate vector, the system is stable, i.e., the maximal server load satisfies $\max_i \rho_i < 1$, whenever the number $q$ of virtual servers satisfies $q > (n-1) \cdot \frac{\rho}{1-\rho}$. We want to confirm the soundness of the result. To do so, we plot $\max_i \rho_i$ as a function of $q$ for a large number of instances of random service rate vectors. Specifically, we set $n = 3$ and $\rho = 0.95$. We draw 50 random heterogeneous service rate vectors by drawing each non-normalized service-rate component $\tilde{\mu_i}$ u.a.r. on integer set $[1, 100]$.

**Thm. 3 (main result).** Fig. 5 shows the evaluation results. The left scale illustrates Thm. 3. As established by Thm. 3, when $q > (3-1) \cdot \frac{0.95}{1-0.95} = 38$, M3 is always stable, i.e., $\max_i \rho_i < 1$. We can also see that the typical performance

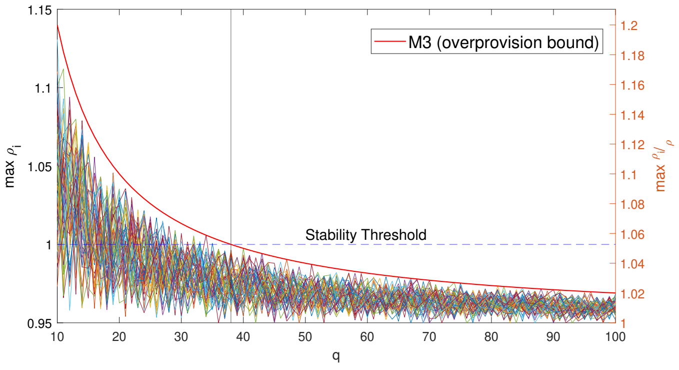


Fig. 5. Plots of the maximal load $\max_i \rho_i$ (left scale) and overprovision factor $\frac{\max_i \rho_i}{\rho}$ (right scale), as a function of $q$ in 50 random heterogeneous systems, with $\rho = 0.95$ and $n = 3$. As established by Thm. 4, the red line is an upper bound.

improves as $q$ grows, although each of the 30 lines does not decrease monotonically. For a smaller $q$, some rate vectors encounter instability, with $\max_i \rho_i \geq 1$. Thm. 3 tells us that there is actually a counter-example with $q = 38$, but it is not among our random instances.

**Thm. 4 (overprovision).** In addition, the right scale of Fig. 5 illustrates a validation of Thm. 4. Its formula follows the red line, which provides an upper-bound on the overprovision factor $\frac{\max_i \rho_i}{\rho}$. For instance, for $q = 100$, the overprovision factor is 1.02.

### B. Maximal stable load measured at the load-balancer

**Fig. 3 baseline example.** We evaluate the performance of the algorithm in terms of the maximal stable load that can be asserted upon the load-balancer. We use the settings of Fig. 3 as a baseline for comparison, with $\rho = 0.8$, and observe the impact of $q$. The blue line represents the maximal stable load at the load-balancer as measured numerically for each $q$ value (corresponding to the fixed-point condition of Thm. 5). In particular, the left side illustrates the results of Table II. For instance, we can see that for $q \in [6, 9]$, the maximal stable $\rho$ is above 0.8, while for $q = 10$, $\rho = 0.8$ would be unstable.

We also compare the results to the result of Thm. 3, re-written as $\rho < \frac{q}{q+n-1}$ stating that any load lower than the right side of the inequality would keep any system stable. The right side of the inequality for the baseline vector (4-server system) is represented by the orange plot, and it is a lower bound on the blue plot, as it guarantees stability for *any* vector, and in particular for this baseline vector. For instance, beyond the dotted vertical line at $q = 13$, M3 is guaranteed to provide stability when $\rho = 0.8$ (Thm. 3). Indeed, the blue plot does not cross below the value of $\rho = 0.8$ (horizontal dotted line) when $q \geq 12$ (vertical dotted line).

### C. Performance against other algorithms: distributed storage

**Settings.** We now compare the performance of M3 against two existing algorithms, both based on the weighted Ring: NGINX's Ketama library [22] and ACH [2]. As mentioned in Table I, there is no other practical heterogeneous CH algorithm

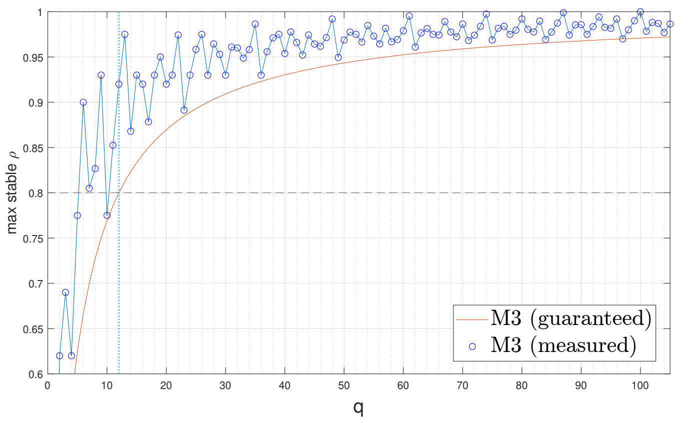
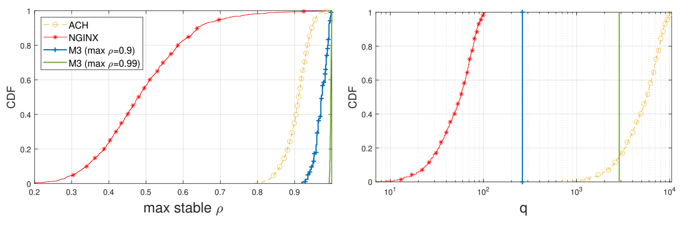


Fig. 6. Measured maximal stable load $\rho$ (blue line, Thm. 5) as a function of the number of virtual servers $q$ given the service rate vector of Fig. 3, vs. the lower bound (orange line, Thm. 3) that is guaranteed by M3 for *any* service rate vector.



(a) CDF of max stable $\rho$ (b) CDF of $q$

Fig. 7. Distributed-storage evaluation with up to $n = 30$ heterogeneous storage servers: Empirical CDF of (a) maximal stable $\rho$ load values and (b) their corresponding number $q$ of virtual servers, comparing ACH, NGINX's Ketama, and M3 for $\rho < 0.9$ and $\rho < 0.99$.

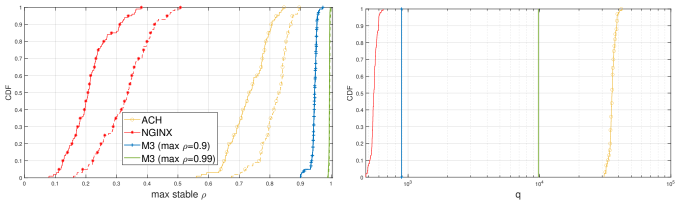
to compare against. There is also no available benchmark for heterogeneous CH with a variable number of servers, so we base our evaluations settings on those from previous papers. We start with the distributed-storage setting from [2], which consists of 15 weak storage servers (weight of 2, Intel SSD) and 15 strong ones (weight of 5, Seagate SATA HDD). We run 1,000 simulations, and make one change: Before each simulation, we independently draw the numbers of strong and weak servers u.a.r. in the integer set $[1, 15]$. For Ketama, we assume that weights also indicate the number of virtual servers, since there is no other indication to the user of how to set this number. For ACH, we use $c_{throttle} = 0.01$ as in the paper [2], resp. yielding $2/0.01 = 200$ and $5/0.01 = 500$ virtual servers per weak and strong storage server. For our M3 algorithm, we use Thm. 3 with stability guarantees for either all $\rho < 0.9$ or all $\rho < 0.99$. In all simulations, the maximal stable $\rho$ values were measured by brute-force sweep with resolution of $10^{-5}$. For NGINX and ACH, we rely on the crc32 [40] hash function for the placement of virtual servers on the ring. Hash collisions are treated as errors and trigger the generation of new random virtual-server identifiers. For all algorithms, we follow the SUHA assumption for request keys (Sec. II), which means that the hash results on the keys are distributed among the hash space in a perfect uniform way (or in other words, we assume an infinite number of requests).

**Results.** Fig. 7(a) and Fig. 7(b) respectively illustrate the CDF of the measured maximal stable $\rho$ and of the number $q$ of virtual servers. This empirical CDF is taken over the 1,000 simulations for each of the algorithms. We also focus on the performance tail at the 1st-percentile for the maximal stable $\rho$ and the 99th-percentile for $q$. We can see that NGINX obtains poor results, with the 1st-percentile stable $\rho$ at 0.24. But it uses a low number of virtual servers in Fig. 7(b), with a 99th-percentile at 100. ACH obtains better stability results, with a 1st-percentile stable $\rho$ at 0.821, but it uses a disproportionate number of virtual servers, with a 99th-percentile at 10,300. M3 obtains better stability results. In the blue plot, M3 guarantees stability for any $\rho < 0.9$, and achieves a 1st-percentile stable $\rho$ at 0.926, with $(30-1)\cdot\frac{0.9}{1-0.9}+1 = 262$ virtual servers. Finally, in the green plot, M3 guarantees stability for any $\rho < 0.99$, with a high number of $(30 - 1) \cdot \frac{0.99}{1-0.99} + 1 = 2872$ virtual servers.

## D. Performance against other algorithms: load-balancer

**Settings.** We now compare the performance of the same algorithms in a load-balancing framework in two modes: SUHA and real trace. First, the SUHA mode is implemented as in the previous evaluation (Fig. 7) and its results are shown in Fig. 8(b), using the same colors and patterns but in dotted lines. Second, we also use real traces of requests recorded in a Facebook datacenter [41]. Contrarily to the SUHA mode, in the real trace mode we simulate a finite number of requests (154M unique 4-tuples). In both modes, the heterogeneous CH load-balancer spreads traffic among $n = 100$ servers. For each server in each simulation, we assume that its service rate is proportional to an integer weight drawn u.a.r. in [1,10]. As in the previous evaluation, the maximal stable load represents a worst-case value of load at the load balancer for which at least one of the servers experiences instability event. This evaluation involves a higher number of physical servers and hence the worst-case scenario is expected to be even worse. We use the FNV-1a hash function for the NGINX, ACH and M3 variants (results were similar for the crc32 hash function). We run 100 simulations. As previously, for Ketama, we assume that weights also indicate the number of virtual servers. For ACH, we use $c_{throttle} = \frac{10\cdot 100}{65{,}537} \approx \frac{1}{65}$, so that there are enough virtual servers in the worst case, as indicated in the ACH paper [2]. All other assumptions stay the same.

**Results.** Fig. 8(a) and Fig. 8(b) respectively illustrate the CDF of the measured maximal stable $\rho$ and of the number $q$ of virtual servers. Both NGINX and ACH obtain poor stability results, in each of the two simulation modes. The real trace mode degrades the performance of all algorithms but hurts ACH's and NGINX's performance the most. The cause for the sub-optimal performance in the real trace mode compared to the SUHA mode is the non-optimal (finite) key distribution of any hash function (in this case FNV-1a), which distributes the keys among the hash space in approximately uniform way, but not exactly uniform for any number of finite keys. Hence,



(a) CDF of max stable $\rho$ (b) CDF of $q$

Fig. 8. Load-balancer evaluation with $n = 100$ heterogeneous servers: Empirical CDF of (a) maximal stable $\rho$ load values and (b) their corresponding number $q$ of virtual servers, comparing ACH, NGINX's Ketama, and M3 for both $\rho < 0.9$ and $\rho < 0.99$. The dotted lines represent the maximal stable $\rho$ when relying on the SUHA assumption, while the solid lines represent simulations that were run using real traces from Facebook's datacenter.

the real trace mode provides a more realistic setting, while we would expect the trace results to converge to the SUHA results for an infinite trace. It can be clearly seen from the figures that the Ring-based algorithms are more sensitive to this sub-optimal phenomenon. NGINX performs with a 1st-percentile stable $\rho$ at 0.08 in real trace mode and at 0.16 in SUHA mode. ACH performs with 1st-percentile stable $\rho$ at 0.56 in real trace mode and at 0.67 in SUHA mode. As in the previous simulation, we can observe the implication on the max stable $\rho$ caused by the inherent imbalance of the weighted Ring. Moreover, simulating the results using real traces and real hash functions degrades the performance of the weighted ring algorithms substantially, while the M3 variants were affected only slightly. M3 with guaranteed stability for $\rho < 0.9$ (in blue) achieves a 1st-percentile stable $\rho$ at approx. 0.9 in both modes, while M3 with guaranteed stability for $\rho < 0.99$ (in green) achieves a 1st-percentile stable $\rho$ at 0.99 in both modes, but of course with a larger $q$. It is worth noting that although the ACH algorithm makes extensive use of virtual servers (≈3.5x more than the M3 stringent variant), the ACH algorithm cannot achieve similar results to any of the M3 variants. A notable observation arising from this evaluation is that whenever naively using the common industry standard for heterogeneous load balancing (NGINX), instability events are very likely to occur if the system load is higher than 20% of its capacity. However, according to the same evaluation, adapting a scalable M3 variant in such cases is expected to prevent instability events completely (100% throughput) whenever the load is less than 90% of its capacity.

### E. Key lessons from the evaluation results

Beyond the compliance of the evaluation results with the paper's theoretical results, three key lessons emerge from the evaluations.

First, the weighted Ring algorithm presents surprisingly weak stability results, especially when using smaller memory resources.

Second, these results are even weaker in real-world finite traces, in which the average load at each given server is much more variable, especially when compared to often-used theoretical assumptions with implied infinite streams of packets (as seen in the solid vs. dotted lines in Fig. 8(a)).

Finally, the M3 algorithm can achieve a strong guaranteed stability result even with limited memory resources.

## VII. Conclusion & Discussion

In the paper, we provided a fundamental analysis of heterogeneous consistent hashing, a key component of current load-balancers. We introduced the H&M framework for quantized heterogeneous CH. Within this framework, we also presented the M3 algorithm for providing min-max fairness. We then demonstrated necessary and sufficient conditions for guaranteeing stability over the set of all arbitrary normalized heterogeneous service rates. Finally, the evaluations illustrate how existing solutions suffer from significant stability issues, while our suggested practical M3 algorithm reaches strong stability performance together with proved stability guarantees.

At the same time, it is worth bearing in mind the following limitations of M3:

**CH limitations.** M3 shares some intrinsic limitations of existing CH algorithms (Table I). It is vulnerable to long-tailed flows that send many packets to the same server, and long-tailed packets that may require significantly more processing than other packets. This is because it is oblivious to the server queue sizes. Solving this could involve sometimes sending new flows to other servers than the CH algorithm calls for, together with a connection tracking mechanism that keeps track of these changes [37].

**Quantization limitations.** Given a finite number $q$ of virtual servers, Cor. 1 establishes that M3 cannot fully reach 100% throughput for all service rate vectors, even though it can get close to it.

## Acknowledgment

We would like to thank Roy Mitrani, Gal Mendelson, Tsahi Birk, our shepherd Deep Medhi, and our anonymous reviewers for their valuable comments. This work was partly supported by the Israel Science Foundation (grant No. 1119/19), the Hasso Plattner Institute Research School, and the Louis and Miriam Benjamin Chair in Computer-Communication Networks.

## References

[1] G. Mendelson, S. Vargaftik, K. Barabash, D. H. Lorenz, I. Keslassy, and A. Orda, "AnchorHash: A scalable consistent hash," *IEEE/ACM Transactions on Networking*, vol. 29, no. 2, pp. 517–528, 2020.

[2] J. Zhou, Y. Chen, and W. Wang, "Attributed consistent hashing for heterogeneous storage systems," in *ACM PACT*, 2018, pp. 1–12.

[3] J. Zhou, W. Xie, Q. Gu, and Y. Chen, "Hierarchical consistent hashing for heterogeneous object-based storage," in *IEEE Trustcom*, 2016, pp. 1597–1604.

[4] M. Sackman, "Perfect consistent hashing," *arXiv preprint arXiv:1503.04988*, 2015.

[5] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu *et al.*, "Ananta: Cloud scale load balancing," *ACM SIGCOMM*, vol. 43, no. 4, pp. 207–218, 2013.

[6] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: a fast and reliable software network load balancer," in *Usenix NSDI*, 2016, pp. 523–535.

[7] NGINX, "Nginx http load balancing documentation," https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer, 2023.

[8] HAProxy, "Haproxy documentation: hash-type parameter," https://cbonte.github.io/haproxy-dconv/2.1/configuration.html#hash-type, 2023.

[9] P. Goel, K. Rishabh, and V. Varma, "An alternate load distribution scheme in DHTs," in *IEEE CloudCom*, 2017.

[10] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Transactions on Networking*, 2003.

[11] D. Halperin, V. Teixeira de Almeida, L. L. Choo *et al.*, "Demonstration of the Myria big data management service," in *ACM SIGMOD*, 2014.

[12] R. Fonseca, S. Ratnasamy, J. Zhao, C. T. Ee, D. Culler, S. Shenker, and I. Stoica, "Beacon vector routing: Scalable point-to-point routing in wireless sensornets," in *Usenix NSDI*, 2005.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[14] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *IPTPS*, 2002.

[15] J. T. Araujo, L. Saino, L. Buytenhek, and R. Landa, "Balancing on the edge: Transport affinity without network state," in *Usenix NSDI*, 2018.

[16] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *ACM STOC*, 1997, pp. 654–663.

[17] D. G. Thaler and C. V. Ravishankar, "Using name-based mappings to increase hit rates," *IEEE/ACM Transactions on Networking*, vol. 6, no. 1, pp. 1–14, 1998.

[18] V. Mirrokni, M. Thorup, and M. Zadimoghaddam, "Consistent hashing with bounded loads," in *ACM-SIAM SODA*, 2018, pp. 587–604.

[19] Y. Nakatani, "Structured allocation-based consistent hashing with improved balancing for cloud infrastructure," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 9, pp. 2248–2261, 2021.

[20] J. Zhou, L. Su, W. Wang, and Y. Chen, "Hashing based data distribution in heterogeneous storage," in *IEEE ICPADS*, 2021, pp. 652–659.

[21] therb1 (username), "Does nginx respect the weight attribute with consistent hashing?" https://stackoverflow.com/questions/42003787/does-nginx-respect-the-weight-attribute-with-consistent-hashing, 2018.

[22] Richard Jones, "Ketama consistent hashing project," https://github.com/RJ/ketama, 2014.

[23] R. Jones, "libketama: Consistent Hashing library for memcached clients," https://www.metabrew.com/article/libketama-consistent-hashing-algo-memcached-clients, 2007.

[24] C. Dong and F. Wang, "Dxhash: A scalable consistent hash based on the pseudo-random sequence," *arXiv preprint arXiv:2107.07930*, 2021.

[25] P. Berenbrink, A. Brinkmann, T. Friedetzky, and L. Nagel, "Balls into non-uniform bins," *Journal of Parallel and Distributed Computing*, vol. 74, no. 2, pp. 2065–2076, 2014.

[26] S. Vargaftik, I. Keslassy, and A. Orda, "LSQ: Load balancing in large-scale heterogeneous systems with multiple dispatchers," *IEEE/ACM Transactions on Networking*, vol. 28, no. 3, pp. 1186–1198, 2020.

[27] R. Atar, I. Keslassy, G. Mendelson, A. Orda, and S. Vargaftik, "Persistent-idle load-distribution," *Stochastic Systems*, vol. 10, no. 2, pp. 152–169, 2020.

[28] K. Gardner, J. Abdul Jaleel, A. Wickeham, and S. Doroudi, "Scalable load balancing in the presence of heterogeneous servers," *ACM SIGMETRICS Performance Evaluation Review*, vol. 48, no. 3, pp. 37–38, 2021.

[29] X. Zhou, N. Shroff, and A. Wierman, "Asymptotically optimal load balancing in large-scale heterogeneous systems with multiple dispatchers," *ACM SIGMETRICS Performance Evaluation Review*, vol. 48, no. 3, pp. 57–58, 2021.

[30] G. Goren, S. Vargaftik, and Y. Moses, "Stochastic coordination in heterogeneous load balancing systems," in *ACM PODC*, 2021, pp. 403–414.

[31] R. Atar, I. Keslassy, G. Mendelson, A. Orda, and S. Vargaftik, "On the persistent-idle load distribution policy under batch arrivals and random service capacity," *arXiv preprint arXiv:2103.12140*, 2021.

[32] D. Gamarnik, J. N. Tsitsiklis, and M. Zubeldia, "Stability, memory, and messaging trade-offs in heterogeneous service systems," *Mathematics of Operations Research*, vol. 47, no. 3, pp. 1862–1874, 2022.

[33] J. Abdul Jaleel, S. Doroudi, K. Gardner, and A. Wickeham, "A general "power-of-d" dispatching framework for heterogeneous systems," *Queueing Systems*, vol. 102, no. 3-4, pp. 431–480, 2022.

[34] S. Bhambay and A. Mukhopadhyay, "Asymptotic optimality of speed-aware JSQ for heterogeneous service systems," *Performance Evaluation*, vol. 157, p. 102320, 2022.

[35] S. Borst, "Load balancing in large-scale heterogeneous systems," *Queueing Systems*, vol. 100, no. 3-4, pp. 397–399, 2022.

[36] Z. Zhao, D. Mukherjee, and R. Wu, "Exploiting data locality to improve performance of heterogeneous server clusters," *arXiv preprint arXiv:2211.16416*, 2022.

[37] G. Mendelson, S. Vargaftik, D. H. Lorenz, K. Barabash, I. Keslassy, and A. Orda, "Load balancing with JET: just enough tracking for connection consistency," in *ACM CoNEXT*, 2021, pp. 191–204.

[38] (2023) M3 code. [Online]. Available: https://github.com/YoavLevi/M3-algorithm-implementation

[39] W. Collins, *Data structures and the Java collections framework*. McGraw-Hill, 2004.

[40] R. Singhal, "Inside intel core microarchitecture (nehalem)," in *IEEE HotChips*, 2008.

[41] C. Avin and S. Schmid, "DC Traces - Trace collection (fb_clustera_full.csv)," https://trace-collection.net/dc-traces/, 2023.