LSQ: Load Balancing in Large-Scale Heterogeneous Systems with Multiple Dispatchers

Shay Vargaftik, Isaac Keslassy, and Ariel Orda

***Abstract*—Nowadays, the efficiency and even the feasibility of traditional load-balancing policies are challenged by the rapid growth of cloud infrastructure and the increasing levels of server heterogeneity. In such heterogeneous systems with many load-balancers, traditional solutions, such as JSQ, incur a prohibitively large communication overhead and detrimental incast effects due to herd behavior. Alternative low-communication policies, such as $JSQ(d)$ and the recently proposed JIQ, are either unstable or provide poor performance.**

We introduce the *Local Shortest Queue* (LSQ) family of load balancing algorithms. In these algorithms, each dispatcher maintains its own, local, and possibly outdated view of the server queue lengths, and keeps using JSQ on its local view. A small communication overhead is used infrequently to update this local view. We formally prove that as long as the error in these local estimates of the server queue lengths is bounded *in expectation*, the entire system is strongly stable. Finally, in simulations, we show how simple and stable LSQ policies exhibit appealing performance and significantly outperform existing low-communication policies, while using an equivalent communication budget. In particular, our simple policies often outperform even JSQ due to their reduction of herd behavior. We further show how, by relying on smart servers (*i.e.,* advanced pull-based communication), we can further improve performance and lower communication overhead.

***Index Terms*—Local Shortest Queue, Load Balancing, Heterogeneous Systems, Multiple Dispatchers.**

I. Introduction

Background. In recent years, due to the rapidly increasing size of cloud services and applications [6], [10], [16], [26], the design of load balancing algorithms for parallel server systems has become extremely challenging. The goal of these algorithms is to efficiently load-balance incoming jobs to a large number of servers, even though these servers display large *heterogeneity* for two reasons. First, current large-scale systems increasingly contain, in addition to multiple generations of CPUs (central processing units) [15], various types of accelerated devices such as GPUs (graphics processing units), FPGAs (field-programmable gate arrays) and ASICs (application-specific integrated circuit), with significantly higher processing speeds. Second, VMs (virtual machines) or containers are commonly used to deploy different services that may share resources on the same servers, potentially leading to significant and unpredictable heterogeneity [13], [17].

In a traditional server farm, a centralized load-balancer (dispatcher) can rely on a full-state-information policy with strong theoretical guarantees for heterogeneous servers, such as join-the-shortest-queue (JSQ), which routes emerging jobs to the server with the shortest queue [7], [8], [16], [36], [37]. This is because in such single-centralized-dispatcher scenarios, the dispatcher forms a single access point to the servers. Therefore, by merely receiving a notification from each server upon the completion of each job, it can track all queue lengths, because it knows the exact arrival and departure patterns of each queue (neglecting propagation times) [18]. The communication overhead between the servers and the dispatcher is at most a single message per job, which is appealing and does not increase with the number of servers. Note that unless Direct Server Return (DSR) is employed, there is not even a need for this additional message per job, since all job responses return through the single dispatcher anyway.

However, in current clouds, which keep growing in size and thus have to rely on multiple dispatchers [12], implementing a policy like JSQ may incur two main problems. (1) It involves a prohibitive implementation and communication overhead as the number m of dispatchers increases [18]; this is because each server needs to keep all m dispatchers updated as jobs arrive and complete, leading to $O(m)$ communication messages per job (and this still holds even when DSR is not employed, since any reply only transits through a single dispatcher). (2) Also, it may suffer from incast issues when all/many dispatchers send at once all incoming traffic to the currently-shortest queue. These two problems force cloud dispatchers to rely on policies that do not provide any service guarantees with multiple dispatchers and heterogeneous servers [20], [25]. For instance, two widely-used open-source load balancers, namely HAProxy and NGINX, have recently introduced the "power of two choices" ($JSQ(2)$) policy into their L7 load-balancing algorithms [27], [32].[1]

Related work. Despite their increasing importance, scalable policies for heterogeneous systems with multiple dispatchers have received little attention in the literature. In fact, as we later discuss, the only suggested scalable policies that address the many-dispatcher scenario in a heterogeneous setting are based on join-the-idle-queue (JIQ) schemes, and none of them is stable [39].

Shay Vargaftik is with VMware Research, E-mail: shayv@vmware.com. Isaac Keslassy and Ariel Orda are with the Technion, E-mail: {isaac@ee, ariel@ee}.technion.ac.il.

[1]Quoting [27] (by Owen Garrett, Head of Products at NGINX): "*Classic load-balancing methods such as Least Connections [JSQ] work very well when you operate a single active load balancer which maintains a complete view of the state of the load-balanced nodes. The "power of two choices" approach is not as effective on a single load balancer, but it deftly avoids the bad-case "herd behavior" that can occur when you scale out to a number of independent load balancers. This scenario is not just observed when you scale out in high-performance environments; it's also observed in containerized environments where multiple proxies each load balance traffic to the same set of service instances.*"

In the $JSQ(d)$ (power-of-choice) policy, to make a routing decision, a dispatcher samples $d \geq 2$ queues uniformly at random and chooses the shortest among them [4], [5], [11], [19], [23], [38]. $JSQ(d)$ is stable in systems with homogeneous servers. However, with heterogeneous servers, $JSQ(d)$ leads to poor performance and even to instability, both with a single as well as with multiple dispatchers [9].

In the $JSQ(d, m)$ (power-of-memory) policy, the dispatcher samples the m shortest queues from the previous decision in addition to $d \geq m \geq 1$ new queues chosen uniformly-at-random [22], [29]. The job is then routed to the shortest among these $d + m$ queues. $JSQ(d, m)$ has been shown to be stable in the case of a single dispatcher when $d = m = 1$, even with heterogeneous servers. However, it offers poor performance in terms of job completion time, and it has not been studied in the multiple-dispatcher realm, thus has no theoretical guarantees.

A recent study [39] proposes a class of policies that are both throughput optimal and heavy-traffic delay optimal. However, their assumptions are not aligned with our system model and motivation due to several reasons: (1) For heterogeneous servers, [39] requires the knowledge of the server service rates, which may not be achievable in practice. (2) [39] assumes that the number of jobs that a server may complete in a time slot, as well as the number of jobs that may arrive at a dispatcher in a time slot, are deterministically upper-bounded, which rules out important modeling options with unbounded support, such as geometric services or Poisson arrivals. (3) Most importantly, they consider only a single dispatcher, and it is unclear whether their analysis and performance guarantees can be extended to multiple dispatchers.

In addition, to address the communication overhead in systems with multiple dispatchers, the JIQ policy has been proposed [18], [21], [30], [31], [33]. Roughly speaking, in JIQ, each dispatcher routes jobs to an idle server, if it is aware of any, and to a random server otherwise. Servers may only notify dispatchers when they become idle. JIQ achieves low communication overhead of at most a single message per job, irrespective of the number of dispatchers, and good performance at low and moderate loads when servers are homogeneous [18]. However, for heterogeneous servers, JIQ is not stable, *i.e.,* it fails to achieve 100% throughput [39].

Finally, two recent studies on low-communication load balancing [3], [34] propose to use local memory as well to hold the possibly-outdated server states. As we later show, these policies are, in fact, special cases of LSQ. That is, they only consider a single dispatcher and homogeneous servers while we consider multiple dispatchers and heterogeneous servers, which hold the main motivation and contribution of our work. Moreover, there are additional significant differences between our model assumptions and theirs that affect the analysis. For example, they consider a continuous-time model with Poisson arrivals and exponential service rates in which incast is impossible, whereas our model is in discrete-time and we only assume the existence of a first and a second moment of the processes. In particular, we do not assume any specific distribution of the arrivals or service rates by the servers. Also, they analyze their algorithms in a large-system limit, whereas our analysis deals with a finite number of servers and dispatchers.

Contributions. This paper makes the following contributions:

Local Shortest Queue (LSQ). We introduce LSQ, a new family of load balancing algorithms for large-scale heterogeneous systems with multiple dispatchers. As Figure 1 illustrates, in LSQ, each dispatcher keeps a local view of the server queue lengths and routes jobs to the shortest among them. Communication overhead among the servers and the dispatchers is used only to update the local views and make sure they are not too far from the real server queue lengths.

Sufficient stability condition and stability proof. We prove that all LSQ policies that keep a *bounded distance in expectation* between the real queue lengths and the local views are strongly stable, *i.e.,* keep bounded expected queue lengths. The main difficulty in the proof arises from the fact that the decisions taken by an LSQ policy depend on the *local view of each dispatcher*, hence on a potentially long history of system states. To address this challenge, we introduce two additional stable policies into our analysis: (1) JSQ and (2) Weighted-Random (WR). Roughly speaking, we show that our policy is sufficiently similar to JSQ which, in turn, is better than WR. We complete the proof by using the fact that, unlike JSQ, WR takes routing decisions that do not depend on the system state.

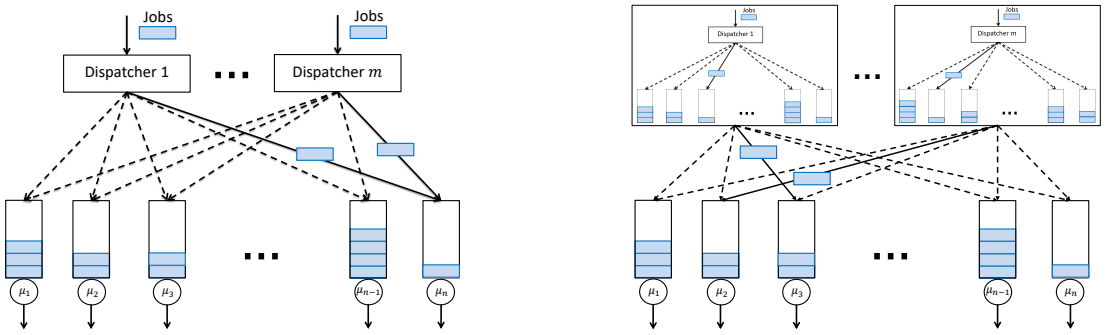
Simplified stability conditions. It can be challenging to prove that an LSQ policy is stable, *i.e.,* that in expectation, the local dispatcher views are not too far from the real queue lengths. Therefore, we develop simpler sufficiency conditions to prove that an LSQ policy is stable and exemplify their use.

Stable LSQ policies. Since LSQ is not restricted to work with either push– (*i.e.,* dispatchers sample the servers) or pull– (*i.e.,* servers update the dispatchers) based communication, we aim to achieve the same communication overhead as the lowest-overhead/best-known examples in each class. Accordingly, we show how to construct new stable LSQ policies with communication patterns similar to those of other low-communication policies such as the push-based $JSQ(2)$, but with significantly stronger theoretical guarantees.

Simulations. Using simulations we show how simple and stable LSQ policies present appealing performance, and significantly outperform other low-communication policies using an equivalent communication budget. Our simple policies often outperform even JSQ. This is achieved by sending jobs to less loaded servers but also by reducing herd behavior when compared to JSQ, as different dispatchers have different views.

Smart servers. We show how relying on smart servers (*i.e.,* advanced pull-based communication) allows us to improve performance and communication overhead even further and to *consistently* outperform JSQ in terms of both mean queues lengths and job completion time delay tail distribution. We rely on two main elements to achieve this: (1) fine-tuning the probabilities at which servers send messages, such that less loaded servers send messages with a higher probability; (2) when a message is sent by a server, it is sent to the dispatcher with the worst local view of this server.

Simulation code. To benefit the research community, we made our evaluation code available online [1].



(a) The JSQ approach.

(b) The LSQ approach.

Fig. 1: JSQ vs. LSQ. **(a)** JSQ requires instant knowledge of all server queues by all dispatchers, resulting in substantial communication overhead and possible bad-case "herd-behaviour" leading to incast issues. **(b)** At each dispatcher, LSQ relies on limited current and past information from the servers to construct a local view of all the server queue lengths. For instance, dispatcher 1 believes that the queue length at server 3 is 1, while it is 2. It then sends jobs to the shortest queue as dictated by its view (here, to server 3 rather than server n). Communication is used only to update the local views of the dispatchers, *i.e.,* to improve their local views.

II. System model

We consider a system with a set $M = \{1, 2, \ldots, m\}$ of dispatchers load-balancing incoming jobs among a set $N = \{1, 2, \ldots, n\}$ of possibly-heterogeneous servers.

Time slots. We assume a time slotted system with the following order of events within each time slot: (1) jobs arrive at each dispatcher; (2) a routing decision is taken by each dispatcher and it immediately forwards its jobs to one of the servers; (3) each server performs its service for this time-slot.

Note that, for simplicity and ease of exposition, we assume that all dispatcher time slots are synchronized; in practice, such synchronization can be achieved by well established techniques such as the Precision Time Protocol (PTP) [2] that is commonly used in order to establish sub-microsecond network-wide time synchronization. For instance, [28] relies on NIC-based synchronization, which achieves a time accuracy of a few hundred nanoseconds even across a multi-hop network.

Nonetheless, we believe that our results may be extended to a framework where time slots are not synchronized among the dispatchers, and leave such a generalization to future work.

Dispatchers. As mentioned, each of the m dispatchers does not store incoming jobs, and instead immediately forwards them to one of the n servers. We denote by $a^j(t)$ the number of exogenous job arrivals at dispatcher j at the beginning of time slot t. We make the following assumption:

$$\left\{ a(t) = \sum_{j=1}^{m} a^j(t) \right\}_{t=0}^{\infty} \text{ is an } i.i.d. \text{ process} \tag{1}$$

$$\mathbb{E}[a(0)] = \lambda^{(1)} \tag{2}$$

$$\mathbb{E}\left[\Big(a(0)\Big)^2\right] = \lambda^{(2)} \tag{3}$$

That is, we only assume that the total job arrival process to the system is $i.i.d.$ over time slots and admits finite first and second moments. Note that we do not assume any specific process or any deterministic bound on the number of arrived jobs at a given time slot. The division of arriving jobs among the dispatchers is assumed to follow any arbitrary policy that does not depend on the system state (*i.e.,* queue lengths). Furthermore, we just assume that there is a positive probability of job arrivals at all dispatchers. That is, we assume that there exists a strictly positive constant ϵ_0 such that

$$\mathbb{P}(a^j(t) > 0) > \epsilon_0 \quad \forall (j,t) \in M \times \mathbb{N}. \tag{4}$$

This, for example, covers complex scenarios with time-varying arrival rates to the different dispatchers that are not necessarily independent. We are not aware of previous work covering such general scenarios with possibly correlated arrivals at the different dispatchers.

We further denote by $a_i^j(t)$ as the number of jobs forwarded by dispatcher j to server i at the beginning of time slot t. Let

$$a_i(t) = \sum_{j=1}^{m} a_i^j(t)$$

be the total number of jobs forwarded to server i by all dispatchers at time slot t. Finally, we assume that the arrival and service rates are unknown to the dispatchers.

Servers. Each server has a FIFO queue for storing incoming jobs. Let $Q_i(t)$ be the queue length of server i at the beginning of time slot t (before any job arrivals and departures at time slot t). We denote by $s_i(t)$ the potential service offered to queue i at time slot t. That is, $s_i(t)$ is the maximum number of jobs that can be completed by server i at time slot t. We assume that, for all $i \in N$,

$$\{s_i(t)\}_{t=0}^{\infty} \text{ is } i.i.d. \text{ over time slots} \tag{5}$$

$$\mathbb{E}[s_i(0)] = \mu_i^{(1)} \tag{6}$$

$$\mathbb{E}\left[\Big(s_i(0)\Big)^2\right] = \mu_i^{(2)} \tag{7}$$

Namely, we assume that the service process of each server is $i.i.d.$ over time slots and admits finite first and second moments. Note that we do not assume any specific process or any

deterministic bound on the number of completed jobs at a given time slot. We also assume that all service processes are mutually independent across the different servers and, furthermore, they are independent of the arrival processes.

Admissibility. We assume the system is sub-critical, i.e., that there exists an $\epsilon > 0$ such that

$$\sum_{i=1}^{n} \mu_i^{(1)} - \lambda^{(1)} = \epsilon. \tag{8}$$

III. LSQ LOAD BALANCING

Next, we formally introduce the LSQ family of load balancing policies. Then, we introduce our main theoretical result of the paper: namely, we establish a sufficient, easy to satisfy, condition for an LSQ policy to be stable.

A. The LSQ Family

We assume that each dispatcher $j \in M$ holds a local view of each server's $i \in N$ queue length. We denote by $\tilde{Q}_i^j(t)$ the queue length of server i as dictated by the local view of dispatcher j at the beginning of time slot t (before any arrivals and departures at that time slot). Finally, we can define LSQ.

Definition 1 (Local Shortest Queue (LSQ))**.** *We term a load balancing policy as an LSQ policy iff at each time slot, each dispatcher j follows the JSQ policy based on its local view of the queue lengths,* i.e., *$\{\tilde{Q}_i^j(t)\}_{i=1}^n$. That is, dispatcher j forwards all of its incoming jobs at the beginning of time slot t to a server i^* such that $i^* \in argmin_i\{\tilde{Q}_i^j(t)\}_{i=1}^n$ (ties are broken randomly).*

As we later show, this broad definition provides appealing flexibility when designing a load balancing policy, i.e., there are numerous approaches for how to *update the local views* of the dispatchers.

B. Sufficient stability condition

We proceed to introduce a sufficient condition for an LSQ policy to be stable. This condition essentially states that the difference between the local views of the dispatchers and the real queue lengths of the servers should be *bounded in expectation*. Note that the actual difference between the local views and the real queue states may be unbounded. As we later discuss, this loose assumption allows flexibility in the algorithm design with strong theoretical guarantees and appealing performance. Formally:

Assumption 1. *There exists a constant $C > 0$ such that at the beginning of each time slot (before any arrivals and departures), it holds that*

$$\mathbb{E}\left[\left|Q_i(t) - \tilde{Q}_i^j(t)\right|\right] \leq C \quad \forall\,(i,j,t) \in N \times M \times \mathbb{N}. \tag{9}$$

We will later rely on it to prove the main theoretical result of this paper, *i.e.,* that any LSQ load balancing policy that satisfies this condition is strongly stable.

C. Stability of LSQ

We begin by formally stating our considered concept of stability.

Definition 2 (Strong stability)**.** *We say that the system is strongly stable* iff *there exists a constant $K \geq 0$ such that*

$$\limsup_{T\to\infty} \frac{1}{T} \sum_{t=0}^{T-1} \sum_{i=1}^{n} \mathbb{E}\left[Q_i(t)\right] \leq K.$$

That is, the system is strongly stable when the expected time averaged sum of queue lengths admits a constant upper bound.

Strong stability is a strong form of stability that implies finite average backlog and (by Little's theorem) finite average delay. Furthermore, under mild conditions, it implies other commonly considered forms of stability, such as steady state stability, rate stability, mean rate stability and more (see [24]). Note that strong stability has been widely used in queuing systems (see [14] and references therein) whose state does not necessarily admit an irreducible and aperiodic Markov chain representation (therefore positive-recurrence may not be considered).

We proceed to prove the strong stability of LSQ whenever Assumption 1 holds.

Theorem 1. *Assume that the system uses LSQ and Assumption 1 holds. Then the system is strongly stable.*

Proof. A server can work on a job immediately upon its arrival. Therefore, the queue dynamics at server i are given by

$$Q_i(t+1) = [Q_i(t) + a_i(t) - s_i(t)]^+, \tag{10}$$

where $[\cdot]^+ \equiv \max\{\cdot, 0\}$. Squaring both sides of (10) yields

$$\begin{aligned}\Big(Q_i(t+1)\Big)^2 \leq \Big(Q_i(t)\Big)^2 + \Big(a_i(t)\Big)^2 + \Big(s_i(t)\Big)^2 \\ + 2a_i(t)Q_i(t) - 2s_i(t)Q_i(t) - 2s_i(t)a_i(t).\end{aligned} \tag{11}$$

Rearranging (11) and omitting the last term yields

$$\begin{aligned}&\Big(Q_i(t+1)\Big)^2 - \Big(Q_i(t)\Big)^2 \leq \\ &\Big(a_i(t)\Big)^2 + \Big(s_i(t)\Big)^2 - 2Q_i(t)\Big(s_i(t) - a_i(t)\Big).\end{aligned} \tag{12}$$

Summing over the servers yields

$$\begin{aligned}&\sum_{i=1}^{n} \Big(Q_i(t+1)\Big)^2 - \sum_{i=1}^{n} \Big(Q_i(t)\Big)^2 \leq \\ &B(t) - 2\sum_{i=1}^{n} Q_i(t)\Big(s_i(t) - a_i(t)\Big),\end{aligned} \tag{13}$$

where

$$B(t) = \sum_{i=1}^{n} \Big(a_i(t)\Big)^2 + \sum_{i=1}^{n} \Big(s_i(t)\Big)^2. \tag{14}$$

We would like to proceed by taking the expectation of (13). To that end, we need to analyze the term

$$\sum_{i=1}^{n} Q_i(t)\Big(s_i(t) - a_i(t)\Big),$$

since $\{Q_i(t)\}_{i=1}^n$ and $\{a_i(t)\}_{i=1}^n$ are dependent.

We shall conduct the following plan. We will introduce two additional policies into our analysis: (1) JSQ and (2) Weighted-Random (WR). Roughly speaking, we will show that the routing decision that is taken by our policy at each dispatcher and each time slot t is sufficiently similar to the decision that would have been made by JSQ *given the same system state*, which, in turn, is no worse than the decision that WR would make at that time slot. Since in WR the routing decisions taken at time slot t do not depend on the system state at time slot t, we will obtain the desired independence, which allows us to continue with the analysis.

We start by introducing the corresponding JSQ and WR notations. Let

$$a_i^{JSQ}(t) = \sum_{j=1}^{m} a_i^{j,JSQ}(t)$$

be the number of jobs that will be routed to server i at time slot t when using JSQ *at time slot* t. That is, each dispatcher forwards its incoming jobs to the server with the shortest queue (ties are broken randomly). Formally, let $i^* \in argmin_i \{Q_i(t)\}$, then $\forall j \in M$

$$a_i^{j,JSQ}(t) = \begin{cases} a^j(t), & i = i^* \\ 0, & i \neq i^*. \end{cases} \tag{15}$$

Let

$$a_i^{WR}(t) = \sum_{j=1}^{m} a_i^{j,WR}(t)$$

be the number of jobs that will be routed to server i at time slot t when using WR *at time slot* t. That is, each dispatcher forwards its incoming jobs to a single randomly-chosen server, where the probability of choosing server i is $\frac{\mu_i^{(1)}}{\sum_{i=1}^n \mu_i^{(1)}}$. Formally, $\forall j \in M$, $i = i^*$ with probability $\frac{\mu_i^{(1)}}{\sum_{i=1}^n \mu_i^{(1)}}$ and

$$a_i^{j,WR}(t) = \begin{cases} a^j(t), & i = i^* \\ 0, & i \neq i^* \end{cases} \tag{16}$$

With these notations at hand, we continue our analysis by adding and subtracting the term $2\sum_{i=1}^n a_i^{JSQ}(t)Q_i(t)$ from the right hand side of (13). This yields

$$\begin{aligned} &\sum_{i=1}^{n}\Big(Q_i(t+1)\Big)^2 - \sum_{i=1}^{n}\Big(Q_i(t)\Big)^2 \leq \\ &B(t) - 2\sum_{i=1}^{n} Q_i(t)\Big(s_i(t) - a_i^{JSQ}(t)\Big) + \\ &2\sum_{i=1}^{n} Q_i(t)\Big(a_i(t) - a_i^{JSQ}(t)\Big). \end{aligned} \tag{17}$$

We would like to take the expectation of (17). However, as mentioned, since the actual queue lengths and the local views of the dispatchers and the routing decisions that are made both by our policy and JSQ are dependent, we shall rely on the WR policy and the expected distance of the local views from the actual queue lengths to evaluate the expected values. To that end, we introduce the following lemmas. We present their proofs in an online extended version of this paper [35].

Lemma 1. *For all time slots t, it holds that*

$$\sum_{i=1}^{n} a_i^{JSQ}(t)Q_i(t) \leq \sum_{i=1}^{n} a_i^{WR}(t)Q_i(t).$$

Lemma 2. *For all servers $i \in N$ and all time slots t, it holds that*

$$\sum_{i=1}^{n} Q_i(t)\Big(a_i(t) - a_i^{JSQ}(t)\Big) \leq \sum_{i=1}^{n}\sum_{j=1}^{m} a(t)\Big|Q_i(t) - \tilde{Q}_i^j(t)\Big|.$$

Applying Lemmas 1 and 2 to (17) yields

$$\begin{aligned} &\sum_{i=1}^{n}\Big(Q_i(t+1)\Big)^2 - \sum_{i=1}^{n}\Big(Q_i(t)\Big)^2 \leq \\ &B(t) - 2\sum_{i=1}^{n} Q_i(t)\Big(s_i(t) - a_i^{WR}(t)\Big) \\ &+ 2\sum_{i=1}^{n}\sum_{j=1}^{m} a(t)\Big|Q_i(t) - \tilde{Q}_i^j(t)\Big|. \end{aligned} \tag{18}$$

Taking the expectation of (18) yields

$$\begin{aligned} &\mathbb{E}\Big[\sum_{i=1}^{n}\Big(Q_i(t+1)\Big)^2\Big] - \mathbb{E}\Big[\sum_{i=1}^{n}\Big(Q_i(t)\Big)^2\Big] \leq \\ &\mathbb{E}\Big[B(t)\Big] - 2\,\mathbb{E}\Big[\sum_{i=1}^{n} Q_i(t)\Big(s_i(t) - a_i^{WR}(t)\Big)\Big] \\ &+ 2\,\mathbb{E}\Big[\sum_{i=1}^{n}\sum_{j=1}^{m} a(t)\Big|Q_i(t) - \tilde{Q}_i^j(t)\Big|\Big]. \end{aligned} \tag{19}$$

We observe that both $a(t)$ (according to (1)) and $\{a_i^{WR}(t)\}_{i=1}^n$ (according to the definition of the WR policy) are independent of $\{Q_i(t)\}_{i=1}^n$ and $\Big\{\tilde{Q}_i^j(t)\ \Big|\ (i,j) \in N \times M\Big\}$. Specifically, by (1), $a(t)$ is independent of any events prior to time t, including the number of accumulated jobs in the system by time t. Applying this observation to (19) and using the linearity of expectation yields

$$\begin{aligned} &\mathbb{E}\Big[\sum_{i=1}^{n}\Big(Q_i(t+1)\Big)^2\Big] - \mathbb{E}\Big[\sum_{i=1}^{n}\Big(Q_i(t)\Big)^2\Big] \leq \\ &\mathbb{E}\Big[B(t)\Big] - 2\sum_{i=1}^{n}\mathbb{E}\Big[Q_i(t)\Big]\,\mathbb{E}\Big[\Big(s_i(t) - a_i^{WR}(t)\Big)\Big] \\ &+ 2\sum_{i=1}^{n}\sum_{j=1}^{m}\mathbb{E}\Big[a(t)\Big]\,\mathbb{E}\Big[\Big|Q_i(t) - \tilde{Q}_i^j(t)\Big|\Big]. \end{aligned} \tag{20}$$

Next, since for any non-negative $\{x_1, x_2, \ldots, x_n\}$ such that

$$x = x_1 + x_2 + \ldots + x_n$$

it always holds that $x^2 \geq \sum_{i=1}^n x_i^2$, using (5)-(7), the linearity of expectation and (1)-(3), we obtain

$$\begin{aligned} &\mathbb{E}\Big[B(t)\Big] = \mathbb{E}\Big[\sum_{i=1}^{n}\Big(a_i(t)\Big)^2\Big] + \mathbb{E}\Big[\sum_{i=1}^{n}\Big(s_i(t)\Big)^2\Big] \leq \\ &\mathbb{E}\Big[\Big(a(t)\Big)^2\Big] + \sum_{i=1}^{n}\mathbb{E}\Big[\Big(s_i(t)\Big)^2\Big] = \lambda^{(2)} + \sum_{i=1}^{n}\mu_i^{(2)}. \end{aligned} \tag{21}$$

Additionally, using (1), (2) and (9) yields

$$\sum_{i=1}^{n}\sum_{j=1}^{m}\mathbb{E}\left[a(t)\right]\mathbb{E}\left[\left|Q_i(t)-\tilde{Q}_i^j(t)\right|\right]\leq \sum_{i=1}^{n}\sum_{j=1}^{m}\lambda^{(1)}C=mn\lambda^{(1)}C. \tag{22}$$

Finally, since the decisions taken by the WR policy are independent of the system state, we can introduce the following lemma (proved in [35]).

Lemma 3. *For all $i\in N$ and t it holds that*

$$\mathbb{E}\left[s_i(t)-a_i^{WR}(t)\right]=\frac{\epsilon\mu_i^{(1)}}{\sum_{i=1}^{n}\mu_i^{(1)}}. \tag{23}$$

Using (21), (22) and Lemma 3 in (20) yields

$$\mathbb{E}\Big[\sum_{i=1}^{n}\Big(Q_i(t+1)\Big)^2\Big]-\mathbb{E}\Big[\sum_{i=1}^{n}\Big(Q_i(t)\Big)^2\Big]\leq \lambda^{(2)}+\sum_{i=1}^{n}\mu_i^{(2)}+2mn\lambda^{(1)}C - 2\sum_{i=1}^{n}\frac{\epsilon\mu_i^{(1)}}{\sum_{i=1}^{n}\mu_i^{(1)}}\mathbb{E}\left[Q_i(t)\right]. \tag{24}$$

For ease of exposition, denote the constants

$$D=\lambda^{(2)}+\sum_{i=1}^{n}\mu_i^{(2)}+2mn\lambda^{(1)}C, \tag{25}$$

and

$$\delta=\frac{\epsilon}{\sum_{i=1}^{n}\mu_i^{(1)}}. \tag{26}$$

Rearranging (24) and using (25) and (26) yields

$$2\delta\sum_{i=1}^{n}\mu_i^{(1)}\cdot\mathbb{E}\left[Q_i(t)\right]\leq D+\left(\mathbb{E}\Big[\sum_{i=1}^{n}\Big(Q_i(t)\Big)^2\Big]-\mathbb{E}\Big[\sum_{i=1}^{n}\Big(Q_i(t+1)\Big)^2\Big]\right). \tag{27}$$

Summing (27) over time slots $[0,1,\ldots,T-1]$, noticing the telescopic series at the right hand side of the inequality and dividing by $2\delta T$ yields

$$\frac{1}{T}\sum_{t=0}^{T-1}\sum_{i=1}^{n}\mu_i^{(1)}\cdot\mathbb{E}\left[Q_i(t)\right]\leq\frac{D}{2\delta}+\frac{1}{2\delta T}\left(\mathbb{E}\Big[\sum_{i=1}^{n}\Big(Q_i(0)\Big)^2\Big]-\mathbb{E}\Big[\sum_{i=1}^{n}\Big(Q_i(T)\Big)^2\Big]\right). \tag{28}$$

Taking limits of (28) and making the standard assumption that the system starts its operation with finite queue lengths, *i.e.,*

$$\mathbb{E}\Big[\sum_{i=1}^{n}\Big(Q_i(0)\Big)^2\Big]<\infty$$

yields

$$\limsup_{T\to\infty}\frac{1}{T}\sum_{t=0}^{T-1}\sum_{i=1}^{n}\mu_i^{(1)}\cdot\mathbb{E}\left[Q_i(t)\right]\leq\frac{D}{2\delta}. \tag{29}$$

Now, dividing both sides of (29) by $\min_i\{\mu_i^{(1)}\}$ yields

$$\limsup_{T\to\infty}\frac{1}{T}\sum_{t=0}^{T-1}\sum_{i=1}^{n}\frac{\mu_i^{(1)}\cdot\mathbb{E}\left[Q_i(t)\right]}{\min_i\{\mu_i^{(1)}\}}\leq\frac{D}{2\delta\cdot\min_i\{\mu_i^{(1)}\}}. \tag{30}$$

Finally, using the fact that

$$\frac{\mu_i^{(1)}\cdot\mathbb{E}\left[Q_i(t)\right]}{\min_i\{\mu_i^{(1)}\}}\geq\mathbb{E}\left[Q_i(t)\right]\quad\forall i$$

in (30) we obtain

$$\limsup_{T\to\infty}\frac{1}{T}\sum_{t=0}^{T-1}\sum_{i=1}^{n}\mathbb{E}\left[Q_i(t)\right]\leq\frac{D}{2\delta\cdot\min_i\{\mu_i^{(1)}\}}. \tag{31}$$

This implies strong stability and concludes the proof. □

Note that the result in (29) yields a slightly stronger bound than the one achieved in (31). This is because in (29) we have a bound on the service-rate-weighted time-averaged expected sum of queue lengths that is not sensitive to the skew among server service rates. That is, even for an arbitrary large skew (*i.e.,* when $\frac{\min_i\{\mu_i^{(1)}\}}{\max_i\{\mu_i^{(1)}\}}$ is arbitrary small), the bound does not grow to infinity. Nevertheless, we provide the result in (30) to obtain the standard form of strong stability (without service-rate normalization of the queue sizes).

IV. Simplified stability conditions

As mentioned, in order to establish that a system that uses an LSQ policy is strongly stable, it is sufficient to show that Assumption 1 holds. Generally, it may be challenging to establish this. To that end, we now develop simplified sufficient conditions. As we later demonstrate, these simplified conditions capture broad families of communication techniques between the servers and the dispatchers, and allow for the design of stable policies with appealing performance and extremely low communication budgets.

Throughout the proofs, we assume that our sufficient condition always holds when the system starts its operation, namely that there exists a constant $C_0\geq 0$ such that

$$\mathbb{E}\left[\left|Q_i(0)-\tilde{Q}_i^j(0)\right|\right]\leq C_0\ \forall\,(i,j)\in N\times M. \tag{32}$$

Also, we denote $\mathbb{1}_i^j(t)$ as an indicator function that obtains the value 1 iff server i updates dispatcher j (via the push-based sampling or the pull-based update message from the server) with its actual queue length at the end of time slot t (after arrivals and departures at time slot t).

A. Stochastic updates

We now prove that in LSQ, it is sufficient for the system to be strongly stable if for any server i, any dispatcher j, and any current local state error at dispatcher j for server i, there is a strictly positive probability that dispatcher j receives an update from server i. Intuitively, it means that the dispatcher may be rarely updated, but expected times between updates are still finite, and therefore errors do not grow unbounded in expectation.

Theorem 2. *Assume that there exists $\bar{\epsilon} > 0$ such that*

$$\mathbb{E}\left[\mathbb{1}_i^j(t) \;\middle|\; \left|Q_i(t) - \tilde{Q}_i^j(t)\right|\right] > \bar{\epsilon} \quad \forall (i,j,t) \in N \times M \times \mathbb{N}. \tag{33}$$

Then, Assumption 1 holds and the system is strongly stable.

Proof. Fix server i and dispatcher j. Denote

$$Z(t) = \left|Q_i(t) - \tilde{Q}_i^j(t)\right|.$$

Now, for all t it holds that

$$\begin{aligned} &Z(t+1) \leq \left(1 - \mathbb{1}_i^j(t)\right) \cdot \Big(Z(t) + a_i(t) + s_i(t)\Big) \leq \\ &\left(1 - \mathbb{1}_i^j(t)\right) \cdot Z(t) + a(t) + s_i(t). \end{aligned} \tag{34}$$

Taking expectation of (34) yields

$$\mathbb{E}[Z(t+1)] \leq \mathbb{E}[(1 - \mathbb{1}_i^j(t)) \cdot Z(t)] + \lambda^{(1)} + \mu_i^{(1)}. \tag{35}$$

Next, using the law of total expectation

$$\begin{aligned} &\mathbb{E}[(1 - \mathbb{1}_i^j(t)) \cdot Z(t)] = \\ &\mathbb{E}\left[\mathbb{E}\left[(1 - \mathbb{1}_i^j(t)) \cdot Z(t) \,\middle|\, Z(t)\right]\right] = \\ &\mathbb{E}\left[Z(t) \cdot \mathbb{E}\left[(1 - \mathbb{1}_i^j(t)) \,\middle|\, Z(t)\right]\right] \leq (1 - \bar{\epsilon})\, \mathbb{E}[Z(t)], \end{aligned} \tag{36}$$

where the last inequality follows from the linearity of expectation and (33). Now, using (36) in (35) yields

$$\mathbb{E}[Z(t+1)] \leq (1 - \bar{\epsilon})\, \mathbb{E}[Z(t)] + \lambda^{(1)} + \mu_i^{(1)}. \tag{37}$$

We now introduce the following lemma (proved in [35]).

Lemma 4. *Fix $\epsilon \in (0, 1]$, $C_1 \geq 0$ and $C_2 \geq 0$. Consider the recurrence*

$$T(n+1) \leq (1 - \epsilon) \cdot T(n) + C_1,$$

with the initial condition

$$T(0) \leq C_2.$$

Then,

$$T(n) \leq \max\left\{\frac{C_1}{\epsilon}, C_2\right\} \; \forall\, n.$$

Finally, using Lemma 4 in (37) yields

$$\mathbb{E}[Z(t)] \leq \max\left\{\frac{\lambda^{(1)} + \mu_i^{(1)}}{\bar{\epsilon}}, C_0\right\} \; \forall\, t.$$

This concludes the proof. □

B. Deterministic updates

We proceed to establish that any LSQ policy, in which each local view entry is updated at least once every fixed number C_{up} of time slots, is strongly stable.

Theorem 3. *Assume that each local entry is updated at least once every $C_{up} \in \mathbb{N}$ time slots. Then Assumption 1 holds and the system is strongly stable.*

Proof. Fix server i and dispatcher j. Then,

$$|Q_i(t + C_{up}) - \tilde{Q}_i^j(t + C_{up})| \leq \sum_{\tau=t}^{t+C_{up}-1} (a(\tau) + s_i(\tau)) \tag{38}$$

This is because the last update of this entry happened at most C_{up} time slots ago. Now, by taking the expectation, we obtain

$$\begin{aligned} &\mathbb{E}\left[|Q_i(t + C_{up}) - \tilde{Q}_i^j(t + C_{up})|\right] \leq \\ &\mathbb{E}\left[\sum_{\tau=t}^{t+C_{up}-1} (a(\tau) + s_i(\tau))\right] = C_{up}(\lambda^{(1)} + \mu_i^{(1)}) \end{aligned} \tag{39}$$

On the other hand, for any $t < C_{up}$ it holds that

$$\begin{aligned} \mathbb{E}\left[|Q_i(t) - \tilde{Q}_i^j(t)|\right] &\leq C_0 + \mathbb{E}\left[\sum_{\tau=0}^{C_{up}-1} (a(\tau) + s_i(\tau))\right] \\ &\leq C_0 + C_{up}(\lambda^{(1)} + \mu_i^{(1)}) \end{aligned} \tag{40}$$

This concludes the proof. □

V. Example LSQ policies

Since LSQ is not restricted to work with either pull- or push-based communications, in this section we provide examples for both. In a push-based policy, the dispatchers sample the servers for their queue lengths, whereas in a pull-based policy the servers update the dispatchers with their queue lengths. While empirically we will see that the pull-based approach can provide better performance in many scenarios, it may also incur additional implementation overhead, as it requires the servers to actively update the dispatchers given some state conditions, rather than passively answer sample queries. Therefore, we consider both the push and pull frameworks.

A. Push-based LSQ example

The power–of-choice $JSQ(d)$ policy forms a popular low-communication push-based load balancing approach, but it is not stable in heterogeneous systems, even for a single dispatcher. Instead, we will now analyze a push-based LSQ policy that uses exactly the same communication pattern between the servers and the dispatchers. It essentially extends the policy of [3], which considers a single dispatcher and homogeneous servers, to multiple dispatchers with heterogeneous servers.

In this policy, which we call $LSQ\text{-}Sample(d)$ and describe in Algorithm 1, each dispatcher holds a local, possibly outdated, array of the server queue lengths, and sends jobs to the minimum one among them. The array entries are updated as follows: (1) when a dispatcher sends jobs to a server, these jobs are added to the respective local approximation; (2) at each time slot, if new jobs arrive, the dispatcher randomly samples d distinct queues and uses this information only to update the respective d distinct entries in its local array to their actual value.

The simplicity of $LSQ\text{-}Sample(d)$ may be surprising. For instance, there is no attempt to guess or estimate how the other dispatchers send traffic or how the queue drains to get a better estimate, *i.e.,* our estimate is based only on the jobs that the specific dispatcher sends and the last time it sampled a queue. We also do not take the age of the information into account.

Furthermore, as we find below, the stability proof of $LSQ\text{-}Sample(d)$ only relies on the sample messages and not on the job increments. We empirically find that these increments help improve the estimation quality and therefore the performance.

Algorithm 1: $LSQ\text{-}Sample(d)$ (push-based comm.)

Code for dispatcher $j \in M$:

Route jobs and update local state:

foreach *time slot* t **do**
  Forward jobs to server $i^* \in argmin_i \left\{\tilde{Q}_i^j(t)\right\}$;
  Update $\tilde{Q}_{i^*}^j(t) \leftarrow \tilde{Q}_{i^*}^j(t) + a^j(t)$;
end

Sample servers and update local state:

foreach *time slot* t **do**
  if *new jobs arrive at time slot* t **then**
    Uniformly at random pick distinct $i_1, \ldots, i_d \in N$;
    For each $i \in \{i_1, \ldots, i_d\}$ update $\tilde{Q}_i^j(t) \leftarrow Q_i(t)$;
end

Algorithm 2: $LSQ\text{-}Update(p)$ (pull-based comm.)

Code for dispatcher $j \in M$:

Route jobs and update local state:

foreach *time slot* t **do**
  Forward jobs to server $i^* \in argmin_i \left\{\tilde{Q}_i^j(t)\right\}$;
  Update $\tilde{Q}_{i^*}^j(t) \leftarrow \tilde{Q}_{i^*}^j(t) + a^j(t)$;
end

Update local state:

foreach *arrived message* $\langle i, q\rangle$ *at time slot* t **do**
  Update $\tilde{Q}_i^j(t) \leftarrow q$;
end

Code for server $i \in N$:

Send update message:

foreach *time slot* t **do**
  if *completed jobs at time slot* t **then**
    Uniformly at random pick $j \in M$;
    if *idle* **then**
      Send $\langle i, Q_i(t)\rangle$ to dispatcher j;
    else
      Send $\langle i, Q_i(t)\rangle$ to dispatcher j *w.p.* p;
end

We proceed to establish that using $LSQ\text{-}Sample(d)$ at each dispatcher results in strong stability in multi-dispatcher heterogeneous systems. Interestingly, this result holds even for $d = 1$.

Proposition 1. *Assume that the system uses* $LSQ\text{-}Sample(d)$. *Then, it is strongly stable.*

Proof. Fix dispatcher j and server i. Consider time slot t. By (4), with probability of at least ϵ_0, dispatcher j samples d out of n servers uniformly at random disregarding the system state at time slot t. Therefore, we obtain

$$\mathbb{E}\left[\mathbb{1}_i^j(t) \;\middle|\; \left|Q_i(t) - \tilde{Q}_i^j(t)\right|\right] \geq \frac{\epsilon_0 \cdot d}{n}.$$

This respects the simplified probabilistic sufficiency condition and thus concludes the proof. □

B. Pull-based LSQ example

JIQ is a popular, recently proposed, low-communication pull-based load balancing policy. It offers a low communication overhead that is upper-bounded by a single message per job [18]. However, as mentioned, for heterogeneous systems, JIQ is not stable even for a single dispatcher.

We now propose a different pull-based LSQ policy that conforms with the same communication upper bound, namely a single message per job, and leverages the important idleness signals from the servers. It essentially follows similar lines to the policy presented in [34], which considers a single dispatcher and homogeneous servers, and extends it to multiple dispatchers with heterogeneous servers.

Specifically, each server, upon the completion of one or several jobs at the end of a time slot, sends its queue length to a dispatcher, which is chosen uniformly at random, using the following rule: (1) if the server becomes idle, then the message is sent with probability 1; (2) otherwise, the message is sent with probability $0 < p \leq 1$ where p is a fixed, arbitrary small, parameter.

Algorithm 2 (termed $LSQ\text{-}Update(p)$) depicts the actions taken by each dispatcher at each time slot.

The intuition behind this approach is to always leverage the idleness signals in order to avoid immediate starvation as done by JIQ; yet, in contrast to JIQ, even when no servers are idle, we want to make sure that the local views are not too far from the real queue lengths, which provides significant advantage at high loads.

We now formally prove that using $LSQ\text{-}Update(p)$ results in strong stability in multi-dispatcher heterogeneous systems. Interestingly, this result holds for any $p > 0$.

Proposition 2. *Assume that the system uses* $LSQ\text{-}Update(p)$. *Then, it is strongly stable.*

Proof. We prove that (33) holds. Fix dispatcher j, server i and time slot t. We examine two possible events at the beginning of time slot t: (1) $Q_i(t) = 0$ and (2) $Q_i(t) > 0$.

(1) Since $Q_i(t) = 0$, the server updated at least one dispatcher in a previous time slot, *i.e.,* for at least one dispatcher j^* we have that $\tilde{Q}_i^{j^*}(t) = 0$. This must hold since there is a dispatcher that received the update message after this queue got empty (that is, when a server becomes idle, a message is sent $w.p.$ 1). Now consider the event $A_1 = \left\{a_i^{j^*}(t) > 0 \cap s_i(t) > 0\right\}$. Since the tie breaking rule is random, by (1), (2), (4), (5) and (6), there exists $\bar{\epsilon}_i > 0$ such that $\mathbb{P}(A_1) > \bar{\epsilon}_i$. Since $a(t)$ and $s_i(t)$ are not dependent on any system information at the beginning of time slot t we obtain

$$\mathbb{E}\left[\mathbb{1}_i^j(t) \;\middle|\; \left|Q_i(t) - \tilde{Q}_i^j(t)\right|, Q_i(t) = 0\right] \geq p \cdot \mathbb{P}(A_1) > p \cdot \bar{\epsilon}_i. \tag{41}$$

(2) Since $Q_i(t) > 0$, there is a strictly positive probability that a job would be completed at this time slot. That is, since $s_i(t)$ is not dependent on any system information at the beginning of time slot t we obtain

$$\mathbb{E}\left[\mathbb{1}_i^j(t) \;\middle|\; \left|Q_i(t) - \tilde{Q}_i^j(t)\right|, Q_i(t) > 0\right] \geq p \cdot P(s_i(t) > 0) > p \cdot \bar{\epsilon}_i. \tag{42}$$

Algorithm 3: $LSQ\text{-}Smart(f, A)$ (smart servers)

Code for server $i \in N$:

Send update message:

```
foreach time slot t do
    if completed jobs at time slot t then
        with probability f:
            send ⟨i, Q_i(t)⟩ to dispatcher dictated by A;
end
```

Finally, since $\mathbb{E}\left[\mathbb{1}_i^j(t) \;\middle|\; \left|Q_i(t) - \tilde{Q}_i^j(t)\right|\right]$ is a convex combination of the left hand sides of (41) and (42) we obtain that

$$\mathbb{E}\left[\mathbb{1}_i^j(t) \;\middle|\; \left|Q_i(t) - \tilde{Q}_i^j(t)\right|\right] > p \cdot \bar{\epsilon}_i.$$

Now fix $\bar{\epsilon} = \min_i \{\bar{\epsilon}_i\}$. We have that

$$\mathbb{E}\left[\mathbb{1}_i^j(t) \;\middle|\; \left|Q_i(t) - \tilde{Q}_i^j(t)\right|\right] > p \cdot \bar{\epsilon} \quad \forall (i,j,t) \in N \times M \times \mathbb{N}.$$

Again, the probabilistic sufficiency condition holds and this concludes the proof. □

C. LSQ with smart servers

We next propose a more advanced LSQ variant that relies on smart servers. Namely, when the system uses a pull-based communication type, the servers update the dispatchers. As a result, the servers know the dispatcher states and how bad their local views are. This is because a server knows what was the last update message it sent to a dispatcher and how many jobs it has received from it since then.

In terms of additional resource requirements, an array of size m is sufficient in order to keep this information at a server. Namely, each server i holds a single entry for each dispatcher j. This entry is updated when server i sends an update message to dispatcher j or when new jobs from dispatcher j arrive to server i.

Specifically, we propose $LSQ\text{-}Smart(f, A)$. This LSQ variant is defined via two parameters, as follows.

1) A function f used by each server. Namely, f determines the probability by which a server, at each time slot in which it completes at least one job, sends an update message.
2) An algorithm A used by each server. Namely, A determines which dispatcher will be updated if a message is sent.

Both f and A may depend on the server's identity and its state. The pseudo-code for $LSQ\text{-}Smart(f, A)$ server update messages is presented by Algorithm 3. The remainder of the dispatcher's code is identical to Algorithm 2.

For concreteness, we next choose specific f and A and prove that they result in a stable LSQ policy. Note that we choose f and A heuristically and not via optimization. We leave further investigation regarding smart servers to future work.

Load dependent updates. For server i and dispatcher j denote

$$Z_i(t) = \max_j \{\left|Q_i(t) - \tilde{Q}_i^j(t)\right|\}.$$

Our heuristic choice for f is given by

$$f(i,p,t) = \begin{cases} p & Z_i(t) < Q_i(t) \\ 1 & Z_i(t) \geq Q_i(t), \end{cases}$$

where $p \in (0,1]$ is a fixed constant. In other words, at each time slot in which server i completes at least one job, it decides whether to update a dispatcher: (1) the server only updates a dispatcher w.p. (with probability) p whenever all local-state errors at dispatchers are relatively small, *i.e.,* strictly smaller than the server queue state; (2) Else, it sends an update w.p. 1.

For instance, if at the beginning of a time slot, the server queue size is 10 and a dispatcher thinks that it is 0 or 20 (*i.e.,* its local error is not small), then the server necessarily sends an update whenever it completes a job at that time slot. Another example is if the server *becomes idle*, and therefore the error cannot be smaller than the queue size of 0, hence the server also sends an update w.p. 1.

Worst approximation first (WAF). We set A such that a server always updates the dispatcher that has the worst local view when the message is sent. That is, if server i sends an update message at time slot t, it is sent to dispatcher $j^* \in argmax_j\{\left|Q_i(t) - \tilde{Q}_i^j(t)\right|\}$. Ties are broken arbitrarily.

For ease of exposition, we abuse notation and denote the function $f^*(p) = \{f(i,p,t)\}_{i \in N, t \in \mathbb{N}}$. We now turn to prove that using $LSQ\text{-}Smart(f^*(p), \text{WAF})$ results in strong stability for any $p > 0$.

Proposition 3. *Assume that the system is sub-critical and uses $LSQ\text{-}Smart(f^*(p), WAF)$. Then, the system is strongly stable.*

Proof. Herein is an outline. For the full proof see [35].

We already saw in the proof of Theorem 2 that, if

$$\mathbb{E}\left[\mathbb{1}_i^j(t) \;\middle|\; \left|Q_i(t) - \tilde{Q}_i^j(t)\right|\right] > \bar{\epsilon} \quad \forall (i,j,t) \in N \times M \times \mathbb{N}, \quad (43)$$

then Assumption 1 holds and we have strong stability. However, when using $LSQ\text{-}Smart(f^*(p), \text{WAF})$, such a condition may fail to hold. Indeed, when a server sends an update message, it is sent to the dispatcher that holds the worst local view at that time slot, thus leaving other dispatchers, with possible better local views, with a probability of 0 for an update. Therefore, in [35] we show that updating the worst local view is at least as good as updating any dispatcher that is chosen randomly. We do so by examining the *sum of errors* over the dispatchers for a specific server i. □

VI. Evaluation

Algorithms. We proceed to present an evaluation study of three stable LSQ schemes, namely: $LSQ\text{-}Sample(2)$, $LSQ\text{-}Update(2m/n)$ and $LSQ\text{-}Smart(f^*(2m/n), \text{WAF})$. We compare them to the baseline full-information JSQ and to the low-communication $JSQ(2)$ and JIQ. We note that all our three LSQ schemes are configured to have roughly the same expected communication overhead as the scalable $JSQ(2)$. Table I summarizes the stability properties and the worst-case communication requirements of the evaluated load balancing techniques as established by our analysis and verified by our evaluations.

	Throughput opt.		Comm. overhead	
	Homo-geneous	Hetero-geneous	Per time slot	Per job arrival
JSQ	✓	✓	$m \cdot n$	m
$JSQ(d)$	✓	×	$d \cdot m$	d
JIQ	✓	×	n	1
$LSQ\text{-}Sample(d)$	✓	✓	$d \cdot m$	d
$LSQ\text{-}Update(p)$	✓	✓	n	1
$LSQ\text{-}Update(f^*(p), \text{WAF})$	✓	✓	n	1

TABLE I: Comparing stability and *worst-case* communication overhead of the evaluated load balancing techniques.

System. In all our experiments, we consider a system of 100 servers and 10 dispatchers. Recall that the system operates in time slots with the following order of events within each time slot: (1) jobs arrive at each dispatcher; (2) a routing decision is taken by each dispatcher and it immediately forwards its jobs to one of the servers; (3) each server performs its service for this time-slot.

Arrivals. The number of jobs that arrive at each dispatcher at each time slot is sampled from a Poisson distribution with parameter $0.1 \cdot \lambda$, hence the total number of arrivals to the system at each time slot is a sample from a Poisson distribution with parameter λ. Each of the 10 dispatchers does not store incoming jobs, and instead immediately forwards them to one of the 100 servers.

Departures. Each server has an unbounded FIFO queue for storing incoming jobs. We divide the servers into the following two groups: (1) weak and (2) strong. We then consider different mixes of their numbers: (1) 10% strong - 90% weak; (2) 50% strong - 50% weak; (3) 90% strong - 10% weak. We consider the case where the ratio between the service rates of weak and strong servers is 1:10. In [35] we also provide results for a ratio of 1:2 and reach similar conclusions.

A. Evaluation results

As mentioned, we examine scenarios with a high degree of heterogeneity. Specifically, in these scenarios, the server service processes are geometrically distributed with a parameter $10p$ for a weak server and a parameter p for a strong server. In a simulation with n_s strong servers and n_w weak servers, we set $p = \frac{n_s+0.1n_w}{100}$ and sweep $0 \leq \lambda < 100$. The results are presented in Figure 2.

Stability. In all three scenarios, JIQ is not stable. Also, JIQ's stability region is significantly decreased due to the higher levels of heterogeneity. $JSQ(2)$ is unstable as well, with even worse degradation in the stability region. Specifically, it is stable only when there are only 10% weak servers in the mix such that the probability of not sampling a strong server upon arrival is sufficiently low. As implied by mathematical analysis, JSQ and all our three LSQ schemes are stable.

Performance. In all three scenarios and over all loads, our pull-based schemes exhibit the best performance. At high loads, our push-based scheme outperforms JSQ in two out of the three scenarios, whereas it under-performs when there is a low number of strong servers in the mix.

Communication overhead. As expected, all our LSQ schemes incur roughly the same communication overhead as the unstable $JSQ(2)$ policy. Recall that this is two orders of magnitude less than JSQ.

Delay tail distribution. Another finding of our simulation results is that *all* of the three LSQ policies consistently provide a better delay tail distribution than JSQ. For example, in Figure 3a, we present the CCDF of all stable policies at a normalized load of 0.95 in the scenario of Figure 2a (marked by a dashed grey line). It should be noted that JSQ has a lower average job completion time than our $LSQ\text{-}Sample(2)$ policy. Nevertheless, JSQ has a worse delay tail distribution. As illustrated in Figure 3b, this is due to the incast effect, where the majority (or even all) of the dispatchers forward their incoming jobs to a single (least loaded) server. It is notable how in all three LSQ policies the incast is nearly eliminated by allowing different dispatchers to have a different view of the system.

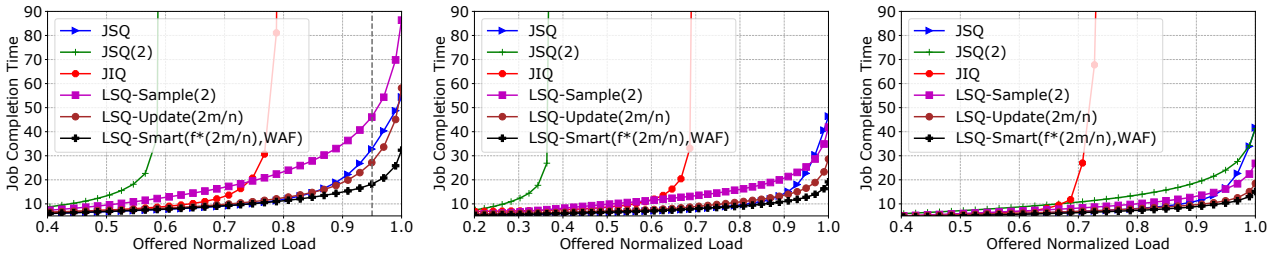
Interestingly, Figure 3a also shows that, for a small fraction that accounts for less than 0.00001 of the jobs (i.e., a rare event), the completion time under $LSQ{-}Smart(f^*(2m/n, WAF))$ is slightly larger than under $LSQ{-}Update(2m/n)$. Namely, even though $LSQ{-}Smart(f^*(2m/n, WAF))$ has a lower mean job completion time, it has a slightly slower decreasing delay tail. Essentially, the reason for this phenomenon is that even though for both policies incast is rare, in this specific scenario for $LSQ{-}Smart(f^*(2m/n, WAF))$, it is less rare than for $LSQ{-}Update(2m/n)$. For example, when using $LSQ{-}Smart(f^*(2m/n, WAF))$, for a small time fraction of ≈0.0001, five dispatchers send their jobs to the same server. But for $LSQ{-}Update(2m/n)$, it decreases by order of magnitude to ≈0.00001.

B. Evaluation takeaways

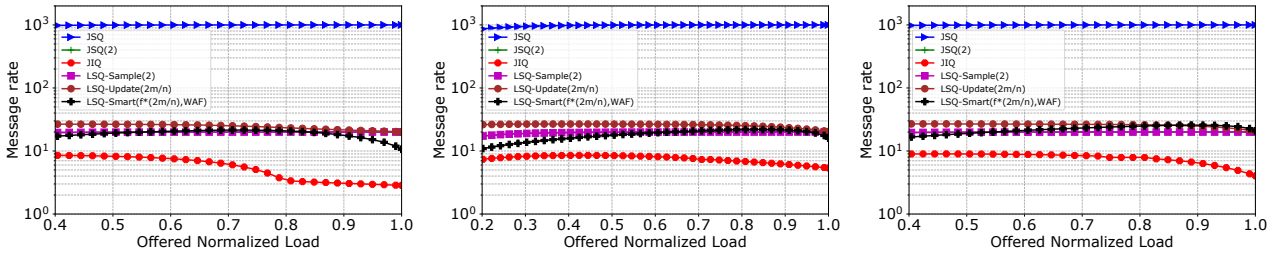
The three tested LSQ approaches always guarantee stability and do so using roughly the same communication budget as the non-throughput-optimal $JSQ(2)$. Moreover, the simulations indicate that, under these low-communication requirements, the tested LSQ policies consistently exhibit good performance in different scenarios and even admit better performance and delay tail distribution than the full-state information JSQ that uses more communication overhead by orders of magnitude.

Additionally, the evaluation results indicate how having pull-based communication and smart servers can further improve performance while using a similar communication budget. This is because pull-based communication and smart servers allow us to tune the system towards sending more messages from less loaded servers and directing them to less updated dispatchers, hence making better use of the communication budget.

Remark. In this paper, we target heterogeneous servers. Nevertheless, we have simulation results that indicate that LSQ techniques provide better performance in terms of job completion times for the homogeneous case (where all the aforementioned techniques are stable) as well. Furthermore, they indicate that, in the homogeneous case, JSQ(2) offers better performance than JSQ at high loads and even delay tails that are competitive with LSQ due to the reduced incast.

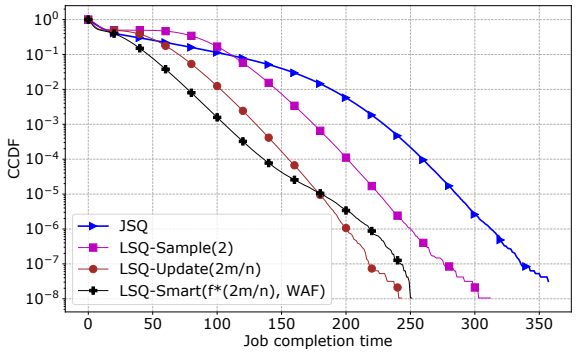


(a) 10 strong servers, 90 weak servers. (b) 50 strong servers, 50 weak servers. (c) 90 strong servers, 10 weak servers.

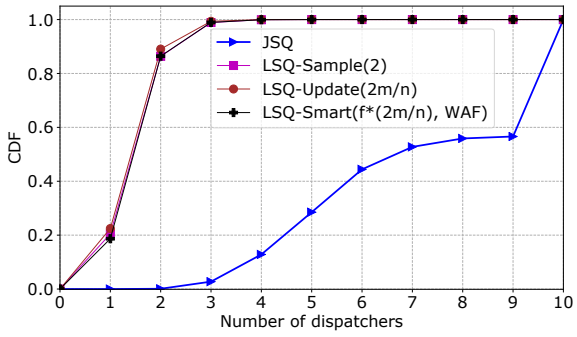


(d) 10 strong servers, 90 weak servers. (e) 50 strong servers, 50 weak servers. (f) 90 strong servers, 10 weak server.

Fig. 2: High heterogeneity scenario with 10 dispatchers and 100 heterogeneous servers.



(a) Job completion time CCDF.



(b) Maximum number of dispatchers that forward their arriving jobs to the same server at a time slot.

Fig. 3: Illustration of the job completion time CCDF and the incast effect at a normalized load of 0.95 for the scenario in Figure 2a (marked by a dashed grey line).

VII. Arbitrarily low communication

We have shown, both formally and by way of simulations, how different LSQ schemes offer strong theoretical guarantees and appealing performance with low communication overhead. In particular, by virtue of Theorems 2 and 3, we can construct various strongly stable LSQ policies with any arbitrarily low communication budget, disregarding whether the system uses pull or push messages (or both).

Achieving strong stability with arbitrary low communication is known to be possible with homogeneous servers, since even a uniform random load-balancing policy is stable in that case, and this was indeed strengthened in [34] for a scheme similar to LSQ and a different stability criterion *in continuous time*. However, establishing this for heterogeneous servers and multiple dispatchers is far from straightforward, and constitutes one of the main contributions of this paper.

Let $M(t)$ be the number of queue length updates performed by all dispatchers up to time t. Fix any arbitrary small $r > 0$. Suppose that we want to achieve strong stability, such that the average message rate is at most r, *i.e.,* for all t we have that $\mathbb{E}[M(t)] \leq rt$. Then, the two following per-time-slot dispatcher sampling rules trivially achieve strong stability (by Theorem 2) and respect the desired bound, *i.e.,* $\mathbb{E}[M(t)] \leq rt$.

Example 1 (push-based communication example)**.** *Dispatcher sampling rule upon job(s) arrival:*

(1) *pick a server $i \in N$ uniformly at random.*
(2) *sample server i with probability $\frac{r}{m}$.*

Example 2 (pull-based communication example)**.** *Server messaging rule upon job(s) completion:*

(1) *pick a dispatcher $j \in M$ uniformly at random.*
(2) *update dispatcher j with probability $\frac{r}{n}$.*

These theorems also enable us to design stable LSQ policies with hybrid communication (*e.g.,* push and pull) that attempt to maximize the benefits of both approaches. For example, the following policy leverages both the advantages of pull-based communication (*i.e.,* being immediately notified that a server becomes idle) and push-based communication (*i.e.,* random exploration of shallow queues when no servers are idle).

Example 3 (Hybrid communication example)**.** *Dispatcher sam-*

pling rule:

(1) *pick a server $i \in N$ uniformly at random.*
(2) *sample server i with probability $\frac{r}{m}$.*

Server messaging rule:

(1) *if got idle, pick a dispatcher $j \in M$ uniformly at random and send it an update message.*

The above examples demonstrate the wide range of possibilities that the LSQ approach offers to the design of stable, scalable policies with arbitrarily low communication overhead.

VIII. Discussion

Before concluding the study, we discuss several additional properties of the different load balancing approaches.

Instantaneous routing. An appealing property of any LSQ policy, similarly to JIQ, is that a dispatcher can immediately take routing decisions upon a job arrival. This is in contrast to common push-based policies that have to wait for a response from the sampled servers to be able to make a decision. For example, when using the $JSQ(2)$ policy, when a job arrives the dispatcher cannot immediately send the job to a server but must pay the additional delay of sampling two servers.

Space requirements. To implement an LSQ policy, similarly to JSQ, each dispatcher has to hold an array of size n (local views). When smart servers are used, each server also has to hold an array of size m (dispatcher states). Such a space requirement incurs negligible overhead on a modern server. For example, nowadays, any commodity server has tens to hundreds of GBs of DRAM. But even a hypothetical cluster with 10^6 servers requires only a few MB of the dispatcher's memory and much less (at least by 1-2 orders of magnitude) of server's memory, which is negligible in comparison to the DRAM size.

Computational complexity. To implement an LSQ policy, similarly to JSQ, each dispatcher has to repeatedly find the minimum (local) queue length. By using a priority queue (*e.g.,* min-heap), finding the minimum results in only a single operation (i.e., simply looking at the head of the priority queue). For a queue length update operation, $O(\log n)$ operations are required in the worst case (*e.g.,* decrease-key operation in a min-heap).[2] Even with $n = 10^6$, just a few operations are required in the worst case per queue length update. This results in a single commodity core being able to perform tens to hundreds of millions of such updates per second, hence resulting in negligible overhead, especially for a low-communication policy in which queue length updates are not too frequent.

Inaccurate information can lead to better performance. Although all tested LSQ variants in this paper are proved to be throughput-optimal, it is still surprising that using inaccurate information can lead to better performance than JSQ. In fact, we have found that allowing outdated information in the multi-dispatcher scenario not only reduces communication overhead significantly but also often results in better performance when compared to the full-state JSQ. This is because the incast effect can significantly degrade the performance of JSQ when many dispatchers forward their jobs to the shortest queue(s) simultaneously. On the other hand, as the simulation results indicate, having inaccurate information reduces the incast effect, since each dispatcher may believe that a different queue is the shortest (*e.g.,* Fig. 3a and 3b). Intuitively, in LSQ, by allowing inaccurate local states of the queue lengths, jobs are being forwarded to queues that may not be the least loaded ones but still have low load, which therefore reduces incast.

Alleviating incast with noise. Another natural way to reduce incast may be to use the JSQ policy with some sophisticated i.i.d. noise addition scheme that locally modifies the approximated queue lengths at each dispatcher in order to break synchronization while preserving performance. In fact, we can reuse our proof of Theorem 1 to show that such a scheme is ensured to be strongly stable if the added noise is bounded in expectation. In a sense, this is related to the issue discussed above, namely, that inaccurate information can lead to better performance. Nonetheless, such a JSQ-based solution is still not scalable in terms of communication overhead.

IX. Conclusions

In this paper, we introduced the LSQ family of load balancing algorithms. We formally established an easy-to-satisfy sufficient condition for an LSQ policy to be strongly stable. We further developed easy-to-verify sufficient stability conditions and exemplified their use. Then, using simulations, we showed how different LSQ schemes significantly outperform well-known low-communication policies, such namely $JSQ(d)$ and JIQ, while consuming a similar communication budget. We further demonstrated how relying on pull-based communication and, even further, on smart servers, allows LSQ to outperform even JSQ in terms of both the means and tail distributions of the job completion times, while using orders of magnitude less communication.

We believe that further investigation of performance guarantees beyond throughput optimality (*e.g.,* delay bounds) may lead to even better LSQ techniques. Also, it will be of interest to test the LSQ concept on real systems as well as explore how LSQ performs under different adversarial settings or additional knowledge of the system in comparison to current practice. For example, in systems where full or partial knowledge regarding these rates can be obtained, it is of interest to investigate how such knowledge can be used in order to provide improved load balancing solutions to the many-dispatcher case.

Acknowledgments

The authors would like to thank associate editor Paolo Giaccone and the anonymous reviewers for their helpful comments. This work was partly supported by the Hasso Plattner Institute Research School, the Israel Ministry of Science and Technology, the Technion Hiroshi Fujiwara Cyber Security Research Center, the Israel Cyber Bureau, and the Israel Science Foundation (grant No. 1119/19).

[2] A more sophisticated data structure, such as the Fibonacci heap, may offer even O(1) operations per update yet incur a higher constant.

REFERENCES

[1] LSQ code. https://github.com/LocalShortestQueue/LocalShortestQueue.

[2] IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pages 1–269, July 2008.

[3] J. Anselmi and F. Dufour. Power-of-d-choices with memory: Fluid limit and optimality. *arXiv preprint arXiv:1802.06566*, 2018.

[4] M. Bramson, Y. Lu, and B. Prabhakar. Randomized load balancing with general service time distributions. In *ACM SIGMETRICS*, 2010.

[5] M. Bramson, Y. Lu, and B. Prabhakar. Asymptotic independence of queues under randomized load balancing. *Queueing Systems*, 71(3):247–292, 2012.

[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] R. D. Foley and D. R. McDonald. Join the shortest queue: stability and exact asymptotics. *Annals of Applied Probability*, pages 569–607, 2001.

[8] G. Foschini and J. Salz. A basic dynamic routing problem and diffusion. *IEEE Transactions on Communications*, 26(3):320–327, 1978.

[9] S. Foss and N. Chernova. On the stability of a partially accessible multi-station queue with state-dependent routing. *Queueing Systems*, 29(1):55–73, 1998.

[10] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *IEEE Grid Computing Environments Workshop*, pages 1–10, 2008.

[11] D. Gamarnik, J. N. Tsitsiklis, and M. Zubeldia. Delay, memory, and messaging tradeoffs in distributed service systems. *Stochastic Systems*, 8(1):45–74, 2018.

[12] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. *ACM SIGCOMM Computer Communication Review*, 44(4):27–38, 2015.

[13] S. K. Garg, J. Lakshmi, and J. Johny. Migrating vm workloads to containers: Issues and challenges. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 778–785. IEEE, 2018.

[14] L. Georgiadis, M. J. Neely, and L. Tassiulas. Resource allocation and cross-layer control in wireless networks. *Foundations and Trends® in Networking*, 1(1):1–144, 2006.

[15] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *ACM SIGCOMM*, pages 58–72, 2016.

[16] V. Gupta, M. Harchol-Balter, K. Sigman, and W. Whitt. Analysis of join-the-shortest-queue routing for web server farms. *Performance Evaluation*, 64(9-12):1062–1081.

[17] R. S. Kannan, A. Jain, M. A. Laurenzano, L. Tang, and J. Mars. Proctor: Detecting and investigating interference in shared datacenters. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 76–86. IEEE, 2018.

[18] Y. Lu, Q. Xie, G. Kliot, A. Geller, J. R. Larus, and A. Greenberg. Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services. *Performance Evaluation*, 68(11):1056–1071, 2011.

[19] S. T. Maguluri, R. Srikant, and L. Ying. Heavy traffic optimal resource allocation algorithms for cloud computing clusters. *Performance Evaluation*, 81:20–39, 2014.

[20] T. McMullen. Load Balancing is Impossible. https://www.youtube.com/watch?v=kpvbOzHUakA, published on May 22, 2016.

[21] M. Mitzenmacher. Analyzing distributed join-idle-queue: A fluid limit approach. In *IEEE Allerton*, pages 312–318, 2016.

[22] M. Mitzenmacher, B. Prabhakar, and D. Shah. Load balancing with memory. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 799–808. IEEE, 2002.

[23] A. Mukhopadhyay, A. Karthik, and R. R. Mazumdar. Randomized assignment of jobs to servers in heterogeneous clusters of shared servers for low delay. *Stochastic Systems*, 6(1):90–131, 2016.

[24] M. J. Neely. Stability and probability 1 convergence for queueing networks via lyapunov optimization. *Journal of Applied Mathematics*, Volume 2012:Article ID 831909, 35 pages, 2012.

[25] NetScaler. NetScaler 11.1 and the Least Connection Method. https://docs.citrix.com/en-us/netscaler/11-1/load-balancing/load-balancing-customizing-algorithms/leastconnection-method.html, published on January 6, 2019.

[26] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *NSDI*, 2013.

[27] I. Owen Garrett of NGINX. NGINX and the "Power of Two Choices" Load-Balancing Algorithm. https://www.nginx.com/blog/nginx-power-of-two-choices-load-balancing-algorithm, published on November 12, 2018.

[28] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized zero-queue datacenter network. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 307–318. ACM, 2014.

[29] D. Shah and B. Prabhakar. The use of memory in randomized load balancing. In *IEEE International Symposium on Information Theory*, page 125, 2002.

[30] A. L. Stolyar. Pull-based load distribution in large-scale heterogeneous service systems. *Queueing Systems*, 80(4):341–361, 2015.

[31] A. L. Stolyar. Pull-based load distribution among heterogeneous parallel servers: the case of multiple routers. *Queueing Systems*, 85(1-2):31–65, 2017.

[32] W. Tarreau. HAProxy. Test Driving "Power of Two Random Choices" Load Balancing. https://www.haproxy.com/blog/power-of-two-load-balancing/, published on February 15, 2019.

[33] M. van der Boor, S. Borst, and J. van Leeuwaarden. Load balancing in large-scale systems with multiple dispatchers. In *IEEE INFOCOM*, 2017.

[34] M. van der Boor, S. Borst, and J. van Leeuwaarden. Hyper-scalable jsq with sparse feedback. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2019.

[35] S. Vargaftik, I. Keslassy, and A. Orda. LSQ: Load Balancing in Large-Scale Heterogeneous Systems with Multiple Dispatchers. *arXiv preprint arXiv:2003.02368 [cs.NI]*, 2020. https://arxiv.org/abs/2003.02368.

[36] R. R. Weber. On the optimal assignment of customers to parallel servers. *Journal of Applied Probability*, 15(2):406–413, 1978.

[37] W. Winston. Optimality of the shortest line discipline. *Journal of Applied Probability*, 14(1):181–189, 1977.

[38] L. Ying, R. Srikant, and X. Kang. The power of slightly more than one sample in randomized load balancing. *Mathematics of Operations Research*, 42(3):692–722, 2017.

[39] X. Zhou, F. Wu, J. Tan, Y. Sun, and N. Shroff. Designing low-complexity heavy-traffic delay-optimal load balancing schemes: Theory to algorithms. *ACM POMACS*, 1(2):39, 2017.

Shay Vargaftik received his B.Sc. and Ph.D. degrees from the Viterbi department of Electrical Engineering, Technion -– Israel Institute of Technology, in 2012 and 2019, respectively. He was the recipient of the Hasso Plattner Institute and the IBM Ph.D. fellowship awards. He is currently a postdoctoral researcher in the VMware Research Group (VRG). He is mainly interested in the theory and practice of networking and machine learning with an emphasis on scalability and efficient resource usage.

Isaac Keslassy (M'02, SM'11) received his M.S. and Ph.D. degrees in Electrical Engineering from Stanford University, Stanford, CA, in 2000 and 2004, respectively. He is currently a full professor in the Viterbi department of Electrical Engineering at the Technion, Israel. His recent research interests include the design and analysis of data-center networks and high-performance routers. He was the recipient of an ACM SIGCOMM test-of-time award, of an ERC Starting Grant, and of the Allon, Mani, Yanai, and Taub awards. He was associate editor for the IEEE/ACM Transactions on Networking.

Ariel Orda (S'84, M'92, SM'97, F'06) received the BSc (summa cum laude), MSc, and DSc degrees in electrical engineering from the Technion, Haifa, Israel, in 1983, 1985, and 1991, respectively. During 1.1.2014-31.12.2017, he was the dean of the Viterbi Department of Electrical Engineering, Technion. Since 1994, he has been with the Department of Electrical Engineering, Technion, where he is the Herman and Gertrude Gross professor of communications. His research interests include network routing, the application of game theory to computer networking, survivability, QoS provisioning, wireless networks, and network pricing. He served as program co-chair of IEEE INFOCOM 2002, WiOpt 2010 and Netgcoop 2020, and general chair of Netgcoop 2012. He was an editor of the IEEE/ACM Transactions on Networking and Computer Networks. He received several awards for research, teaching, and service.