Distributed Dispatching in the Parallel Server Model

Guy Goren
Technion – Israel Institute of Technology, Haifa, Israel
sgoren@campus.technion.ac.il

Shay Vargaftik
VMware Research
shayv@vmware.com

Yoram Moses
Technion – Israel Institute of Technology, Haifa, Israel
moses@ee.technion.ac.il

Abstract

With the rapid increase in the size and volume of cloud services and data centers, architectures with multiple job dispatchers are quickly becoming the norm. Load balancing is a key element of such systems. Nevertheless, current solutions to load balancing in such systems admit a paradoxical behavior in which more accurate information regarding server queue lengths degrades performance due to herding and detrimental incast effects. Indeed, both in theory and in practice, there is a common doubt regarding the value of information in the context of multi-dispatcher load balancing. As a result, both researchers and system designers resort to more straightforward solutions, such as the power-of-two-choices to avoid worst-case scenarios, potentially sacrificing overall resource utilization and system performance. A principal focus of our investigation concerns the value of information about queue lengths in the multi-dispatcher setting. We argue that, at its core, load balancing with multiple dispatchers is a distributed computing task. In that light, we propose a new job dispatching approach, called *Tidal Water Filling*, which addresses the distributed nature of the system. Specifically, by incorporating the existence of other dispatchers into the decision-making process, our protocols outperform previous solutions in many scenarios. In particular, when the dispatchers have complete and accurate information regarding the server queue lengths, our policies significantly outperform all existing solutions.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms; Networks → Network algorithms; Theory of computation → Online algorithms

Keywords and phrases Distributed load balancing, Join the Shortest Queue, Tidal Water Filling, Parallel Server Model

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.14

Related Version An extended version of this paper appears in `https://arxiv.org/abs/2008.00793`.

Funding *Guy Goren*: Partly supported by a grant from the Technion Hiroshi Fujiwara cyber security research center and the Israel cyber bureau, as well as by a Jacobs fellowship.
Yoram Moses: Yoram Moses is the Israel Pollak academic chair at the Technion. His work was supported in part by the Israel Science Foundation under grant 2061/19, and by the U.S.-Israel Binational Science Foundation under grant 2015820.

1 Introduction

Large software systems that govern the operations of data centers and of cloud-based services are important components of modern-day computing infrastructure. Such systems process a large volume of jobs that often arrive at distinct locations and are serviced by a multitude

of servers. How jobs are assigned to servers, and in particular load balancing the servers' job queues, plays a central role in determining the effectiveness of the system's overall performance.

The traditional approach to load balancing in this setting, known as the *supermarket model* [38, 39, 5], employs a single centralized dispatcher to which all requests are forwarded, and from which they are assigned to the servers. In the last decade, with the increasing size of cloud services and applications, using a single dispatcher has become a problematic bottleneck. System designers have consequently shifted to architectures that employ multiple dispatchers [6, 4, 26]. The load balancing problem for such multi-dispatcher systems is a very natural and urgent distributed systems problem. However, to the best of our knowledge, it has received only limited attention in the literature (see Section 1.1). The current paper considers the problem from a distributed computing perspective. We consider a setting with M dispatchers and N servers. Jobs arrive at the dispatchers according to a stochastic process, and the servers complete jobs at a stochastic rate. Each dispatcher observes the jobs that it receives, but not those received by the other dispatchers. Dispatchers interact with the servers when they send them jobs, and can also communicate with them in order to obtain information regarding the lengths of the servers' queues.

Load balancers are typically evaluated in terms of their behavior under heavy load, which occurs when the rate of job arrivals approaches the rate at which servers are able to process the jobs. A central factor in determining the quality of a load balancing protocol is the response time that it offers the clients. A natural measure is thus the expected response time (or *latency*) for jobs submitted to the system. In some cases, however, a client's task is broken up into multiple jobs, and the task is successfully served only when the last of these jobs is processed. For such instances, it is also important to meet a desired tail latency for the 95th, 99th, or even the 99.9th percentile of the distribution [3, 25]. Indeed, according to publications by Google and Amazon, even a small sub-second addition to response time in dynamic content websites has led to a persistent loss of users and revenue [16, 28].

A principal focus of our investigation concerns the value of information about queue lengths in the multi-dispatcher setting. As discussed below, several well-known load balancing policies behave poorly when the dispatchers' information is highly correlated. They suffer from so-called "herd behavior," in which, when dispatchers identify the same servers as having short queues, they all send their jobs to these servers. As a consequence, the servers become overloaded, resulting in poor performance. This is especially acute when dispatchers share considerable information about the queue lengths. Indeed, in describing their solutions, recent papers have made statements such as "*Inaccurate information can lead to better performance*" [35], or "*Inaccurate information can improve performance*" [40]. Interestingly, the value of accurate information is doubted not only by theoreticians. In fact, open source load balancer deployments such as HAProxy [33] and NGYNX [7] as well as cloud service companies such as Fastly [18] and Netflix [30] report making use of limited queue-size information in order to avoid detrimental herd behavior effects. This all appears to be rather perplexing and counter-intuitive. Is it indeed the case that there is such thing as too much information in the load-balancing context? Naïvely, at least, we would expect information to be valuable in a resource allocation context such as this.

In this paper, we consider why popular policies suffer from herd behavior and suggest new policies that avoid it. We argue that in policies that exhibit herd behavior, a dispatcher can be thought of as optimizing its behavior w.r.t. its information in a manner that is oblivious to the presence of other dispatchers. We suggest that this can be avoided by an analysis that explicitly accounts for the fact that other dispatchers are present. To focus on these issues, we

start by considering an idealized setting in which all dispatchers have complete information regarding the queue sizes. Each dispatcher has access to the job requests that it receives, but does not know how many jobs each of the other dispatchers receives. (The distributions of arrival rates at dispatchers are unknown but assumed to be the same, as are those of server processing rates.) In this setting, we propose a new load-balancing policy called *Tidal Water Filling* (TWF), and prove that it is in a precise sense optimal for the case of complete information. Indeed, we show that it improves on all known policies. Finally, we demonstrate that its performance improves as the amount of information available to the dispatchers increases. **These results establish that, when used appropriately, information can be gainfully used for load balancing in the multi-dispatcher setting**. These are the main contributions of the paper.

Since the performance of Tidal Water Filling improves with available information regarding queue sizes, we turn to the question of how limited communication can be exploited to boost this information, thereby further improving the performance of TWF. To this end, we design a variant of the TWF protocol in which both servers and dispatchers keep track of queue size information. Moreover, whenever a dispatcher interacts with a server, the two share the information they have. The information is consistently updated by using standard distributed systems techniques such as timestamping the size data. We show using simulations that this approach allows a significant increase in the amount of relevant data used by dispatchers even when communication is limited, and, in turn, markedly improves the load balancing performance.

1.1 Related Work

In the standard supermarket (i.e., parallel server) model, it is well known that if the (single) dispatcher has complete information regarding the server queues, the protocol that routes each job to the server with the shortest queue (called *Join the Shortest Queue* and denoted by JSQ) offers strong performance and strong theoretical guarantees [38, 39, 5]. JSQ has also motivated the design of reduced-state load balancing techniques for resource-constrained scenarios in which the dispatcher is exposed to only partial information about the server queue lengths. For example, in *Power-of-d-choices*, denoted by JSQ(d) [17, 36, 20], when a job arrives, a dispatcher randomly probes d servers and assigns the job to a server with the shortest queue among them. A related strategy is called *Power of memory*, denoted by JSQ(d, m) [29, 22]. In JSQ(d, m), the dispatcher samples the m shortest queues to whom it sent jobs in the latest round, in addition to $d \geq m \geq 1$ new randomly chosen servers. The job is then routed to the shortest among these $d + m$ queues.

In the last decade, with the increasing size of cloud services and applications, the need to scale horizontally drove system designers to introduce multiple dispatchers into their design as a single dispatcher could no longer utilize hundreds and thousands of servers [6, 4, 26]. In such multi-dispatcher systems, traditional solutions such as JSQ suffer from detrimental herd behaviour and therefore systems operators abandon the use of readily available information and turn to reduced-state approaches such as JSQ(d) potentially sacrificing overall system performance and reduced resource utilization in order to prevent worst-case scenarios [30, 33, 7, 18, 16, 28].

In the search for a better alternative for the multi-dispatcher load balancing scenario, Join-the-idle-queue (JIQ) [31, 16, 21, 32, 37] was recently proposed. In JIQ, dispatchers are notified only by idled servers. In turn, a dispatcher sends jobs to an idle server when it is aware of one, or to a randomly selected server otherwise. JIQ was shown to significantly

improve performance at low and moderate loads over JSQ(d) due to its immediate prevention of server starvation [16]. However, at higher loads, its performance resembles random routing due to the absence of idle servers and its performance deteriorates quickly [41].

The most recent advances on load balancing for multi-dispatcher systems appeared in [35, 40]. In the Local Shortest Queue policy (LSQ), proposed in [35], each dispatcher keeps a local state with possibly outdated queue size information, which is infrequently updated. A dispatcher then sends jobs to a shortest queue by its local estimation. This can be viewed as a generalization of similar ideas suggested for single-dispatcher systems [2, 34]. LSQ was followed by LED [40], which extended the theoretical performance guarantees to a wider family of *tilted* dispatching policies. These local-state-driven policies were shown to outperform previous policies considered for the multi-dispatcher model such as JSQ, JSQ(d) and JIQ. Intuitively, this is because they use considerably less information than JSQ, which reduces herding, and, on the other hand, maintain local states at the dispatchers allowing for a long term memory and preventing many events in which no single good server is discovered.

However, the aforementioned policies admit a paradoxical behavior in which accurate information, or even partial but correlated information among the dispatchers, degrades their performance. This is because, like their complete state-information JSQ counterpart, having shared information about good servers leads them to herd-behavior and detrimental incast effects that increase tail latency. That is, in the multiple-dispatchers case, when updated information is available to different dispatchers, they all send at once all incoming jobs to the servers with the currently-shorter queues, overwhelming them with the accumulated traffic. This phenomenon has already been pointed out in [19], which suggested the importance of using randomness to break the symmetry.

Another line of work concerning load balancing in distributed systems is based on the balls-into-bins model [1]. Recent approaches commonly apply regret minimization (e.g., [11]) and adaptive techniques (e.g., [15]), producing more precise theoretical bounds. However, their model assumptions are not aligned with our model (e.g., we consider stochastic arrivals at each dispatcher and stochastic departures at each server). Consequently, their analysis does not directly apply in our model and vice versa.

2 Model

We consider a system with a set $\mathcal{D}$ of M dispatchers and a set $\mathcal{S}$ of N servers. Dispatchers can communicate with servers over communication channels or via shared memory, but there is no direct communication among dispatchers. The network is the complete undirected bipartite graph with edge set $E = \mathcal{D} \times \mathcal{S}$, and both jobs and standard messages can be sent over the edges of E.[1] The system proceeds over discrete synchronous rounds. Time starts at time 0, and round $t + 1$ occurs between time t and time $t + 1$. Each round consists of four phases: First, every dispatcher receives an external input with a set of job requests. Second, every dispatcher sends each of the jobs it received to a server for processing, and every job received by a server is added to its job queue. Third, each of the servers completes processing a set of (zero or more) jobs from the head of its queue and reports the results to the appropriate clients. (This is where jobs depart from the system.) Finally, in a potential fourth phase of the round, dispatchers and servers may communicate information about the status of the server queues. More formally, the four phases are:

[1] The high rate of incoming jobs at the dispatchers makes interaction among them undesirable. As a result, the assumption that dispatchers do not interact is standard practice [19, 40, 35, 37, 32, 21, 31, 16, 33, 7].

1. **Arrivals:** Some number, $a^{(m)}(t)$, of exogenous jobs arrive at dispatcher m at the beginning of round t. (We denote $a(t) = \sum_{m\in\mathcal{D}} a^{(m)}(t)$.) Job arrivals are governed by stochastic processes. For simplicity, we assume that $a^{(m)}(t)$ are *i.i.d.* random variables governed by the same distribution which, according to standard practice, *is not assumed to be known*.
2. **Dispatching.** In every round, each dispatcher forwards the jobs it received to the servers for service. We consider two variants of dispatching that address two distinct affinity constraints. In one, termed *splittable dispatching*, the dispatcher assigns each of the jobs to a server of its choice. This handles a setting with no affinity constraints. That is, when the jobs arriving at a dispatcher m can be processed independently. We separately consider *unsplittable dispatching*, in which all jobs that arrive at m in a given round must be forwarded to the same server. This handles a setting with strict affinity constraints. Where, e.g., the jobs arriving at m in a given round share data or resources.
3. **Departures.** Each server maintains a FIFO queue that keeps track of its pending jobs. We denote by $Q_n(t)$ the length of server n's queue at the beginning of round t (before any job arrivals and departures) and denote $Q(t) \triangleq \langle Q_1(t), \ldots, Q_N(t)\rangle$. Moreover, we denote by $g_n^{(m)}(t)$ the number of jobs that server n receives from dispatcher m in round t. Moreover, $g_n(t) = \sum_{m\in\mathcal{D}} g_n^{(m)}(t)$ denotes the total number of jobs sent to server n in round t. We denote server n's job completion rate (i.e., the number of jobs that it is able to complete) in round t by $s_n(t)$. We thus have that $Q_n(t+1) = \max\{0, Q_n(t)+g_n(t)-s_n(t)\}$. I.e., server n completes $s_n(t)$ jobs in round t if it has that many jobs to process. Otherwise, it completes all $Q_n(t) + g_n(t) < s_n(t)$ jobs in its possession. As in the case of arrivals, $s_n(t)$ is assumed to be stochastically determined. Again, for ease of exposition their distribution is assumed to be the same for all servers.
4. **Communication.** In the fourth phase of a round, dispatchers obtain information about the queue sizes at the servers. We will consider two settings. In the *complete information* case, every dispatcher is informed of all queue sizes[2] and so, at the start of round t, it knows $Q(t)$. The *incomplete information* setting is one in which queue information is communicated over the channels of $E = \mathcal{D} \times \mathcal{S}$, without all servers communicating with all dispatchers in every round.

Admissibility. For a setting as above to be feasible, it must be the case that the servers' processing power is sufficient for handling the incoming job requests. Denoting $s(t) = \sum_{n\in\mathcal{S}} s_n(t)$, we define the *load* to be $\rho = \mathbb{E}[a(0)]/\mathbb{E}[s(0)]$. To be admissible, it must hold that $\rho < 1$.[3]

3 Stochastic Coordination

As a first step towards designing an effective multi-dispatcher policy, let us consider the problem in the simpler, single dispatcher, case. In round t, this dispatcher has access to the vector $Q(t)$ of queue sizes at the servers, and to the number $a(t)$ of jobs that have been submitted to the system. If the dispatcher is free to send different jobs to different servers then it will, intuitively, dispatch jobs one by one according to the JSQ principle. It will send the first job to a queue of shortest length, and then iterate through the jobs, each time sending the next job to a queue of smallest size given the jobs that it has already assigned.

[2] In practice, such information could be gathered in different ways. E.g., by a shared bulletin board or shared memory, as well as by having servers update a central process, who can forward a single message to with $Q(t)$ to each dispatcher.

[3] In [8], for the purpose of mathematical stability analysis, we also make the standard assumption that the arrival and departure processes admit a finite variance, i.e., $\mathrm{Var}(a(0)) < \infty$ and $\mathrm{Var}(s(0)) < \infty$.

Algorithm 1 Computing the water level.

```
1: function WaterLevel(Q, a)  ▷ Q is the multiset of queue lengths; a is the total number of arrivals;
2:                            ▷ Min returns the minimal value in a multiset.
3:     while a > 0 do
4:         MinSet ← {Q_n ∈ Q | Q_n = Min(Q)}        ▷ The set of all minimal queues.
5:         if |MinSet| = |Q| then                    ▷ |·| denotes the cardinality of a set.
6:             return Min(Q) + a/|Q|
7:         NextMin ← Min(Q\MinSet)
8:         δ ← NextMin − Min(Q)
9:         if δ · |MinSet| < a then
10:            a ← a − δ · |MinSet|
11:            for Q_n ∈ MinSet do
12:                Q_n ← Q_n + δ
13:        else
14:            return Min(Q) + a/|MinSet|
```

Consequently, all queues to which jobs are sent end up with the same sizes (give or take 1). We can view this as being analogous to a process of filling water into a container: Consider a water container with an uneven bottom at heights that correspond to the histogram (rearranged in sorted order) defined by $Q(t)$. If we should pour a volume of $a(t)$ units of water into the container, then the water level will coincide with the height of the queues to which jobs were dispatched, up to a rounding error due to the fact that water is continuous and jobs are discrete. The largest amount of water would be poured into the deepest column, just as the largest number of jobs would be sent to the shortest queue.

On a given input (Q, a), Algorithm 1 computes the water level $\mathrm{WL} = \textsc{WaterLevel}(Q, a)$ that results from pouring a units of water into a container with bottom shaped according to Q. The height of each column once the water is poured would be $Q^* \triangleq (Q_1^*, \ldots, Q_N^*)$ where $Q_n^* \triangleq \max\{Q_n, \mathrm{WL}\}$. We remark that Q_n^* is not always an integer but might also be a rational number. If queue lengths are maintained in sorted order, then WL is efficiently computable in $O(\min(N, a))$ time complexity.

In a multi-dispatcher system dispatchers make decisions independently of each other. It is thus only natural for them to independently optimize their load balancing decisions. Namely, to dispatch jobs in exactly the same manner as if they were alone in the system. This is indeed the case in current solutions. However, in a system where one is not alone, such oblivious behavior of disregarding the others may result in sub-optimal performance [7, 18, 35, 40, 30]. This occurs when the same server is identified as the best destination by different dispatchers. These dispatchers then simultaneously forward jobs to this server, causing its queue to grow rapidly, increasing delay times and sometimes even causing the server to drop jobs. However, the fact that dispatchers make decisions independently does not mean that their decision making protocols must be independently optimized.

In the multi-dispatcher context that we are considering, dispatchers cannot directly coordinate their actions in every given round, since they do not directly communicate with each other. Nevertheless, it is possible to design their protocols in such a way that their actions will be compatible with each other, and will not conflict. The key to doing so is employing *randomized protocols*, in which the dispatchers' moves are stochastic. Indeed, randomization has been a standard tool for symmetry breaking in distributed computing for over four decades [27, 14]. Our goal will be to design probabilistic load balancing protocols that will provide good performance by optimizing the cumulative behavior of the dispatchers. This will provide the dispatchers with a silent form of stochastic coordination.

4 Tidal Water Filling

Focusing on the complete information setting, we assume that each dispatcher m has access to the number $a^{(m)} = a^{(m)}(t)$ of jobs that it has received in the current round, and to the vector of server queue sizes $Q = Q(t)$ (we shall omit the round number t when it is clear from context). Based on Q and $a^{(m)}$, it needs to decide where to send each job. We seek a solution that will be feasible to compute and amenable to analysis. In particular, we seek a policy that (1) is uniform for all dispatchers; given the same Q and $a^{(m)} = a^{(m')}$, both m and m' should act in the same way, and in which (2) a dispatcher treats all jobs uniformly. Hence, the output of m's computation is a vector $\langle p_1, \ldots, p_N \rangle$, where p_n is the probability that any given job will be sent by m to server n, for every $n \in \mathcal{S}$.

We denote by $\bar{g}_n^{(m)}$ the random variable specifying the number of jobs sent to server n by dispatcher m. The total number of jobs received by n is $\bar{g}_n = \sum_{m \in \mathcal{D}} \bar{g}_n^{(m)}$, and its queue size once it receives them is the random variable $\bar{Q}_n \triangleq Q_n + \bar{g}_n$. Finally, we shall denote $\bar{Q} \triangleq \langle \bar{Q}_1, \ldots, \bar{Q}_N \rangle$.

Recall that Q and the total number of jobs $a = a(t)$ determine a water-filling solution $Q^* = Q^*(Q, a)$ as described in Section 3. Our goal will be to design a policy that computes the dispatching probabilities P in such a way that the resulting queue sizes $\bar{Q}$ approximate Q^* as well as possible. More formally, we wish to minimize the L_2 distance between Q^* and $\bar{Q}$. Intuitively, a large distance from the water level WL=WaterLevel(Q,a) induces a large delay in response times (for a positive difference) or server starvation and lesser resource utilization (negative difference). Since we seek to avoid long delay tails as well as unnecessary server idleness, we consider a large deviation from the WL to be worse than several small ones. This rules out linear or sub-linear distance measures such as the L_1 distance. On the other hand, giving too much weight to large deviations may miss opportunities to optimize the mean. For example, the L_∞ distance (i.e., min-max) addresses only the largest deviation. We therefore choose to use the L_2 distance, since it balances these two desires and is amenable to formal analysis.

We denote the vector of job arrivals at the dispatchers by $\vec{a} = \langle a^{(1)}, \ldots, a^{(M)} \rangle$. As an interim step, we derive a policy that computes the dispatching probabilities based on Q and the full vector $\vec{a}$ of jobs that arrive in the round, and not only the allocation $a^{(m)}$ of a single dispatcher m. We will later discuss how this analysis can be applied to an individual dispatcher's computation. Notice that Q and $\vec{a}$ uniquely determine a (fixed) vector Q^* resulting from water filling Q with $a = \sum_{m \in \mathcal{D}} a^{(m)}$ new jobs. Now, a policy $P(Q, \vec{a})$ gives rise to the random variable vector $\bar{Q}$ of queue sizes, as described above.

Recall that $\bar{g}_n = \bar{Q}_n - Q_n$. Similarly, we denote $g_n^* \triangleq Q_n^* - Q_n$. Our goal is to minimize

$$\begin{aligned}
\mathbb{E}\|Q^* - \bar{Q}\|_2^2 \;=\; & \mathbb{E}\|(Q_1^* - \bar{Q}_1, \ldots, Q_N^* - \bar{Q}_N)^T\|_2^2 \;= \\
& \mathbb{E}\|(Q_1 + g_1^* - Q_1 - \bar{g}_1, \ldots, Q_N + g_N^* - Q_N - \bar{g}_N)^T\|_2^2 \;= \\
& \mathbb{E}\|(g_1^* - \bar{g}_1, \ldots, g_N^* - \bar{g}_N)^T\|_2^2 \;= \sum_{n \in \mathcal{S}} \mathbb{E}\left[(g_n^* - \bar{g}_n)^2\right] \;= \\
& \sum_{n \in \mathcal{S}} {g_n^*}^2 - 2 \sum_{n \in \mathcal{S}} \left(g_n^* \mathbb{E}\left[\bar{g}_n\right]\right) + \sum_{n \in \mathcal{S}} \mathbb{E}\left[\bar{g}_n^2\right]
\end{aligned} \tag{1}$$

We perform separate analyses for the splittable and for the unsplittable cases.

4.1 The Splittable Case

In the splittable case, every job is sent to a server n with a probability of p_n. This implies, in particular, that the random variable $\bar{g}_n^{(m)}$ admits a binomial distribution, that is, $\bar{g}_n^{(m)} \sim \text{Bin}\left(a^{(m)}, p_n\right)$. Since each decision at each dispatcher is done independently, $\{\bar{g}_n^{(m)} \mid m \in \mathcal{D}\}$

are independent binomial variables with probability p_n. Thus, $\bar{g}_n = \sum_{m\in\mathcal{D}} \bar{g}_n^{(m)}$, where $\bar{g}_n \sim \text{Bin}\left(\sum_{m\in\mathcal{D}} a^{(m)}, p_n\right) \sim \text{Bin}\left(a, p_n\right)$. Hence,

$$\mathbb{E}[\bar{g}_n] = ap_n \quad \text{and} \quad \mathbb{E}[\bar{g}_n^2] = ap_n(1-p_n) + a^2{p_n}^2. \tag{2}$$

Given Q and a we can rewrite (1) using (2) as a function of $P = \langle p_1, \ldots, p_N\rangle$:

$$\begin{aligned} f(P) \;&=\; \mathbb{E}\|Q^* - \bar{Q}\|_2^2 \;=\; \sum_{n\in\mathcal{S}} {g_n^*}^2 - 2a\sum_{n\in\mathcal{S}} g_n^* p_n + \sum_{n\in\mathcal{S}} \left(ap_n - a{p_n}^2 + a^2{p_n}^2\right) \\ &=\; \sum_{n\in\mathcal{S}} {g_n^*}^2 - 2a\sum_{n\in\mathcal{S}} g_n^* p_n + a - a\sum_{n\in\mathcal{S}} {p_n}^2 + a^2\sum_{n\in\mathcal{S}} {p_n}^2 \end{aligned} \tag{3}$$

Now, to simplify the analysis, we first make the observation that for any strictly positive number of arrivals to the system (i.e., $a > 0$),

$$\arg\min f(P) \;=\; \arg\min (a-1)\sum_{n\in\mathcal{S}} {p_n}^2 - 2\sum_{n\in\mathcal{S}} g_n^* p_n. \tag{4}$$

We thus turn to solve the expression on the right-hand side. As can be seen from (4), for a single arrival to the system (i.e., $a = 1$), the solution would be to divide the probabilities arbitrarily among all shortest queues. Thus, we next assume that $a > 1$. Recall that we aim to compute a probability assignment P that optimizes $\arg\min f(P)$. In particular, we have that $\sum_{n\in\mathcal{S}} p_n = 1$ and $p_n \geq 0\ \forall n \in \mathcal{S}$. The optimization problem to solve in standard form is,

$$\begin{aligned} \min_P \quad & \tilde{f}(P) = (a-1)\sum_{n\in\mathcal{S}} {p_n}^2 - 2\sum_{n\in\mathcal{S}} g_n^* p_n \\ \text{s.t.} \quad & \sum_{n\in\mathcal{S}} p_n - 1 = 0, \quad -p_n \leq 0\ \forall n \in \mathcal{S}. \end{aligned} \tag{5}$$

Notice that this is not a linear program, because the objective function is not linear. However, the problem is convex with affine constraints. To solve the problem, we employ the Karush-Kuhn-Tucker method (KKT) [10, 12]. The associated Lagrangian function is

$$L(P,\Lambda) = (a-1)\sum_{n\in\mathcal{S}} {p_n}^2 - 2\sum_{n\in\mathcal{S}} g_n^* p_n - \sum_{n\in\mathcal{S}} \Lambda_n p_n + \Lambda_0\left(\sum_{n\in\mathcal{S}} p_n - 1\right). \tag{6}$$

The respective KKT conditions are

$$\begin{aligned} &\frac{\partial L}{\partial p_n} = 2(a-1)p_n - 2g_n^* - \Lambda_n + \Lambda_0 = 0 \quad \forall\, n \in \mathcal{S} && \text{(Stationarity)} \\ &\sum_{n\in\mathcal{S}} p_n - 1 = 0 \text{ and } p_n \geq 0 \quad \forall n \in \mathcal{S} && \text{(Primal feasibility)} \\ &\Lambda_n \geq 0 \quad \forall n \in \mathcal{S} && \text{(Dual feasibility)} \\ &p_n\Lambda_n = 0 \quad \forall n \in \mathcal{S} && \text{(Complementary slackness)} \end{aligned} \tag{7}$$

By Stationarity in (7) we obtain that, for any p_n,

$$p_n = \frac{2g_n^* - \Lambda_0 + \Lambda_n}{2(a-1)}, \tag{8}$$

and adding Complementary slackness from (7) yields that for any $p_n > 0$ we have,

$$p_n = \frac{2g_n^* - \Lambda_0}{2(a-1)}. \tag{9}$$

We can now substitute for p_n according to (9) in our objective function (5) to obtain a function of a single variable Λ_0. This yields,

$$\tilde{f}(P(\Lambda_0)) = (a-1)\sum_{p_n>0}\left(\frac{2g_n^*-\Lambda_0}{2(a-1)}\right)^2 - 2\sum_{p_n>0} g_n^*\left(\frac{2g_n^*-\Lambda_0}{2(a-1)}\right) = \frac{\sum_{p_n>0}\left(\Lambda_0^2-(g_n^*)^2\right)}{a-1}. \quad (10)$$

Clearly, to minimize the objective function, we seek the smallest Λ_0 that satisfies the KKT conditions given in (7). Observe that we can lower bound Λ_0 by combining (8) with the Primal feasibility in (7) to obtain:

$$1 \;=\; \sum_{n\in\mathcal{S}} p_n \;=\; \sum_{n\in\mathcal{S}} \frac{2g_n^*-\Lambda_0+\Lambda_n}{2(a-1)} \;\geq\; \sum_{g_n^*>0} \frac{2g_n^*-\Lambda_0+\Lambda_n}{2(a-1)}. \quad (11)$$

Using $\sum_{n\in\mathcal{S}} g_n^* \;=\; \sum_{g_n^*>0} g_n^* \;=\; a$, and rearranging (11) yields

$$2a - \Lambda_0 \sum_{g_n^*>0} 1 \;+\; \sum_{g_n^*>0} \Lambda_n \;\leq\; 2(a-1). \quad (12)$$

Thus, due to the Dual feasibility in (7), we obtain

$$\Lambda_0 \geq \frac{2+\sum_{g_n^*>0}\Lambda_n}{g_+^*} \geq \frac{2}{g_+^*}, \quad \text{where} \quad g_+^* \triangleq \sum_{g_n^*>0} 1. \quad (13)$$

Setting $\Lambda_0 = \frac{2}{g_+^*}$ and $\Lambda_n = 0$ for all $g_n^* > 0$ respects the KKT conditions and minimizes the objective function with respect to Λ_0. Finally, substituting $\frac{2}{g_+^*}$ for Λ_0 in Equation (9), we obtain that the optimal solution for $a > 1$ in the splittable case is

$$p_n \;=\; \max\{0, \frac{g_n^* - 1/g_+^*}{a-1}\}\ . \quad (14)$$

▶ **Definition 1** (Splittable tidal water filling). *Given $Q = Q(t)$ and $a = a(t) > 1$, a stochastic dispatching policy $P(Q, a)$ that, in every round t sends each job to server $n \in \mathcal{S}$ with probability $p_n = \max\{0, \frac{g_n^* - 1/g_+^*}{a-1}\}$, implements tidal water filling (sTWF) in the splittable setting.*

Notice that sTWF depends only on $a = a(t)$ and $Q = Q(t)$. It does not depend on the full detail of $\vec{a}$. In the context of complete information, Q is available to the dispatcher. However, a is not. In order to use sTWF an individual dispatcher m must replace a with some estimate. If $\mathbb{E}[a(0)]$, the expected value of a, is known, it can be used. Similarly, if $\mathbb{E}[s(0)]$, the total expected completion rate of the servers is known, it may also be used to replace a. Since, by assumption, dispatcher m has access to $a^{(m)}(t)$, it can use $Ma^{(m)}(t)$ for $a(t)$. This has the nice property that the average of what the dispatchers use equals exactly the total arrivals at that round. That is, $\frac{1}{M}\sum_{m\in\mathcal{D}} Ma^{(m)}(t) = a(t)$. We will hereafter assume that dispatcher m estimates $a(t)$ in this manner.[4] In Section 4.3 we shall discuss the properties and the intuitive interpretation of the probabilities used in sTWF.

[4] In Appendix C of [8], we show that the resulting protocol satisfies the desirable *strong stability property* for discrete-time queuing systems. We do the same for all the policies introduced in this paper.

4.2 The Unsplittable Case

In the unsplittable case, we again assume that every dispatcher m knows the vector $Q(t)$ of queue sizes (complete information). It differs from the splittable case only in that m must send all of the jobs that it receives in a given round to a single server. This affects the mathematics of the optimization problem. First of all, knowing the complete vector $\vec{a}$ of arrivals makes a significant difference in this case. Indeed, given Q and $\vec{a}$, computing an optimal job assignment to the servers essentially requires solving an instance of Bin-Packing. In Appendix A of [8] we prove its NP-hardness by a reduction from the Partition problem [9]. More precisely, we show the following.

▶ **Theorem 2.** *Given Q and $\vec{a}$, it is NP-hard to decide if $\min \mathbb{E}\|Q^* - \bar{Q}\|_2^2 = 0$ for a system with two servers.*

In general, the values $a^{(1)}, a^{(2)}, \ldots, a^{(M)}$ may be different from each other. Theorem 2 implies that optimizing for general Q and $\vec{a}$ is intractable. Instead, we optimize the unsplittable problem for the case that $a^{(1)} = a^{(2)} = \cdots = a^{(M)}$, which is consistent with the assumption that $a = Ma^{(m)}$, made in the splittable setting. We consider this case as a heuristic means to derive the dispatching probabilities. Our experiments in Section 4.4 show that the resulting policy works well, even when arrivals are governed by *i.i.d.* Poisson distributions, under which the arrival values $a^{(m)}$ are rarely identical.

We now reformulate the optimization problem in the unsplittable setting for this heuristic case. The difference from the splittable setting arises following (1) since now $\bar{g}_n^{(m)}$ does not admit a Binomial distribution. Instead, we have

$$\bar{g}_n^{(m)} = \begin{cases} a^{(m)}, & \text{w.p. } p_n, \\ 0, & \text{otherwise.} \end{cases}$$

Namely, a dispatcher m send all $a^{(m)}$ of its jobs to server n with probability p_n. Therefore, the total number of jobs that server n receives is $\bar{g}_n = \sum_{m\in\mathcal{D}} \bar{g}_n^{(m)}$. Since $\{\bar{g}_n^{(m)} \mid m \in \mathcal{D}\}$ are *i.i.d.* random variables, $\bar{g}_n$ has the following first and second moments,

$$\begin{aligned} &\mathbb{E}[\bar{g}_n] = Ma^{(m)}p_n, \\ &\mathbb{E}[\bar{g}_n^2] = \sum_{m\in\mathcal{D}} (a^{(m)})^2 p_n + \sum_{\substack{m,m'\in\mathcal{D},\\ m\neq m'}} a^{(m)} g^{(m')} {p_n}^2 = M(a^{(m)})^2 p_n + M(M-1)(a^{(m)}p_n)^2. \end{aligned} \tag{15}$$

Given Q and $a^{(m)}$ we rewrite (1) substituting the first and second moments according to (15). This yields

$$\begin{aligned} f(P) &= \sum_{n\in\mathcal{S}} {g_n^*}^2 - 2Ma^{(m)} \sum_{n\in\mathcal{S}} g_n^* p_n + \sum_{n\in\mathcal{S}} \left(M(a^{(m)})^2 p_n + M(M-1)(a^{(m)}p_n)^2 \right) \\ &= \sum_{n\in\mathcal{S}} {g_n^*}^2 - 2Ma^{(m)} \sum_{n\in\mathcal{S}} g_n^* p_n + M(a^{(m)})^2 \sum_{n\in\mathcal{S}} p_n + (M^2 - M)(a^{(m)})^2 \sum_{n\in\mathcal{S}} {p_n}^2. \end{aligned} \tag{16}$$

Recall that we aim to minimize $f(P)$ under the constraint that P is a probability assignment. That is, $\sum_{n\in\mathcal{S}} p_n = 1$ and $p_n \geq 0\ \forall n \in \mathcal{S}$. Observe that,

$$\arg\min f(P) \;=\; \arg\min (M-1)a^{(m)} \sum_{n\in\mathcal{S}} {p_n}^2 - 2\sum_{n\in\mathcal{S}} g_n^* p_n. \tag{17}$$

Again, we proceed to solve for the right hand side. As can be seen from (17), for a single-dispatcher system (i.e., $M = 1$), the solution would be to arbitrarily divide the probabilities

among all shortest queues, i.e., JSQ. Thus, we next assume that $M > 1$. The optimization problem in standard form becomes

$$\begin{aligned} \min_{P} \quad & \tilde{f}(P) = (M-1)a^{(m)} \sum_{n\in\mathcal{S}} {p_n}^2 - 2\sum_{n\in\mathcal{S}} g_n^* p_n \\ \text{s.t.} \quad & \sum_{n\in\mathcal{S}} p_n - 1 = 0, \quad -p_n \leq 0 \ \forall n \in \mathcal{S}. \end{aligned} \tag{18}$$

Once more, we employ the KKT method. Since the solution mainly follows similar lines to those of the splittable case, the full derivation is deferred to Appendix B of [8]. For the unsplittable case, however, the solution is more involved. In particular, it involves identifying the subset of servers $\mathcal{U} \subseteq \mathcal{S}$ for which positive probabilities should be assigned. We then use a summation on $\mathcal{U}$ to derive a lower bound on Λ_0 as in (11). We find this set of servers to be $\mathcal{U} = \{n \mid Q_n < \textsc{WaterLevel}(Q, (M-1)a^{(m)})\}$.[5] Namely, $\mathcal{U}$ is the set of servers that would be strictly below the water level if the total arrivals were $(M-1)a^{(m)}$ instead of $Ma^{(m)}$.

The solution for the optimization problem in (18) yields the following notion of tidal water filling for the unsplittable case:

▶ **Definition 3** (Unsplittable tidal water filling). *Given $Q = Q(t)$ and $a^{(m)} = a^{(m)}(t)$, let $\mathcal{U} = \{n \mid Q_n < \textsc{WaterLevel}(Q, (M-1)a^{(m)})\}$. An individual stochastic dispatching policy $P(Q, a^{(m)})$ for dispatcher m that, in every round t sends all its jobs to server $n \in \mathcal{S}$ with probability*

$$p_n = \max\left\{0, \frac{g_n^* - (a^{(m)} - \sum_{n'\notin\mathcal{U}} g_{n'}^*)/|\mathcal{U}|}{(M-1)a^{(m)}}\right\} \tag{19}$$

implements tidal water filling (uTWF) in the unsplittable setting.

Observe that, when at most one job arrives at each dispatcher, i.e., $a^{(m)} \leq 1$ for all $m \in \mathcal{D}$, there is no difference between the splittable and unsplittable problems. Indeed, in this case Definition 3 and Definition 1 coincide.

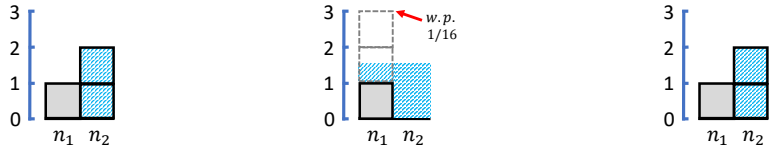
4.3 TWF vs Water Filling in Expectation

Recall that we aim to approximate water filling (i.e., Q^*). Consider the policy by which every dispatcher sends a job to each server n with a probability proportional to the amount of "water" it would receive in the pure water-filling solution. More formally, we define the Water Filling in Expectation policy (WFiE) to assign probabilities $P(Q, a) = \langle p_1, \ldots, p_N \rangle$, where for every n we have

$$p_n = \frac{g_n^*}{a} \ . \tag{20}$$

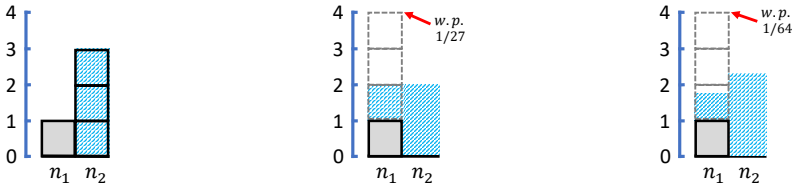
The expected length of each server n's queue under WFiE is precisely Q_n^*. Tidal water filling, however, does not produce the pure water-filling solution in expectation. Nevertheless, the math does not lie, and TWF improves on WFiE. We use the following two examples to demonstrate that WFiE is suboptimal, and to provide intuition for why TWF is better.

[5] Recall that $\textsc{WaterLevel}(\cdot,\cdot)$ is given by Algorithm 1.



(a) JSQ. **(b)** WFiE. **(c)** TWF.

Figure 1 A scenario with 2 dispatchers each of which receives a single job. Illustrating the expected arrivals at each queue for JSQ (Figure 1a), WFiE (Figure 1b) and TWF (Figure 1c).



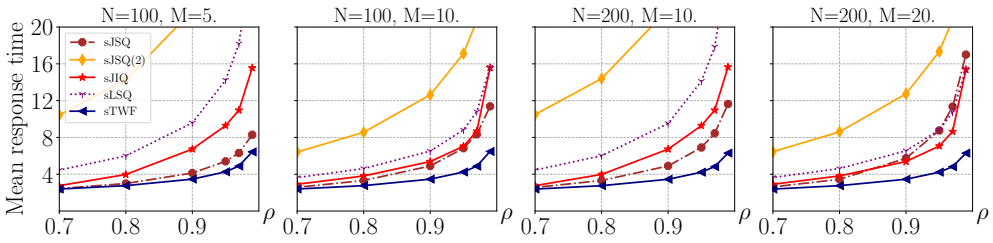
(a) JSQ. **(b)** WFiE. **(c)** TWF.

Figure 2 A scenario with 3 dispatchers each of which receives a single job. Illustrating the expected arrivals at each queue for JSQ (Figure 2a), WFiE (Figure 2b) and TWF (Figure 2c).

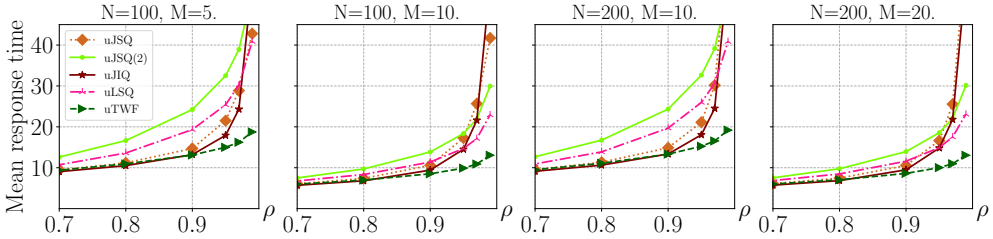
Example 1. Figure 1 depicts a system with $N = 2$ servers. At the beginning of the round, server n_1 has a single job in its queue, and server n_2 is idle (its queue is empty). There are $M = 2$ dispatchers, each of which receives a single job to dispatch. In this scenario, a dispatcher that uses JSQ will send its job to n_2; a dispatcher that uses WFiE will send its job to n_1 with probability $p_{n_1} = 1/4$ and to n_2 with probability $p_{n_2} = 3/4$; a dispatcher that uses TWF (since each dispatcher has a single job to dispatch in this scenario, sTWF and uTWF coincide) will send its job to n_2 with probability $p_{n_2} = 1$. The resulting *expected lengths* of the queues are depicted in Figure 1a for JSQ, in Figure 1b for WFiE, and in Figure 1a for TWF. Observe that both JSQ and TWF guarantee the favorable solution in which the longest queue has size 2, while in WFiE there is a non-negligible probability of 1/16 that both jobs will be forwarded to n_1, creating a queue of size 3.

Example 2. Figure 2 illustrates a system with the same two servers and the same initial state, but with $M = 3$ dispatchers. Each of the three dispatchers receives a single job to dispatch. In this scenario, a dispatcher that uses JSQ will again send its job to n_2; a dispatcher that uses WFiE will send its job to n_1 with probability $p_{n_1} = 1/3$ and to n_2 with probability $p_{n_2} = 2/3$; a dispatcher that uses TWF will send its job to n_1 with probability $p_{n_1} = 1/4$ and to n_2 with probability $p_{n_2} = 3/4$. The resulting *expected lengths* of the queues are depicted in Figure 1a for JSQ, in Figure 1b for WFiE, and in Figure 1a for TWF. In this case, JSQ results in herding towards n_2, creating a queue of size 3 there. WFiE results is a probability of 1/27 for ending with 4 jobs queuing at n_1, while n_2 remains idle. Observe that TWF reduces the probability of this worst-case allocation from 1/27 to 1/64. This is precisely the advantage that TWF provides over TFiE in general. We note that TWF also provides, with high probability, a favorable allocation in comparison to JSQ. In the second example, for instance, JSQ will produce a better outcome than TWF with probability $1.56\% = 1/64$, while TWF will be better than JSQ with probability $42.2\% = 27/64$.

Roughly speaking, our TWF policies have a greater bias towards short queues than WFiE does. As illustrated by the examples this, in turn, reduces the probability that queues will grow excessively long, and reduces the probability for short queues to become idle.



(a) Splittable dispatching.



(b) Unsplittable dispatching.

Figure 3 Average job response time as a function of the load over four different systems. The x-axis represents load ρ. The y-axis represents the average response time.

4.4 Evaluation

We conducted an empirical study of our TWF policies via simulations. In all of the simulations, at round t a dispatcher m has access only to $a^{(m)}(t)$. Recall that in both sTWF and uTWF dispatcher m uses $M \cdot a^{(m)}(t)$ as an estimate for $a(t)$.

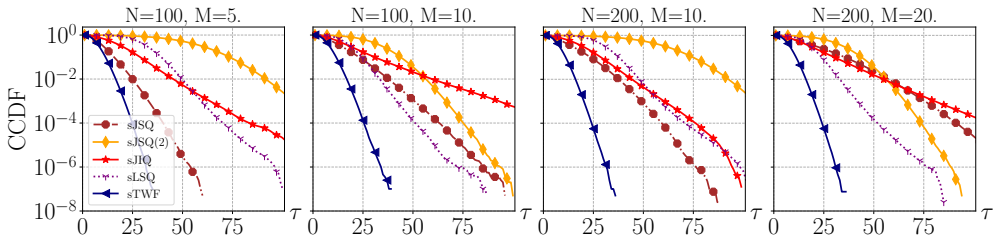
Arrivals and departures. Our modeling of the arrival and departure processes follows standard practice (e.g., [35, 16, 22, 34, 2]). In each round, we set $a^{(m)}(t) \sim Poisson(\lambda)$ for each dispatcher $m \in \mathcal{D}$, and $s^{(n)}(t) \sim Geometric(\mu)$ for each server $n \in \mathcal{S}$. Therefore, the load on the system is $\rho \triangleq M\lambda/(N\frac{\mu}{1-\mu})$.

Dispatching policies. We compare our policies to JSQ, the Power-of-two-choices denoted by JSQ(2), JIQ and the recently proposed LSQ.[6] We use a prefix of s and u to denote the splittable and the unsplittable case. For the splittable case, similarly to sTWF, other policies are also allowed to split jobs for parallel processing.

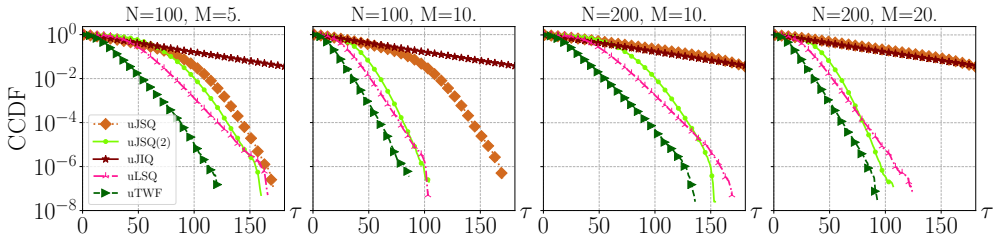
Setup. We consider systems with different numbers of servers N and dispatchers M. For each system, we run a set of simulations. Each simulation is configured with a different load and lasts for 10^5 time slots (rounds).

Results. Figure 3 shows the mean job response time (y-axis) across the different loads (x-axis) for different systems. It is notable that sTWF outperforms all other policies in the splittable case across all systems, and uTWF does the same in the unsplittable case. Moreover, as the load increases the gap between the TWF policies and the rest grows significantly. We remark that none of the other policies is in second place across the board.

[6] Specifically, we use LSQ-Sample(2). See [35] for details.



(a) Splittable dispatching.



(b) Unsplittable dispatching.

Figure 4 Response-time tail distribution over four different systems at high load ($\rho = 0.99$). The x-axis represents the response time (denoted by τ). The y-axis represents the CCDF.

For example, sJSQ is the second best at the smaller system ($N = 100, M = 5$) while sJIQ is the second best at the largest system ($N = 200, M = 20$).

As mentioned in Section 1 and in [3, 25, 16, 28], tail distributions play a crucial role in the parallel-server setting. We proceed to measure the tail distributions under various system parameters under the challenging scenario of a high load of ($\rho = 0.99$). This is depicted in Figure 4 using the complementary cumulative distribution function (CCDF), which shows for each response time (x-axis, denoted by τ), what is the fraction of jobs that surpass it (y-axis). E.g, for sLSQ in system $(200, 10)$ the response time of the 99^{th} percentile (i.e., 10^{-2}) is ≈ 50 rounds. Again, our TWF policies outperform all other techniques in all systems. Notice, that while uJSQ(2) is competitive in comparison to the other techniques when considering delay-tail decay, it is less favorable compared to them in terms of the average response time (see, e.g., Figure 3b). Intuitively, this is because its increased randomness helps to reduce worst-case herding, but at the price of often not utilizing good available servers. In summary, in the complete information case, our simulations show that the TWF policies consistently outperform the previous approaches when the load is high.

5 Enhancing Performance for Partial Information

In this section we relax the requirement that dispatchers have complete and accurate information regarding Q. That is, we now consider situations in which a dispatcher does not know the exact state of all servers. In line with recent work [35, 40, 2, 34], we consider a system where each dispatcher keeps a local array representing the servers' queue lengths, which may contain inaccurate (e.g., outdated) information. Communication is used to update array entries in the following manner: At the end of each round (i.e., in the fourth phase), a dispatcher establishes communication links with a fraction $\eta \leq 1$ of the servers, which are chosen uniformly at random. The corresponding entries in the dispatcher's local state are then updated with these servers' queue lengths. Additionally, a dispatcher that establishes a communication link to send jobs to server n during round t learns $Q_n(t)$. Notice that $\eta = 1$

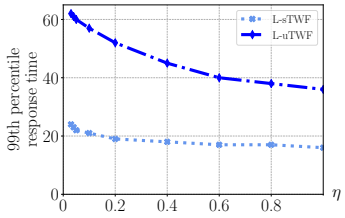


Figure 5 Response times as a function of η at high load ($\rho = 0.99$) for a system with $(N, M) = (100, 10)$.

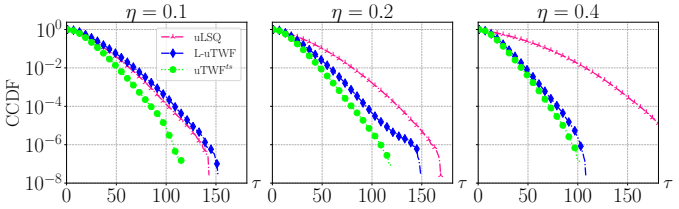


Figure 6 The effect of a distributed communication protocol on performance in a $(N, M) = (200, 20)$ system. Measuring the response time tail distribution (i.e., CCDF) at high load ($\rho = 0.99$).

corresponds to the complete information assumption; we use η as a parameter designating the extent of partial information available to the dispatchers. A dispatcher that uses uTWF based on its local array is said to implement *Local* uTWF (L-uTWF). *Local* sTWF is defined in the same manner. Figure 5 illustrates the simulations results. It shows that the response time improves monotonically as η increases.

This motivates us to increase the available information to the dispatchers. We attempt to do so without increasing the number of links that a dispatcher establishes. This is of interest since in many system the cost of communication lies mainly in the connection establishment rather than in the size of its content [24, 23]. To that end, we keep track of queue-size information at the servers, in a local array of size N. A server updates its local array based on its own queue length and information that it receives from dispatchers with which it has connections. Whenever a communication link between a dispatcher and a server is established, they merge their arrays. This is obtained by assigning time stamps to array entries, and maintaining the most recent information per entry upon the merge (cf. [13]).[7]

To test the effectiveness of the above scheme, we conduct an experiment comparing our protocols with the state-of-the-art LSQ-Sample of [35, 40] for different values of η. We denote by uTWF^{ts} the unsplittable policy from Definition 3 based on local arrays at both dispatchers and servers with time stamps. Figure 6 shows how the performance of uTWF^{ts} improves on that of L-uTWF for given values of η. The figure also illustrates that even the simpler L-uTWF policy is competitive with LSQ already at $\eta = 0.1$. As η grows (and the queue information improves), our protocols perform better, while LSQ's performance degrades.

6 Discussion

This work has demonstrated that, contrary to popular belief, queue-size information can be judiciously used to improve the quality of load balancing. In particular, we provided new policies that avoid herding and outperform all previous solutions for the case of complete information.

More generally, load balancing in the multi-dispatcher parallel server model is a natural question to explore using distributed systems tools and techniques. Our analysis in Section 5, for example, made use of time-stamping and flooding to improve the load balancing performance when information is partial. There are many additional aspects of the partial information setting that should be explored. For example, we note that when queue size information is sparse, the TWF's advantages do not come into play, and its performance

[7] To the best of our knowledge, this method was not previously employed in the parallel server model.

is not better than that of previous load-balancing policies. We state as an open problem how the advantages of previous approaches can be combined with those of TWF to obtain a policy that would make the best use of information across the whole spectrum of possibilities.

In another vein, it would be interesting to investigate how information about the distributions governing a multi-dispatcher systems can be obtained, and how they can be used to improve load-balancing performance. Can they provide good estimates for the TWF policies, and if so, how much benefit can they bring? Much is clearly left to be done.

References

1 Micah Adler, Soumen Chakrabarti, Michael Mitzenmacher, and Lars Rasmussen. Parallel randomized load balancing. *Random Structures & Algorithms*, 13(2):159–188, 1998.

2 Jonatha Anselmi and Francois Dufour. Power-of-d-choices with memory: Fluid limit and optimality. *Mathematics of Operations Research*, 2020.

3 Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

4 Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 523–535, 2016.

5 Atilla Eryilmaz and Rayadurgam Srikant. Asymptotically tight steady-state queue length bounds implied by drift conditions. *Queueing Systems*, 72(3-4):311–359, 2012.

6 Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. *ACM SIGCOMM Computer Communication Review*, 44(4):27–38, 2014.

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

8 Guy Goren, Shay Vargaftik, and Yoram Moses. Distributed dispatching in the parallel server model, 2020. `arXiv:2008.00793`.

9 Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

10 William Karush. Minima of functions of several variables with inequalities as side conditions. *Master's Thesis, Department of Mathematics, University of Chicago*, 1939.

11 Robert Kleinberg, Georgios Piliouras, and Éva Tardos. Load balancing without regret in the bulletin board model. *Distributed Computing*, 24(1):21–29, 2011.

12 Harold W Kuhn and Albert W Tucker. Nonlinear programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492, Berkeley, Calif., 1951. University of California Press. URL: `https://projecteuclid.org/euclid.bsmsp/1200500249`.

13 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. `doi:10.1145/359545.359563`.

14 Daniel Lehmann and Michael O Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In John White, Richard J Lipton, and Patricia C Goldberg, editors, *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 133–138. ACM Press, 1981. `doi:10.1145/567532.567547`.

15 Christoph Lenzen and Roger Wattenhofer. Tight bounds for parallel randomized load balancing. In *Proceedings of the 43rd annual ACM Symposium on Theory of Computing*, pages 11–20, 2011.

16 Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R Larus, and Albert Greenberg. Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services. *Performance Evaluation*, 68(11):1056–1071, 2011.

17 Malwina J Luczak, Colin McDiarmid, et al. On the maximum queue length in the supermarket model. *The Annals of Probability*, 34(2):493–527, 2006.

18 Tyler McMullen. Load Balancing is Impossible. Scaleconf 2016. URL: `https://www.youtube.com/watch?v=kpvbOzHUakA`.

19 Michael Mitzenmacher. How useful is old information? *IEEE Transactions on Parallel and Distributed Systems*, 11(1):6–20, 2000.

20 Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

21 Michael Mitzenmacher. Analyzing distributed join-idle-queue: A fluid limit approach. In *2016 54th Annual Allerton Conference on Communication, Control, and Computing*, pages 312–318. IEEE, 2016.

22 Michael Mitzenmacher, Balaji Prabhakar, and Devavrat Shah. Load balancing with memory. In *43rd Annual IEEE Symposium on Foundations of Computer Science.*, pages 799–808. IEEE, 2002.

23 YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 77–92, 2020.

24 David Murray, Terry Koziniec, Kevin Lee, and Michael Dixon. Large MTUs and internet performance. In *13th IEEE International Conference on High Performance Switching and Routing*, pages 82–87, 2012.

25 Rajiv Nishtala, Paul Carpenter, Vinicius Petrucci, and Xavier Martorell. Hipster: Hybrid task manager for latency-critical cloud workloads. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 409–420, 2017.

26 George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *26th Symposium on Operating Systems Principles (SOSP)*, pages 325–341, 2017.

27 Michael O Rabin. N-process synchronization by $4\log_2 N$-valued shared variables. In *21st Annual Symposium on Foundations of Computer Science*, pages 407–410. IEEE Computer Society, 1980. `doi:10.1109/SFCS.1980.26`.

28 Eric Schurman and Jake Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity Web Performance and Operations Conference*. O'Reilly, 2009.

29 Devavrat Shah and Balaji Prabhakar. The use of memory in randomized load balancing. In *Proceedings IEEE International Symposium on Information Theory*, page 125, 2002.

30 Mike Smith. Netflix Technology Blog. Rethinking Netflix's Edge Load Balancing. September 2018. URL: `https://netflixtechblog.com/netflix-edge-load-balancing-695308b5548c`.

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

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

33 Willy Tarreau. HAProxy. Test Driving "Power of Two Random Choices" Load Balancing. , published on February 15, 2019. URL: `https://www.haproxy.com/blog/power-of-two-load-balancing/`.

34 Mark van der Boor, Sem Borst, and Johan van Leeuwaarden. Hyper-scalable jsq with sparse feedback. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(1):1–37, 2019.

35 Shay Vargaftik, Isaac Keslassy, and Ariel Orda. LSQ: Load Balancing in Large-Scale Heterogeneous Systems With Multiple Dispatchers. *IEEE/ACM Transactions on Networking*, 2020.

36 Nikita Dmitrievna Vvedenskaya, Roland L'vovich Dobrushin, and Fridrikh Izrailevich Karpelevich. Queueing system with selection of the shortest of two queues: An asymptotic approach. *Problemy Peredachi Informatsii*, 32(1):20–34, 1996.

37 Chunpu Wang, Chen Feng, and Julian Cheng. Distributed join-the-idle-queue for low latency cloud services. *IEEE/ACM Transactions on Networking*, 26(5):2309–2319, 2018.

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

39 Wayne Winston. Optimality of the shortest line discipline. *Journal of Applied Probability*, 14(1):181–189, 1977.

40 Xingyu Zhou, Ness Shroff, and Adam Wierman. Asymptotically optimal load balancing in large-scale heterogeneous systems with multiple dispatchers. *arXiv preprint arXiv:2002.08908*, 2020.

41 Xingyu Zhou, Jian Tan, and Ness Shroff. Heavy-traffic delay optimality in pull-based load balancing systems: Necessary and sufficient conditions. *ACM SIGMETRICS Performance Evaluation Review*, 47(1):5–6, 2019.