Join-Idle-Queue: A Novel Load Balancing Algorithm for Dynamically Scalable Web Services

Yi Lu[a], Qiaomin Xie[a], Gabriel Kliot[b], Alan Geller[b], James R. Larus[b], Albert Greenberg[c]

[a]Department of Electrical and Computer Engineering,University of Illinois at Urbana-Champaign
[b]Extreme Computing Group, Microsoft Research
[c]Microsoft Azure

Abstract

The prevalence of dynamic-content web services, exemplified by *search* and *online social networking*, has motivated an increasingly wide web-facing front end. Horizontal scaling in the Cloud is favored for its elasticity, and distributed design of load balancers is highly desirable. Existing algorithms with a centralized design, such as Join-the-Shortest-Queue (`JSQ`), incur high communication overhead for distributed dispatchers.

We propose a novel class of algorithms called Join-Idle-Queue (`JIQ`) for distributed load balancing in large systems. Unlike algorithms such as Power-of-Two, the `JIQ` algorithm incurs no communication overhead between the dispatchers and processors at job arrivals. We analyze the `JIQ` algorithm in the large system limit and find that it effectively results in a reduced system load, which produces 30-fold reduction in queueing overhead compared to Power-of-Two at medium to high load. An extension of the basic `JIQ` algorithm deals with very high loads using only local information of server load.

Keywords: Load balancing · queueing analysis · randomized algorithm · cloud computing

1 Introduction

With affordable infrastructure provided by the Cloud, an increasing variety of dynamic-content web services, including search, social networking and e-commerce, are offered via the cyber space. For all service-oriented applications, short response time is crucial for a quality user experience. For instance, the average response time for web search is approximately 1.5 seconds. It was reported by Amazon and Google [14] that an extra delay of 500 ms resulted in a 1.2% loss of users and revenue, and the effect persisted after the delay was removed. Load balancing mechanisms have been widely used in traditional web server farms to minimize response times. However, the existing algorithms fall short with the large scale and distinct structure of service-oriented data centers. We focus on load balancing on the web-facing front end of Cloud service data centers in this paper.

In traditional small web server farms, a centralized hardware load balancer, such as F5 Application Delivery Controller [13], is used to dispatch jobs evenly to the front end servers. It uses the Join-the-Shortest-Queue (`JSQ`) algorithm that dispatches jobs to the processor with the least number of jobs. The `JSQ` algorithm is a greedy algorithm from the view of an incoming job, as the algorithm grants the job the highest instantaneous processing speed, assuming a processor sharing (PS) service discipline. The `JSQ` algorithm is not optimal, but is shown to have great performance in comparison to algorithms with much higher complexity [8]. Another benefit of the `JSQ` algorithm is its low communication overhead in traditional web server farms, as the load balancer can easily track the number of jobs at each processor: All incoming requests connect through the centralized load balancer and all responses are also sent through the load balancer. The load balancer is hence aware of all arrivals of jobs to a particular processor, and all departures as well, which makes tracking a simple task. No extra communication is required for the `JSQ` algorithm.

The growth of dynamic-content web services has resulted in an increasingly wide web-facing front end in Cloud data centers, and load balancing poses new challenges:

1. Distributed design of load balancers. A traditional web server farm contains only a few servers, while Cloud data centers have hundreds or thousands of processors for the front end alone. The ability to scale horizontally in and out to adapt to the elasticity of demand is highly valued in data centers. A single hardware load balancer that accommodates hundreds of processors are both expensive and at times wasteful as it increases the granularity of scaling. It is difficult to turn off a fraction of servers when the utilization of the cluster is low as it will require the reconfiguration of the load balancer. In addition, hardware load balancers lack the robustness of distributed load balancers, and programmability of software load balancers. The drawbacks have prompted the development of distributed software load balancers in the Cloud environment [1].

Fig. 1 illustrates load balancing with distributed dispatchers. Requests are routed to a random dispatcher via, for instance, the Equal-Cost-Multi-Path (ECMP) algorithm in a router. Load balancing of flows across dispatchers is less of a problem as the number of packets in web requests are similar. The service time of each request, however, varies much more widely, as some requests require the processing of a larger amount of data. Each dispatcher independently attempts to balance its jobs. Since only a fraction of jobs go through a particular dispatcher, the dispatcher has no knowledge of the current number of jobs in each server, which makes the implementation of the `JSQ` algorithm difficult.

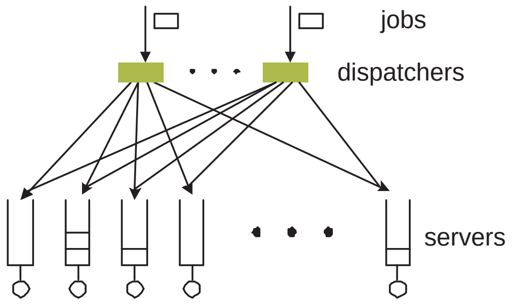


Figure 1: Distributed dispatchers for a cluster of parallel servers.

2. Large scale of the front end. The large scale of the front end processors exacerbates the complexity of the `JSQ` algorithm, as each of the distributed dispatchers will need to obtain the number of jobs at *every* processor before *every* job assignment. The amount of communication over the network between dispatchers and thousands of servers is overwhelming.

A naive solution is for each dispatcher to assign a request to a randomly sampled processor. We call it the `Random` algorithm. It requires no communication between dispatchers and processors. However, the response time is known to be large. For instance, even with a light-tailed distribution such as exponential, the mean response time at 0.9 load is 10 times the mean service time of a single job (For a given load ρ and service rate μ, the mean response time is $(1-\rho)^{-1}\mu^{-1}$ while the mean service time is μ^{-1}.) The queuing overhead tremendously outweighs the service time due to uneven distribution of work in the cluster.

1.1 Previous Work

There have been two lines of work on load balancing with only partial information of the processor loads, hence potentially adaptable for distributed implementation. Both are based on randomized algorithms, but are developed for different circumstances.

The Power-of-*d* (`SQ(d)`) algorithm. The randomized load balancing algorithm, `SQ(d)`, has been studied theoretically in [16, 10, 3, 7, 9]. It was conceived with the large web server farms in mind and all results have been asymptotic in the system size, hence it is a good candidate for load balancing in Cloud service data centers. At each job arrival, the dispatcher samples d processors and obtains the number of jobs at each of them. The job is directed to the processor with the least number of jobs among the d sampled. The `SQ(d)` algorithm produces exponentially improved response time over the `Random` algorithm, and the communication overhead is greatly reduced over the `JSQ` algorithm. However, the gap between its performance and `JSQ` remains significant. More importantly, the `SQ(d)` algorithm requires communication between dispatchers and processors *at the time* of job assignment. The communication time is on the critical path, hence contributes to the increase in response time.

Work stealing and work sharing. Another line of work [15, 2, 11] was developed for shared-memory multi-processor systems and is considered for high-performance compute clusters [5, 17, 6]. Work stealing is defined as idle processors pulling jobs from other randomly sampled processors, and work sharing is defined as overloaded processors pushing jobs to other randomly sampled processors. The main distinction between shared-memory systems and web server clusters lies in the way work arrives to the system: In a shared-memory system or compute cluster, new threads are generated on each processor independently, while for web services, new jobs arrive from external networks at the dispatcher. Hence for web services, assigning jobs to processors and then allowing the processors to redistribute jobs via either pull or push mechanism introduces additional overhead. Moreover, moving a job in process is easy in a shared-memory system [15], but difficult for web services, as the latter involves migration of TCP connections and subsequent synchronization with subtasks at the back end of the cluster. As a result, work stealing and work sharing algorithms are not directly applicable to Cloud service data centers.

1.2 Our Approach

We propose a class of algorithms called Join-Idle-Queue (`JIQ`) for large-scale load balancing with distributed dispatchers. The central idea is to decouple discovery of lightly loaded servers from job assignment. The basic version involves idle processors informing dispatchers at the time of their idleness, without interfering with job arrivals. This removes the load balancing work from the critical path of request processing.

The challenge lies in the distributed nature of the dispatchers as the idle processors need to decide which dispatcher(s) to inform. Informing a large number of dispatchers will increase the rate at which jobs arrive at idle processors, but runs the risk of inviting too many jobs to the same processor all at once and results in large queuing overhead. The processor can conceivably remove itself from the dispatchers once it receives the first job, but this will require twice as much communication between processors and dispatchers. On the other hand, informing only one dispatcher will result in wasted cycles at idle processors and assignment of jobs to occupied processors instead, which adversely affects response times.

To solve the problem, the proposed `JIQ` algorithm *load balances* idle processors across dispatchers, which we call the *secondary* load balancing problem. In order to solve the primary load balancing problem of assigning jobs to processors, we first need to solve the secondary problem of assigning idle processors to dispatchers, which curiously takes place in the reverse direction. While the primary problem concerns the reduction of average queue length at each processor, the secondary problem concerns the *availability* of idle processors at each dispatcher. It is not a priori obvious that load balancing idle processors across dispatchers will outperform the `SQ(d)` algorithm because of the challenges outlined above.

We consider the PS service discipline, which approximates the service discipline of web servers. The analysis also applies to the FIFO service discipline, which can be interesting in its own right: certain dynamic web services incur busty processing at the front end where each burst is shorter than processor sharing granularity, hence the queues behave as a FIFO re-entrant queue. Analyzing the performance of FIFO re-entrant queues is outside the scope of this paper, but the analysis of FIFO queues is a necessary first step. The main contributions of the paper are as follows:

1. We analyze both the primary and secondary load balancing systems and find that the `JIQ` algorithm reduces the *effective load* on the system. That is, a system with 0.9 load behaves as if it has, say 0.5 load. The mean queue length in the system is shown to be *insensitive* to service time distributions for the PS discipline in the large system limit.

2. The proposed `JIQ` algorithm incurs no communication overhead at job arrivals, hence does not increase actual response times. With equal or less complexity, and not taking into account the communication overhead of `SQ(2)`, `JIQ` produces smaller queueing overhead than `SQ(2)` by an order of magnitude, with the actual value depending on the load and processor-to-dispatcher ratio. For instance, the `JIQ` algorithm with `SQ(2)` in the reverse direction yields 30-fold reduction in mean queueing overhead over `SQ(2)` for both PS and FIFO disciplines, at 0.9 load and with the processors-to-dispatchers ratio fixed at 40.

Remark. This paper considers homogeneous processors and the analysis assumes Poisson arrivals of requests. The `JIQ` algorithm can be readily extended in both directions to include heterogeneous processors and general arrivals. In particular, reporting decisions are made locally by each processor, which can take into account its heterogeneity. Note that data locality is not a problem in the front end where the servers are responsible for gathering data from the back end and organizing them into pages. The general distribution of arrival intervals does not change the analysis in the large system limit as the arrivals at an individual dispatcher become Poisson. The evaluation of the performance of `JIQ` algorithm is based on simulation with a variety of service time distributions, corresponding to different application workloads. Since the `JIQ` algorithm exploits the large scale of the system, a testbed capturing its behavior will need to contain at least hundreds, if not thousands of servers, which is not available at the moment. We defer the implementation details of the `JIQ` algorithm to future publications.

Section 2 describes the `JIQ` algorithm with distributed dispatchers. We analyze the algorithm in the large system limit with general service time distributions in Section 3 and discuss design extensions and implementation issues in Section 4. Section 5 compares the performance of the `JIQ` algorithm with the `SQ(2)` algorithm via simulation.

2 The Join-Idle-Queue Algorithm

Consider a system of n parallel homogeneous processors interconnected with commodity network components. There are an array of m dispatchers, with $m << n$. Requests arrive at the system as a rate-$n\lambda$ Poisson process. Each request is directed to a randomly chosen dispatcher, which assigns it to a processor. The service time of a request is assumed to be i.i.d. with a general service time distribution $B(\cdot)$ of mean 1. We consider both PS and FIFO service disciplines.

The objective of the load balancing algorithm is to provide fast response time at each processor without incurring excessive communication overhead. In particular, communication overhead on the critical path, *i.e.*, at the arrival of a request, is to be avoided as it adds to the overall response time. Communication off the critical path is much less costly as it can ride on *heartbeats* sent from processors to job dispatchers signalling the health of the nodes.

2.1 Preliminary

There are two ways to think about load balancing in a system of parallel processors. As an entire system, the processors *collaborate* to adapt quickly to the randomness in the arrival process and service times. On the other hand, when focusing on a single processor, efficient load balancing changes the *arrival rate* to the processor based on the number of jobs in its queue. In particular, it increases the arrival rate to idle processors and decreases that to processors with a large queue size. This results in shorter busy cycles for each processor and faster response time.

To illustrate the effect of length of busy cycles on response times, compare the two busy cycle patterns on a single processor illustrated in Fig. 2. The letter 'b' denotes 'busy' and the letter 'i' denotes 'idle'. The two patterns can result from different load balancing schemes in the system. The load is the same for both patterns, as they share the same mean idle time. However, pattern 2 indicates a much larger arrival rate than pattern 1 when the processor is idle. This results in shorter busy cycles and a much shorter response time.

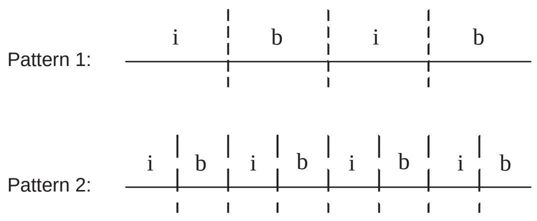


Figure 2: Busy (b) / idle (i) patterns of a processor.

Motivated by the above, we compare the rate of arrival to an idle processor, λ_0, for the following three algorithms. With rate-$n\lambda$ arrivals and n processors of service rate 1, the load on the system is λ. The `Random` algorithm produces the worst response times and the `JSQ` algorithm produces the best.

$$\begin{aligned}
\texttt{Random.} \quad & \lambda_{0,R_1} && = \lambda, \qquad \forall\, n. \\
\texttt{SQ(d).} \quad & \lim_{n\to\infty} \lambda_{0,R_d} && = \lambda \frac{1-\lambda^d}{1-\lambda} = \lambda + \lambda^2 + \cdots + \lambda^d. \text{ [3].} \\
\texttt{JSQ.} \quad & \lim_{n\to\infty} \lambda_{0,JSQ} && = \frac{\lambda}{1-\lambda} = \lambda + \lambda^2 + \lambda^3 + \cdots \text{ [8].}
\end{aligned}$$

Under the `Random` algorithm, the stochastic queueing process at each processor is *independent* of each other. There is no collaboration and the arrival rate is constant for all queue sizes. The `SQ(d)` algorithm changes the arrival rate probabilistically. Each increase in d, the number of processors it compares, adds one term to the arrival rate. However, the marginal effect of each additional comparison on the arrival rate decreases exponentially. The `JSQ` algorithm compares all processors in the system and we have an interesting observation,

Corollary 1

$$\lim_{d\to\infty} \lim_{n\to\infty} \lambda_{0,R_d} = \lim_{n\to\infty} \lambda_{0,JSQ}.$$

Observe that in the large system limit as n goes to infinity, the queue sizes under `JSQ` never exceed 1 and every job is directed to an idle processor. This motivates the focus on idle processors: As the cluster increases in size, arrival rates for the larger queue sizes become less important. Random distribution suffices at the very small chance of all processors being occupied.

2.2 Algorithm Description

The algorithm consists of the primary and secondary load balancing systems, which communicate through a data structure called *I-queue*. Together, they serve to decouple the discovery of idle servers from the process of job assignment. Fig. 3 illustrates the overall system with an I-queue at each dispatcher. An I-queue is a list of a subset of processors that have reported to be idle. All processors are accessible from each of the dispatchers.

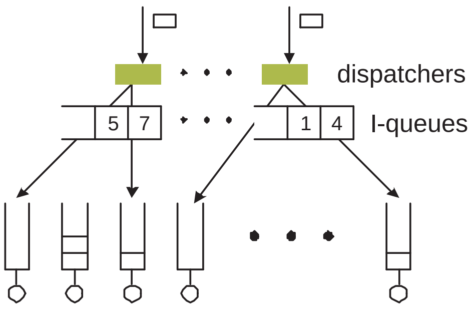


Figure 3: The JIQ algorithm with distributed dispatchers, each of which is equipped with an I-queue.

Primary load balancing. The primary load balancing system exploits the information of idle servers present in the I-queues, and avoids communication overhead from probing server loads. At a job arrival, the dispatcher consults its I-queue. If the I-queue is non-empty, the dispatcher removes the first idle processor from the I-queue and directs the job to this idle processor. If the I-queue is empty, the dispatcher directs the job to a randomly chosen processor.

When a processor becomes idle, it informs an I-queue of its idleness, or *joins* the I-queue. For all algorithms in this class, each idle processor joins only one I-queue to avoid extra communication to withdraw from I-queues. The challenge with distributed dispatchers is the uneven distribution of incoming jobs and idle processors at the dispatchers: It is possible that a job arrives at an empty I-queue while there are idle processors in other I-queues. This poses a new load balancing problem in the *reverse* direction from processors to dispatchers:

How can we assign idle processors to I-queues so that when a job arrives at a dispatcher, there is high probability that it will find an idle processor in its I-queue?

Secondary load balancing. When a processor becomes idle, it chooses one I-queue based on a load balancing algorithm and informs the I-queue of its idleness, or joins it. We consider two load balancing algorithms in the reverse direction: Random and SQ(d). We call the algorithm with Random load balancing in the reverse direction `JIQ-Random` and that with SQ(d) load balancing `JIQ-SQ(d)`. With `JIQ-Random`, an idle processor chooses an I-queue uniformly at random, and with `JIQ-SQ(d)`, an idle processor chooses d random I-queues and joins the one with the smallest queue length. While all communication between processors and I-queues are off the critical path, `JIQ-Random` has the additional advantage of having a one-way communication, without requiring messages from the I-queues. We study the performance of both algorithms with analysis and simulation in the rest of the paper.

3 Analysis in the Large System Limit

We analyze the performance of the `JIQ` algorithms in the large system limit as we are interested in the regime of hundreds or thousands servers. In particular, fix the ratio of the number of servers to the number of I-queues

$$r \equiv \frac{n}{m},$$

and let $n, m \to \infty$. We make two simplifying assumptions: 1. Assume there is exactly *one copy* of each idle processor in the I-queues. 2. Assume there are *only* idle processors in the I-queues. We discuss the validity and consequence of the assumptions below.

We can modify the algorithm without adding too much complexity so that the first assumption always holds. There can be more than one copy of an idle processor in the I-queues when an idle processor receives a random arrival, becomes idle again, and joins another I-queue. This can be prevented if we let an idle processor keep track of the I-queue it currently registers with, and does not join a new I-queue after finishing service of a randomly directed job. This modification is not included in the actual algorithm as no visible difference in performance is observed in simulations.

The second assumption is violated when an idle server receives a random arrival. In the current algorithm, the server does not withdraw from the I-queue as this will incur too much communication. As a result, this server is no longer idle, but still present in the I-queue it registered with. We show in Corollary 2 that the mean arrival rate from occupied I-queues is r times more than that from empty I-queues for `JIQ-Random`. The difference is even larger for `JIQ-SQ(d)`. As a result, there is only a small chance to have random arrivals at idle servers, and no visible difference is observed from computed values and simulation results. The difference in the arrival rates also explains why events violating assumption 1 are rare as they are a subset of events violating assumption 2. Moreover, the number of copies of an idle processor will not increase without bound as the arrival rate from occupied I-queues is proportional to the number of copies.

For the rest of the section, consider a system of n single-processor servers. Jobs arrive at the system in a Poisson process of rate $n\lambda$, $\lambda < 1$, hence the load on the system is λ. Let there be m dispatchers and a job is randomly directed to a dispatcher. The arrival process at each dispatcher is Poisson with rate $n\lambda/m$. The service times of jobs are drawn i.i.d. from a distribution $B(\cdot)$ with mean 1. It is easy to establish stability for the system with the `JIQ` algorithm, as the queue size distribution at the servers is dominated by that with the `Random` algorithm, which is known to be stable.

The system consists of a primary and secondary load balancing subsystems. The primary system consists of server queues with mean arrival rate λ and service rate 1. The secondary system consists of I-queues, where idle servers arrive with some *unknown* rate that depends on the distribution of the server queue length, and are "served" at rate $n\lambda/m$. There are two types of arrivals to a server: the arrivals assigned to an idle server from the I-queues and the arrivals randomly directed from a dispatcher with an empty I-queue. The latter arrivals form a Poisson process, but the former arrivals are not Poisson as it depends on the queueing process of I-queues, which has memory. This makes analysis challenging.

3.1 Analysis of Secondary Load Balancing System

For the secondary load balancing system, we are interested in the proportion of occupied I-queues when the system is in equilibrium, as it determines the rates of both arrivals through occupied I-queues and arrivals that are randomly directed. The two rates will determine the response time of the primary system. For all service time distributions, we have the following theorem.

Theorem 1 Proportion of occupied I-queues. *Let ρ_n be the proportion of occupied I-queues in a system with n servers in equilibrium. We show that $\rho_n \to \rho$ in expectation as $n \to \infty$, where for the `JIQ-Random` algorithm,*

$$\frac{\rho}{1-\rho} = r(1-\lambda), \tag{1}$$

for the `JIQ-SQ(d)` algorithm,

$$\sum_{i=1}^{\infty} \rho^{\frac{d^i-1}{d-1}} = r(1-\lambda). \tag{2}$$

We defer the full proof to the appendix and provide an outline below. The secondary load balancing system consists of m I-queues that serve virtual jobs corresponding to each idle server entering and leaving the I-queues. Since an idle server is removed at a job arrival, and the distribution of inter-arrival times is exponential, we observe that the system of I-queues have exponential "service" times. The expected proportion of occupied I-queues is equal to the (virtual) load on I-queues.

The load on I-queues depends on the arrival rate of idle servers, which in turn depends on the load on I-queues: A large load implies a larger proportion of occupied I-queues, hence a larger arrival rate to idle servers. This leads to improved load balancing on the primary system, and an increased arrival rate of idle servers to I-queues, or an increased load. To break this circular dependence, we observe that the expected proportion of idle servers goes to $1-\lambda$, regardless of the load on I-queues. As a result, the mean queue length of an I-queue goes to $r(1-\lambda)$. This is shown in Appendix A.1. Note that a better load balancing scheme in the secondary system does not change the queue lengths, but rather changes the rate of arrival, hence the expected proportion of occupied I-queues. In particular, a better load balancing algorithm induces a larger rate of arrival of idle processors to I-queues, which corresponds to shorter busy cycles in the primary system and shorter response times.

In order to relate the mean queue length of I-queues to the load, we need a better understanding of the queueing process at I-queues. The arrival process to I-queues is not Poisson in any finite system. However, we show in Appendix A.2 that the arrival process goes to Poisson as $n \to \infty$, with the `Random` or `SQ(d)` load balancing scheme in the secondary system. The expected proportion of occupied I-queues, ρ, is then obtained from the expression of mean queue length for the `Random` and `SQ(d)` algorithms. Note that the exponential distribution of the virtual service times at I-queues allows an explicit expression of the mean queue length in both cases. In particular, since `SQ(2)` produces most of the performance improvement with exponential service time distributions [11], probing two I-queues is sufficient.

Corollary 2 *In the large system limit, the arrival rate to idle servers with the `JIQ-Random` algorithm is $(r+1)$ times more than that to an occupied server.*

Proof. The arrival rate to idle servers equals

$$\frac{\lambda\rho}{1-\lambda} + \lambda(1-\rho) = \lambda(1-\rho)(r+1),$$

and the arrival rate to occupied servers equals $\lambda(1-\rho)$. ∎

The difference in arrival rates is even larger with the `SQ(2)` algorithm.

Fig. 4 shows the proportion of empty I-queues with $r = 10$ for different load balancing algorithms, both from the formulae given in Theorem 1 and simulations of a finite system with $n = 500$ and $m = 50$. We can see that the values obtained from Theorem 1 in the large system limit predict that of a finite system very well, as the coincidence of the markers with the corresponding curves is almost exact. The simulation results also verify that the proportion of empty I-queues is invariant with different service time distributions of the actual jobs, as predicted by Theorem 1.

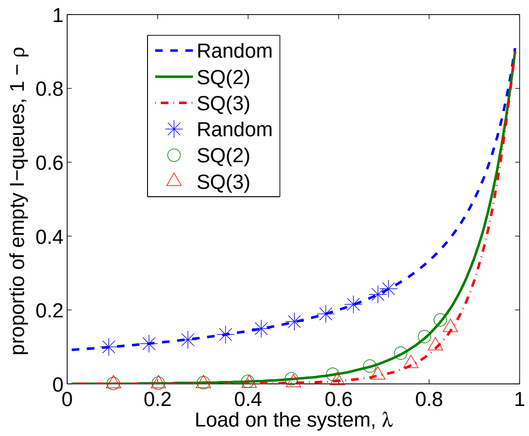


Figure 4: Proportion of empty I-queues with $r = 10$. Line curves are obtained from Theorem 1. Markers are from simulations with $n = 500$ and $m = 50$. The values for different service time distributions are indistinguishable.

We observe a significant reduction in the proportion of empty I-queues with a better load balancing algorithm such as `SQ(2)`. At moderate load, there are a large number of idle processors and the `SQ(d)` algorithms result in the proportion of empty I-queues being close to 0. For instance, at $\lambda = 0.6$, the proportion of empty I-queues is 0.2 under the `Random` algorithm, but only 0.027 under the `SQ(2)` algorithm. The further reduction of the proportion of empty I-queues by `SQ(3)` over `SQ(2)` is not as significant.

At high load, the number of idle processors becomes small. In particular, at $\lambda > 0.9$, there are fewer idle processors than the number of I-queues, and the effect of better load balancing diminishes. The proportion of empty I-queues under the `SQ(d)` algorithms converges to that under the `Random` algorithm as the load goes to 1. The I-queues will be better utilized at high load if servers report when they have a light load, instead of being absolutely idle. We explore this extension with simulation in Section 5.

3.2 Analysis of Primary Load Balancing System

Using the results from the secondary load balancing system, we can solve for the response time of the primary system. Let

$$s = \lambda(1 - \rho),$$

where ρ is the proportion of occupied I-queues computed in Theorem 1.

Theorem 2 Queue Size Distribution. *Let the random variable Q_n denote the queue size of the servers of an n-system in equilibrium. Let Q_s denote the queue size of a M/G/1 server at the reduced load s with the same service time distribution and service discipline, then*

$$\mathbb{P}(Q_n = k) \;\rightarrow\; \frac{\mathbb{P}(Q_s = k)}{1 - \rho} \;\;\forall\, k \geq 1 \;\; \textit{as} \;\; n \rightarrow \infty. \tag{3}$$

Corollary 3 Mean Response Time. *Let the mean queue size at the servers in the large system limit be $\bar{q}$. It is given by*

$$\bar{q} \;=\; \frac{q_s}{1 - \rho}, \tag{4}$$

with q_s being the mean queue size of the M/G/1 server with the same service time distribution and service discipline.

The mean response time

$$\bar{T} = \frac{\bar{q}}{\lambda} = \frac{q_s}{s},$$

assuming a mean service time of 1.

Corollary 4 Insensitivity. *The queue size distribution of the* `JIQ` *algorithm with PS service discipline in the large system limit depends on the service time distributions only through its mean.*

We defer the proof of Theorem 2 to the appendix and provide an outline below. Recall that there are two types of arrivals to the processors: one arrival process occurs through the I-queues and only when the processor is idle. The other arrival process occurs regardless of the processor queue length, when a job is randomly dispatched. The rate of each type of arrivals depends on the proportion of occupied I-queues, ρ. With probability ρ, an incoming job finds a occupied I-queue, and with probability $1-\rho$, it finds an empty I-queue. Hence the arrival process due to random distribution is Poisson with rate $s = \lambda(1-\rho)$.

Let the arrivals at empty I-queues be colored green, and the arrivals at occupied I-queues colored red. For an idle processor, the first arrival can be red or green, but the subsequent arrivals can only be green. The arrival process of the green jobs is Poisson with rate s, but the arrival process of the red jobs is not Poisson. However, observe that a busy period of a server in the `JIQ` system is indistinguishable from a busy period of an M/G/1 server with load s. Hence, the queue size distribution differs from that of an M/G/1 server with load s only by a constant factor λ/s.

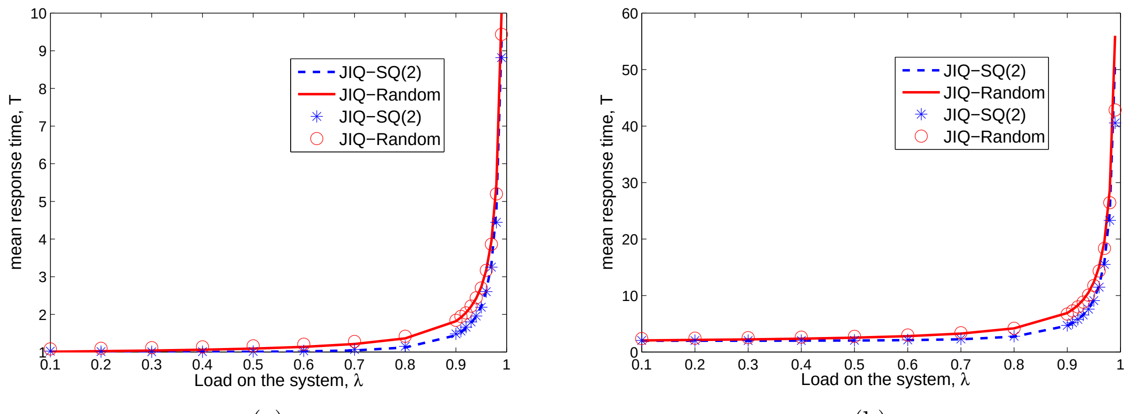


Figure 5: Mean response time for the `JIQ-Random` and `JIQ-SQ(2)` algorithms with $r = 10$. Fig. (a) has an exponential service time distribution with mean 1 (this makes the minimum possible mean response time 1), and Fig. (b) has a Weibull service time distribution with mean 2 and variance 20 (minimum possible mean response time is 2) and FIFO service discipline. Line curves are obtained from Theorem 2. Markers are from simulations with $n = 500$ and $m = 50$.

Fig. 5(a) shows the mean queue size for general service disciplines with exponential service time distributions and Fig. 5(b) shows the mean queue size for FIFO service discipline with Weibull service time distribution with mean 2 and variance 20. In both cases, the computed curve fits well with the simulation results in a system of 500 servers. The error bars are not plotted as the largest standard deviation is smaller than 0.001.

Contrary to the improvement in ρ observed in Fig. 4, the performance gain of `JIQ-SQ(2)` over `JIQ-Random` is not significant at moderate load, and the magnitude of reduction in response time remains small even at higher load with $r = 10$. For instance, the reduction in response time is only 17% at $\lambda = 0.9$ in Fig. 5(a). This is because improvement of `JIQ-SQ(2)` over `JIQ-Random` is most conspicuous when the random arrivals incur large queue sizes *and* there is significant improvement in ρ. This is expected to happen at high load with a big processor to I-queue ratio, r. With $r = 10$, the queue sizes incurred by random arrivals is small at low loads, and the number of idle processors per I-queue is too small at high loads, resulting in similar mean response times for `JIQ-Random` and `JIQ-SQ(2)`.

4 Design and Extension

In this section, we seek to provide understanding of the relationship between the analytical results and system parameters, and to discuss extensions with local estimation of load at the servers.

4.1 Reduction of Queueing Overhead

We can compute the reduction of queueing overhead by `JIQ-Random` as Eqn. (1) gives an explicit expression for ρ. We compute q_s for exponential service time distributions and general service discipline,

$$q_s = \frac{s}{1-s}. \tag{5}$$

Eqn. (5) also holds for PS service discipline and general service time distributions in the large system limit due to insensitivity. The mean response time for `JIQ-Random` is hence

$$\frac{1}{1-s} = \frac{1}{1-\frac{\lambda}{1+r(1-\lambda)}} = 1 + \frac{\lambda}{(1-\lambda)(1+r)}.$$

Compare this to the mean response time of a M/M/1 queue with rate-λ arrival

$$\frac{1}{1-\lambda} = 1 + \frac{\lambda}{1-\lambda},$$

since the mean service time of a job is 1, the queueing overhead is $\frac{\lambda}{1-\lambda}$ for `Random` and $\frac{\lambda}{(1-\lambda)(1+r)}$ for `JIQ-Random`. This is a $(1+r)$-fold reduction.

For the FIFO service discipline, by the Pollaczek-Khinchin formula,

$$q_s = s + s^2\frac{1+C^2}{2(1-s)}, \tag{6}$$

where $C^2 = \sigma^2/(\bar{x})^2$, the ratio of the variance of the particular service time distribution to its mean squared.

The mean response time for `JIQ-Random` with FIFO is

$$1 + s\frac{1+C^2}{2(1-s)} = 1 + \frac{1+C^2}{2}\frac{\lambda}{(1-\lambda)(1+r)}.$$

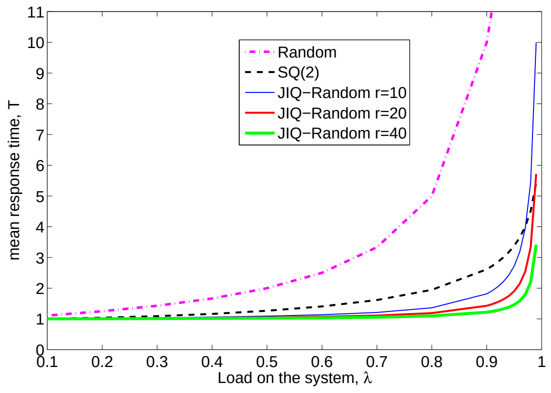
Compare this to the mean response time of a M/G/1 queue with rate-λ arrival

$$1 + \lambda\frac{1+C^2}{2(1-\lambda)} = 1 + \frac{1+C^2}{2}\frac{\lambda}{1-\lambda}.$$

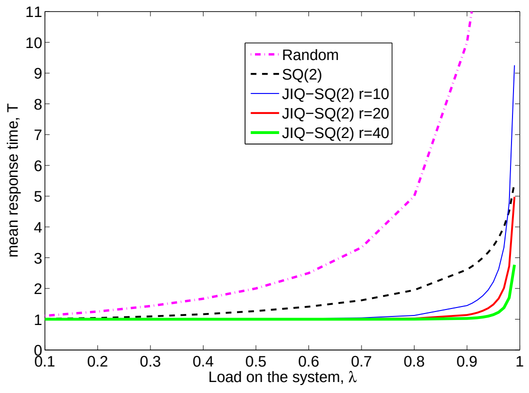
Again, we observe a $(1+r)$-fold reduction in queueing overhead. With a larger C for general service time distributions, the absolute saving in queueing overhead from `SQ(2)` will be much more significant. We are not showing the comparison in explicit forms as we do not have an explicit expression of the mean queue size of `SQ(2)` for general service time distributions. The performance of `JIQ-SQ(2)` is plotted with ρ obtained numerically in Fig. 6.

Fig. 6(a) compares the computed mean response time for `Random`, `SQ(2)` and `JIQ-Random` with $r = 10$, 20 and 40. Fig. 6(b) compares that with `JIQ-SQ(2)`, with ρ obtained numerically. The `JIQ` algorithms have similar performance as `SQ(2)` at low load, but outperform it significantly at medium to high load. At very high load such as 0.99, however, the `JIQ` algorithms do not perform as well as `SQ(2)` due to the lack of idle servers in the system. We propose the following extension to alleviate the problem.

Extension to `JIQ`. At very high load, a server reports to the I-queues when it is lightly loaded. For instance, a server can report to one I-queue when it has one job in the queue and report again when it becomes idle. This will insert one copy of all servers with one job and two copies of all idle servers in the I-queues, and further increase the arrival rate to idle servers with zero or one job.



(a)



(b)

Figure 6: Fig. (a) compares the computed mean response time for `Random`, `SQ(2)` and `JIQ-Random` with $r = 10$, 20 and 40. Fig. (b) compares the mean response time for `Random`, `SQ(2)` and `JIQ-SQ(2)` with $r = 10$, 20 and 40. Both figures are for PS service discipline and general service time distributions.

The advantage of delegating the decisions on arrival rates to servers is that the load information is readily available at each server. It is much harder for a dispatcher to estimate the load on the system as the service times are not available. It is also easy for heterogeneous servers to respond differently based on local load.

To determine the reporting threshold, the rule of thumb is half of the resulting mean queue size. Decreasing the reporting threshold will increase the rate of random arrivals, which results in a larger queue size. On the other hand, increasing the reporting threshold will attract too many arrivals at once to an idle server and result in unbalanced load. The mean queue size with a threshold other than one remains difficult to analyze. We evaluate this extension using simulation in Section 5.

Comparison of complexity. The complexity of `JIQ-Random` is strictly smaller than that of `SQ(2)`, where the only communication is by the idle processors joining a random I-queue. There is no back and forth communication between dispatchers and processors. The `JIQ-SQ(2)` algorithm has exactly the same complexity as `SQ(2)`. Both `JIQ-Random` and `JIQ-SQ(2)` have all communication off the critical path.

4.2 Implementation Discussion

The `JIQ` algorithm is simple to implement with asynchronous message passing between idle processors and I-queues. Persistent TCP connections between dispatchers and processors for job dispatching can also be used for transmitting the messages. No synchronization is needed among the dispatchers and I-queues as an idle processor is allowed to receive randomly dispatched jobs even when it has joined an I-queue.

The system is easy to scale by adding servers to the cluster. Dispatchers, I-queues and processors can scale independently and the numbers of each can depend on the nature of applications, the style of requests and the load on the system. Although the analysis is performed with one I-queue attached to each dispatcher, it is possible to implement several dispatchers sharing the same I-queue. For instance, in a multi-processor system, each dispatcher can be implemented on an independent processor co-located with the I-queue processor and sharing memory between them. Analytically, this is the same as aggregating several dispatchers to form one with larger bandwidth, and the ratio r only depends on the number of processors and number of I-queues.

5 Evaluation

We evaluate the class of `JIQ` algorithms against the `SQ(d)` algorithm for a variety of service time distributions via simulation. Note that the mean response time for the `SQ(d)` algorithm with general service time distributions does not have an explicit expression, hence we can only simulate its performance.

We choose the following service time distributions that occur frequently in practice. This is the same as the distributions simulated in [8]. To simplify the control of the variance of service time distributions, we let *all distributions have mean* 2, and they are listed in increasing order of C^2.

Service time distributions.

1. `Deterministic`: point mass at 2 (variance = 0)
2. `Erlang2`: sum of two exponential random variables with mean 1 (variance = 2)
3. `Exponential`: exponential distribution with mean 2 (variance = 4)
4. `Bimodal-1`: (mean = 2,variance = 9)

$$X = \begin{cases} 1 & w.p. \;\; 0.9 \\ 11 & w.p. \;\; 0.1 \end{cases}$$

5. `Weibull-1`: Weibull with shape parameter = 0.5 and scale parameter = 1 (heavy-tailed, mean = 2, variance = 20)
6. `Weibull-2`: Weibull with shape parameter = $\frac{1}{3}$ and scale parameter = $\frac{1}{3}$ (heavy-tailed, mean = 2, variance = 76)
7. `Bimodal-2`: (mean = 2,variance = 99)

$$X = \begin{cases} 1 & w.p. \;\; 0.99 \\ 101 & w.p. \;\; 0.01 \end{cases}$$

The deterministic distribution models applications with constant job sizes. The Erlang and exponential distributions model waiting time between telephone calls. The bimodal distributions model applications where jobs have two distinct sizes. The Weibull distribution is a heavy-tailed distribution with finite mean and variance. It has a decreasing hazard rate, *i.e.*, the longer it is served, the less likely it will finish service. The heavy-tailed distribution models well many naturally occurring services, such as file sizes [4].

We evaluate the `JIQ-Random` and `JIQ-SQ(2)` algorithms against the `SQ(2)` algorithm, since the `JIQ` algorithms evaluated have strictly lower communication overhead than `SQ(2)`. Moreover, the overhead does not interfere with job arrivals, as in the case of `SQ(2)`.

For readability of the figures, the labels for the `JIQ` algorithms are shortened with 'R' standing for Random and 'S' standing for SQ(2). The number after the letter is the value of r, the ratio of number of processors to number of dispatchers. The experiments with $r = 10$ are run with 500 processors and 50 I-queues, $r = 20$ with 500 processors and 25 I-queues, and $r = 40$ with 600 processors and 15 I-queues. We simulate a moderate load $\lambda = 0.5$ and a high load $\lambda = 0.9$.

Fig. 7 compares the performance of the algorithms with the seven service time distributions listed above. We simulate two service disciplines, PS and FIFO, and the results are very different. Note that $T = 2$ is the minimum mean response time achievable as all the jobs have mean service time 2.

1. `JIQ-Random` outperforms `SQ(2)`.

For both service disciplines and both loads, `JIQ-Random` consistently outperforms `SQ(2)`. Consider the reduction in queueing overhead. We tabulate the percentage of improvement of the `JIQ-Random` algorithms over `SQ(2)` for the distribution `Bimodal-2`, which has the largest variance. Let t_j be the response time of `JIQ-Random` and t_s be that of `SQ(2)`, as obtained from Fig. 7, the percentage of improvement in queuing overhead is computed as

$$\frac{(t_s - 2) - (t_j - 2)}{t_s - 2},$$

where 2 is the mean service time.

Table 1 shows that the `JIQ-Random` algorithm with $r = 10$ reduces queueing overhead by at least 33.2% from the `SQ(2)` algorithm, as for FIFO service discipline at load 0.9. For the PS service discipline, the improvement is almost 2-fold. The `JIQ-Random` algorithm achieves a 4-fold $(1/(1-73.3\%) \simeq 4)$ reduction with $r = 20$ and a 7-fold $(1/(1-85.9\%) \simeq 7)$ reduction with $r = 40$, for PS discipline at load 0.9. It achieves a 3-fold $(1/(1-65.2\%) \simeq 3)$ reduction with $r = 20$ and a 5-fold $(1/(1-81.2\%) \simeq 5)$ reduction with $r = 40$, for FIFO discipline at load 0.9.

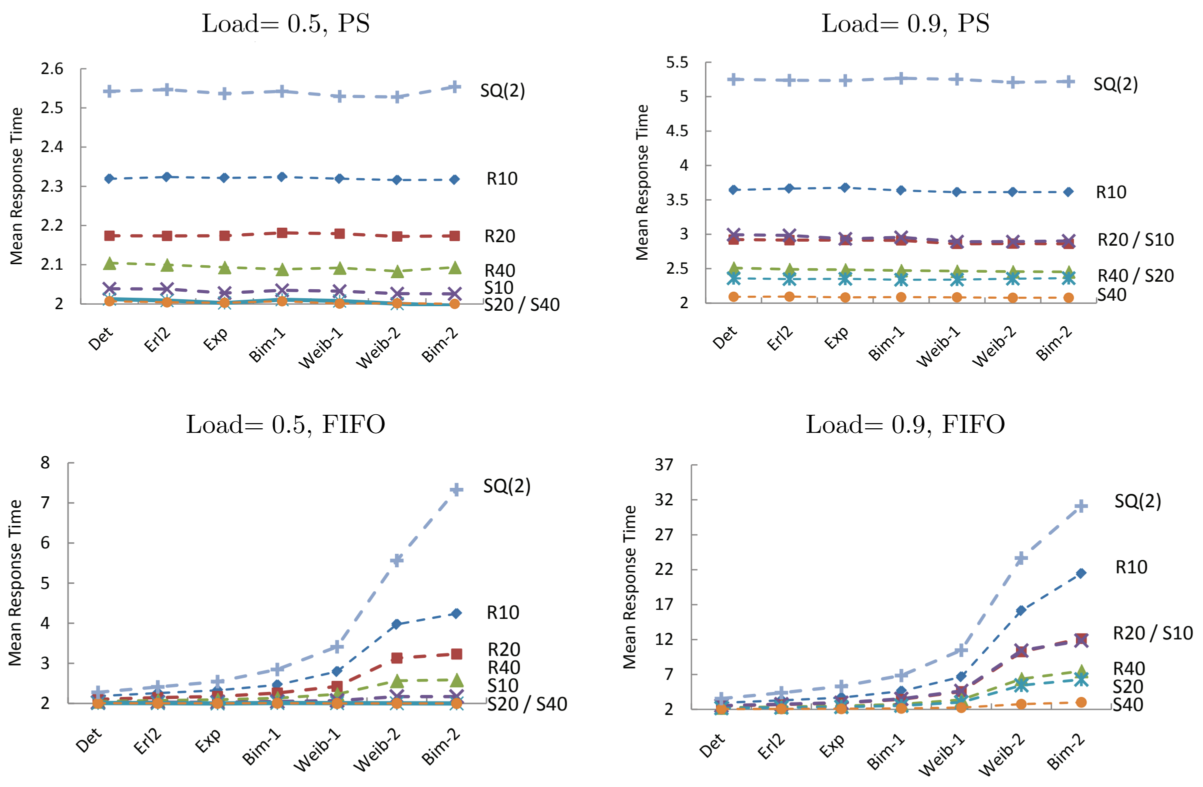


Figure 7: Mean response time comparison for `SQ(2)`, `JIQ-Random` and `JIQ-SQ(2)`, with 7 different service time distributions. The smallest possible mean response time is 2 with a mean service time of 2.

	PS		FIFO	
	0.5	0.9	0.5	0.9
R10	42.8%	49.9%	58.0%	33.2%
R20	68.7%	73.3%	76.9%	65.2%
R40	83.1%	85.9%	88.9%	81.2%

Table 1: Percentage of improvement in queuing overhead of `JIQ-Random` over `SQ(2)`.

2. `JIQ-SQ(2)` achieves close to minimum response time.

At a load of 0.5, the `JIQ-SQ(2)` algorithm achieves a mean queueing overhead less than 5% of the mean service time for the PS service discipline. For both disciplines, the mean response times with $r = 10$ never exceed 2.2, and those with $r = 20$ and $r = 40$ are *essentially* 2.

At a load of 0.9, for the PS service discipline, the mean response time of the `JIQ-SQ(2)` algorithm remains close to 2. At $r = 10$, it is around 2.9, and at $r = 40$, it never exceeds 2.1. The mean response time varies more under the FIFO service discipline. Even there, the `JIQ-SQ(2)` algorithm has mean response time below 3 for all service time distributions. This represents a **30-fold reduction** ($3.3/0.01 \simeq 30$ for PS and $30/1 \simeq 30$ for FIFO) for both disciplines at $r = 40$ and 0.9 load.

3. The `JIQ` algorithms are near-insensitive with PS in a finite system.

Based on the simulation, the `JIQ` algorithms are near-insensitive to the service time distributions under the PS service discipline. We showed in Section 3 that the response times are insensitive to the service time distributions in the large system limit. The simulation verifies that in a system of $500 - 600$ processors, the mean response times do not vary with service time distributions.

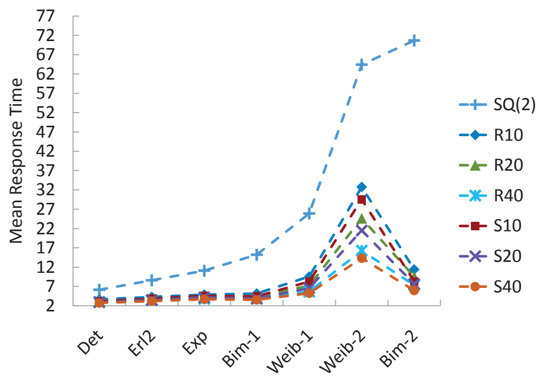


Figure 8: Mean response time comparison for `SQ(2)`, `JIQ-Random` and `JIQ-SQ(2)` with reporting threshold equal to two, with 7 different service time distributions. The smallest possible mean response time is 2 with a mean service time of 2. Note that the mean queue lengths are not monotonically increasing with variance of service times.

`JIQ` algorithms with extension.
We evaluate the extension of the `JIQ` algorithms with reporting threshold equal to two at a high load of 0.99. This is the region where the performance of the original `JIQ` algorithms is similar to that of `SQ(2)`, as shown in Fig. 6. However, with reporting threshold equal to two, the `JIQ` algorithms significantly outperforms `SQ(2)`. For instance, with exponential distribution, for which service disciplines do not affect response times, `SQ(2)` outperforms `JIQ-Random` with threshold equal to one in Fig. 6, but is outperformed by `JIQ-Random` with $r = 10$ and threshold equal to two, with 88% reduction in queueing overhead.

Observe the interesting phenomenon that the mean queue sizes are no longer monotonically increasing with variance of service times. In particular, the two bimodal distributions have smaller mean queue sizes than distributions with smaller variance. For the bimodal distributions with variance 99, `JIQ-Random` with $r = 10$ reduces the mean queue size from that of `SQ(2)` by 89%, and `JIQ-SQ(2)` with $r = 40$ reduces the mean queue size from that of `SQ(2)` by 97.1%. On the other hand, for the Weibull distribution with variance 76, `JIQ-SQ(2)` with $r = 40$ reduces the mean queue size from that of `SQ(2)` by only 83%. Apparently higher moments of the distribution start to have an effect when the reporting threshold is more than 1. We defer the analysis of the extended algorithm to future work.

6 Conclusion and Future Work

We proposed the `JIQ` algorithms for web server farms that are dynamically scalable. The `JIQ` algorithms significantly outperform the state-of-the-art `SQ(d)` algorithm in terms of response time at the servers, while incurring no communication overhead on the critical path. The overall complexity of `JIQ` is no greater than that of `SQ(d)`.

The extension of the `JIQ` algorithms proves to be useful at very high load. It will be interesting to acquire a better understanding of the algorithm with a varying reporting threshold. We would also like to understand better the relationship of the reporting frequency to response times, as well as an algorithm to further reduce the complexity of the `JIQ-SQ(2)` algorithm while maintaining its superior performance.

References

[1] N. Ahmad, A. G. Greenberg, P. Lahiri, D. Maltz, P. K. Patel, S. Sengupta, and K. V. Vaid. *DISTRIBUTED LOAD BALANCER.* Google Patents, Aug. 2008. US Patent App. 12/189,438.

[2] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proc. 35th IEEE Conference on Foundations of Computer Science*, pages 356–368, 1994.

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

[4] M. Crovella and A. Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Trans. Networking*, 5(6):835–846, 1997.

[5] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Proc. ACM Conf. on High Performance Computing Networking, Storage and Analysis*, 2009.

[6] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. on Software Engineering*, SE-12(5), May 1986.

[7] C. Graham. Chaoticity on path space for a queueing network with selection of the shortest queue among several. *Journal of Appl. Prob.*, 37:198–211, 2000.

[8] V. Gupta, M. Harchol-Balter, K. Sigman, and W. Whitt. Analysis of join-the-shortest-queue routing for web server farms. *Perforance Evaluation*, (64):1062–1081, 2007.

[9] M. Luczak and C. McDiarmid. On the maximum queue length in the supermarket model. *The Annals of Probability*, 34(2):493–527, 2006.

[10] M. Mitzenmacher. The power of two choices in randomized load balancing. *Ph.D. thesis, Berkeley*, 1996.

[11] M. Mitzenmacher. Analyses of load stealing models based on differential equations. In *Proc. 10th ACM Symposium on Parallel Algorithms and Architetures*, pages 212–221, 1998.

[12] J. Nair, A. Wierman, and B. Zwart. Tail-robust scheduling via limited processor sharing. *Performance Evaluation*, page 1, 2010.

[13] K. Salchow. Load balancing 101: Nuts and bolts. *White Paper, F5 Networks, Inc.*, 2007.

[14] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes and http chunking in web search. *O'Reilly Velocity Web performance and operations conference*, June 23rd 2009.

[15] M. S. Squillante and R. D. Nelson. Analysis of task migration in shared-memory multiprocessor scheduling. In *Proc. ACM Conference on the Measurement and Modeling of Computer Systems*, pages 143–155, 1991.

[16] N. D. Vvedenskaya, R. L. Dobrushin, and F. I. Karpelevich. Queueing system with selection of the shortest of two queues: An asymptotic approach. *Probl. Inf. Transm*, 32(1):20–34, 1996.

[17] Y.-T. Wang and R. Morris. Load sharing in distributed systems. *IEEE Trans. on Computer*, c-34(3), March 1985.

Appendix

A. Proof for Secondary Load Balancing System

We need the following lemmas for the proof of Theorem 1.

Lemma 1 *Consider a M/G/n system with arrival rate $n\lambda$. Let the random variable μ_n denote the number of idle servers in equilibrium. Let $q_n = \frac{\mu_n}{n}$, then*

$$\mathbb{E}(|q_n - (1-\lambda)|) \to 0 \quad as \quad n \to \infty. \tag{7}$$

Proof. Since the system load is λ, we have $\mathbb{E}(\mu_n) = (1-\lambda)n$ for all n. Let the number of occupied servers be $X_n = n - \nu_n = Q_n \wedge n$, where Q_n is the number of jobs in the equilibrium system. In order to show the lemma, it is sufficient to show that for any ϵ, there exists n_0 such that for all $n > n_0$,

$$\mathbb{E}(X_n - n\lambda)^2 \to 2n\lambda.$$

Consider a M/G/∞ process with arrival rate $n\lambda$. Let the number of occupied servers be Y, whose distribution is Poisson with mean $n\lambda$. We can bound the probability

$$\begin{aligned}
\mathbb{P}(Y \geq n) &= \sum_{k=n}^{\infty} \frac{(n\lambda)^k e^{-n\lambda}}{k!} \leq \sum_{k=n}^{\infty} \frac{(n\lambda)^k e^{-n\lambda}}{\left(\frac{k}{e}\right)^k} = \sum_{k=n}^{\infty} \left(\frac{n\lambda e}{k}\right)^k e^{-n\lambda} \\
&= \sum_{k=n}^{\lfloor 2n\lambda e \rfloor} \left(\frac{n\lambda e}{k}\right)^k e^{-n\lambda} + \sum_{k=\lceil 2n\lambda e \rceil}^{\infty} \left(\frac{n\lambda e}{k}\right)^k e^{-n\lambda} \\
&\leq (2\lambda e - 1)\, n\, (\lambda e)^n\, e^{-n\lambda} + \frac{1}{2^{2n\lambda e - 1}} e^{-n\lambda} \\
&\leq C_1 n e^{-nC_2}
\end{aligned}$$

for large enough n and some constants C_1, C_2 not depending on n. Since $\sum_{n=0}^{\infty} \mathbb{P}(Y \geq n) < \infty$, by the Borel-Cantelli lemma, we have that $Y < n$ infinitely often. With a standard coupling, and since $X_n = Y$ if $Y < n$, we can show that for each sample path, $X_n = Y$ when n is large enough, hence $\mathbb{E}(X_n - Y)^2 \to 0$.

We can now bound the variance of X_n. For some $\epsilon_n \to 0$,

$$\begin{aligned}
\mathbb{E}(X_n - n\lambda)^2 &\leq 2\mathbb{E}(Y - n\lambda)^2 + 2\mathbb{E}(X_n - Y)^2 \\
&\leq 2n\lambda + \epsilon_n \to 2n\lambda \quad \text{as} \quad n \to \infty.
\end{aligned}$$

This proves Lemma 1. ∎

Lemma 2 *Consider a system of n parallel servers with arrival rate $n\lambda$ and the following assignment algorithm. Let $\rho_n(t)$ be a function of t on $[0,1]$. For a job arriving at the system at time t, with probability $1-\rho_n(t)$, it joins a random processor. With probability $\rho_n(t)$, it joins an idle processor if one exists and joins a random processor otherwise. Let the random variable $\nu_n(t)$ denote the number of idle processors at time t in the n-system. Let $p_n(t) = \frac{\nu_n(t)}{n}$, then*

$$\mathbb{E}(|p_n(t) - (1-\lambda)|) \to 0 \quad \text{as} \quad n \to \infty \quad \text{and} \quad t \to \infty, \tag{8}$$

independent of the function $\rho_n(t)$.

Proof. Since the load on the system is λ, we have $\mathbb{E}(p_n(t)) = 1 - \lambda$ as $t \to \infty$, which is independent of $\rho_n(t)$. We examine the term $Var(p_n(t))$. We color the jobs directed to random processors (with probability $1-\rho_n(t)$) green and the jobs directed to idle processors (with probability $\rho_n(t)$) red. Observe that variance of the proportion of occupied processors depends on $\rho_n(t)$ as the arrivals of green jobs are independent, but the arrivals of red jobs are dependent. In particular, with $\rho_n(t) = \rho$, let the proportion of processers occupied by green jobs as t becomes large be $u^g_{n,\rho}$ and that by red jobs be $u^r_{n,\rho}$. Hence, $\mathbb{E}u^g_{n,\rho} = (1-\rho)\lambda$ and

$$Var(u^g_{n,\rho}) = \frac{n(1-(1-\rho)\lambda)(1-\rho)\lambda}{n^2} = \frac{(1-(1-\rho)\lambda)(1-\rho)\lambda}{n}.$$

As red jobs are directed to idle processors, the system with red jobs only is a M/G/k system with $k = n(1-u^g_{n,\rho})$ being a random number. As k becomes large, by Lemma 1, the variance

$$Var(u^r_{n,\rho}) \leq \frac{2\lambda\rho}{n}$$

independent of k. Together we have

$$\begin{aligned}
\lim_{t\to\infty} Var(p_n(t)) &\leq \sup_{\rho} \{\, Var(u^g_{n,\rho}) + Var(u^r_{n,\rho}) \,\} \\
&\leq \frac{2\lambda}{n} \to 0 \quad \text{as} \quad n \to \infty.
\end{aligned}$$

This proves Lemma 2. ∎

Fix $m/n = r$, where m is the number of dispatchers.

Corollary 5 *Let the random variable σ_n denote the average queue length of the m I-queues as $t \to \infty$, and let a_n denote the total arrival rate of idle servers at the I-queues. Note that the arrival process is not necessarily Poisson. We have*

$$\mathbb{E}(|\sigma_n - r(1-\lambda)|) \to 0 \quad as \quad n \to \infty, \tag{9}$$

and there exists a constant a such that

$$\mathbb{E}(|\frac{a_n}{m} - a|) \to 0 \quad as \quad n \to \infty. \tag{10}$$

Proof. Eqn. (9) follows directly from Eqn. (8) as $\sigma_n = \frac{p_n n}{m} = p_n r$. Since the average I-queue length goes to $r(1-\lambda)$ as n goes to infinity, and the service rate of each I-queue is constant at $\frac{n\lambda}{m} = r\lambda$, the average arrival rate for each I-queue becomes constant. ■

Lemma 3 *Let the arrival process at I-queue 1 be $\Lambda_n(t)$. For `JIQ-Random`, $\Lambda_n(t)$ goes to a Poisson process as n goes to infinity. For `JIQ-SQ(2)`, $\Lambda_n(t)$ goes to a state-dependent Poisson process where the rate depends on the I-queue lengths.*

Proof. First we show that $\Lambda_n(t)$ goes to a Poisson process for `JIQ-Random`. The rate at which processors become idle depends on the distribution of queue sizes and remaining service time of jobs in the queues, and is difficult to compute. However, from Corollary 5, we have that the arrival rate of idle processors at an I-queue goes to a constant value as n goes to infinity.

We show that for arbitrary times t_0 and t_1, with $t_0 < t_1$, $\Lambda_n(t_1) - \Lambda_n(t_0)$ goes to a Poisson distribution with mean $a(t_1 - t_0)$. This implies stationarity and independence of increments for the point process. For a given ϵ, there exists n_0 large enough such that for all $n > n_0$,

$$\mathbb{P}(|a_n(t_1 - t_0) - ma(t_1 - t_0)| > o(m)) < \epsilon,$$

and $a(t_1 - t_0) \le n(t_1 - t_0)$. Without loss of generality, assume that $ma(t_1 - t_0)$ and $n(t_1 - t_0)$ are integers. Denote the number of idle processors joining I-queue 1 by N_n.

$$\begin{aligned} \mathbb{P}(N_n = k) \quad &\le \quad (1-\epsilon)\binom{ma(t_1-t_0)+o(m)}{k}\left(\frac{1}{m}\right)^k\left(1-\frac{1}{m}\right)^{ma(t_1-t_0)+o(m)-k} \\ &\quad +\epsilon\binom{n(t_1-t_0)}{k}\left(\frac{1}{m}\right)^k\left(1-\frac{1}{m}\right)^{n(t_1-t_0)-k} \to \frac{e^{-a(t_1-t_0)}(a(t_1-t_0))^k}{k!}, \end{aligned}$$

as we let $\epsilon \to 0$ and $n, m \to \infty$. Similarly,

$$\mathbb{P}(N_n = k) \quad \ge \quad (1-\epsilon)\binom{ma(t_1-t_0)+o(m)}{k}\left(\frac{1}{m}\right)^k\left(1-\frac{1}{m}\right)^{ma(t_1-t_0)+o(m)-k} \to \frac{e^{-a(t_1-t_0)}(a(t_1-t_0))^k}{k!}.$$

Hence, the distribution of N_n goes to Poisson with mean $a(t_1 - t_0)$.

For the `JIQ-SQ(d)` algorithm, consider the number of arrivals that choose I-queue 1 as one of the d choices, which we call *potential arrivals*. A similar argument shows that the potential arrival process goes to Poisson with mean $da(t_1 - t_0)$. The potential arrival process is further thinned by a coin toss whose probability depends on the I-queue length, yielding a state-dependent Poisson process[3]. ■

Proof of Theorem 1.

From Lemma 3, we conclude that each I-queue has the same load and a Poisson arrival process. Let it be u. Since the service time distribution of each I-queue is exponential, we can compute the mean I-queue length for `JIQ-Random` and `JIQ-SQ(2)` respectively. Corollary 5 yields that the mean I-queue length goes to $r(1-\lambda)$ hence we have $\frac{u}{1-u} = r(1-\lambda)$ for `JIQ-Random` and $\sum_{i=1}^{\infty} u^{\frac{d^i-1}{d-1}} = r(1-\lambda)$ [16] for `JIQ-SQ(2)`.

Lemma 3 also implies the asymptotic independence of the arrival process for `JIQ-Random` and the potential arrival process for `JIQ-SQ(2)`. The asymptotic independence of the queue size distribution for `JIQ-Random` follows directly and that for `JIQ-SQ(2)` follows from [3]. Hence the proportion of occupied I-queues $\rho_n \to \rho = u$ as $n \to \infty$. This shows Eqn. (1) and (2). ■

B. Proof for Primary Load Balancing System

We prove Theorem 2 in this section. We need the following lemmas.

Lemma 4 *Let Λ_n be the arrival process of randomly directed jobs at server 1. The point process Λ_n is Poisson with rate $s_n \to s = \lambda(1-\rho)$.*

Proof. A job arrives at a empty I-queue with probability $1-\rho_n$, where ρ_n is a random variable denoting the proportion of occupied I-queues. The job is randomly directed with probability $1-\rho_n$, and arrives at server 1 with probability $1/n$. Hence the process of randomly directed jobs is a result of thinning the rate-$n\lambda$ Poisson process with a coin toss with varying probability $(1-\rho_n)/n$. Since each coin toss is independent, and the expected number of arrivals in a period of length t is $\lambda(1-\mathbb{E}\rho_n)t$, the random arrival process is Poisson with rate $s_n = \lambda(1-\mathbb{E}\rho_n)$. Since $\rho_n \to \rho$ in expectation, $\mathbb{E}\rho_n \to \rho$ and $s_n \to s$. This proves the lemma. ■

Lemma 5 *Consider a server with load λ. The arrival rate when the server idle is unknown and not necessarily Poisson. The arrival process when the server is occupied is Poisson with rate s. Let Q be the equilibrium queue size of the server and Q_s be the equilibrium queue size of a rate-s M/G/1 queue, then*

$$\mathbb{P}(Q=k) = \frac{\lambda\mathbb{P}(Q_s=k)}{s}. \tag{11}$$

Proof. Consider a busy period for this server and a busy period of an M/G/1 queue. Observe that a busy period starts when an arrival enters the idle server. Since the arrival process to the server is Poisson with rate s once it is occupied, we can show that the queue size distribution within a busy period is the same as that of an M/G/1 queue with rate s, using a standard coupling. Hence, $\mathbb{P}(Q=k|Q>0) = \mathbb{P}(Q_s=k|Q_s>0)$. Since the server has load λ, $\mathbb{P}(Q>0)=\lambda$ and $\mathbb{P}(Q_s>0)=s$. Hence

$$\begin{aligned}\mathbb{P}(Q=k) &= \mathbb{P}(Q=k|Q>0)\mathbb{P}(Q>0) = \mathbb{P}(Q_s=k|Q_s>0)\mathbb{P}(Q>0)\\ &= \frac{\mathbb{P}(Q_s=k)}{\mathbb{P}(Q_s>0)}\mathbb{P}(Q>0) = \frac{\lambda\mathbb{P}(Q_s=k)}{s}.\end{aligned}$$

This proves Lemma 5. ■

Proof of Theorem 2.
Using Lemma 5 and $s_n = \lambda(1-\mathbb{E}\rho_n)$, we have

$$\mathbb{P}(Q_n=k) = \frac{\lambda\mathbb{P}(Q_{s_n}=k)}{s_n} = \frac{\mathbb{P}(Q_{s_n}=k)}{1-\mathbb{E}\rho_n}.$$

Using Lemma 3, $s_n \to s = \lambda(1-\rho)$, we have

$$\mathbb{P}(Q_n=k) \to \frac{\lambda\mathbb{P}(Q_s=k)}{s} = \frac{\mathbb{P}(Q_{s_n}=k)}{1-\rho}.$$

This proves Theorem 2. ■