Performance Modeling and Design of Computer Systems

Queueing Theory in Action

Mor Harchol-Balter

Book DOI: http://dx.doi.org/10.1017/CBO9781139226424

Online ISBN: 9781139226424

Hardback ISBN: 9781107027503

Chapter

24 - Task Assignment Policies for Server Farms pp. 408-432

Chapter DOI: http://dx.doi.org/10.1017/CBO9781139226424.031

Cambridge University Press

CHAPTER 24

Task Assignment Policies for Server Farms

In this chapter we revisit server farms, however this time in the context of high-variability job sizes (indicative of the workloads described in Chapter 20), rather than Exponential job sizes.

The server farm architecture is ubiquitous in computer systems. Rather than using a single, powerful server to handle all incoming requests (assuming such a beast can even be purchased), it is more cost efficient to buy many slow, inexpensive servers and pool them together to harness their combined computational power. The server farm architecture is also popular for its flexibility: It is easy to add servers when load increases and easy to take away servers when the load drops. The term **server farm** is used to connote the fact that the servers tend to be co-located, in the same room or even on one rack.

Thus far, we have primarily studied server farms with a *central queue*, as in the M/M/k system, which we initially examined in Chapter 14 and then revisited from a capacity provisioning perspective in Chapter 15. In the M/M/k, jobs are held in a central queue, and only when a server is free does it take on the job at the head of the queue.

By contrast, in computer systems, most server farms do *immediate dispatching* (also known as *task assignment*), whereby incoming jobs are immediately assigned to servers (also known as *hosts*). There is typically no central queue; instead the queues are at the individual hosts.

Such server farms with immediate dispatching of jobs require an important policy decision, known as the **task assignment policy**. This is the rule that is used by the front-end router (also known as a *dispatcher* or *load balancer*) to assign incoming jobs to servers. For example, incoming jobs may be assigned to servers in round-robin order, or each incoming job might be assigned to the server with the shortest queue. The choice of task assignment policy hugely influences the response time of jobs at the server farm, sometimes by orders of magnitude. Using the right task assignment policy is particularly important when the job size distribution has high variability. A major question in computer systems design is finding a good task assignment policy to minimize mean response time (or some other performance variant).

Figure 24.1 illustrates the server farm model that is the focus of this chapter. The high-speed front-end router deploys the *task assignment policy*, which assigns incoming jobs to hosts. Observe that the *scheduling policy* deployed at an individual host is not fixed, but is typically application dependent. For example, in web server farms, the servers are typically time-sharing servers, and each server "simultaneously" serves all

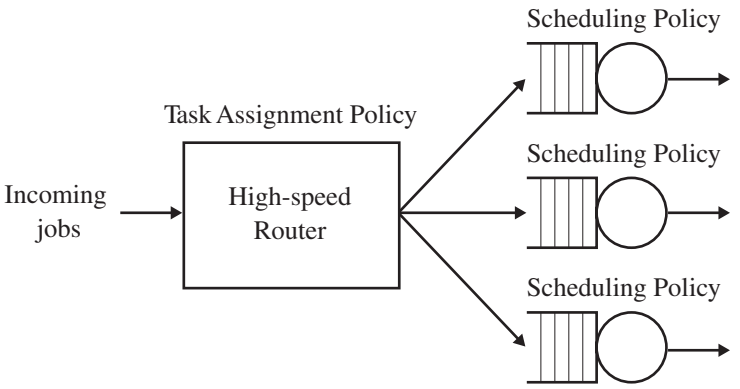


Figure 24.1. Server farm model.

jobs in its queue; that is, each server deploys the PS scheduling policy. By contrast, in manufacturing systems and in many supercomputing settings, jobs are often non-preemptible, and each server serves one job at a time in FCFS order.

The goal of this chapter is to understand the performance of different task assignment policies. The literature is so vast in this area that we can only hope to highlight some important results. Our goal throughout is to provide intuition. The readings noted in Section 24.4 contain greater depth.

In Section 24.1, we consider the case where jobs are non-preemptible and each server serves the jobs in its queue in FCFS order, as shown in Figure 24.2. In Section 24.2, we assume that jobs are preemptible and each server serves the jobs in its queue in PS order, as shown in Figure 24.6. For each of the settings in Sections 24.1 and 24.2, we look for task assignment policies that minimize mean response time. To facilitate analysis in these sections, we assume a Poisson arrival process. In Section 24.3, we ask more broadly how one could design optimal server farms in the case where jobs are preemptible and all design decisions are open (i.e., we can use any task assignment policy and any scheduling policy at the servers). Throughout we assume that job

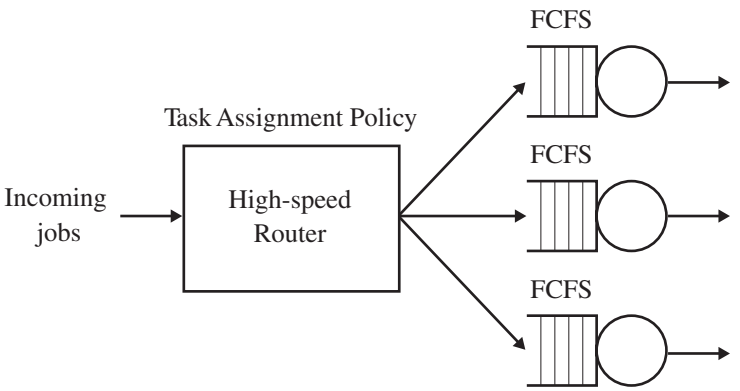


Figure 24.2. Server farm model with FCFS scheduling at hosts.

sizes are drawn from a high-variability distribution, such as the Bounded Pareto from Chapter 20, because such distributions reflect empirically measured job sizes.

24.1 Task Assignment for FCFS Server Farms

In this section, we assume the server farm model as shown in Figure 24.2, with k servers. In particular, we assume that *jobs are not preemptible* and that each server processes jobs in its queue in FCFS order. For simplicity, we assume that servers are *homogeneous*. We assume that job sizes are independently and identically distributed according to some high-variability distribution, G, with mean $\frac{1}{\mu}$. We also assume that jobs arrive according to a Poisson process with average rate λ. As usual, we denote the system utilization, ρ, by

$$\rho = \frac{\lambda}{k\mu}, \qquad 0 \leq \rho \leq 1$$

and the resource requirement, R, by

$$R = \frac{\lambda}{\mu} = k\rho, \qquad 0 \leq R \leq k$$

in accordance with Definition 14.4.

This model is common in manufacturing settings where it is often difficult, if not impossible, to preempt a job in progress. It is also common in supercomputing settings, where jobs are typically parallel computations, which makes them very difficult to preempt.

There are many choices for task assignment policies under this model. Many policies are not analytically tractable. However, in some cases their performance can be approximated. In others, even approximations are poor. Our discussion of these policies therefore sometimes relies on empirical results.

We can divide task assignment policies into those that make use of knowing the size (service requirement) of an arrival and those that do not assume any knowledge of the size of an arrival.

Question: What are some examples of task assignment policies for Figure 24.2 that do not assume any knowledge of the size of an arrival?

Answer: We list some common policies:

Under the **RANDOM** policy, each job is assigned to one of the k hosts with equal probability. The aim of the RANDOM policy is to equalize the expected number of jobs at each host.

Under the **ROUND-ROBIN** policy, jobs are assigned to hosts in a cyclical fashion with the ith job being assigned to host number $(i \mod k) + 1$. This policy also aims to equalize the expected number of jobs at each host.

Under the **JSQ (Join-the-Shortest-Queue)** policy, each incoming job is assigned to the host that has the shortest queue (the queue with the fewest *number* of jobs) at the

time when the job arrives. If several hosts have the same fewest number of jobs, then JSQ picks one of these hosts at random. This policy tries to equalize the instantaneous number of jobs at each host.

All three policies immediately dispatch jobs to hosts, where there is a queue of jobs at each host. Alternatively, we could instead imagine a single *central* queue (like in the M/M/k), where a host, when free, picks the job at the head of the queue to run. Because we are assuming generally distributed i.i.d. job sizes, this architecture is referred to as the **M/G/k**. Note that, although the M/G/k is not strictly within our model because there are not queues at the hosts, it still obeys the general framework of our model, because jobs are non-preemptible and the (single) queue is serviced in FCFS order, and we do not need to know job sizes. Thus we include M/G/k as one of our policies.

Question: Which of these policies – RANDOM, ROUND-ROBIN, JSQ, or M/G/k – would you guess has the lowest mean response time?

Answer: We will discuss and compare the policies one at a time . . . but hold on to your guess.

Question: Which do you think is superior: ROUND-ROBIN or RANDOM? Why?

Answer: It turns out that ROUND-ROBIN slightly outperforms RANDOM. To see why this is so, observe that the arrival process into each queue under RANDOM is a Poisson process, by Poisson splitting. By contrast, the interarrival time into each queue under ROUND-ROBIN is a sum of Exponentials – namely, an Erlang-k distribution – which has less variability than an Exponential. Hence, in RANDOM each queue behaves like an M/G/1, with average arrival rate λ/k, where G is the job size distribution, whereas under ROUND-ROBIN each queue behaves like an E_k/G/1, with average arrival rate λ/k. The lower variability of the E_k, compared to the M (Exponential), results in ROUND-ROBIN having lower mean response time.

Comparing ROUND-ROBIN with JSQ is more difficult, because for JSQ it is not possible to boil down the behavior of each queue to some simple G/G/1 queue. The issue is that the arrival process into a given queue under JSQ depends on the state of the other queues. To precisely analyze JSQ, one would need a k-dimensional Markov chain that tracks the number of jobs in each of the k queues. Unfortunately, no one knows how to analyze such a k-dimensional chain that grows unboundedly in all k dimensions. Even the case of just $k = 2$ dimensions and Exponential service times does not yield a closed form, and, for higher k $(k > 2)$, only approximations exist (see Section 24.4 for details). Based on the approximations that are known, it seems clear that JSQ is far superior to ROUND-ROBIN with respect to mean response time under higher job size variability. In fact, for high-variability job size distributions, JSQ can lower mean response time by an order of magnitude compared to ROUND-ROBIN.

Question: Intuitively, why does it make sense that JSQ should outperform ROUND-ROBIN under higher job size variability?

Answer: JSQ balances the instantaneous number of jobs at each queue, whereas ROUND-ROBIN balances the expected number of jobs. The difference is that JSQ can react quickly. Imagine that all queues have 5 jobs, but there is a lot of variability

among job sizes and one of the queues empties suddenly. JSQ can quickly remedy the situation by sending the next 5 arrivals to that queue, whereas ROUND-ROBIN would have to wait for $k/2$ more arrivals on average before even one job could be sent to the empty queue. During that period of waiting for $k/2$ arrivals, the server corresponding to the empty queue is not being utilized, increasing the load on all other servers and increasing overall mean response time. When job size variability is high, queues can empty very suddenly. This is why the dynamic properties of JSQ are so important.

Definition 24.1 A ***dynamic*** policy is one that adapts based on changes in the state of the system (e.g., the number of jobs at each queue), whereas a ***static*** policy is oblivious to the changes in state.

Thus JSQ is *dynamic*, whereas ROUND-ROBIN is *static*.

Question: How would you guess that JSQ compares with M/G/k under high job size variability? Why?

Answer: Both M/G/k and JSQ are dynamic policies, and both are good at making sure that no host is left idle. However M/G/k has a big additional advantage: It holds off on assigning jobs to hosts as long as possible. Observe that in JSQ it could happen that all queues have 5 jobs, and suddenly one queue empties, creating a situation where one server is unutilized. This underutilization can never happen under M/G/k, because whenever there are $\geq k$ jobs, every host is busy.

Empirical results show that M/G/k can outperform JSQ by an order of magnitude with respect to mean response time when job size variability is high.

Now suppose that we know the *size* of a job when it arrives. Clearly there are many more task assignment policies possible if we know the job size. One obvious example is the LWL policy.

Under the **LWL (Least-Work-Left)** policy, each job goes to the queue where it will achieve the lowest possible response time. This is a *greedy* policy, because each job is acting in its own best interest. Specifically, each incoming job is assigned to the queue that has the least total work at the time when the job arrives. Note that the work a job sees ahead of it is exactly its waiting time. Unlike some of the policies we saw earlier that aim to equalize the *number* of jobs at each host, the LWL policy aims to equalize the total *work* at each host.

The LWL policy is exactly what we do when we go to the supermarket. We look at each line and count not the number of people (jobs) there, but rather we look at the number of items in each person's basket (the job size) for every person in each line. We then join the line with the smallest total number of items (least work remaining).

Recall that under JSQ we only look at the number of jobs in each queue in deciding where to route a job. When job size variability is high, the number of jobs in a queue

can be a poor estimate of the total work in that queue. In this sense LWL is far superior to JSQ.

Question: How do LWL and M/G/k compare?

Answer: We will prove in Exercise 24.4 that LWL and M/G/k are actually equivalent, (i.e., LWL = M/G/k). Specifically, if both policies are fed the same arrival sequence of jobs, and ties are resolved in the same way in both systems, then it can be shown that the same job goes to the same host at the same time under both policies. Observe that when a job arrives to the M/G/k system, it may sit in the central queue for a while before being dispatched. However, the host that it will eventually go to is exactly the host that had the least work in front of it under LWL.

Unfortunately the analysis of M/G/k (and hence LWL) is a long-standing open problem in queueing theory. It is hard to imagine why the M/G/k is so hard to analyze, given that the M/M/k is so simple. Many young queueing theorists have devoted several years to beating their heads against the problem of analyzing the M/G/k system. Of course, one can always replace the job size distribution G with some phase-type distribution, PH, and use matrix-analytic methods to solve the M/PH/k system (see Exercise 21.7). Although this yields numerical solutions, it does not provide insight into which properties of the job size distribution matter and how these properties affect the solution. Also, even from a numerical standpoint, matrix-analytic solutions are not a panacea, because they can become very unstable (the matrices become near singular) when the distributions are highly skewed (e.g., when the squared coefficient of variation of the job size distribution, C^2, is very high).

The first closed-form approximation for waiting time in an M/G/k was proposed by Lee and Longton [118] over a half-century ago; it says that the waiting time in an M/G/k is basically the same as that in an M/M/k, but scaled up by a simple factor related to C^2:

$$\mathbf{E}\left[T_Q^{\text{M/G/k}}\right] \approx \left(\frac{C^2+1}{2}\right) \mathbf{E}\left[T_Q^{\text{M/M/k}}\right] \tag{24.1}$$

Many other authors have also proposed closed-form approximations for mean delay in the M/G/k, all involving only the first 2 moments of the job size distribution (see Section 24.4). Unfortunately, any approximation of mean delay based on using only the first 2 moments is provably inaccurate for some job size distributions; in addition, the inaccuracy of the approximation can be off by a factor proportional to C^2 [76].

Table 24.1 (borrowed from [76]) illustrates why two moments of the job size distribution are insufficient for predicting $\mathbf{E}[T_Q]$. The first row of the table shows $\mathbf{E}[T_Q]$ for an M/G/10 with mean job size of 1 and $C^2 = 19$ (first column) or $C^2 = 99$ (second column) according to approximation (24.1). The remaining rows show various distributions, all of which have been parameterized to have the same mean job size, $\mathbf{E}[S] = 1$, and appropriate C^2. As shown, the difference in $\mathbf{E}[T_Q]$ across distributions can be very high – $\mathbf{E}[T_Q]$ differs by a factor of close to 2 when $C^2 = 19$ and by a factor of more than 3 when $C^2 = 99$.

Table 24.1. *Simulation results for the 95% confidence intervals in an M/G/k, with $k = 10$, $\rho = 0.9$, and $\mathbf{E}[S] = 1$*

	$\mathbf{E}[T_Q]$ for $C^2 = 19$	$\mathbf{E}[T_Q]$ for $C^2 = 99$
2-Moment Approximation (24.1)	6.6873	33.4366
Weibull	6.0691 ± 0.01	25.9896 ± 0.18
Bounded Pareto ($\alpha = 1.1$)	5.5277 ± 0.02	24.6049 ± 0.28
Lognormal	4.994 ± 0.025	19.543 ± 0.42
Bounded Pareto ($\alpha = 1.3$)	4.879 ± 0.025	18.774 ± 0.36
Bounded Pareto ($\alpha = 1.5$)	3.947 ± 0.032	10.649 ± 0.54

The first line shows $\mathbf{E}\left[T_Q\right]$ for the 2-moment approximation given in (24.1). The remaining lines show various distributions with appropriate C^2.

Table 24.2 summarizes the task assignment policies that we have considered so far. There is one more commonly employed policy shown in the table that also makes use of knowing the size of jobs, namely the SITA policy.

Table 24.2. *Examples of common task assignment policies*

RANDOM	Each job is assigned to one of the k hosts with equal probability.
ROUND-ROBIN	The ith job is assigned to host $(i \mod k) + 1$.
JSQ	Each job is assigned to the host with the fewest number of jobs.
LWL	Each job is assigned to the host with the least total work.
M/G/k	When a server is free, it grabs the job at the head of the central queue.
SITA	Small jobs go to host 1, mediums to host 2, larges to host 3, etc.

Under the **SITA (Size-Interval-Task-Assignment)** policy [83], each host is assigned to a size interval, where the size intervals are non-overlapping and span the full range of possible job sizes. For example, the first host is assigned only "small" jobs (those of size between 0 and s, for some s); the second host is assigned only "medium" jobs (those of size between s and m, for some $m > s$); the third host is assigned only "large" jobs (those of size between m and l, for some $l > m$), etc. Every incoming job is routed to the appropriate host based on its size.

Question: What is the point of the SITA policy? Why does it make sense?

Answer: The SITA policy is similar to the "express lane" in your local supermarket, where one or two queues are reserved for "short" jobs only. When job size variability is high, there can be some very large jobs and some very small ones. By dedicating certain queues to short jobs only, we provide isolation for short jobs, so that they do not get stuck waiting behind long jobs.

Observe that we have not fully specified the SITA policy, because we have not specified the size cutoffs.

Question: Under SITA, what size cutoffs make sense?

Answer: One might think that choosing cutoffs that balance expected load among the queues makes sense. That is, we would choose s, m and l such that

$$\int_0^s tf(t)dt = \int_s^m tf(t)dt = \int_m^l tf(t)dt.$$

However, it turns out that choosing the cutoffs to balance expected load can be very far from optimal. This point is explored in Exercise 24.6.

Finding the optimal cutoff is very counterintuitive and often involves severely unbalancing the load between the servers. For example, for a Bounded Pareto with $\alpha < 1$, one wants to choose cutoffs that unbalance the load, favoring small jobs by underloading the servers of small jobs, whereas for a Bounded Pareto with $\alpha > 1$, one wants to choose cutoffs that unbalance the load, favoring large jobs by underloading the servers of large jobs [93, 82]. In the case of just $k = 2$ servers, the optimal cutoff can be obtained by search; however, the search becomes less feasible with more than 2 servers. As of the date of this book, the problem of finding closed-form cutoffs that minimize mean response time for general job size distributions is still wide open [93].

Question: How can we analyze SITA given that we know the cutoffs?

Answer: Once we are given size cutoffs, the analysis of SITA (under a Poisson arrival process) is very straightforward. Because the size of the incoming job is drawn at random from the size distribution, G, we can view the splitting of jobs into queues as probabilistic Poisson splitting of the arrival process. The ith queue can then be modeled as an M/G$_i$/1 queue, where G_i represents the job size distribution of jobs arriving at queue i. An example is provided in Exercise 24.1.

Question: How does the performance of SITA and LWL = M/G/k compare?

Answer: This is a surprisingly difficult question. Part of the problem, of course, is that neither SITA (with unknown cutoffs) nor LWL is analytically tractable in closed form. We start with some intuitions about each policy and then move on to what results exist in the literature.

One advantage of the LWL policy over any other policy is that it is ideal at keeping servers utilized. This can be seen by viewing LWL as M/G/k. It is impossible under LWL for one server to have zero jobs while another server has a job queueing.

One advantage of the SITA policy is that it is ideally suited to reducing variability at each queue. Suppose that the original job size distribution, G, has high variability. Under most policies, this same high variability is transferred to all the queues. This is problematic because we know, from the P-K formula (Chapter 23), that queueing delay is directly proportional to the variability of the job size distribution. SITA specifically divides up the job size distribution so that each queue sees only a portion of the domain of the original distribution, greatly decreasing the job size variability at each queue. Another way of putting this is that SITA provides short jobs *protection* from long jobs. Because most jobs in computing-based systems are short jobs, and because long jobs can be very, very long, isolating the many short jobs from long jobs greatly reduces mean response time.

The SITA policy and its variants have been part of the common wisdom for a long time and have been the focus of many papers (see Section 24.4 for references). Because of SITA's benefits in reducing job size variability, for a very long time it was believed that SITA, or some SITA-like variant, was far superior to LWL = M/G/k with respect

to mean response time when the job size variability was very high. Many papers specifically compared the performance of SITA to LWL and found that as job size variability is *increased*, SITA becomes far superior to LWL.

Figure 24.3 illustrates SITA's superiority on a server farm with $k = 2$ servers. The job size distribution shown is a Bounded Pareto (k,p,α) with $\alpha = 1.4$. The resource requirement is $R = 0.95$, equivalent to $\rho = 0.95/2$. C^2 is increased while holding $\mathbf{E}[S]$ fixed by increasing the upper limit, p, while decreasing the lower limit, k. The SITA mean response times are computed analytically by first numerically deriving the optimal splitting cutoff. The LWL mean response times are not analytically tractable, so we use a (very loose) upper bound on LWL performance, given in [157]. Figure 24.3 shows the effect on mean response time, $\mathbf{E}[T]$, as C^2 is increased, under LWL and SITA. According to the figure, SITA is far superior to LWL. Although the results for LWL are loose, the trend of SITA's superiority over LWL is in good agreement with simulation results and those in earlier research papers.

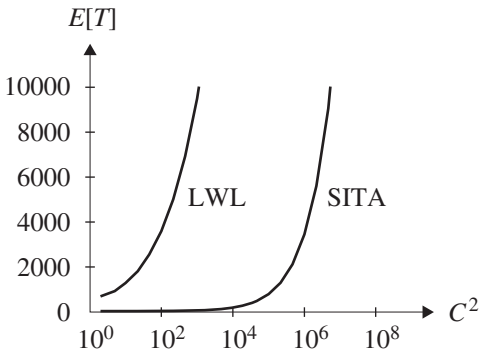


Figure 24.3. Expected response time, $\boldsymbol{E}[T]$, for SITA and LWL versus C^2 in a 2-server system with Bounded Pareto job size distribution with $\alpha = 1.4$ and resource requirement $R = 0.95$.

To more clearly illustrate SITA's superiority, we consider a server farm, again with $k = 2$ servers, where this time $R = 1.8$ and the job size distribution is a Hyperexponential, H_2, with unbalanced means (specifically $Q = 0.7$ fraction of the load is contained in one of the Hyperexponential branches). The advantage of using an H_2 is that we can analytically solve for the (nearly) exact mean response time under LWL via matrix-analytic methods, so that we do not have to use any upper bound. The H_2 also lends itself nicely to increasing C^2 while holding $\mathbf{E}[S]$ constant. Figure 24.4 clearly illustrates SITA's superiority over LWL for this H_2 job size distribution.

Despite comparisons such as those depicted in Figures 24.3 and 24.4 which show that SITA outperforms LWL by orders of magnitude for high job size variability, a *proof* of SITA's superiority over LWL never materialized. SITA itself is difficult to analyze, even for Poisson arrivals, because in general there is no closed-form expression for the optimal size cutoffs and for the resulting response time. Furthermore, LWL (which is equivalent to M/G/k) is in general only approximable. Thus, many of the existing comparisons have used simulation to assert their claims or have compared response time only under heavy-traffic regimes.

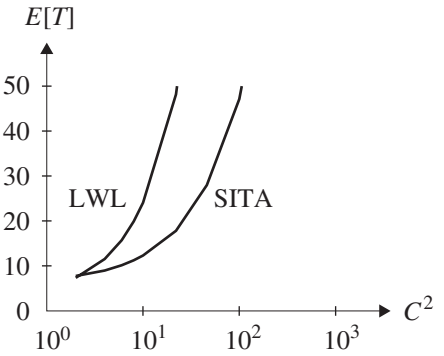


Figure 24.4. Expected response time, $\boldsymbol{E}[T]$, for SITA and LWL versus C^2 in a 2-server system with $R = 1.8$ and job size distribution H_2 with unbalanced branches ($Q = 0.7$).

In 2009, a surprising result was proven, showing that the common wisdom is actually *wrong* [90]: There are cases where SITA is not superior to LWL under high C^2, and in fact, SITA is provably *unboundedly worse* than LWL as $C^2 \to \infty$ in some regimes. An example of such a regime is provided in Figure 24.5.

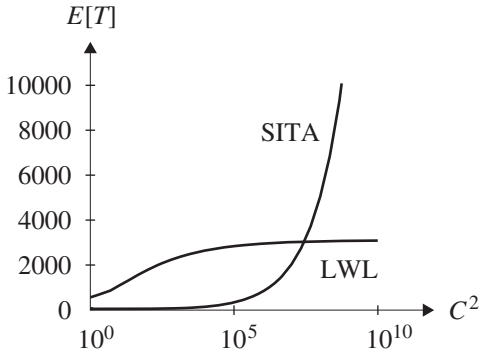


Figure 24.5. Expected response time, $\boldsymbol{E}[T]$, for SITA and LWL versus C^2 in a 2-server system with Bounded Pareto job size distribution with $\alpha = 1.6$ and $R = 0.95$.

Figure 24.5 considers a server farm identical to that of Figure 24.3, except that the Bounded Pareto parameter α has changed from $\alpha = 1.4$ to $\alpha = 1.6$. As in the case of Figure 24.3, the mean response time for SITA is computed analytically, while we use an upper bound from [157] for the mean response time for LWL. This time, however, the comparison between SITA and LWL looks very different. For lower C^2, SITA still improves on LWL; however, there is a crossover point, at sufficiently high C^2, after which SITA's response time diverges, whereas LWL's response time converges. This crossover point is actually at a much lower C^2 than it appears in the graph, because the upper bound for LWL is loose.

This crossover point, after which SITA's response time diverges but LWL's response time converges, was not observed in prior work (which mostly relies on simulation, numerical methods, heavy-traffic approximations or 2-moment M/G/2 approximations), possibly because the earlier literature did not consider the very high C^2 regions, thus (incorrectly) concluding that SITA is always superior to LWL.

Question: But why would SITA be inferior to LWL under high job size variability? Isn't SITA specifically designed to combat high variability?

Answer: Consider a server farm with two hosts, and the Bounded Pareto (k, p, α) job size distribution from Figure 24.5 as $p \to \infty$ and $C^2 \to \infty$. SITA needs to place its size cutoff somewhere. If it places the size cutoff at any finite value, x, then the first host sees a job size distribution with finite variance (because sizes range from k to x); however, the second host sees a job size distribution with infinite variance (because sizes range from x to ∞, as $p \to \infty$). Since mean response time is a weighted sum of the response time at the two hosts, the mean response time under SITA will tend to infinity as $C^2 \to \infty$. Note that we can instead make the cutoff, x, increase with p. This however, means that as $p \to \infty$, the first host experiences infinite variability, which again means that SITA has infinite mean response time.

By contrast, under LWL performance is only bad if two big jobs arrive near to each other, blocking off both servers. But the probability of such a bad event can be very low if the resource requirement is sufficiently light (e.g., $R < 1$ with 2 servers). In this case the second server is not really needed, and is thus available to serve shorter jobs if one server gets blocked with a long job.

In general, for a system with k servers, we define the number of **spare servers** as

$$\text{Number spare servers } = k - \lceil R \rceil.$$

These spare servers can be extremely effective in combating variability. Even if C^2 for the job size distribution approaches infinity, the spare servers can be used to let the smaller jobs have a "freeway" so that they do not get stuck behind large jobs, allowing the response time of LWL to converge.

One might think that SITA could similarly benefit from spare servers, but the strict routing in SITA makes it unable to enjoy this benefit.

Question: But why was there a difference between the Bounded Pareto with $\alpha = 1.4$, shown in Figure 24.3, and the Bounded Pareto with $\alpha = 1.6$, shown in Figure 24.5, given that both cases were run with one spare server?

Answer: The Bounded Pareto with $\alpha = 1.4$ has a fatter tail, implying there are more medium and large jobs. This increases the probability of a "bad event" where two large jobs arrive at near the same time. It turns out that in this case one spare server is insufficient for LWL. The difference is captured more precisely in the $3/2$ moment of the job size distribution. Observe that $\mathbf{E}\left[S^{\frac{3}{2}}\right]$ is infinite for $\alpha = 1.4$, whereas $\mathbf{E}\left[S^{\frac{3}{2}}\right]$ is finite for $\alpha = 1.6$. Theorem 24.2 explains that the $3/2$ moment of S is crucial in understanding the response time of the M/G/2 (LWL).

Theorem 24.2 *[155, 156, 157, 158] For (almost) all job size distributions with r.v. S, the mean response time of the M/G/2 is bounded (finite) if and only if $\mathbf{E}\left[S^{\frac{3}{2}}\right]$ is finite and there is at least one spare server.*

Although we do not have space to prove this theorem, in Section 24.4 we elaborate on generalizations of the result to $k > 2$ servers. Note that this stability result is very

different from an M/G/1, whose mean response time is finite if and only if $\mathbf{E}[S^2]$ is finite.

Summary

This section has dealt with finding task assignment policies for server farms in the case where jobs are not preemptible and job size variability is high. Our discussion covers only the highlights of a vast body of literature in the area. The main point is that task assignment is non-obvious and can be counterintuitive. Many common, well-known policies, such as RANDOM, ROUND-ROBIN, and JSQ, are virtually worthless when job size variability is high. Furthermore, it is difficult to *rank* the policies: As we have seen, sometimes SITA can be *far* superior to LWL, and sometimes the reverse is true. Even a seemingly innocuous and obvious goal like "load balancing" is questionable, because SITA can perform far better when the load across servers is purposely unbalanced. Finally, the analysis of task assignment policies is often very difficult and is still in its infancy. We have collected some important references in Section 24.4.

Question: So far we have only considered the case of high job size variability. Suppose that instead the job size variability is very low. How do the policies that we have considered compare in this situation?

Answer: When job sizes are Deterministic (e.g., all jobs have size 1), the ROUND-ROBIN policy is optimal, because the arrivals to a server are maximally spaced out; in fact, if job sizes and interarrival times are both Deterministic, then no job will be delayed under ROUND-ROBIN, assuming that the system is not in overload. The JSQ policy will actually end up doing the same thing as ROUND-ROBIN, because the shortest queue will be that which has not received a new job in the longest time. By the same logic, LWL will end up doing the same thing as ROUND-ROBIN. In contrast, RANDOM will sometimes make the "mistake" of sending two consecutive arrivals to the same queue, incurring some delay. SITA will reduce to RANDOM, because all jobs have the same size. Even with occasional mistakes by RANDOM, we expect mean response time to be very low. To see this, consider the case of RANDOM with Deterministic job sizes and Poisson arrivals. By Poisson splitting, each queue becomes an M/D/1 queue, which we know has only half the delay of an M/M/1.

24.2 Task Assignment for PS Server Farms

We now turn to a very different model of a server farm. Figure 24.6 depicts a queueing model of a web server farm. Here the incoming requests are HTTP requests. These must be immediately dispatched to one of the server hosts, because they are connections that need immediate attention. Requests are fully preemptible in that any request can be stopped and restarted where we left off. Given that this is a network-based application, running on TCP, it is important that the service seem immediate and constant. For this reason, we cannot have requests waiting in a FCFS queue. Instead, each host

time-shares among all the requests in its queue, so that each HTTP request receives "constant" service. This scheduling is modeled as Processor-Sharing (PS).

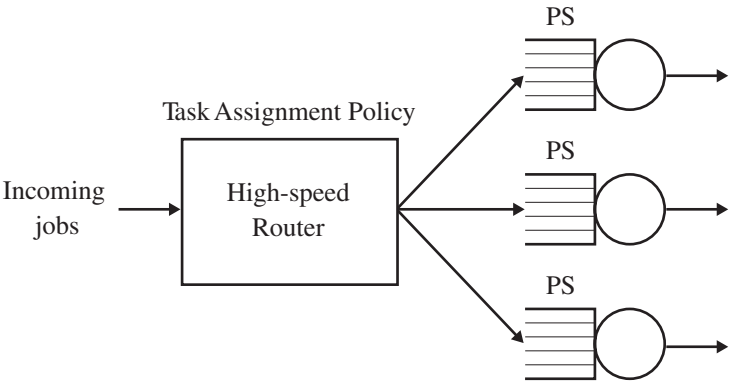


Figure 24.6. Server farm model with PS scheduling at hosts.

There are many common high-speed routers used for dispatching HTTP requests in a web server farm. Some examples are Cisco's LocalDirector [42], IBM's Network Dispatcher [140], and F5's BIG-IP [21]. The job size distribution for websites is known to be highly variable and heavy-tailed [47, 48]. To ease the analysis, we assume that the arrival process is a Poisson process with average rate λ and that job sizes are i.i.d. Assuming a Poisson arrival process is not necessarily unrealistic, particularly if the arrival process is the merge of many disparate users. However, our assumptions that job sizes are independent of each other and independent of the interarrival times are not true in practice, as pointed out in several papers, [70, 47, 169]. As before, we assume k identical servers and use S to denote the job size, where $\mathbf{E}[S] = \frac{1}{\mu}$. The system load is denoted by $\rho = \frac{\lambda}{k\mu}$ and the resource requirement by $R = \frac{\lambda}{\mu}$.

Given the huge prevalence of web server farms, it is important to consider what task assignment policies are best for the PS server farm model and how they compare. We again consider the policies in Table 24.2; all except for the M/G/k (which by definition uses a FCFS queue) are reasonable options for a PS server farm. Let's see how these compare.

Question: Consider first the RANDOM policy and the SITA policy. Recall that for the FCFS server farm, the SITA policy was far superior to RANDOM when the job size distribution was highly variable. How do these policies compare for the PS server farm, again assuming high job size variability?

Hint: Analyzing SITA of course depends on the size cutoffs. It turns out that, in contrast to FCFS server farms, the optimal size cutoffs for PS server farms are those that *balance* load between the servers (see Exercise 24.2). Thus the load at every server is ρ, just like the overall system load. It is therefore easiest to express the response time of each policy in terms of ρ.

Hint: Both RANDOM and SITA experience Poisson splitting, because the size cutoffs in SITA can be viewed as sending a fraction, p_i, of jobs to server i.

Answer: For RANDOM, we note that an arrival goes to a random queue with load ρ and arrival rate λ/k. By Poisson splitting, this random queue is an M/G/1/PS queue,

but the mean response time for M/G/1/PS is the same as that of M/M/1/FCFS (see Chapter 22). Thus, the random arrival goes to a queue with (on average) $\frac{\rho}{1-\rho}$ jobs. By Little's Law, its response time at this queue is then

$$\mathbf{E}\left[T\right]^{\text{RANDOM}} = \frac{1}{\lambda/k} \cdot \frac{\rho}{1-\rho} = \frac{k}{\lambda} \cdot \frac{\rho}{1-\rho}.$$

For SITA, WLOG assume that the job size distribution ranges from 0 to ∞ and that the size cutoffs are $s_1, s_2, \ldots s_{k-1}$, where jobs in the interval $(0, s_1)$ go to host 1, jobs of size (s_{i-1}, s_i) go to host i, and jobs of size (s_{k-1}, ∞) go to host k. The fraction of jobs that go to host i is p_i where $p_i = \int_{s_{i-1}}^{s_i} f(t)dt$, where $f(t)$ is the density of the job size distribution. The load at queue i is ρ (see the first hint). By Poisson splitting, queue i is an M/G/1/PS queue (see second hint). The arrival rate into queue i is λ_i, where $\lambda_i = \lambda p_i$. Putting these facts together we have

$$\mathbf{E}\left[T \mid \text{ job goes to host } i\right]^{\text{SITA}} = \frac{1}{\lambda_i} \cdot \frac{\rho}{1-\rho}.$$

$$\begin{aligned}
\mathbf{E}\left[T\right]^{\text{SITA}} &= \sum_{i=1}^{k} p_i \cdot \mathbf{E}\left[T \mid \text{ job goes to host } i\right] \\
&= \sum_{i=1}^{k} p_i \cdot \frac{1}{\lambda_i} \cdot \frac{\rho}{1-\rho} \\
&= \sum_{i=1}^{k} \frac{1}{\lambda} \cdot \frac{\rho}{1-\rho} \\
&= \frac{k}{\lambda} \cdot \frac{\rho}{1-\rho}.
\end{aligned}$$

Thus we have that

$$\mathbf{E}\left[T\right]^{\text{RANDOM}} = \mathbf{E}\left[T\right]^{\text{SITA}}. \tag{24.2}$$

The fact that RANDOM and SITA have the same mean response time (for server farms with PS servers) might be surprising, because these policies yield very different performance for FCFS servers. The reason that RANDOM and SITA were so different for server farms with FCFS servers is that RANDOM does nothing to reduce job size variability, whereas SITA does a lot to reduce variability. However, PS scheduling is invariant to job size variability, and hence the benefit of SITA in reducing job size variability is superfluous.

Question: For server farms with PS servers, which is better: JSQ or LWL? Which was better for server farms with FCFS servers?

Answer: Recall that for the case of server farms with FCFS servers, LWL was superior to JSQ. For FCFS servers, LWL represented the *greedy* policy, whereby each job was routed to the host where it would itself experience the lowest response time, namely the host with the least total work.

For the case of PS servers, JSQ represents the *greedy* policy that routes jobs to the host where it will likely experience the lowest response time. Specifically, under JSQ, each job is routed to the host where it will time-share with the fewest jobs. By contrast,

knowing the total work at a PS host does not necessarily have any bearing on the job's response time at that host.

Unfortunately, analyzing JSQ is no easier for a PS server farm than for a FCFS server farm, even when the job size distribution is Exponential. Modeling JSQ requires tracking the number of jobs at each queue, so that we can determine to which host an arrival should be routed. But tracking the number of jobs in each queue necessitates a state space that grows unboundedly in k dimensions (one for each queue), making it intractable. The problem is only amplified for LWL where we need to track the total work at each queue.

One idea for analyzing JSQ in a PS server farm is to approximate the dependence between the queues while only tracking what is going on in a single *one* of the k queues, WLOG queue 1 [79]. The dependence is captured by making the arrival rate into queue 1 be dependent on the number of jobs at queue 1. For example, the average arrival rate into queue 1 should be λ/k. However, if queue 1 currently has 0 jobs, then the arrival rate into queue 1 should be greater than λ/k, because it is likely that the other queues have more jobs than queue 1. Likewise, if queue 1 currently has many jobs, then the arrival rate into queue 1 will be less than λ/k, because the other queues likely have fewer jobs. By deriving the correct load-dependent arrival rate into queue 1, one can approximate the influence of the other $k - 1$ queues on queue 1. Finally, since queue 1 was chosen WLOG, the delay experienced by an arrival to queue 1 is the system delay.

A recent finding is that JSQ is surprisingly (nearly) insensitive to job size variability for PS server farms [79]. At first, this may seem to follow from the insensitivity of the M/G/1/PS queue. However, there is more to it than that, because LWL, as we will soon see, is not at all insensitive to job size variability for PS server farms. A proof of the near insensitivity of JSQ has not yet been found, as of the time of writing of this book.

Figure 24.7 shows simulation results for the performance of all the task assignment policies we have considered over a range of job size distributions, described in

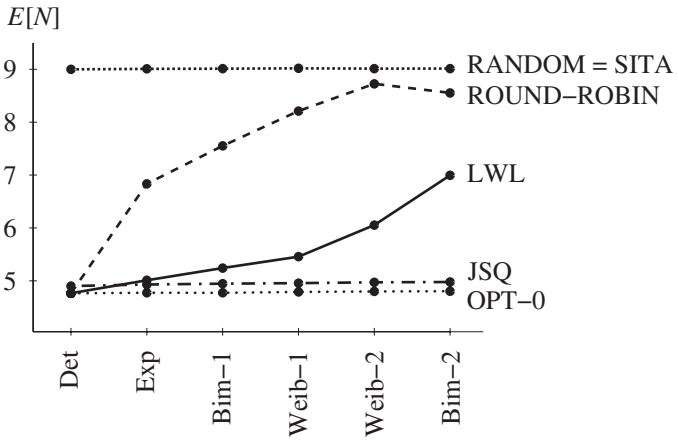


Figure 24.7. Simulation results for a server farm with two PS hosts, under different task assignment policies. The x-axis shows a variety of job size distributions (described in Table 24.3) in order of increasing variability from left to right. The y-axis depicts the mean number of jobs *per host* of the server farm, under each job size distribution. The server farm load is $\rho = 0.9$.

Table 24.3. *Job size distributions, each with mean* 2*, but increasing variance from top to bottom*

Distribution	Mean	Variance
Deterministic: point mass at 2	2	0
Erlang-2: sum of two Exp(1) random variables	2	2
Exponential: Exp(0.5) random variable	2	4
Bimodal-1: $\begin{cases} 1 & \text{w.p. } 0.9 \\ 11 & \text{w.p. } 0.1 \end{cases}$	2	9
Weibull-1: (shape parameter 0.5, scale parameter 1)	2	20
Weibull-2: (shape parameter $\frac{1}{3}$, scale parameter $\frac{1}{3}$)	2	76
Bimodal-2: $\begin{cases} 1 & \text{w.p. } 0.99 \\ 101 & \text{w.p. } 0.01 \end{cases}$	2	99

Table 24.3.[1] Each distribution has mean 2, but the distributions have increasing variance, ranging from a variance of 0 for the Deterministic distribution to a variance of 99 for the Bimodal-2 distribution. As we consider distributions with higher and higher variance, we see that the performance of ROUND-ROBIN and LWL both deteriorate. By contrast, the performance of SITA, RANDOM, and JSQ appear insensitive to the job size variability (JSQ is not actually insensitive, because there is a 1% variation, not visible by eye). The JSQ policy is clearly the best of the policies we have considered thus far.

To gauge the optimality of JSQ, we also compare policies against the **OPT-0** policy. The OPT-0 policy, introduced in [25], assigns each incoming job so as to minimize the mean response time for all jobs currently in the system, *assuming that there are* 0 *future arrivals*. Note that we are not being greedy from the perspective of the incoming job, but rather trying to minimize across all the jobs in the system. This policy is followed for each successive incoming arrival. Although the JSQ policy is far simpler than OPT-0, its performance is within about 5% of OPT-0, for all job size distributions in the table. Thus the JSQ policy appears to be near optimal.

Summary

This section has dealt with finding task assignment policies for server farms in the case where jobs are preemptible, the scheduling at the servers is PS, and job size variability is high. Unlike the previous section that dealt with FCFS scheduling at the servers, there is presently very little literature on this topic, see [6], [79], and [101], probably because the operations research community deals far less with PS servers than with FCFS servers.

The main point of this section is that task assignment is very different for server farms composed of PS servers as compared to FCFS servers. Whereas JSQ is a pretty bad policy for server farms of FCFS servers, because of its ineffectiveness in alleviating delays created by high job size variability, JSQ is an excellent policy for server farms of PS servers. Similarly, SITA, a top performer for server farms with FCFS servers, is

[1] The Weibull distribution has p.d.f. $f(t) = \frac{\alpha}{\lambda}\left(\frac{t}{\lambda}\right)^{\alpha-1} e^{-\left(\frac{t}{\lambda}\right)^{\alpha}}$, for $t > 0$, where $\alpha > 0$ is called the shape parameter and $\lambda > 0$ is called the scale parameter; see [181]. The parameters that we chose in the table result in heavy-tailed distributions.

among the worst performers for server farms with PS servers. Under server farms of PS servers, job size variability is not a big problem, and some policies, like JSQ, are nearly insensitive to job size variability.

The analysis of server farms with PS servers is a wide open area, full of open problems. For example, we did not even discuss the very interesting problem of task assignment policies for the case of heterogeneous servers.

24.3 Optimal Server Farm Design

We now turn to the more theoretical question of how to optimally design a server farm if one is allowed to choose *both* the task assignment policy and the scheduling policy at the individual hosts; both these decisions are shown in Figure 24.1. This is a theoretical question, because typically the scheduling policy at the individual hosts is dictated by the operating system at the servers and the application. To give ourselves maximum flexibility, we further assume that jobs are fully preemptible and that we know a job's size when it arrives (of course we do not know future jobs). Finally, we allow ourselves the flexibility of having a central queue at the router, if we want one. As usual, we assume a job size distribution with high variability, mean job size $\mathbf{E}[S]$, and a Poisson arrival process with average rate λ.

Unfortunately, there exists almost no stochastic analysis in the area of optimal server farm design. All the work in the area of optimal server farm design deals with worst-case analysis and competitive ratios. In **worst-case analysis**, one is no longer looking at the performance of a policy, $\mathcal{P}$, under a Poisson arrival process with i.i.d. job sizes taken from some distribution, as we have been assuming. Instead, one imagines an adversary who can generate any arrival sequence, where the arrival sequence consists of arrival times of jobs and their sizes. The policy $\mathcal{P}$ is now evaluated on each possible arrival sequence and is compared with the optimal policy for that arrival sequence. Specifically, we imagine some algorithm **OPT** that behaves optimally on each arrival sequence. We do not know what OPT looks like, and it does not have to be consistent across arrival sequences (that is, OPT can follow a different algorithm on each arrival sequence); we just use OPT to denote the best possible solution for every arrival sequence. Now consider the whole space of possible arrival sequences. For each arrival sequence, $\mathcal{A}$, consider the following ratio:

$$r_{\mathcal{P}}(\mathcal{A}) = \frac{\mathbf{E}\left[T(\mathcal{A})\right]^{\mathcal{P}}}{\mathbf{E}\left[T(\mathcal{A})\right]^{\text{OPT}}},$$

where $\mathbf{E}\left[T(\mathcal{A})\right]^{\mathcal{P}}$ is the expected response time of policy $\mathcal{P}$ on arrival sequence $\mathcal{A}$. Then the **competitive ratio** of policy $\mathcal{P}$ is defined as

$$\text{Competitive ratio of } \mathcal{P} = \max_{\mathcal{A}} r_{\mathcal{P}}(\mathcal{A}).$$

In worst-case analysis, the higher the competitive ratio of a policy, the "worse" that policy is. Note that a policy $\mathcal{P}$ can have a high competitive ratio even if it performs poorly on just a single arrival sequence.

Worst-case analysis can yield a very different ranking of policies than that obtained by the stochastic analysis described in most of this book. A policy $\mathcal{P}$ can be viewed as very poor in a worst-case sense, because it performs badly on one particular arrival sequence, but that arrival sequence can be a very low-probability event in a stochastic sense.

Question: Returning to the question of optimal server farm design, what are good routing/scheduling policy choices?

Hint: For the case of a single queue, with fully preemptible jobs, where we know the size of the job, what is the best scheduling policy on every arrival sequence?

Answer: In Exercise 2.3, we proved that the SRPT policy, which always (preemptively) runs that job with the shortest remaining processing time, is optimal with respect to mean response time, for the case of a single queue. This result was originally proved in [159] and holds under any arrival sequence of job sizes and arrival times.

The optimality of SRPT for a single queue inspires the server farm configuration shown in Figure 24.8. It is like an M/G/k, except that the central queue is served in SRPT order. Specifically, the k hosts are always serving those k jobs with the currently shortest remaining processing times. Suppose, WLOG, that server i is working on a job with remaining processing requirement r_i, where $r_i > r_j$, for all active servers j. Then, if a job comes in with shorter remaining time than r_i, that arrival is immediately put into service at the ith server, and the prior job being served at the ith server is put back into the queue. We refer to this as the **Central-Queue-SRPT** policy.

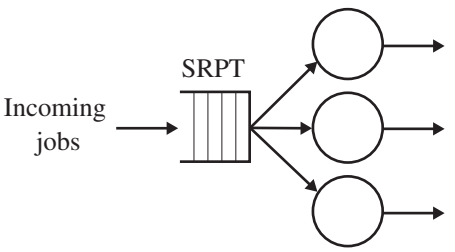


Figure 24.8. Server farm with Central-Queue-SRPT policy.

The Central-Queue-SRPT policy looks very good. Because at every moment of time the k jobs with shortest remaining processing time are those in service, the server farm behaves very much like a single queue with a server that is k times the speed.

Question: Is Central-Queue-SRPT optimal in the worst-case sense? That is, does Central-Queue-SRPT minimize $\mathbf{E}[T]$ on every arrival sequence?

Answer: Sadly, the answer is no. The following is an example of a "bad" arrival sequence, for the case of a 2-server system, where Central-Queue-SRPT does not produce the minimal mean response time. This is adapted from [119]:

- At time 0, 2 jobs of size 2^9 arrive, as well as 1 job of size 2^{10}.
- At time 2^{10}, 2 jobs of size 2^8 arrive, as well as 1 job of size 2^9.
- At time $2^{10} + 2^9$, 2 jobs of size 2^7 arrive, as well as 1 job of size 2^8.
- At time $2^{10} + 2^9 + 2^8$, 2 jobs of size 2^6 arrive, as well as 1 job of size 2^7.
- And so forth . . .

Let's name our two servers, server A and server B. The optimal algorithm, at time 0, will run the 2 jobs of size 2^9 on server A and will simultaneously run the job of size 2^{10} on server B. By time 2^{10}, all work that arrived at time 0 will be complete. At time 2^{10}, the optimal algorithm will run the 2 jobs of size 2^8 on server A and will simultaneously run the job of size 2^9 on server B, and so forth. The point is that the optimal algorithm is always able to pack the jobs in such a way that the servers are both fully utilized at all times.

By contrast, Central-Queue-SRPT makes a mess of this arrival sequence. At time 0, it tries to run one of the jobs of size 2^9 on server A and the other job of size 2^9 on server B, because these are the jobs with smallest remaining time. Only when these complete does Central-Queue-SRPT start to run the job of size 2^{10}, leaving one of the servers idle. Unfortunately, this does not leave enough time to complete the job of size 2^{10}, which must be preempted at time 2^{10} when the next batch of jobs comes in. Central-Queue-SRPT continues its mistakes, by now running one of the jobs of size 2^8 on server A and the other job of size 2^8 on server B. Only when these complete does it start to run the job of size 2^9. Unfortunately, there is not enough time for the job of size 2^9 to complete before the new batch of jobs arrive, etc. Central-Queue-SRPT packs the jobs badly, so that the two servers are not both fully utilized; hence, resources are wasted and jobs do not complete.

Although the Central-Queue-SRPT algorithm performs particularly badly on this arrival sequence, Leonardi and Raz [119] prove that it is still the best possible online algorithm from a worst-case competitive-ratio perspective. It is shown in [119] that the competitive ratio of Central-Queue-SRPT is proportional to $\log\left(\frac{b}{s}\right)$, where b is the biggest job size possible and s is the smallest job size possible, and that no online algorithm can improve on this competitive ratio by more than a constant multiplicative factor.

It is important to note that, although the Central-Queue-SRPT algorithm is not optimal in a worst-case sense, that does not mean that it is not optimal in a *stochastic* sense. For example, it might be the best possible policy given a Poisson arrival process and i.i.d. job sizes from any general distribution. Unfortunately, we do not know the answer to this question, because no one to date has been able to analyze the Central-Queue-SRPT policy from a stochastic perspective, even under a Poisson arrival process and Exponentially distributed job sizes. The closest approximation to this is [89], which analyzes an M/PH/k queue with an arbitrary number of preemptive priority classes, where jobs are prioritized in order of shortest expected remaining time. Unfortunately, the results in [89] are numerical in form; no closed-form analysis exists. Analyzing Central-Queue-SRPT stochastically is an important open problem in queueing.

A related open problem is the question of optimal task assignment under the restriction that jobs need to be **immediately dispatched** to hosts, meaning that they cannot be held in a central queue. This restriction is of practical importance, because, in many applications, like web servers, the request needs to be assigned to a host that can immediately establish a connection with the client. In the case where jobs need to be

immediately assigned to hosts, it is optimal to run SRPT scheduling at the individual hosts.[2] Thus we have the architecture shown in Figure 24.9.

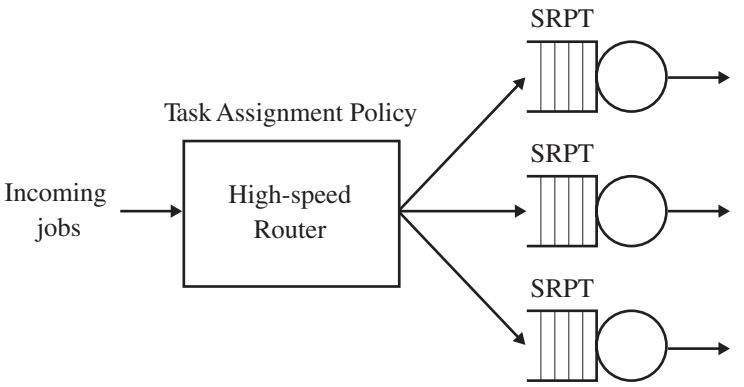


Figure 24.9. Server farm with immediate dispatch and SRPT scheduling at the hosts.

Question: Given SRPT scheduling at the individual hosts as in Figure 24.9, what is a good (immediate dispatch) task assignment policy for minimizing mean response time?

Hint: SRPT is very effective at getting short jobs out.

Answer: Simple RANDOM task assignment is not a bad policy here, because each queue then looks like an M/G/1/SRPT queue with arrival rate λ/k and mean job size $\mathbf{E}[S]$. The idea proposed independently by [9] and by [49] is to go a step further and make sure that the short jobs are spread out over all the SRPT servers, so that the servers can each be working on getting as many short jobs out as possible. Specifically, the **IMD** algorithm in [9] divides jobs into size classes (small, medium, large, etc.) and then assigns each incoming job to the server with the smallest number of jobs in that size class. The point is to make sure that each server has some smalls, some mediums, and some larges, so that their SRPT scheduling can be maximally effective.

Avrahami and Azar [9] prove that when the server farm in Figure 24.9 is run with the IMD task assignment policy, the performance is on the order of that achieved by Central-Queue-SRPT in a worst-case sense, meaning that the two algorithms result in competitive ratios within a constant factor of each other. An exact stochastic analysis of IMD is not known, although an approximation is given in [49].

Summary

This section has looked at server farm configurations using "optimal" task assignment/scheduling policy pairings. Almost all the available analysis is worst-case analysis.

[2] Using proof by contradiction, one can argue that not running SRPT scheduling at the host will only increase mean response time.

Unfortunately, there is almost no stochastic analysis known for any of the models that we considered. This is also a wide-open area. A reader interested in working on such analysis should first read Chapters 32 and 33, which consider SRPT and related variants for a single-server system.

24.4 Readings and Further Follow-up

Below we list additional references on the topic of task assignment for server farms.

More on SITA and Its Variants for Server Farms with FCFS Servers

It is not clear where the idea of size-based task assignment originated, because it has been part of the common wisdom for a long time. Size-based splitting was used in the Cornell Theory Center [99]. The SITA policy was formally introduced by Harchol-Balter, Crovella, and Murta in [83]. In a follow-up paper [82], Harchol-Balter introduced a variant of SITA called TAGS that does not require knowing the size of the job, but nevertheless achieves response times close to those of SITA. The SITA policy and its variants have been the focus of many papers including [99, 83, 82, 177, 50, 134, 41, 172, 65, 32, 11, 59, 36, 161]. Because of SITA's benefits in reducing job size variability, for a very long time it was believed that SITA, or some SITA-like variant, was far superior to LWL = M/G/k with respect to mean response time when the job size variability was very high. Many papers specifically compared the performance of SITA to LWL and found that, as job size variability is *increased*, SITA becomes far superior to LWL [32, 41, 50, 65, 82, 83, 134, 172, 177]. Although the above papers suggest that SITA should be superior to LWL under high job size variability, [90] finds that the opposite can actually be true for certain job size distributions and loads, as explained in this chapter. This work was further extended in [91], where variants of SITA were considered that combine the strengths of SITA and LWL. For example [91] considers an M/G/2 server farm, where the top server only serves small jobs, but the bottom server can serve any job, referred to as **Hybrid**. Surprisingly, [91] proves that even Hybrid can sometimes be inferior to LWL.

More on M/G/k and G/G/k

The mean response time for the M/G/k, and hence also G/G/k, remains a longstanding open problem in the queueing literature [76]. After the Lee and Longton [118] approximation in 1959, many other authors proposed alternative simple closed-form approximations for mean waiting time; see [97, 98, 112, 132, 26, 196]. Unfortunately, all of these closed-form approximations also involve only the first 2 moments of the job size distribution, which is shown to be insufficient in [76].

There are several key analytical papers that are concerned with the G/G/k under high job size variability. Scheller-Wolf and Sigman [156, 155] prove an upper bound on mean delay in a G/G/k system where this upper bound does not depend on any moment

of service time higher than the $\frac{k+1}{k}$ moment, and it particularly does not depend on the variance of job size. The [155] result requires that the resource requirement, R, is not too high: $R < \lfloor k/2 \rfloor$ (where $R = k\rho$). However, [156] generalizes the result to allow for higher load, $R < k - 1$. The converse of the [156, 155] results was presented by Scheller-Wolf and Vesilo in [158] for a large class of distributions. It is known that if $R > k - 1$, then the G/G/k diverges as $C^2 \to \infty$ [157], where C^2 is the squared coefficient of variation of the job size.

Whitt [184] and Foss and Korshunov [64] consider a G/G/2 and study the delay tail behavior when job size is heavy-tailed. They find that for low load, the delay tail grows like the tail of the equilibrium distribution, squared, whereas for high load the delay tail grows like the tail of the equilibrium distribution. These results are consistent with [155] and [158].

More on JSQ

For the case of server farms with PS servers, there has been very little analysis of JSQ, see [79]. However, there is a substantive simulation study by Bonomi [25] that examines both JSQ and a few policies that improve slightly over JSQ (5% improvement) by exploiting knowledge of the remaining service times of jobs.

By contrast, there is a lot of work on JSQ for server farms with FCFS scheduling at the servers. For workloads with non-decreasing failure rate, where job sizes are not known a priori, the JSQ policy is provably optimal [180, 194, 52]. As we have pointed out, however, JSQ is far from optimal for FCFS servers with *highly variable* job sizes [83, 82].

Almost all papers analyzing JSQ for server farms with FCFS servers are limited to $k = 2$ servers, an Exponential job size distribution, and the mean response time metric. Even here, the papers are largely approximate and often require truncation of the state space or of some infinite sums [108, 61, 73, 44, 146, 122, 3]. Sometimes the results are exact, but are not computationally efficient and do not generalize to higher values of k [27].

For analyzing JSQ with more than $k = 2$ servers, for the case of server farms with FCFS servers, again with Exponential job sizes, only approximations exist. Nelson and Philips [128] use the following idea: They look at the steady-state probability of the M/M/k queue (with a central queue) as an estimate for the total number of jobs in the JSQ/FCFS system; they then assume that the jobs in the system are divided equally (within 1) among each of the queues. Lin and Raghavendra [120] follow the approach of approximating the number of busy servers by a Binomial distribution and then also assume that the jobs are equally divided among each of the queues (within 1). Both approximations are surprisingly accurate, with reported accuracy in the 2% to 8% error range. There are also some numerical methods papers that do not lead to a closed-form solution, but are accurate and computationally efficient for not-too-large k; see for example [2, 122, 4].

Finally, Bramson, Lu, and Prabhakar [30] consider the limiting regime where the number of queues goes to infinity ($k \to \infty$). In this regime they prove that, for any fixed number of queues, the numbers of jobs at each of the queues become independent. This allows them to derive various performance metrics, including queue lengths.

Cycle Stealing in Server Farms

The performance of server farms can be improved dramatically by allowing servers to share their work. *Cycle stealing* is the idea of allowing an idle server to take on some work from a busy server's queue. Analyzing the performance of a server farm with cycle stealing has the same difficulties as analyzing JSQ, because one needs to track the number of jobs at each server, resulting in a Markov chain that grows unboundedly in k dimensions. Even for $k = 2$ this is only approximable. There are a long list of approximations for different variants of cycle stealing, including approximations based on truncating the state space along one of the dimensions [74, 170, 171]; approximations based on boundary-value methods [55, 113, 43]; heavy-traffic techniques [17, 63]; approximations based on the idea of Dimensionality Reduction of Markov chains [86, 136, 89, 88, 192, 20, 137, 138, 87, 135]; and others [168, 193] .

24.5 Exercises

24.1 Server Farm with Size-Interval-Task-Assignment

We are given a server farm with 2 identical FCFS hosts. Arrivals into the system occur according to a Poisson process with rate λ. Job sizes follow a power-law distribution, denoted by random variable S, where $\mathbf{P}\{S > x\} = x^{-2.5}$, for $1 \leq x < \infty$. Small jobs (those of size < 10) are routed to the first server, and large jobs (those of size ≥ 10) are routed to the second server.

(a) Derive the mean response time, $\mathbf{E}[T]$, for this system.

(b) What would $\mathbf{E}[T]$ be if the job size distribution were changed to $\mathbf{P}\{S > x\} = x^{-1.5}$, for $1 \leq x < \infty$.

24.2 PS Server Farm

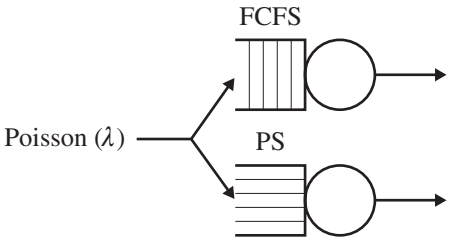
Consider a server farm with 2 identical PS hosts and SITA task assignment. Arrivals into the system occur according to a Poisson process with rate λ. Job sizes follow some (general) distribution. Prove that the SITA cutoff which minimizes mean response time is that which balances load between the two hosts.

24.3 Hybrid Server Farm

We are given a server farm with 2 identical hosts. Arrivals into the system occur according to a Poisson process with rate λ. Job sizes are denoted by the random variable S, with p.d.f. $f_S(t)$, where $0 \leq t \leq \infty$, and c.d.f. $F_S(t) = \mathbf{P}\{S < t\}$.

Because our job size distribution F_S has high variability, we decide to combat this by using a variant of size-interval scheduling, where small jobs are sent to server 1, where they are scheduled in FCFS order, and large jobs are sent to server 2, where they are scheduled according to PS. Assume that the size

cutoff is chosen such that load is balanced between "small" and "large" jobs. Specifically, $\rho = \frac{\lambda \mathbf{E}[S]}{2}$ represents both the load of the server farm and the load at each host. Suppose that this (balanced-load) size cutoff is 10. Thus a job is considered "small" if its size is less than 10, and is called "large" otherwise.



Write an expression for $\mathbf{E}[T]$, the mean response time experienced by an arriving job, as a function of ρ, λ, $f_S(t)$, and $F_S(t)$.

24.4 Equivalence of LWL and M/G/k

Assume that LWL and M/G/k are fed the same arrival sequence of jobs and are both run on k servers. Assume also that ties are resolved the same way in both systems. Prove by induction that each job is served by the same server in both the systems. Furthermore, the job begins and ends service at the same time in both systems.

24.5 One Fast Machine versus Two Slow Ones

Consider the question of whether one fast machine or two slow ones is better for minimizing mean waiting time, $\mathbf{E}[T_Q]$. Each slow machine works at half the rate of the single fast machine. Specifically, a job that requires s seconds of processing time on the fast machine will require $2s$ seconds of processing time on a single slow machine.

(a) Back in Chapter 14, we addressed this question for the M/M/k, where job sizes were drawn from an Exponential distribution. Which architecture (one fast or two slow) was superior there?

(b) Does the answer change when the job size distribution is not Exponential? Specifically, consider the following distribution of job sizes:

- $\frac{100}{101}$ fraction of the jobs have service requirement (size) 0.01 seconds when run on a slow machine. These are called small jobs.
- $\frac{1}{101}$ fraction of the jobs have service requirement (size) 1 second when run on a slow machine. These are called large jobs.

Observe that this job size distribution has the *heavy-tail* property, whereby the 1% largest jobs comprise about half the total load. Assume jobs arrive according to a Poisson process with rate λ, and consider two ways of processing the workload:

1. Use a single "fast" machine: M/G/1.
2. Use two slow machines and split the incoming jobs so that small jobs go to machine 1 and large jobs go to machine 2.

Compute $\mathbf{E}[T_Q]$ as a function of λ in both of these cases. Which case results in a lower $\mathbf{E}[T_Q]$? Explain what is going on. If we had defined the job size distribution to be less heavy-tailed, would the answer change?

24.6 To Balance Load or Not to Balance Load?

The purpose of this problem is to determine whether load balancing between two *identical* FCFS hosts is always a good idea for minimizing $\mathbf{E}[T_Q]$, or whether we might want to purposely *unbalance* load. Assume that jobs arrive from outside according to a Poisson process with rate λ, where S denotes the job size, and the system load is $\rho = \frac{\lambda \mathbf{E}[S]}{2} = 0.5$. Assume a Bounded Pareto$(k = .0009, p = 10^{10}, \alpha = 0.5)$ job size distribution with mean 3,000.

(a) First consider the SITA-E task assignment policy, where the size cutoff, x, is chosen to equalize the load at the two hosts (E stands for "equalize load").
   i. What is the cutoff x under SITA-E?
   ii. What fraction of jobs are sent to each host under SITA-E?
   iii. What is the mean delay, $\mathbf{E}[T_Q]$, under SITA-E?
   iv. What is $\mathbf{E}[T_Q]$ under RANDOM, and how does this compare with $\mathbf{E}[T_Q]$ under SITA-E?

(b) Returning to SITA-E, now purposely unbalance the load by dropping x, the SITA-E cutoff.
   i. Try different values of x and record the corresponding $\mathbf{E}[T_Q]$. Approximately how low should x be to minimize $\mathbf{E}[T_Q]$?
   ii. What is the load at each host under this new cutoff x?
   iii. What fraction of jobs are sent to each host under this new cutoff x?

24.7 A Better SITA-E?

Based on [10]. Consider a server farm with two identical FCFS hosts and SITA-E task assignment (see Exercise 24.6). Eitan proposes a new cutoff heuristic: rather than finding cutoffs which balance load, we instead derive cutoffs which equalize the expected number of queued jobs, $\mathbf{E}[N_Q]$, at each host. Determine whether Eitan's idea is useful by evaluating it on the workload from Exercise 24.6: Assume that jobs arrive from outside according to a Poisson process, with system load $\rho = \frac{\lambda \mathbf{E}[S]}{2} = 0.5$. Assume a *BP*$(k = .0009, p = 10^{10}, \alpha = 0.5)$ job size distribution with mean 3,000. Which cutoff scheme (balanced load or Eitan's scheme) is better for minimizing overall mean delay, $\mathbf{E}[T_Q]$? How does Eitan's heuristic for finding cutoffs compare with using the optimal cutoff?

24.8 Additional Recommended Problem

Exercise 21.7 on understanding the effect of job size variability in an M/G/2 system is recommended as well.