On Scheduling Redundant Requests with Cancellation Overheads

Kangwook Lee, Ramtin Pedarsani, and Kannan Ramchandran
Dept. of Electrical Engineering and Computer Sciences
University of California, Berkeley
{kw1jjang, ramtin, kannanr}@eecs.berkeley.edu

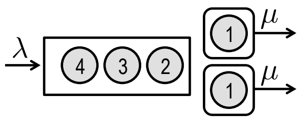
Abstract*—Reducing latency in distributed computing and data storage systems is gaining increasing importance. Several empirical works have reported on the efficacy of scheduling *redundant requests* in such systems. That is, one may reduce job latency by 1) scheduling the same job at more than one server and 2) waiting only until the fastest of them responds. Inspired by the empirically observed gains of such schemes, several theoretical models have been proposed to explain the power of using redundant requests. Although the proposed models in the literature provide useful insights such as when scheduling redundant requests can be beneficial, all these results rely heavily on a common assumption: *all redundant requests of a job can be immediately cancelled as soon as one of them is completed.

In this paper, we study how one should schedule redundant requests when such assumption does *not* hold. This is of great importance in practice since cancellation of running jobs typically incurs non-negligible delays. In order to bridge the gap between the existing models and practice, we propose a new queueing model that captures such cancellation delays. We then find how one can schedule redundant requests to achieve the optimal average job latency when cancellation delay is considered and accounted for. Our results show that even with a small cancellation overhead, the actual optimal scheduling policy differs significantly from the optimal scheduling policy when the overhead is zero.

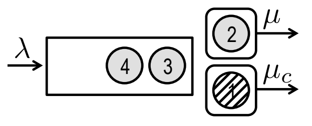
Further, we study optimal dynamic scheduling policies, which appropriately schedule redundant requests based on the number of jobs in the system. Our analysis reveals that for the two-server case, the optimal dynamic scheduler can achieve 7% to 16% lower average job latency, compared with the optimal static scheduler. This observation is in stark contrast to the known fact that the optimal static scheduler performs as well as the optimal dynamic scheduler when cancellation overhead is ignored, affirming that misleading conclusions result if the cancellation overhead is ignored completely.

I. Introduction

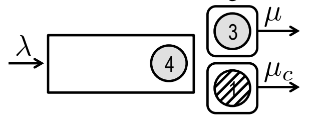
Individual components of a large-scale distributed system exhibit highly variable response times [1]. The high variability is due to many factors: shared resources, queueing delay in multiple layers, and hardware failures. Unfortunately, completely removing such sources of variability in large-scale systems is infeasible. As a result, researchers have proposed several approaches to achieve better latency performance while living with this high variability. One of the most promising approaches is that of scheduling *redundant requests* to multiple components or servers [1]–[5]. That is, one can schedule the same job at different servers and obtain the result from the request that responds first in order to reduce latency. Clearly, the technique of scheduling redundant requests can



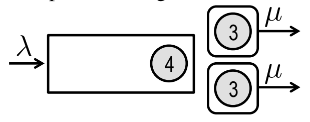
(a) $t = t_1$: 4 jobs are in the system. Job 1 is replicated and both server 1 and 2 are working on it.



(b) $t = t_2$: server 1 completes job 1, and starts working on job 2. Job 1's replica is being cancelled.



(c) $t = t_3$: server 1 completed job 2, and starts working on job 3. The copy of job 2 left the system.



(d) $t = t_4$: server 2 canceled the copy of job 1, and starts working on the replica of job 3.

Fig. 1: An example of scheduling redundant requests with cancellation overheads. Jobs being canceled are shaded. Job 1 experiences a faster response because two servers work on the same job as depicted in 1a, but the other jobs are delayed because server 2 takes a long time to cancel job 1, and some system resource is not utilized during the cancellation period. The scheduling policy used here is called π_∞, and it is formally defined in Section III-D.

be employed only if requests can be simultaneously served at many different servers. Many systems indeed possess this desired property: in a distributed computing and data storage system such as Apache Hadoop, a compute job can be served at any of the servers that stores a copy of the input data; in a network with multiple routes between nodes, the transmitter can choose to establish multiple flows across different paths, and can transmit redundant packets along them.

Many recent works [1]–[5] empirically observe reductions in job latency by scheduling redundant requests. Following the empirical efficacy of redundant requests, several recent works propose theoretical models of redundant requests scheduling, and find the optimal scheduling policies under the proposed models [5]–[7]. However, most existing models commonly assume immediate cancellation of straggling requests; i.e., *all requests except for the finished one are immediately removed from the system.* While this assumption enables an important first step in the analysis of systems with redundant requests, it limits the studied models in several ways. First, cancellation of requests in highly distributed systems typically takes a non-negligible amount of time for exchange of control signals and restoration of server states. Further, job cancellation is not even feasible in some applications where schedulers do not have full control over the entire system. That is, the scheduler can issue redundant requests but cannot revoke them. For instance, [4] considers requesting multiple DNS (Domain

Name Servers) queries to achieve a lower DNS response delay, and these requests cannot be canceled because the scheduler does not have control over them.

Figure 1 illustrates how the cancellation overheads negatively affect the overall system performance. In Figure 1a, job 1 is replicated, and it can be served quickly because two servers work on the same job. However, the other jobs are negatively affected if server 2 takes some time to cancel the copy of job 1: during the cancellation period, the system's resource is being wasted. It is therefore no longer clear whether redundant requests will help or not due to this phenomenon. More generally, it is unclear how one can optimally use redundant requests when accounting for this overhead. We attempt to answer this question in this paper.

Our contributions are as follows. First, bridging the evident gap between the existing models and practice, we propose a new model of redundant requests with cancellation overhead. In order to strike a balance between analytical tractability and capturing key system attributes, we propose a 'memoryless cancellation' model to capture the job cancellation overheads, and study how this cancellation overhead affects the latency performance of several scheduling policies. We show that *maximally scheduling redundant requests is inefficient even with almost negligible cancellation overhead.* This implies the limited applicability of theoretical results that do not consider such cancellation overhead. We also study the latency-optimal dynamic scheduling policy and show the necessity of using dynamic policies over static policies: we show that *the optimal dynamic policy can provide up to 16% additional latency reduction over the optimal static policy.* Our analysis applies a recently developed queueing analysis technique, called the Recursive Renewal Reward (RRR) technique [8].

The remainder of this paper is organized as follows. Section II discusses related literature. Section III describes the queueing model that captures redundant requests and cancellation overhead. Section IV and Section V provide analytical results for static and dynamic schedulers respectively. Finally, Section VI presents conclusions and discusses open problems.

II. Prior work

A. Systems-centric literature

Scheduling policies that try to reduce latency by sending redundant requests have been previously studied, largely empirically, in [2]–[5], [9]–[15]. These works evaluate system performance under redundant requests for several applications, and report reduction in the latency in many cases. Dean and Barroso [13] observe a reduction in latency in Google's system when requests are sent to two servers instead of one. Ananthanarayanan et al. [2] observe that job latency can be reduced by replicating small compute jobs only. Vulimiri et al. [4] observe that a DNS query can be handled quickly by issuing multiple copies of it to multiple servers.

Huang et al. [3] consider a distributed storage system where the data is stored using an $(n = 16,\ k = 12)$ Reed-Solomon code. Exploiting the flexibility provided by the code, they send requests to more than k storage nodes, and observe latency reduction by taking the first responding k reads, and decoding the original data using k coded chunks. Liang and Kozat [5] consider cloud storage services, perform experiments on the Amazon EC2 cloud, and observe latency reduction when arrival rates are lower than certain thresholds.

There has been little work on designing systems in which redundant requests can be efficiently scheduled. A notable exception is the recent works in distributed storage literature [16], [17]. In [16], Rashmi et al. propose storage codes that provides the scheduler with flexibility of connecting to more nodes than required and using the data received from a first subset to respond, treating the other slower nodes as 'erasures'. In [17], Rawat et al. construct codes that support a scaling number of parallel reads while keeping the storage overhead small.

B. Theory-centric literature

There has recently been a significant interest in models for understanding and analyzing redundant-request-based scheduling policies. Most of these models consider using redundant requests in distributed storage systems with a focus on the flexibility of storage codes. Shah et al. [6] study under what settings sending redundant requests helps. They show that when the service time is exponentially distributed, and there are no cancellation overheads, scheduling maximal redundant requests is the latency-optimal policy. They also characterize certain settings where redundant requests will *not* help irrespective of the cancellation cost. Chen et al. [18] present a similar result that any scheme that maximally schedules redundant requests is delay-optimal among all dynamic schedulers. In [7], Joshi et al. provide bounds on the average latency faced by a job when the requests are sent (redundantly) to *all* the servers. Kumar et al. [19] consider a general problem of a heterogeneous distributed storage with multiple classes of data. In [5], Liang and Kozat provide an approximate analysis of scheduling redundant requests using insights from their experiments on the Amazon EC2 cloud.

A few works have focused on the problem of scheduling redundant requests for general systems such as distributed computing clusters or grid computing. Kim considers the problem of scheduling redundant requests in grid computing systems in his thesis [20]. In [21], Koole et al. show that maximizing task replication stochastically maximizes the number of completed jobs if service time is exponentially distributed. In [22], Wang et al. consider the tradeoff between latency performance and resource usage in large-scale distributed computing systems. In [23], Wang considers a few variations of the problem. In [24], Qiu et al. consider a variety of service time distributions and arrival processes, and analyze latency performance using the matrix analytic method.

A few works including [5] and [23] consider dynamic scheduling policies. Liang et al. [5] propose dynamic scheduling policies based on an approximate analysis. In [23], Wang et al. study scheduling of finite number of jobs in the queue.

Our work differs from most of the existing works in two ways. First, we explicitly include the overhead of canceling redundant requests in the model, and find the optimal scheduling policy. Most of the existing works do not consider this overhead, and propose scheduling policies that may not perform well in practice. Second, we also study the dynamic scheduling policy, while most of the existing works have studied static scheduling policies only.

III. Model

In this section, we describe our queueing model that can capture scheduling and canceling redundant requests.

A. Arrival and service model

We assume that jobs arrive according to a Poisson process with rate λ and the service time of a job at each server is exponentially distributed with rate μ. This is a common assumption in queueing theory for analytical tractability. Further, we assume that copies of a job experience independent service times at different servers. This is a valid assumption in many applications: in distributed storage systems, reading an identical file at different disks of the same I/O performance incurs different latencies due to independent random access time; in distributed computing systems, processing an identical task may require different amounts of time depending on the status of servers, disks, and network.

B. Redundant requests

We assume that a job can be scheduled at more than one server: whenever a server becomes available, the scheduler may schedule a new job or a replica of a running job at the available server. When redundant requests are used, the job latency is defined as the difference between the *earliest* departure time of the job and the time of its arrival. When a job is scheduled at multiple servers, we assume that *the redundant jobs experience independent random service times.* The independence assumption is witnessed in previous works. For instance, [25] reports the independence between download times of a file from the cloud server across multiple threads.

C. Memoryless cancellation

Using redundant requests without prompt cancellation may result in resource wastage. Thus, one would ideally like the scheduler to cancel redundant jobs at the other servers when one of the redundant jobs is completed. We model the cancellation delay as random delays that are exponentially distributed. That is, when one of the replicas is fully processed, the other replicas start getting canceled, and the cancellation times are exponentially distributed with rate μ_c, *the cancellation rate*. We name this model of cancellation overhead as *memoryless cancellation model*. Note that the cancellation time of a job does not affect the latency of the job, but may delay the other waiting jobs.

Depending on the value of μ_c, the memoryless cancellation model falls into one of the three following scenarios[1].

1) ($\mu_c = \mu$: Infeasible cancellation) In some systems, it may be impossible or too complicated to cancel the replicas of a job. That is, one cannot cancel a running job even though copies of that job is already served at other servers. This can be simply modeled by setting $\mu_c = \mu$ due to the memoryless property of the exponential distribution. Even though using redundant requests seems very inefficient under this setup, we show that when the arrival rate is smaller than a certain threshold, one can still reduce job latency by using redundant requests.
2) ($\mu_c = \infty$: Immediate cancellation) The immediate cancellation of the redundant requests can be supported by systems that are very centralized and have a negligible communication delay. In other words, one can cancel any job that is submitted to a server if the identical job is served by another server, and the cancellation happens immediately. This can be captured by setting $\mu_c = \infty$.
3) ($\mu_c > \mu$: Slow cancellation) In most systems of interest, cancellation is possible but requires a small yet non-negligible amount of time. For this case, we assume that the cancellation of a job requires a random amount of time that is exponentially distributed with rate μ_c.

[1]We do not consider the degenerate case where $\mu_c < \mu$: by letting redundant jobs continue to run until completion without canceling them, one can achieve $\mu_c = \mu$

D. Scheduling policies

In a queueing system, the scheduler needs to make a scheduling decision when a resource becomes available. In this work, the system allows the scheduler to schedule a redundant request of a running job or a waiting job in the queue. This flexibility makes the problem of finding the optimal scheduling policy challenging. In this work, we restrict all scheduling policies to be First-come-First-served (FCFS), work-conserving, and non-preemptive. We consider both *static* and *dynamic* scheduling policies: a dynamic policy is one that adapts based on changes in the state of the system (e.g., the number of jobs in the system), whereas a static policy is oblivious to the changes in state [26].

Given a scheduling policy, say π, we consider two metrics: the maximum arrival rate and the average job latency. The system is stable under a scheduling policy π at arrival rate λ if $\mathbb{E}^{\pi}[N] < \infty$. The capacity region of a scheduling policy is defined as the set of all arrival rates at which the system is stable under π. The maximum arrival rate of a scheduling policy is defined as the supremum of its capacity region. The average job latency is defined as $\mathbb{E}^{\pi}[D] = \lim_{t\to\infty} \mathbb{E}[(\sum_{i=1}^{Z_t} D_i)/Z_t]$, where Z_t is the number of jobs that are completed before time t, and D_i is the latency of the ith completed job. We define a special class of scheduling policies $\{\pi_\gamma\}$ as follows.

Definition (Definition of π_γ) A scheduling policy π_γ schedules a redundant request of the least-recently arrived job if and only if the number of distinct jobs in the system at the time of decision $N(t)$ is less than or equal to $\gamma \in \{0, 1, \ldots\}$.

Note that π_0 and π_∞ are two special cases, and they are the only static policies among the class of the schedulers: π_0 never schedules a redundant request, which is the classic FCFS scheduler; π_∞ always schedules redundant requests regardless of the system state. All the other policies are dynamic policies: for $1 \leq \gamma < \infty$, π_γ requires the knowledge of the system status. Note that $\{\pi_\gamma\}$ does not cover the entire scheduling policies, and it rather is a specific family of policies.

E. M/M/2 with redundant requests

As a first attempt to understand the latency performance of redundant requests, and to find the latency-optimal scheduling policy, we focus on two-servers systems: M/M/2 queueing systems with redundant requests.

IV. Analysis of Static Schedulers

In this section, we analyze and compare the two static schedulers: π_0, which does not use any redundant request, and π_∞, which always schedules redundant requests. The analysis of these static scheduling policies are still useful when one wants to adapt redundant requests but cannot exploit the status of the system for some reasons. For instance, when the system is designed in a layered way, and the scheduling layer does not have much information about the current load of the system.

By finding the average job latency performance of both schedulers, we can answer important questions such as 'when should one use redundant requests?' and 'when is π_∞ better than π_0?'. In this section, we first analyze performance of π_0 and π_∞. We then present the optimal static scheduling policy.

A. Without redundant requests: $\boldsymbol{\pi_0}$

Since π_0 does not use any redundant request, an M/M/2/π_0 is identical to an M/M/2/FCFS. Thus, the following lemma immediately follows [26].

Lemma 1. *(Stability and job latency of M/M/2/π_0) The maximum arrival rate and the average job latency of an M/M/2/π_0 are as follows:* $\lambda^{\pi_0}_{max} = 2\mu, \quad \mathbb{E}^{\pi_0}[D] = 4\mu/(4\mu^2 - \lambda^2)$.

Figure 2 shows the average job latency of an M/M/2/π_0 system as a function of λ.

B. With redundant requests: $\boldsymbol{\pi_\infty}$

Now, we consider the other static scheduling policy π_∞. We first illustrate how redundant requests are scheduled using a sample evolution of M/M/2/π_∞ system illustrated in Figure 1. In Figure 1a, there are 3 jobs waiting in the queue, and job 1 is replicated, and being served by server 1 and server 2. In Figure 1b, server 1 finishes job 1 earlier than the other, and starts working on job 2 while the other server starts canceling the redundant request of job 1. Note that job 1 is deemed served at this point, and the remaining replica does not affect this job's latency. In Figure 1c, server 1 finishes job 2 and accepts job 3. In Figure 1d, server 2 completes job cancellation, and starts working on job 3's replica.

Since the arrival process, the service process, and the cancellation process are all memoryless and independent, an M/M/2/π_∞ system is Markovian, and the system can be modeled as a Markov chain by defining the state of the system appropriately. We enumerate all the states of an M/M/2 system in Table I. We illustrate the state notation using the example in Figure 1. In Figure 1a, 3 jobs are waiting in the queue, and two servers are working on the same job. We denote this state by $(s, 3)$, where s means that the same job is being processed by both servers, and 3 indicates the number of jobs in the queue. In Figure 1b, server 1 completely served job 1 and starts working on job 2 while server 2 is canceling job 1. We name this state as $(c, 2)$, where c stands for cancellation. Similarly, Figure 1c corresponds to state $(c, 1)$, and Figure 1d corresponds to state $(s, 1)$. With this definition of states, the Markov chain of the system can be found, and it is depicted in Figure 3. The bottom row shows all the states in which one of the two servers is canceling a job, and the top row shows all the states in which two servers are working on the same job. We distinguish the transitions corresponding to scheduling redundant requests with double tips.

S	Meaning	$L(S)$	$N(S)$
(0)	No job in the system	0	0
(s,i), $i \geq 0$	Two servers working on the same job, i waiting jobs in the queue.	$i{+}2$	$i{+}1$
$(c,-1)$	One job being canceled, no waiting job in the queue.	1	0
(c,i), $i \geq 0$	One server working on a job, the other server canceling a job, i waiting jobs in the queue.	$i{+}2$	$i{+}1$
$(d,-1)$	Only one server working on a job, no waiting job in the queue.	2	1
(d,i), $i \geq 0$	Two servers working on different jobs, i waiting jobs in the queue.	$i{+}3$	$i{+}2$

TABLE I: States of an M/M/2 system. $L(S)$ denotes the level of state S. $N(S)$ denotes the number of distinct jobs in the system in state S.

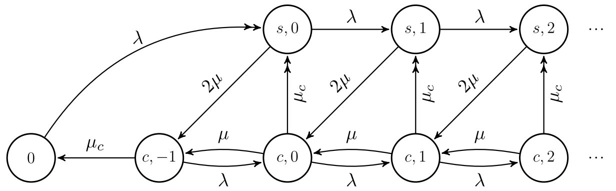


Fig. 3: A Markov chain of an M/M/2/π_∞. States are defined in Table I. States (s,i), $i \geq 0$ represent that both servers are working on the same job, and there are i jobs waiting in the queue. States (d,i), $i \geq 0$ represent that two servers are working on different jobs, and there are i jobs waiting in the queue. State (0) represent there is no job in the system, and state $(c,-1)$ represents there is only one job that is being canceled by one server.

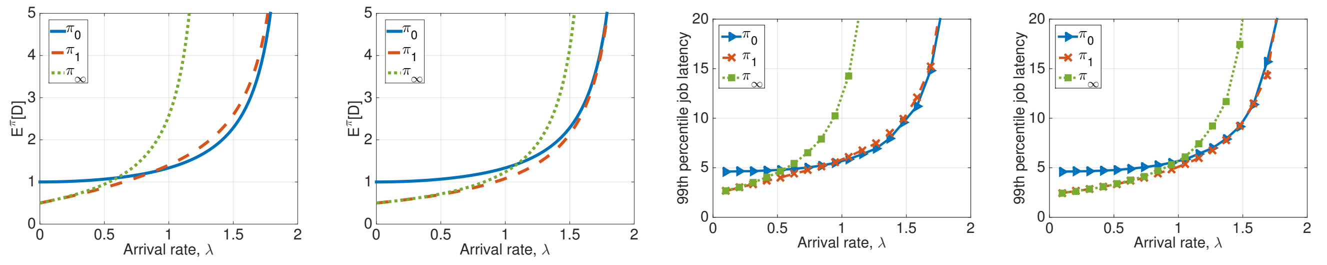
We are now ready to analyze the latency performance of the system. We first establish the relationship between the average number of 'distinct jobs' in the system and the average job latency of the system using the Little's Law [27].

Lemma 2. *(Little's Law with Redundant Requests) The average job latency of a queueing system that schedules redundant requests is* $\mathbb{E}[D] = \mathbb{E}[N]/\lambda$, *where* $\mathbb{E}[N]$ *is the average number of distinct jobs (excluding redundant requests and those being canceled).*

Proof. We ignore how redundant requests are scheduled and replicas are canceled, and consider just when distinct jobs enter and leave the system. Then, the entire system can be thought as a regular queueing system where distinct jobs arrive and depart. The lemma follows from the direct application of the Little's Law to this system. □

Lemma 2 implies that we can analyze the average job latency if we find the average number of distinct jobs in the system. We do so by using the recursive renewal reward technique, developed recently in [8]. Using this technique, we can exactly analyze the system's performance without relying on the numerical methods. We first present the stability region and the average job latency of an M/M/2/π_∞.

Theorem 3. *(Stability and job latency of M/M/2/π_∞) An M/M/2/π_∞ system is stable if and only if the arrival rate λ is strictly less than the maximum arrival rate λ_{max}, where λ_{max}*



(a) Average latency: $\mu_c = \mu$ (b) Average latency: $\mu_c = 5\mu$ (c) 99th percentile latency: $\mu_c = \mu$ (d) 99th percentile latency: $\mu_c = 5\mu$

Fig. 2: Average and tail latency of π_0(solid, blue), π_1(dashed, orange), and π_∞(dotted, green) with $\mu = 1,\ \mu_c = \mu$ and 5μ: π_0 does not schedule redundant requests, π_1 schedules them adaptively, and π_∞ maximally schedules them.

is as follows:

$$\lambda_{max}^{\pi_\infty} = 2\mu(\mu+\mu_c)(2\mu+\mu_c)^{-1}. \tag{1}$$

If the system is stable, the average job latency is as follows.

$$\begin{aligned}&\mathbb{E}^{\pi_\infty}[D] \\ &= \frac{(\mu+\mu_c)\left\{(2\mu_c(\mu+\mu_c)+\lambda(4\mu+\mu_c)\right\}}{\left\{2\mu_c(\mu+\mu_c)+\lambda(2\mu+\mu_c)\right)(2\mu(\mu+\mu_c)-\lambda(2\mu+\mu_c)\}}\end{aligned} \tag{2}$$

Proof. We first choose the state (0) as the home state. A renewal cycle is defined as the process of starting from the home state and returning back to the home state. We define as earning reward *the number of distinct jobs in the system*. Then, the average reward rate is equal to the mean reward earned over a cycle; that is,

$$\mathbb{E}[N] = \frac{\mathbb{E}[\text{accumulated reward over a renewal cycle}]}{\mathbb{E}[\text{length of a renewal cycle}]} \overset{\text{def}}{=} \frac{R}{T}.$$

For notational simplicity, we define $L(S)$, the level of state S, as shown in Table I. Note that $L(S) = N(S)+1$ except $S = (0)$. Also, we denote 'state s' simply by s unless it makes any confusion. Now, the average length of a renewal cycle T can be found as follows. Clearly, any renewal path must visit $(c,-1)$ to visit the home state. Thus, if we find the average length of the process from $(s,0)$ to $(c,-1)$, and that of the process from $(c,-1)$ to the home state, we can find T. By defining the average length of the process from state S to one level left of state S as T_S^L, T can be expressed as follows,

$$T = \lambda^{-1} + T_{s,0}^L + T_{c,-1}^L. \tag{3}$$

Now, consider $T_{s,0}^L$, the average length of the process from $(s,0)$ to one level left of $(s,0)$, which is $(c,-1)$. Depending on the first transition from $(s,0)$, there are two cases: if one of the two servers finishes the same job before a new job arrives, the process moves to $(c,-1)$; otherwise, the process moves to $(s,1)$. The average time staying at $(s,0)$ before any event happens is $(\lambda+2\mu)^{-1}$. With probability $2\mu(\lambda+2\mu)^{-1}$, the service event happens and the process moves to one level left. With probability $\lambda(\lambda+2\mu)^{-1}$, the arrival event happens and the process moves rightward to $(s,1)$. If this happens, as the process must come back to level 1 to visit the home state, the additional time required is $T_{(s,1)}^L + T_{(c,0)}^L$. Similar equations can be found also for $T_{c,-1}^L$ and $T_{c,0}^L$, and we have the following equations.

$$\begin{aligned}T_{s,0}^L &= (\lambda+2\mu)^{-1}\left[1+\lambda(T_{s,1}^L+T_{c,0}^L)+2\mu\cdot 0\right] \\ T_{c,-1}^L &= (\lambda+\mu_c)^{-1}\left[1+\lambda(T_{c,0}^L+T_{c,-1}^L)+\mu_c\cdot 0\right] \\ T_{c,0}^L &= (\lambda+\mu+\mu_c)^{-1}\left[1+\lambda(T_{c,1}^L+T_{c,0}^L)+\mu\cdot 0+\mu_c T_{s,0}^L\right]\end{aligned} \tag{4}$$

We now exploit the repeating structure of the Markov chain. Observe that $T_{s,1}^L$, the mean time from $(s,1)$ to one level left, is equal to $T_{s,0}^L$, the mean time from $(s,0)$ to one level left; two processes cannot experience any difference other than reward because of the repeating structure. Similarly, one can find the equivalence of $T_{c,1}^L$ and $T_{c,0}^L$. Thus, we have $T_{s,1}^L = T_{s,0}^L$ and $T_{c,1}^L = T_{c,0}^L$. Similarly, we find R, the average accumulated reward over a renewal cycle. We define the average accumulated reward of the process from state S to one level left of S as R_S^L. Noting that there is no reward in (0), R can be decomposed as $R = \frac{0}{\lambda} + R_{s,0}^L + R_{c,-1}^L$. Similarly, we can rewrite $R_{c,-1}^L, R_{s,0}^L$, and $R_{c,0}^L$.

$$\begin{aligned}R_{c,-1}^L &= (\lambda+\mu_c)^{-1}\left[\lambda(R_{c,0}^L+R_{c,-1}^L)+\mu_c\cdot 0\right] \\ R_{s,0}^L &= (\lambda+2\mu)^{-1}\left[1+\lambda(R_{s,1}^L+R_{s,0}^L)+2\mu\cdot 0\right] \\ R_{c,0}^L &= (\lambda+\mu+\mu_c)^{-1}\left[1+\lambda(R_{c,1}^L+R_{c,0}^L)+\mu\cdot 0+\mu_c R_{s,0}^L\right]\end{aligned} \tag{5}$$

We exploit the repeating structure again. Similar to the previous argument, the mean reward until from $(s,1)$ to one level left, is equal to $R_{s,0}^L$ plus $T_{s,0}^L$: the process starting from $(s,1)$ always gets one additional reward, and the total additional reward is equal to the length of the process.

$$\begin{aligned}R_{s,1}^L &= R_{s,0}^L + T_{s,1}^L = R_{s,0}^L + T_{s,0}^L \\ R_{c,1}^L &= R_{c,0}^L + T_{c,1}^L = R_{c,0}^L + T_{c,0}^L\end{aligned} \tag{6}$$

Now, the above equations can be solved, and the solutions of these equations are provided in Appendix VII. Then, using Lemma 2, one can find $E^{\pi_\infty}[D]$, and $\lambda_{\max}^{\pi_\infty}$. □

Note that the maximum arrival rate of an M/M/2/π_∞ system is strictly lower than that of an M/M/2/π_0 system since cancellation overheads induce wastage of system resource. However, the job latency performance under the policy π_∞ is still better if the arrival rate is low enough. Figure 2 compares the average job latencies of the two policies with different values for λ and μ_c: the solid (blue) line shows the average job latency of an M/M/2/π_0, and the dotted (green) line shows the average job latency of an M/M/2/π_∞. Observe that there

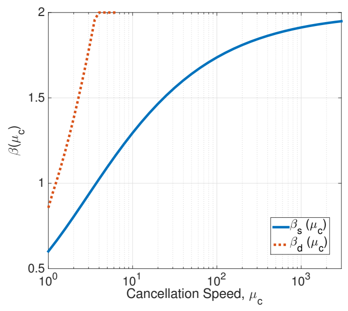


Fig. 4: Threshold function $\beta_s(\mu_c)$ and $\beta_d(\mu_c)$ defined in Theorem 4 and Theorem 6 with $\mu = 1$. The slow convergence of $\beta_s(\mu_c)$ explains how inefficient the policy π_∞ is even with a negligible cancellation overhead. For instance, $\beta_s(\mu_c) \simeq 1.25\mu = (0.625) \cdot 2\mu$ implies that the policy π_∞ becomes inefficient at 62.5% system if $\mu_c = 10\mu$.

exists a threshold after which π_∞ becomes worse than π_0, and this threshold depends on the cancellation overhead; studying this threshold fully characterizes the optimal static scheduling policies.

C. The latency-optimal static scheduler

We now compare π_0 and π_∞ with arbitrary cancellation overhead, and find the optimal static scheduling policy.

Theorem 4. *(The latency-optimal static scheduling policy for M/M/2) If the cancellation of jobs is immediate (i.e., $\mu_c = \infty$), the policy π_∞ is always latency-optimal. If the cancellation of jobs is infeasible (i.e., $\mu_c = 0$), the policy π_∞ is latency-optimal if and only if $\lambda < 0.6013\mu$, and the policy π_0 is latency-optimal otherwise. If the job cancellation rate is $\mu < \mu_c < \infty$, the π_∞ policy is latency-optimal if and only if $\lambda < \beta_s(\mu_c)\mu$, where $\beta_s(\mu_c)$ is the unique solution of the following third-order polynomial equation in β.*

$$\sum_{i=0}^{3} \alpha_i \beta^i = 0 \tag{7}$$

$$\alpha_3 = -(\mu_c+1)(\mu_c+4), \quad \alpha_2 = 4(\mu_c+2)^2 - \ell(\mu_c+1)$$
$$\alpha_1 = 4(2\ell - \mu_c + \ell\mu_c - \mu_c^2), \quad \alpha_0 = -4\ell(\mu_c+1)$$
$$\ell = 2\mu_c(\mu_c+1)$$

We call $\beta_s(\mu_c)$ the threshold function for static scheduling policies. For all $\mu_c > \mu$, $0.6013 < \beta_s(\mu_c) < 2$.

Proof. One can show that $\mathbb{E}^{\pi_0}[D] - \mathbb{E}^{\pi_\infty}[D]$ is strictly decreasing as λ increases. Also, when $\lambda \simeq 0$, the difference is strictly positive; i.e., $\lim_{\lambda\to 0} \mathbb{E}^{\pi_\infty}[D] = (2\mu)^{-1} < (\mu)^{-1} = \lim_{\lambda\to 0} \mathbb{E}^{\pi_0}[D]$. Moreover, π_∞ has a smaller stability region than π_0. Thus, the equation $\mathbb{E}^{\pi_\infty}[D]|_\lambda = \mathbb{E}^{\pi_0}[D]|_\lambda$ has a unique solution, and we call it $\beta(\mu_c)\mu$. Thus, solving the equation gives us the third-order equation. Further, since $\mathbb{E}^{\pi_\infty}[D]|_\lambda$ is a decreasing function of μ_c, $\beta_s(\mu_c)$ is an increasing function. Thus, one can find the minimum value of $\beta_s(\mu)$ by solving the equation with $\mu_c = \mu$. ☐

We plot the threshold function $\beta_s(\mu_c)$ in Figure 4 when $\mu = 1$. *Note that the threshold function $\beta_s(\mu_c)$ does not approach 2μ as quickly as one would expect.* For instance, $\beta_s(\mu_c) \simeq 1.25\mu = (0.625) \cdot 2\mu$, when $\mu_c \simeq 10\mu$, and $\beta_s(\mu_c) \simeq 1.9\mu = (0.95) \cdot 2\mu$, when $\mu_c \simeq 1000\mu$: the policy π_∞ becomes worse than the policy π_0 if the system load is higher than 62.5% even though cancellation overhead is as low as $\mu_c = 10\mu$. *That is, maximally scheduling redundant requests can utterly fail even with a low cancellation overhead.* In Section V, we show that an optimal dynamic scheduling policy can be used to avoid this phenomenon.

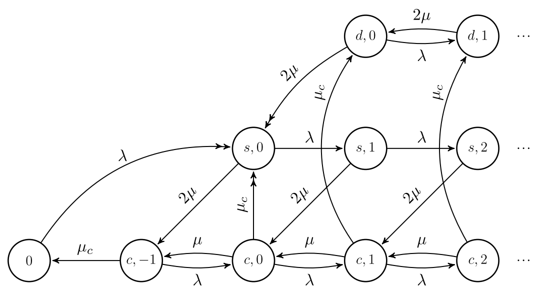


Fig. 5: The Markov chain for an M/M/2/π_1 system

V. Optimal Dynamic Schedulers

In this section, we analyze the performance of optimal dynamic schedulers.

A. The latency-optimal dynamic schedulers

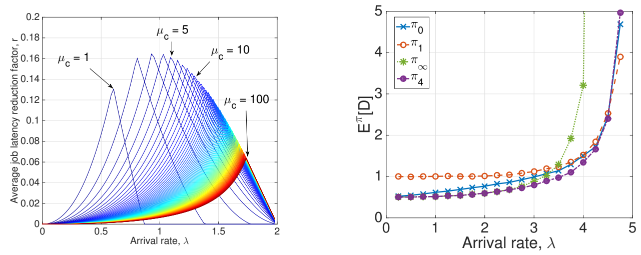
In this section, we provide main results characterizing the optimal dynamic schedulers. The detailed proofs are presented in Appendix. Our first result states that the average latency performance of a dynamic scheduling policy π_1 can be analyzed using the RRR technique.

Theorem 5. *(Stability and job latency of M/M/2/π_1) An M/M/2/π_1 system is stable if and only if the arrival rate λ is strictly less than the maximum arrival rate $\lambda_{max} = 2\mu$. If the system is stable, the average job latency $\mathbb{E}^{\pi_1}[D]$ can be exactly found using the RRR technique.*

Proof. We defer the detailed proof of the theorem to Appendix, and provide a brief outline of the proof. Similar to the proof of Theorem 3, we first draw the Markov chain of the system under the policy π_1. The Markov chain is depicted in Figure 5. Note that when a Markov chain starts from a state in the repeating portion, it can enter the one left level through one of two states, (d,i) or $(c,i+1)$ for some i. Thus, one needs to find with what probabilities the process enters (d,i) and $(c,i+1)$, respectively. This complicates the analysis, but one can still obtain the exact latency performance. ☐

In order to find the optimal dynamic scheduling policy, one could analyze the performance of $\{\pi_\gamma\}$ for all $\gamma \in \{0,1,\dots\}$ and find the best one at a given arrival rate. Surprisingly, it can be shown that the better one between π_1 and π_0 is indeed the optimal dynamic scheduling policy. This can be rigorously proved via dynamic programming. Due to lack of space, we omit the proof. Therefore, the optimal dynamic scheduling policy can be characterized as follows.

Theorem 6. *(The latency-optimal dynamic scheduling policy for M/M/2) The π_1 policy is latency-optimal if and only if $\lambda < \beta_d(\mu_c)\mu$. Otherwise, π_0 is optimal. The threshold function for*



(a) The average job latency reduction by adopting the optimal dynamic policy over the optimal static policy when $\mu = 1$

(b) Average job latency of π_0, π_1, π_∞, and π_4 for M/M/5 systems with $\mu = 1,\ \mu_c = 5\mu$.

Fig. 6: The value of dynamic scheduling and performance of $\{\pi_\gamma\}$ for M/M/n systems.

dynamic scheduling policy $\beta_d(\mu_c)$ is defined as the unique solution of the following equation.

$$\mathbb{E}^{\pi_1}[D]|_{\lambda=\beta\mu} = \mathbb{E}^{\pi_0}[D]|_{\lambda=\beta\mu} \tag{8}$$

For all $\mu_c > \mu$, $0.8685 < \beta_d(\mu_c) < 2$.

Proof. This theorem immediately follows from the application of Lemma 1 and Theorem 5. □

In Figure 4, $\beta_d(\mu_c)$ is plotted. Note that the threshold function $\beta_d(\mu_c)$ approaches 2μ much faster than $\beta_s(\mu_c)$. For instance, $\beta_d(\mu_c) \approx 2\mu$ when $\mu_c \approx 3\mu$: the dynamic policy π_1 becomes optimal at almost all arrival rates if cancellation overhead is $\mu_c \approx 3\mu$. This phenomenon can be also observed in Figure 2: when $\mu_c = 5\mu$, π_1 is strictly better than π_0 at all arrival rates. This observation is in stark contrast to the known fact that the optimal static scheduler performs as good as the optimal dynamic scheduler when cancellation overhead is ignored [6], affirming that misleading results may be derived if the cancellation overhead is completely ignored.

B. Performance gap between the optimal static and the optimal dynamic policies

We found the optimal static schedulers and the optimal dynamic schedulers in Section IV and Section V, respectively. In this section, we compare the optimal static scheduling policy and the optimal dynamic policy. Denote the optimal static policy as π_s^* and the optimal dynamic policy as π_d^*. Theorem 4 and 6 imply that π_s^* achieves the lower envelope of π_0 and π_∞, and π_d^* achieves the lower envelope of π_0 and π_1. In Figure 4, the optimal static policy achieves the lower envelope of the dotted (green) curve and the solid (blue) curve, whereas the dynamic policy achieves the lower envelope of the dashed (orange) curve and the solid (blue) curve.

We define the average job latency reduction factor r by using the optimal dynamic policy over the optimal static policy as follows: $r = (\mathbb{E}^{\pi_s^*}[D] - \mathbb{E}^{\pi_d^*}[D])/\mathbb{E}^{\pi_s^*}[D]$. By studying *the value of dynamic scheduling*, we can help system designers make a right choice between superior performance of dynamic scheduling policies and (possibly) higher cost involved with building and running such dynamic policies.

Using Theorem 4 and Theorem 6, we find r and plot it in Figure 6a. We observe that the reduction factor $r \in [0.07, 0.16]$ for $\mu \leq \mu_c \leq 100\mu$. That is, in general, *the optimal dynamic schedulers can reduce the average job latency by 7% to 16% at the cost of implementation and deployment of dynamic schedulers.* The maximal reduction factor is observed when $3 < \mu_c < 4$, which is an unexpected phenomenon.

VI. Conclusion

We study how one can optimally schedule redundant requests in the presence of cancellation overheads. We propose a new queueing model of redundant requests with cancellation overheads, find the latency-optimal scheduling policies for M/M/2 systems, and present several observations, that may help in the design of a more efficient scheduling algorithm with redundant requests.

There are many open problems related to this work. First of all, we believe that similar structural results of the optimal dynamic policies can be shown for M/M/n, but the exact analysis of latency performance is still open. It is empirically observed that the class of simple threshold policies $\{\pi_\gamma\}$ perform well even for M/M/n systems. Figure 6b shows the average job latency under different scheduling policies. We compare the four policies π_0, π_1, π_∞ and π_4. Upto a certain threshold, π_4 provides the best performance, and then π_1 becomes dominant beyond the threshold. We conjecture that appropriate choice of $\{\pi_\gamma\}$ can achieve near-optimal latency performances for general M/M/n systems.

Another important metric to be studied is tail latency. Figure 2 plots the 99th percentile job latency obtained via simulations. We observe similar tail latency tradeoffs and conjecture that similar optimal policy structures can be found for tail latency, but rigorous analysis is open.

References

[1] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, pp. 74–80, 2013.

[2] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Why let resources idle? Aggressive cloning of jobs with Dolly," in *USENIX HotCloud*, Jun. 2012.

[3] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in Windows Azure Storage," in *USENIX Annual Technical Conference (ATC)*, Jun. 2012.

[4] A. Vulimiri, O. Michel, P. Godfrey, and S. Shenker, "More is less: Reducing latency via redundancy," in *11th ACM Workshop on Hot Topics in Networks*, Oct. 2012, pp. 13–18.

[5] G. Liang and U. Kozat, "Fast cloud: Pushing the envelope on delay performance of cloud storage with coding," *arXiv:1301.1294*, Jan. 2013.

[6] N. B. Shah, K. Lee, and K. Ramchandran, "When do redundant requests reduce latency?" in *Annual Allerton Conference on Communication, Control, and Computing*, 2013.

[7] G. Joshi, Y. Liu, and E. Soljanin, "Coding for fast content download," in *50th Annual Allerton Conference on Communication, Control, and Computing*, Oct. 2012.

[8] A. Gandhi, S. Doroudi, M. Harchol-Balter, and A. Scheller-Wolf, "Exact analysis of the m/m/k/setup class of markov chains via recursive renewal reward," *SIGMETRICS Perform. Eval. Rev.*, vol. 41, no. 1, pp. 153–166, Jun. 2013. [Online]. Available: http://doi.acm.org/10.1145/2494232.2465760

[9] A. C. Snoeren, K. Conley, and D. K. Gifford, "Mesh-based content routing using xml," in *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, 2001, pp. 160–173.

[10] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. N. Rao, "Improving web availability for clients with MONET," in *USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2005, pp. 115–128.

[11] M. J. Pitkänen and J. Ott, "Redundancy and distributed caching in mobile dtns," in *Proceedings of 2nd ACM/IEEE international workshop on Mobility in the evolving internet architecture*. ACM, 2007, p. 8.

[12] D. Han, A. Anand, A. Akella, and S. Seshan, "Rpt: Re-architecting loss protection for content-aware networks," in *USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2011.

[13] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[14] C. Stewart, A. Chakrabarti, and R. Griffith, "Zoolander: Efficiently meeting very strict, low-latency SLOs," in *International Conference on Autonomic Computing*, 2013.

[15] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan, "Reducing web latency: the virtue of gentle aggression," in *ACM SIGCOMM*, 2013.

[16] K. V. Rashmi, N. B. Shah, K. Ramchandran, and P. Kumar, "Regenerating codes for errors and erasures in distributed storage," in *Proc. International Symposium on Information Theory*, Jul. 2012.

[17] A. S. Rawat, D. S. Papailiopoulos, A. G. Dimakis, and S. Vishwanath, "Locality and availability in distributed storage," *CoRR*, vol. abs/1402.2011, 2014.

[18] S. Chen, Y. Sun, U. C. Kozat, L. Huang, P. Sinha, G. Liang, X. Liu, and N. B. Shroff, "When queueing meets coding: Optimal-latency data retrieving scheme in storage clouds," *arXiv preprint arXiv:1404.6687*, 2014.

[19] A. Kumar, R. Tandon, and T. C. Clancy, "On the latency of erasure-coded cloud storage systems," *arXiv preprint arXiv:1405.2833*, 2014.

[20] Y. Kim, "Resource management for large scale unreliable distributed systems," Ph.D. dissertation, University of California, Berkeley, 2009.

[21] G. Koole and R. Righter, "Resource allocation in grid computing," *Journal of Scheduling*, vol. 11, no. 3, pp. 163–173, 2008.

[22] D. Wang, G. Joshi, and G. Wornell, "Efficient task replication for fast response times in parallel computation," *arXiv preprint arXiv:1404.1328*, 2014.

[23] D. Wang, "Computing with unreliable resources: design, analysis and algorithms," Ph.D. dissertation, Massachusetts Institute of Technology, 2014.

[24] Z. Qiu, J. F. Pérez, and P. G. Harrison, "Assessing the impact of concurrent replication with canceling in parallel jobs," 2014.

[25] G. Liang and U. C. Kozat, "Tofec: Achieving optimal throughput-delay trade-off of cloud storage using erasure codes," *CoRR*, vol. abs/1307.8083, 2013.

[26] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*, 1st ed. New York, NY, USA: Cambridge University Press, 2013.

[27] J. D. C. Little, "A proof for the queuing formula: $L = \lambda W$," *Operations Research*, vol. 9, no. 3, pp. 383–387, 1961.

VII. Appendix

Proof of Theorem 3. In order to complete the proof, we solve the equations (3)-(7) in Theorem 5. We can exactly solve them since they are 8 linear equations with 8 variables after replacing several terms using the recursive structure. We provide the solutions of the equations here.

$$\begin{aligned}
T &= -\frac{\mu\,\left(\lambda\,\mu_c + 2\,\lambda\,\mu + 2\,\mu_c\,\mu + 2\,\mu_c{}^2\right)}{\lambda\,\mu_c\,\left(\lambda\,\mu_c + 2\,\lambda\,\mu - 2\,\mu_c\,\mu - 2\,\mu^2\right)}\\
T^L_{c,-1} &= -\frac{2\,\mu\,(\mu_c+\mu)}{\mu_c\,\left(\lambda\,\mu_c + 2\,\lambda\,\mu - 2\,\mu_c\,\mu - 2\,\mu^2\right)}\\
T^L_{s,0} &= -(\mu_c+\mu)\left(\lambda\,\mu_c + 2\,\lambda\,\mu - 2\,\mu_c\,\mu - 2\,\mu^2\right)^{-1}\\
T^L_{c,0} &= -(\mu_c+2\,\mu)\left(\lambda\,\mu_c + 2\,\lambda\,\mu - 2\,\mu_c\,\mu - 2\,\mu^2\right)^{-1}\\
R &= \frac{\mu\,(\mu_c+\mu)\,\left(\lambda\,\mu_c + 4\,\lambda\,\mu + 2\,\mu_c\,\mu + 2\,\mu_c{}^2\right)}{\mu_c\left(\lambda\,\mu_c + 2\,\lambda\,\mu - 2\,\mu_c\,\mu - 2\,\mu^2\right)^2}\\
R^L_{c,-1} &= \frac{\lambda\,\mu\,\left(2\,\mu_c{}^2 + 6\,\mu_c\,\mu - 1\,\lambda\,\mu_c + 4\,\mu^2\right)}{\mu_c\left(\lambda\,\mu_c + 2\,\lambda\,\mu - 2\,\mu_c\,\mu - 2\,\mu^2\right)^2}\\
R^L_{s,0} &= \frac{\mu\,\left(\lambda^2 - \lambda\,\mu_c - \lambda\,\mu + 2\,\mu_c{}^2 + 4\,\mu_c\,\mu + 2\,\mu^2\right)}{\left(\lambda\,\mu_c + 2\,\lambda\,\mu - 2\,\mu_c\,\mu - 2\,\mu^2\right)^2}\\
R^L_{c,0} &= \frac{\mu\,\left(2\,\mu_c{}^2 + 6\,\mu_c\,\mu - 1\,\lambda\,\mu_c + 4\,\mu^2\right)}{\left(\lambda\,\mu_c + 2\,\lambda\,\mu - 2\,\mu_c\,\mu - 2\,\mu^2\right)^2}.
\end{aligned}$$

□

Proof of Theorem 5. Recall that π_1 is the policy that redundant request is used only when the queue is empty. To analyze this policy we use the RRR technique in [8].

We first write the first step equations to find T, which is the expected return time to state (0) starting from state (0). Let T^L_S be the the expected time that the Markov process hits a state that is one level left of state S. Let q be the probability that the first left state that is visited from $(s,1)$ is $(s,0)$. Let r be the probability that the first left state that is visited from $(c,1)$ is $(c,0)$. Then, one can write the following first step equations:

$$\begin{aligned}
T &= (\lambda)^{-1} + T^L_{s,0} + T^L_{c,-1}\\
T^L_{c,-1} &= (\mu_c+\lambda)^{-1}\left[1 + \lambda(T^L_{c,0} + T^L_{c,-1})\right]\\
T^L_{s,0} &= (\lambda+2\mu))^{-1}\left[1 + \lambda(T^L_{s,1} + qT^L_{s,0} + (1-q)T^L_{c,0})\right]\\
T^L_{c,0} &= (\lambda+\mu+\mu_c)^{-1}\\
&\quad\cdot\left[1 + \mu_c T^L_{s,0} + \lambda(T^L_{c,1} + (1-r)T^L_{s,0} + rT^L_{c,0})\right]\\
T^L_{d,0} &= (\lambda+2\mu)^{-1}\left[1 + \lambda(T^L_{d,0} + T^L_{d,0})\right]\\
T^L_{s,1} &= (\lambda+2\mu)^{-1}\left[1 + \lambda(T^L_{s,1} + qT^L_{d,0} + (1-q)T^L_{c,1})\right]\\
T^L_{c,1} &= (\lambda+\mu+\mu_c)^{-1}\\
&\quad\cdot\left[1 + \mu_c T^L_{d,0} + \lambda(T^L_{c,1} + (1-r)T^L_{d,0} + rT^L_{c,1})\right]
\end{aligned}$$

Note that in the last three equations we used the following birth-and-death property of the Markov chain: $T^L_{d,1} = T^L_{d,0}$, $T^L_{c,1} = T^L_{c,0}$, and $T^L_{s,1} = T^L_{s,0}$. Moreover, one can find the probabilities q and r by solving the following equations:

$$q = (\lambda+2\mu)^{-1}\left[2\mu\cdot 0 + \lambda(q + (1-q)(1-r))\right] \tag{9}$$

$$r = (\lambda+\mu+\mu_c)^{-1}\left[\mu\cdot 1 + \mu_c\cdot 0 + \lambda\cdot r^2\right] \tag{10}$$

Thus, solving the above equations one can write T as a long but closed form expression.

Similarly we can write the first step equations to find R, the average accumulated reward over a renewal cycle. Let R^L_S be the expected reward gained from S until the first visit to a left state. Then,

$$\begin{aligned}
R &= R^L_{s,0} + R^L_{c,-1},\quad R^L_{c,-1} = \lambda(\lambda+\mu_c)^{-1}(R^L_{c,0} + R^L_{c,-1})\\
R^L_{s,0} &= (\lambda+2\mu)^{-1}\left[1 + \lambda(R^L_{s,1} + qR^L_{s,0} + (1-q)R^L_{c,0})\right]\\
R^L_{c,0} &= (\lambda+\mu+\mu_c)^{-1}\\
&\quad\cdot\left[1 + \mu_c R^L_{s,0} + \lambda(R^L_{c,1} + rR^L_{c,0} + (1-r)R^L_{s,0})\right]\\
R^L_{d,0} &= (\lambda+2\mu)^{-1}\left[2 + \lambda(R^L_{d,0} + T^L_{d,0} + R^L_{d,0})\right]\\
R^L_{s,1} &= (\lambda+2\mu)^{-1}\left[2 + \lambda(R^L_{s,1} + T^L_{s,1} + qR^L_{d,0} + (1-q)R^L_{c,1})\right]\\
R^L_{c,1} &= (\lambda+\mu+\mu_c)^{-1}\\
&\quad\cdot\left[2 + \mu_c(R^L_{c,1} + T^L_{c,1} + (1-r)R^L_{d,0} + rR^L_{c,1})\right].
\end{aligned}$$

Note that in the last three equations, we used the following relationships exploiting the structure of the Markov chain: $R^L_{d,1} = T^L_{d,0} + R^L_{d,0}$, $R^L_{s,2} = R^L_{s,1} + T^L_{s,1}$, and $R^L_{c,2} = R^L_{c,1} + T^L_{c,1}$. Thus, one can solve the equations exactly to find R in closed form. By applying Lemma 2, we obtain the average job latency in closed form; that is, $\mathbb{E}^{\pi_1}[D] = \frac{1}{\lambda}\frac{R}{T}$. □