Modeling and Analyzing Systems with Redundancy

Kristen Gardner

CMU-CS-17-112

May 2017

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Mor Harchol-Balter, Chair
Guy Blelloch
Anupam Gupta
Alan Scheller-Wolf
Rhonda Righter, UC Berkeley

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation under grant numbers CMM-1538204, CCF-1629444, and DGE-1252522, Intel ISTC-CC, the Siebel Scholarship Foundation, the Google Women Techmakers Scholarship, and Microsoft.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: redundancy, replication, task assignment, dispatching, scheduling, queueing theory, stochastic processes, Markov models, RIQ, Redundancy-d, LRF, PF

To my family.

Abstract

Reducing latency is a primary concern in computer systems. As cloud computing and resource sharing become more prevalent, the problem of how to reduce latency becomes more challenging because there is a high degree of variability in server speeds. Recent computer systems research has shown that the same job can take 12 times or even 27 times longer to run on one machine than another, due to varying background load, garbage collection, network contention, and other factors. This server variability is transient and unpredictable, making it hard to know how long a job will take to run on any given server, and therefore how best to dispatch and schedule jobs.

An increasingly popular strategy for combating server variability is *redundancy*. The idea is to create multiple copies of the same job, dispatch these copies to different servers, and wait for the first copy to complete service. A great deal of empirical computer systems research has demonstrated the benefits of redundancy: using redundancy can yield up to a 50% reduction in mean response time. Unfortunately, there is very little theoretical work analyzing performance in systems with redundancy.

This thesis presents the first exact analysis of response time in systems with redundancy. We begin in the Independent Runtimes (IR) model, in which a job's service times (runtimes) are assumed to be independent across servers. Here we derive exact expressions for the distribution of response time in a certain set of class-based redundancy systems. We also propose two new scheduling policies, Least Redundant First (LRF) and Primaries First (PF), and prove that LRF minimizes overall system response time, while PF is fair across classes of jobs with different redundancy degrees.

While the IR model is appropriate in certain settings, in others it does not make sense because the independence assumption eliminates any notion of an "inherent job size." The IR model leads to the conclusion that more redundancy is always better, which often is not true in practice. Therefore we propose the $S\&X$ model, which is the first model to decouple a job's inherent size (X) from the server slowdown (S). This model is important because, unlike prior models, it allows a job's runtimes to be correlated across servers. The $S\&X$ model makes it evident that redundancy does not always help: in fact, too much redundancy can lead to instability. To overcome this, we propose a new dispatching policy, Redundant-to-Idle-Queue, which is provably stable in the $S\&X$ model, while offering substantial response time improvements compared to systems without redundancy

Acknowledgments

So many people have positively influenced my intellectual, professional, and personal growth during my time as a graduate student. First and foremost, I want to thank Mor Harchol-Balter, who has been an exceptional advisor for the past five years. Mor's enthusiasm for research is unparalleled; I owe a great deal of my intellectual development to her constant willingness to schedule another meeting, try a new proof approach, and when all else fails find a new problem to study. I am equally grateful to Mor for her commitment to promoting my professional development, both by always looking out for opportunities for me to give talks and apply for awards, and by offering wide-ranging advice on almost every topic imaginable.

Alan Scheller-Wolf has effectively been a second advisor to me. In addition to always having a new research idea to explore and providing valuable feedback on my writing, Alan has been endlessly supportive of my career goals and has been an outstanding role model for maintaining balance in an academic career. Thanks to Rhonda Righter for being a wonderful host during my visit to Berkeley and for continuing to be my collaborator and mentor. I am particularly grateful to Rhonda for her persistence when trying to prove tricky results, especially when the proof turns out to require Poisson arrivals after all. I also would like to thank the rest of my thesis committee, Guy Blelloch and Anupam Gupta, for their thoughtful questions and feedback on my work.

Over the course of my graduate career, I have been fortunate to work with a number of collaborators who became my co-authors on the work presented in this thesis: Sherwin Doroudi, Esa Hyytiä, Benny van Houdt, Mark Velednitsky, and Sam Zbarsky. Thanks also to Evan Cavallo, Jan-Pieter Dorsman, Brandon Lieberthal, John Wright, Danny Zhu, and Timmy Zhu for the many hours of discussion and many lines of code they contributed to my work. Thanks to Sem Borst for hosting my visit to Bell Labs in summer 2013; I very much enjoyed our collaboration and learned a great deal from Sem during that summer.

Outside of research, a great many people in the School of Computer Science have played an important role in making my time at CMU the positive experience that it has been. Women@SCS has been an important source of community for me, and I would like to thank Carol Frieze and Mary Widom for everything that they do for the organization and for supporting and encouraging me throughout my time as a student. Thanks also to the wonderful administrative staff in the Computer Science Department, in particular Deb Cavlovich, Nancy Conway, and Catherine Copetas, for keeping the department running and handling every problem and question that arises, no matter how bizarre.

Finally, I have truly enjoyed my years at CMU, and for this I credit my "moral support committee": my parents and sister, Wendy, Tom, and Allie Gardner; and all the friends who have shared my experiences here. Thanks for musicals and knitting, for kayaking and skating, for humoring my love of Harry Potter and chocolate cake, for Hanabi, and for all those hours of Avalon. You have made CMU a home for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Reducing latency is a primary concern in queueing systems. Typically, latency-reducing strategies fall into one of two categories: *dispatching*—to which server should an arriving job be sent?—and *scheduling*—which job in the queue should run next when the server becomes available? Common solutions include dispatching an arriving job to the queue with the fewest jobs (Join-the-Shortest-Queue, JSQ) or the smallest amount of work (Least-Work-Left, LWL), or scheduling the job with the Shortest Remaining Processing Time (SRPT). Ideas like JSQ perform well when the number of jobs in the queue is a good indicator of the amount of work in the queue. Policies like LWL and SPRT require knowing for how long each job will occupy the server at the moment when the job arrives. Unfortunately, often in practice it is difficult to know or estimate job sizes in advance, and, depending on the job size distribution, the number of jobs in the queue can offer a very poor estimate of the amount of work in the queue. Hence policies that sound good in theory can be difficult to implement, or can lead to unexpectedly poor performance.

The problem is exacerbated by the growing prevalence of virtualization and cloud computing, which allow resources to be shared among many users. This resource sharing can cause a great deal of *server-side variability*, meaning that a job's runtime (the time from when the job enters service until it completes service)[1] depends not only on the inherent work associated with the job, but also on server-dependent factors such as resource contention, network overhead, disk seek time, etc. Not only does server-side variability make it difficult to predict job runtimes—making policies like SRPT or LWL impossible to implement—but it also means that a job can experience a very different runtime on one server than that same job would experience on a different server, even if the servers nominally have the same speeds and capabilities. For example, suppose job j runs on a particular server s_1, where it competes for resources with 5 other jobs, and job j takes 10 seconds to complete service. If instead job j ran on server s_2, where it would compete for resources with 2 other jobs, perhaps job j would complete service in only 6 seconds. But it is not as simple as just knowing the number of jobs competing for service at each server. Server-side variability can manifest itself in a variety of forms that are often unpredictable and can change over time, and can significantly affect a job's runtime. For example, a high delay when scheduling a virtual machine on a physical machine, combined with high delay at a network switch, can cause the same job to take 27 times longer to run on one server than on another [76]. In the face of this

[1]The runtime is often called the "service time" in the queueing literature; we use the term "runtime" to avoid confusion with "server-side variability."

server-side variability, we need new strategies for reducing latency.

One increasingly popular strategy for overcoming server-side variability is called *redundancy*. The idea is to create multiple copies of the same job, dispatch these copies to different servers, and wait for only the first copy to complete service. Redundancy helps reduce latency for two reasons. First, it allows a job to experience the shortest queueing time across all servers at which the job joins the queue. Second, a job that enters service on multiple servers gets to experience the minimum runtime across all these servers. Empirical work measuring the effects of redundancy has demonstrated that redundancy can reduce mean response time—defined as the moment from when a job arrives to the system until the moment when its first copy completes service—by 20% [22]. Improvements are even bigger at tail percentiles of response time: redundancy can reduce the 99th percentile of response time by 38% [22].

But redundancy is not free: one of the major concerns about implementing redundancy-based policies is that there are costs associated with redundancy that can outweigh its potential benefits. When a job runs on multiple servers, it uses additional resources, thereby adding load to the system. There is a risk that the extra load can cause subsequent jobs to be delayed or can even cause the system to become unstable. In order to limit the amount of extra load added to the system, a job's extra copies can be cancelled as soon as the first copy completes service. But this, too, has an overhead: it can take time to cancel the redundant copies, particularly if this involves, for example, shutting down a virtual machine. Redundancy also increases the amount of network bandwidth required simply because a redundant request must be sent to multiple servers. The hope is that it is possible to design redundancy-based policies that do not incur prohibitively high costs.

Unfortunately, very little theoretical work exists on analyzing redundancy systems. Without theoretical analysis, it is difficult to develop principled answers to the key questions that arise in designing redundancy systems. How many redundant copies of each job should we create? How much additional response time benefit do we get from increasing the number of copies per job? Is there a point at which we have gone too far, and further additional copies are harmful? Is there a point beyond which the system becomes unstable? How does redundancy compare to other common load balancing strategies? What if not all jobs can create redundant copies? Does redundancy unfairly penalize jobs that cannot become redundant? How can we fix this? Answering these questions, and others, is critical to designing systems that best leverage redundancy's potential to achieve good performance.

The central goals of this thesis are (1) to develop new analysis of systems with redundancy, thereby providing insight regarding the benefits and costs of redundancy, and (2) to design redundancy-based dispatching and scheduling policies that improve performance.

1.1 System Model

Throughout much of this thesis, we discuss redundancy in the context of computer systems. There are many different ways in which computer systems and the jobs served therein can be structured. Here we describe the characteristics of the systems that are our focus in this thesis.

Centralized vs. distributed systems. Multi-server systems follow one of two structures. In a *centralized* system, all servers share a single queue. All jobs that arrive to the system join this queue, and when a server becomes available it chooses its next job from among all jobs in the system according to some policy. In a *distributed* system (e.g., Sparrow [59]), each server has its own queue. When a job arrives to the system it is dispatched to one (or more) server(s) according to some policy and joins the queue at the selected server(s). When a server becomes available it chooses its next job from among those jobs in its own queue. If there are no jobs in the server's queue the server must idle, even if there are jobs in the queues at other servers. Note that it is possible to define system structures that use a hybrid of the centralized and distributed settings.

In this thesis we focus on the **distributed setting**. This makes sense in large systems with hundreds or thousands of servers, in which a single central queue can become a bottleneck.

Single-task jobs vs. multiple-task jobs. There are two common models for how jobs are structured. In the first, each job consists of multiple subtasks, all of which must complete service in order for the job to be considered complete. In a redundancy system, each individual task can be replicated to multiple servers. This model is based on the structure of jobs in systems such as MapReduce. In the second model, each job consists of only a single task. This **single-task** model is a more traditional setting in queueing theory, and is our focus in this thesis.

Constrained vs. flexible redundancy structures. One of our most important policy design decisions is to choose to which servers to dispatch each job's redundant copies. In one system model, this choice is constrained because each job is only capable of running on a fixed subset of the servers. For example, this could be because of data locality constraints: a job can only run on a server at which its data is stored. In the second model, each job can run on any of the servers, and we must choose to how many servers and to which servers to dispatch the job.

In this thesis we study both of these settings. We begin by considering **constrained** redundancy structures (Chapters 3 and 4) and then move on to **flexible** structures (Chapters 5 and 6).

Here we have discussed only our high-level assumptions about the *structure* of the redundancy systems we study. These assumptions all have to do with the types of systems on which we choose to focus: there exist some real systems that follow a distributed structure and others that exhibit a centralized structure. Likewise, in some systems jobs are made up of one task while in others jobs consist of many tasks. The assumptions we make here serve to narrow the focus of this thesis to one particular system structure.

1.2 Policy Design and Theoretical Modeling Assumptions

We differentiate the structural assumptions discussed in Section 1.1, which relate to the inherent characteristics of the underlying systems we are studying, from the modeling assumptions we make when developing mathematical abstractions of redundancy systems, and the policy design assumptions we make about which types of policies are permissible. Much of this thesis is about the ways in which our mathematical modeling and policy design decisions affect performance. Here we provide an overview of some of these modeling choices.

Cancel-on-Start vs. cancel-on-complete. All redundancy policies involve creating multiple copies of a job, but there are several alternatives for when the extra copies are removed from the system. Under *cancel-on-start* policies, a job's copies wait in the queue at multiple servers and the extra copies are cancelled immediately as soon as the first copy *enters* service. Here a job is *not* allowed to have multiple copies in service simultaneously. Under *cancel-on-complete* policies, the extra copies are cancelled immediately as soon as the first copy *completes* service. Here a job may have multiple copies in service simultaneously on different servers. It is also possible to design *no-cancellation* redundancy policies, under which all copies that are created are required to run to completion.

The choice of which cancellation policies are appropriate can depend on the particular system characteristics. For example, when sending redundant packet transmissions in a network, it is not possible to cancel the extra transmissions, so only no-cancellation policies are appropriate. In other settings such as file downloads or MapReduce computations, it is possible to cancel extra copies both while they are still in the queue and while they are in service. Here whether it is better to implement a cancel-on-start or cancel-on-complete policy depends on factors such as the cancellation cost (i.e., how long it takes to cancel extra copies while they are in the queue vs. in service), the system load, and the amount of server-side variability.

In most of this thesis, we focus on **cancel-on-complete** redundancy policies. We briefly consider **cancel-on-start** policies in Section 6.8.

Independent vs. correlated runtimes. A key decision in modeling systems with redundancy is whether a job's runtimes (service times) across different servers are independent or correlated.

In the *Independent Runtimes* (IR) model, when copies of a job enter service on multiple servers, each copy's runtime is drawn *independently* from some distribution. Here there is a single random variable that is meant to capture both job-dependent and server-dependent factors that influence how long the job is in service at a particular server. Using a single random variable to represent runtime is the standard modeling choice in queueing theory in general. Accordingly, nearly all of the existing analytical work on redundancy systems assumes the IR model.

The IR model is appropriate for settings in which the particular server on which a job runs is the dominant factor in determining the job's runtime. One example of a system in which this is the case is the deceased donor organ transplant waitlist. In the United States, a separate waitlist is maintained for each geographic region; typically a patient joins the waitlist for the region in which she lives. But in some cases, patients can redundantly join waitlists for multiple regions. In this setting, a patient's "runtime" represents the time until an organ becomes available when the patient is at the head of the queue. It is reasonable to imagine that organs become available in different regions according to independent processes (as deaths occur). A patient waiting at the head of the queue in multiple different regions receives a transplant after waiting for the minimum "runtime" across these regions.

The assumptions of the IR model are also reasonable in settings in which inherent job sizes—i.e., the amount of work a job requires—are very small, whereas server-side variability may be high. For example, a simple web search query may involve very little computational work and very little bandwidth. However, if there is significant network contention or if the server to which the query is dispatched is highly loaded from other workloads, it may take a long time for the

query to complete. In this case, the job's inherent size is negligible compared to the server-side variability, which dominates the job's runtime. Because the server-side variability is likely to be independent across servers, it is reasonable to assume that the job's runtimes are independent across servers.

However, in applications in which a job's inherent size is not negligible compared to the server-side variability, a *Correlated Runtimes* (CR) model is more appropriate. For example, to access a file stored on disk, first the disk rotates and the disk head seeks the starting location of the file, and then the file is transferred. The rotation and seek times are server-dependent because they are determined by the initial position of the disk head. The transfer time is job-dependent because the same number of bytes must be transferred regardless of which disk is used to access the file. If the file is very large, the job-dependent file size may not be negligible relative to the server-dependent rotation and seek times. Here, our modeling assumptions should reflect the fact that a job's runtimes are correlated, rather than independent, across servers.

In this thesis we study both independent and correlated runtime models. We begin by working within the traditional **Independent Runtimes** model (Chapters 3, 4, and 5). We then turn to settings in which a **Correlated Runtimes** model is more realistic. There is no previously existing CR model in the queueing literature, hence we introduce a new model, called the **S&X model** (Chapter 6).

1.3 Thesis Statement

Redundancy is a powerful technique that has the potential to improve performance significantly in multi-server queueing systems. However, redundancy also is a risky strategy that can degrade performance severely under the wrong design choices. Through *mathematical analysis of redundancy systems*, we obtain a better understanding of *when redundancy helps and hurts*, which allows us to design *new scheduling and dispatching policies* for redundancy systems that are analytically tractable, provably stable, and offer significant response time improvements relative to systems without redundancy.

1.4 Contributions

In this section we outline the major contributions of this thesis. Our contributions fall into three categories: modeling and analysis, policy design, and insights that help us understand redundancy systems.

We begin by studying redundancy systems within the IR model. Initially, we consider constrained redundancy systems in which jobs have a class-based structure, meaning that each class of jobs sends its redundant copies to a fixed subset of the servers (where different classes' subsets may overlap). We assume that servers work in first-come first-served (FCFS) order, and, often, that runtimes are exponentially distributed.

- **[Modeling and analysis]** We derive the first exact analysis of response time in systems with redundancy. We model the system using a Markov chain and derive the limiting distribution on the state space. The state space is very detailed and, in general, difficult to aggregate to

find the distribution of the number of jobs in the system and the distribution of response time. We do so in small systems with 2 or 3 servers. (Chapter 3)

- **[Modeling and analysis]** We develop a state aggregation approach to move from the limiting distribution on the state space (Chapter 3) to the per-class distribution of response time in systems with any number of servers, where the job classes follow a particular redundancy structure called a nested structure. (Chapter 4)
- **[Insights]** We observe that redundancy is not always fair, meaning that redundant classes of jobs can benefit at the expense of non-redundant classes. (Chapter 3)

While FCFS is a natural scheduling policy, scheduling is a powerful technique that can greatly affect system performance. We use scheduling to try to mitigate the negative aspects of redundancy seen above.

- **[Policy design]** We propose the Least Redundant First (LRF) scheduling policy, under which each server processes the job in its queue that has the fewest redundant copies. We prove that LRF is optimal with respect to minimizing overall system response time. (Chapter 4)
- **[Policy design]** We propose the Primaries First (PF) scheduling policy, under which each job designates one primary copy and zero or more secondary copies; each server gives the primary copies in its queue strict preemptive priority over secondary copies. We prove that PF is fair in the sense that it does not hurt any class of jobs relative to a system with no redundancy. (Chapter 4)

The class-based redundancy structure assumes that jobs are constrained in the set of servers to which they can send their copies. Often this is not the case; any server is capable of processing every job, and we need to decide what degree of replication to use. In flexible redundancy systems, we study the Redundancy-$\mathbf{d}$ dispatching policy, under which each job that arrives to the system sends copies to $\mathbf{d}$ servers chosen uniformly at random.

- **[Modeling and analysis]** Starting with our limiting distribution on the system state space (Chapter 3), we develop a state aggregation approach to derive mean response time in a system with any number of servers under the Redundancy-$\mathbf{d}$ policy. (Chapter 5)
- **[Modeling and analysis]** We derive the full distribution of response time under the Redundancy-$\mathbf{d}$ policy. Our analysis uses a novel differential equations approach in the limit as the number of servers approaches infinity. (Chapter 5)
- **[Insights]** Our analysis allows us to evaluate the effect of $\mathbf{d}$ on response time. We find that as $\mathbf{d}$ increases, mean response time decreases, indicating that in the IR model, more redundancy is better. (Chapter 5)

In Chapters 3-5 we assume the IR model. But, as discussed in Section 1.2, in many systems a CR model is more appropriate, and the lessons learned in the IR model do not necessarily carry over to the CR setting. We next turn our attention to understanding redundancy under the CR model.

- **[Modeling and analysis]** We propose a new CR model, called the $S\&X$ model, which is the first model to explicitly decouple the job's inherent size from the server-dependent factors that affect its runtime, thereby enabling runtimes to be correlated across servers. (Chapter 6)

- **[Insights]** We find that in the $S\&X$ model, more redundancy is not always better. Instead, redundancy adds load to the system, potentially leading to poor performance or even overload. We observe that in order to leverage the benefits of redundancy without incurring too many costs, redundancy-based policies must limit the amount of extra load added to the system. (Chapter 6)
- **[Policy design]** We propose the Redundant-to-Idle-Queue (RIQ) dispatching policy, under which each job is only allowed to run replicas on servers that are idle when the job arrives. This ensures that load is only added to the system when the system has sufficient capacity to process the extra load, hence redundancy never leads to instability. We derive an approximate analysis of response time under RIQ. (Chapter 6)
- **[Policy design]** We propose several further improvements on the RIQ policy, all of which are "smart" redundancy policies that use the system state to determine whether it is safe to replicate arriving jobs. (Chapter 6)

1.5 Outline

The remainder of this thesis is organized as follows:

- In Chapter 2 we review prior work on redundancy systems and other related queueing models.
- In Chapter 3 we present our Markov chain analysis and analyze response time in redundancy systems with 2 or 3 servers. [33, 34]
- In Chapter 4 we study scheduling in redundancy systems. We introduce the Least Redundant First policy and prove its optimality, and we introduce the Primaries First policy and prove that it is fair. [29]
- In Chapter 5 we introduce and study the Redundancy-$\mathbf{d}$ dispatching policy. We present our analysis of the mean and distribution of response time under Redundancy-$\mathbf{d}$, and we study the impact of $\mathbf{d}$ on response time. [32, 35, 36]
- In Chapter 6 we introduce the $S\&X$ model. We propose and analyze the Redundant-to-Idle-Queue (RIQ) dispatching policy, and present several variants on RIQ that further improve performance. [30, 31]
- In Chapter 7 we conclude and discuss directions for future work.

Chapter 2

Related Work

While redundancy has been implemented in practice for the past decade, it has only begun to receive attention in the theoretical community in the past few years. In this section, we begin by discussing applications of redundancy and empirical work measuring the benefits and costs of redundancy (Section 2.1). We then review related work on redundancy systems, much of which has been developed concurrently to the work presented in this thesis (Section 2.2). We also review the prior work on several existing queueing models that are related to redundancy systems (Section 2.3).

2.1 Applications of Redundancy

2.1.1 Computer Systems

In recent years, redundancy has become a popular technique for reducing latency in computer systems. For example, Google's BigTable service uses two different forms of redundant requests [22]. *Tied requests* are a cancel-on-start form of redundancy, in which multiple copies of the same job are dispatched to different servers and the extra copies are cancelled as soon as the first copy *enters* service. Similarly, the Sparrow system issues multiple requests per job and proactively cancels the extra requests when the first copy enters service [59]. This is different from the cancel-on-complete form of redundancy studied in most of this thesis, in which extra copies are cancelled as soon as the first copy *completes* service. BigTable's second type of redundancy, called *hedged requests*, fits within our cancel-on-complete model of redundancy. Using hedged requests can reduce mean response time by 21% and the 99th percentile of redundancy by 38%. However, [22] notes that "naive implementations of this technique typically add unacceptable additional load" to the system. Hence there is a need to develop scheduling and dispatching policies that limit the amount of extra load added by redundancy.

In MapReduce systems, too, running redundant copies of tasks is a common technique for straggler mitigation. The original MapReduce system implemented a strategy called *backup tasks*, also known as *speculative execution*, in which if a task runs for too long an additional copy of that task can be launched [23]. This significantly reduces mean response time while only increasing resource usage by a few percent.

Following the introduction of MapReduce, a series of papers presented further improvements on the algorithms used for speculative execution. The LATE system chooses to run a speculative copy of the task that is expected to complete service farthest in the future and caps the number of speculative copies that can run at once [78]. Mantri takes a probabilistic approach that allows a speculative copy to run only if (a) the new copy will complete service before the old copy with high probability, or (b) the expected total amount of resources used by all copies of a task decreases if the new copy runs [6]. A common aspect of each of these approaches is that there is some delay between when the original copy of a task enters service and when the extra speculative copies are launched. The Dolly system proposes an alternative approach: replicate all of a job's tasks as soon as the job arrives to the system, but only replicate jobs that have a small number of tasks [4]. Further variations and improvements on speculative execution include Hopper, which introduces a speculation-aware scheduling policy that reserves some servers to run speculative copies [60], and GRASS, which proposes a speculation policy specifically designed for approximation jobs [5]. In all of this work, an important goal is limiting the amount of redundancy so that the system load does not increase appreciably.

Redundancy has also been proposed as a technique to reduce latency for DNS queries, database queries, and network flows [71]. In these systems it is difficult to cancel the extra copies at all, so all copies are allowed to run to completion. Hence the system load increases by a factor of $\mathbf{d}$ when $\mathbf{d}$ copies are created per job. This causes the system to become unstable at much lower arrival rates than when there is no redundancy [71]. One suggestion for preventing this problem is to give the extra copies low priority over the "original" copies [71]. We consider a similar scheduling policy in Chapter 4.

2.1.2 Organ Transplant Waitlists

In 2009, a unique aspect of Steve Jobs's liver transplant made headlines: even though Jobs lived in California, his transplant was performed in Tennessee [20]. Typically, a patient waiting for a deceased donor organ in the United States puts his name on the waitlist for the geographic region in which he lives. In addition to joining the waitlist in California, Jobs issued a redundant request by also joining the waitlist in Tennessee. This is called *multiple listing* in organ transplant waitlist systems; in Jobs's case, multiple listing reduced the time he had to wait to receive a transplant when a liver became available sooner in Tennessee than in California.

Multiple listing is becoming an increasingly common strategy to reduce the waiting time for deceased donor organ transplants: it allows patients to experience the minimum waiting time across several waitlists [54]. While Steve Jobs's famous multiple listing was for a liver transplant, multiple listing is most common on kidney transplant waitlists. Multiple listing is an extremely effective strategy for reducing the time it takes for a patient to receive a kidney: patients who multiple list wait on average half as long as those who are listed it only one region [54]. In addition, multiple listing can help to reduce geographic disparities in waiting times [7]. Unfortunately, multiple listing is not terribly common. In 2003, only about 6% of kidney patients were multiple listed [54]. Some patients may be unaware that multiple listing is an option; others may be unable to multiple list because it is too costly to travel to alternative transplant centers. The company OrganJet was founded to combat these problems by facilitating transportation of patients to remote transplant centers, thereby increasing the proportion of patients who are able to multiple list [8].

2.2 Analyzing Redundancy Systems

2.2.1 Single-Task Model

In this thesis we focus on the single-task model, in which only one copy of a job needs to complete service; this scenario has been studied in several other papers, all within the IR model. In [71], approximations for response time are derived for a system where each job sends copies to multiple randomly chosen servers. This is similar to the Redundancy-$\mathbf{d}$ policy that we introduce in Chapter 5, but unlike in our work, in [71] extra copies are not cancelled upon completion of the first copy. This no-cancellation assumption greatly simplifies the analysis because as the number of servers grows large, one can view each server as being an independent M/M/1 queue. When extra copies are cancelled, we can no longer view the system as independent M/M/1s.

In the single-task model, most of the work on scheduling focuses on understanding how the runtime distribution affects the optimal number of redundant copies to run. Typically it is assumed that any job can run on any server, and that all jobs wait in a central queue. When runtimes follow a new-better-than-used distribution, it is optimal to schedule the same job on all servers at the same time [48, 62]. Unlike in our model, in [48, 62] jobs do not have classes that restrict the set of servers to which each job can replicate, so the result does not extend to our system. We propose and analyze scheduling policies for class-based redundancy systems in Chapter 4. Furthermore, while from [48, 62] it is clear that more redundancy is better, neither paper analyzes response time as a function of the degree of redundancy, as we do in Chapter 5.

2.2.2 Multiple-Task Model

Most of the theoretical literature on redundancy systems focuses on the multiple-task model, in which each job consists of multiple tasks, all of which must complete service. One example of a multiple-task model is the (n, k) system (also called an MDS queue), in which there are n servers and each arriving job sends a request to all n servers. In order for the job to be complete, $k \leq n$ of the requests must complete service. When $k = n$, the system is equivalent to the fork-join system (see Section 2.3.2). The $k < n$ scenario is motivated by the coding theory community: it is common for a file to be stored using an (n, k) code, which stores the file as n pieces, any k of which must be retrieved in order to reconstruct the entire file.

The (n, k) fork-join system was first proposed in [43], and bounds and approximations were derived in [43, 44, 61]. These bounds typically involve designing bounding Markov chains in the case of exponential runtimes [61], or using order statistics to bound response time [43, 44]. In the more general case in which different classes of jobs can have different replication degrees k_i, bounds on response time have been derived, also using order statistics [49]. Most of this work assumes the IR model. In one case, an alternative CR model is considered in which runtimes are correlated across servers via taking a weighted average of a job-dependent random variable and a server-dependent random variable [43]. However, the paper only considers correlated runtimes in an M/G/∞ system, which greatly simplifies the analysis because there is no queueing.

In [62], a variation on the (n, k) fork-join system was proposed in which each job sends copies to $r \leq n$ of the servers and is complete when $k \leq r$ of these copies finish service. Both [61] and [62] use coupling arguments to identify classes of runtime distributions for which it is optimal

for jobs to be fully redundant ($r = n$) and non-redundant ($r = 1$) in both central-queue and distributed-queue models. However neither [61] nor [62] provides any analysis quantifying mean response time as a function of r. In an alternative version of the (n, k, r) system, the servers are "batched" into smaller (r, k) systems; bounds on response time follow from the (n, k) response time bounds in [44].

Several papers in the multiple-task model also study the question of how to schedule jobs and tasks. The Blocking-one Scheduling policy, under which none of a job's tasks can enter service until the previous job has completed, was proposed as an analytically tractable scheduling policy in the (n, k) system [41]. However, this policy is known to be sub-optimal because it can force servers to idle while there is still work in the queue. Optimal and near-optimal scheduling policies were developed for the (n, k) system [65, 66]. These policies are based on the Shortest Remaining Processing Time scheduling policy, and choose both which job(s) to serve and whether to schedule any redundant copies of tasks. Scheduling has also been used to try to reduce both response time and resource consumption [73].

2.3 Non-redundancy Systems

2.3.1 Coupled Processor/Cycle Stealing

In a *coupled processor* system, there are two servers and two classes of jobs, A and B. Server 1 works on class A jobs in FCFS order, and server 2 works on class B jobs in FCFS order. However, if there are only jobs of one class in the system, the servers "couple" to serve that class at a faster rate. This differs from redundancy systems, in which the same job runs on multiple servers but waits for only the first copy to complete. When runtimes are exponential and independent across servers, "coupling" to run at a faster rate and waiting for the minimum runtime are mathematically equivalent, but they are not the same under general runtime distributions or correlated runtimes. Furthermore, in a coupled processor system class A jobs only get to use server 2 when the system is empty of B's (and vice-versa); the scheduling policies allowed in redundancy systems are much more general.

Generating functions for the stationary distribution of the queue lengths in a two-server coupled processor system with exponential runtimes were derived in [24, 47], but this required solving complicated boundary value problems and provided little intuition for the performance of the systems. The stationary distribution of the workload in the two-server system was derived in [19] using a similar approach. In [40], a power-series approach was used to numerically compute the queue-length stationary distribution in systems with more than two servers under exponential runtimes. Much of the remaining work on coupled processor models involves deriving bounds and asymptotic results (for example, [15]).

In the donor-beneficiary model (one-way cycle stealing), only one class of jobs (the beneficiary) receives access to both servers, typically only when no jobs of the other (donor) class are present. In addition, if there is only one beneficiary job present, one server must idle (the servers do not "couple"). The donor-beneficiary model has been studied, in approximation, in a variety of settings [39, 58]. However, it differs from the redundancy model because a job is never in service at more than one server.

2.3.2 Fork-Join

Another related model is the *fork-join* system, in which each job that enters a system with k servers splits into k pieces, one of which goes to each server. The job is considered complete only when all k pieces have completed service. This is different from the redundancy model because only one redundant request needs to finish service in the redundancy model. Furthermore, a fork-join job sends work to all k servers, whereas in a redundancy system jobs can send copies to any subset of the servers. The fork-join model is known to be very difficult to analyze. Many papers have derived bounds and approximations for such a system (for example, [9, 10, 46, 57, 75]). Exact analysis remains an open problem except for the two-server case [25, 26]; see [17] for a more detailed overview. The (n, k) system discussed above in Section 2.2 is a recent generalization of the fork-join system.

2.3.3 Flexible Server Systems

A third related model is the *flexible server* system, in which each class of jobs has its own queue, and each server can serve some subset of classes. Jobs that arrive to the system wait in a central queue (or possibly multiple central queues, where each queue corresponds to a different job class). When a server becomes available, it chooses the queue from which to take its next job according to some policy. By contrast in the distributed redundancy systems that we study, each server has its own queue and jobs are routed to a subset of servers upon arrival. The design and performance of flexible server systems has been studied in many papers, e.g., [12, 64, 68, 69]. A special case of the flexible server system uses the following policy. When a server becomes available, it chooses the job that arrived earliest from among the jobs it can serve (i.e., servers work in FCFS order among compatible jobs). For this model, under a specific routing assumption when an arriving job sees multiple idle servers, the stationary distribution that satisfies the balance equations is given [2, 70]. Our redundancy model requires no such routing assumption, because arriving redundant jobs enter service at *all* idle servers. Nonetheless, the form of the stationary distribution in our model is mathematically similar to that in [2, 70].

The key difference between flexible server systems and redundancy systems is that redundant jobs may be served by multiple servers simultaneously, whereas in a flexible server system, each job may be processed by only one server. Interestingly, this type of flexible server system can be seen as a redundancy system in which the extra copies are cancelled as soon as the first copy *enters* service rather than when the first copy *completes* service. While most of this thesis focuses on the cancel-on-complete model of redundancy, we consider the cancel-on-start model in Section 6.8.

Chapter 3

First Exact Analysis

3.1 Introduction

Recent empirical computer systems research has demonstrated that redundancy can lead to significant reductions in response time. For example, Google's BigTable service uses redundancy to reduce mean response time by over 20%, with even bigger improvements in the tail percentiles of response time [22]. But in order to design redundancy-based dispatching and scheduling policies that are guaranteed to perform well, we must answer many questions about how redundancy systems behave. By how much can redundancy reduce response time? How should we share resources among jobs with different degrees of redundancy? Does redundancy benefit more-redundant jobs at the expense of less-redundant jobs? How does redundancy compare to other common load-balancing strategies? A principled approach to designing efficient redundancy systems requires answering these questions from a theoretical perspective.

To this end, our goal in this chapter is to provide the first exact analysis of response time in systems with redundancy. We study a system with any number of servers k, and any number of job classes ℓ. A *class* of jobs is associated with a set of servers that indicates the servers to which jobs of that class dispatch their redundant copies. The class can be thought of as being imposed by, for example, data locality constraints that determine a particular set of servers on which each job can run. We allow the system to follow redundancy structure (i.e., a bipartite graph indicating which job classes are associated with which servers), as shown in Figure 3.1.

In this chapter, we study redundancy within the IR model. We further assume that a job's runtimes are exponentially distributed. As discussed in Chapter 1, these assumptions are appropriate in settings in which a job's runtime depends primarily on server-side factors rather than on inherent properties of the job itself. For example, in organ transplant waitlist systems, the time it takes for an organ to become available in one region is independent from the time it takes for an organ to become available in a different region.

Our main contributions are as follows:

A new product-form limiting distribution. Our analytical approach involves modeling the system using a Markov chain. A very complex state space is required to capture the intricate system dynamics. It is not enough to know the number of jobs in each queue, or even the number

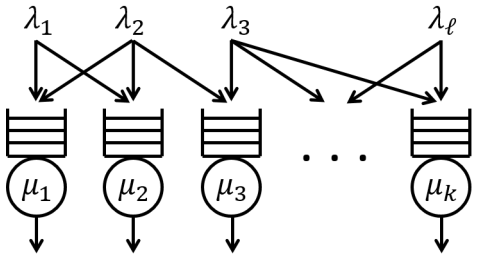


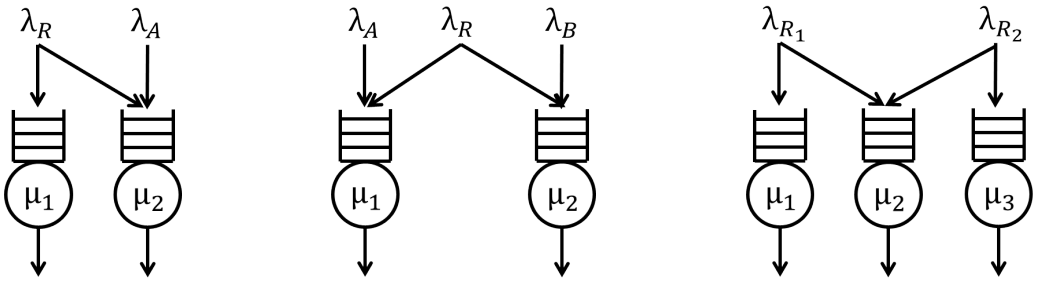
Figure 3.1: An example of our redundancy model. Each server j provides service at rate μ_j. Each class of jobs i arrives to the system as a Poisson process with rate λ_i and joins the queue at all servers in $S_i = \{j \mid \text{server } j \text{ can serve class } i\}$.

of jobs of each class within each queue. Rather, one needs to track the exact position and class of every job in every queue, so that one knows which jobs to delete when a copy of a redundant job completes service. Nonetheless, we derive the limiting distribution on the state space. We find that the limiting distribution exhibits an unusual form that resembles a product-form solution.

First exact analysis of response time in redundancy systems. While our limiting distribution result holds for a very general system structure, it is difficult in general to move from the limiting distribution on the state space to the distribution of response time. We derive *per-class response time distributions* for three simple models, shown in Figure 3.2. In the $\mathbb{N}$ model (Figure 3.2(a)), there are two arrival streams of jobs. One class is non-redundant and joins the queue at only one server, while the other class is redundant at both servers. The $\mathbb{N}$ model illuminates the response time benefit to the redundant class and the pain to the non-redundant class. In the $\mathbb{W}$ model (Figure 3.2(b)), we imagine that we have a stable system, where each server is serving its own class of jobs, when a new class of jobs arrives that can be processed at either server. We then turn to the $\mathbb{M}$ model (Figure 3.2(c)), where each job class has its own dedicated server, and there is an additional shared server which can be used by all job classes.

Understanding when redundancy outperforms alternative dispatching policies. Redundancy is just one option for reducing response time. Other common dispatching policies used for load balancing include optimally probabilistically splitting the load among all allowed servers (Opt-Split) and dispatching each job to the shortest of all allowed servers (JSQ). We investigate how the approach of making $\mathbf{d}$ redundant copies of each job compares with optimally probabilistically splitting the load among $\mathbf{d}$ queues, or with joining the shortest of $\mathbf{d}$ queues. Furthermore, while redundancy may benefit the redundant job class, what is the response time penalty experienced by the non-redundant jobs in the system? Do other approaches create less of a penalty than redundancy? Finally, if one class of jobs is redundant, does that class suffer when other classes become redundant as well? We use the $\mathbb{N}$, $\mathbb{W}$, and $\mathbb{M}$ models as case studies to explore these questions.

The remainder of this chapter is organized as follows. In Section 3.2 we formalize our model; define the $\mathbb{N}$, $\mathbb{W}$, and $\mathbb{M}$ models; and state the theorem that gives the limiting distribution for



(a) $\mathbb{N}$ model (b) $\mathbb{W}$ model (c) $\mathbb{M}$ model

Figure 3.2: (a) The $\mathbb{N}$ model. Class A jobs join the queue at server 2 only, while class R jobs are redundant at both servers. (b) The $\mathbb{W}$ model. Class A jobs join the queue at server 1 only, class B jobs join the queue at server 2 only, and class R jobs are redundant at both servers. (c) The $\mathbb{M}$ model. Class R_1 jobs are redundant at servers 1 and 2, and class R_2 jobs are redundant at servers 2 and 3.

the general system. In Sections 3.3, 3.4, and 3.5, we discuss detailed results for the $\mathbb{N}$, $\mathbb{W}$, and $\mathbb{M}$ models respectively. Section 3.6 addresses how these models scale as the number of servers increases. In Section 3.7 we consider what happens when we relax our modeling assumptions. Finally, in Section 3.8 we discuss the implications of this work and describe future research directions building on this work. This chapter is based on joint work with Sherwin Doroudi, Samuel Zbarsky, Mor Harchol-Balter, Esa Hyytiä, and Alan Scheller-Wolf and appears in the following papers:

- Kristen Gardner, Samuel Zbarsky, Sherwin Doroudi, Mor Harchol-Balter, Esa Hyytia, and Alan Scheller-Wolf. "Reducing Latency via Redundant Requests: Exact Analysis." In *SIGMETRICS*, June 2015. [33]
- Kristen Gardner, Samuel Zbarsky, Sherwin Doroudi, Mor Harchol-Balter, Esa Hyytia, and Alan Scheller-Wolf. "Queueing with redundant requests: exact analysis." *Queueing Systems*, 83(3):227-259, 2016. doi:10.1007/s11134-016-9485-y. [34]

3.2 Model and Limiting Distribution

We consider a system with k servers, denoted $1, 2, \ldots, k$, and ℓ classes of jobs, denoted $1, 2, \ldots, \ell$. (see Figure 3.1). The runtime at server j is distributed exponentially with rate μ_j for all $1 \leq j \leq k$, and each server processes the jobs in its queue in first-come first-served (FCFS) order. Each class of jobs i arrives to the system as a Poisson process with rate λ_i, and replicates itself by joining the queue at some fixed subset of the servers $S_i = \{j \mid \text{server } j \text{ can serve class } i\}$. Jobs in class i cannot join the queue at any server $j \notin S_i$. A job may be in service at multiple servers at the same time; if a job is in service at both servers j and m, it receives service at combined rate $\mu_j + \mu_m$. As soon as the first copy of a job completes service, the job is considered complete and all remaining copies are cancelled. We assume this cancellation occurs instantaneously regardless of whether the copy is in service or in the queue.

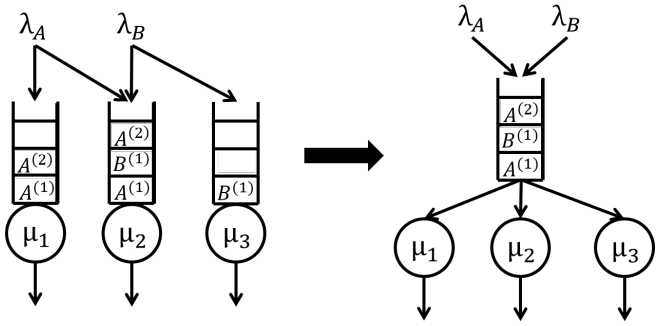


Figure 3.3: Let A and B be two job classes, where $A^{(i)}$ is the ith arrival of class A. We can view the general redundancy system (left) as having a single central queue from which each server works in FCFS order, skipping over those jobs it cannot serve (right). The central queue is an interleaving of the individual servers' queues, where each job appears only once, and appears in the order in which it arrived.

Our analytical approach involves modeling the system using a Markov chain. Looking at Figure 3.1, it is difficult to figure out an appropriate state space. One might think that you could track the number of jobs of each class at each queue, but this state space is missing information about which specific jobs are in service at which servers. Another possibility is to track the order of all jobs in all queues, but this state space because extremely large and unwieldy to work with as the number of servers grows.

The key insight that allows us to model this system is that we can view the system as having a single central queue in which all jobs wait in the order in which they arrived to the system (see Figure 3.3). Each server processes jobs from this central queue in FCFS order, skipping over those jobs it cannot serve. For example, in Figure 3.3, server 3, which can only serve class-B jobs, will skip over job $A^{(1)}$ and move to job $B^{(1)}$ when choosing its next job. We can write the state of the system as $(c_n,\ c_{n-1}, \ldots,\ c_1)$, where there are n jobs in the system, and c_i is the class of the i^{th} job in this central queue; c_1 is the class of the job at the head of the queue, which is also in service at all servers in S_1.

Theorem 3.1. *When for all sets of job classes $C \subseteq \{1, \ldots, \ell\}$,*

$$\sum_{c \in C} \lambda_c < \sum_{m \in \bigcup_{c \in C} S_c} \mu_m,$$

the system is stable, and the limiting probability of being in state $(c_n, c_{n-1}, \ldots, c_1)$ is

$$\pi_{(c_n,\ldots,c_1)} = \mathcal{C} \prod_{i=1}^{n} \frac{\lambda_{c_i}}{\sum_{m \in \bigcup_{j \le i} S_{c_j}} \mu_m},$$

where $\mathcal{C}$ is a normalizing constant.

Proof. Deferred to Section 3.2.1. □

Although $\pi_{(c_n,\, c_{n-1},\ldots,\, c_1)}$ looks like a product-form solution, it is not; we cannot write the limiting probabilities as a product of independent marginal per-server terms, or as a product of independent marginal per-class terms. In fact, the form is quite unusual, as illustrated in Example 3.1.

Example 3.1. *Consider the system shown in Figure 3.3. Here, the current state is* $(A,\ B,\ A)$*, where the head of the queue is at the right, job* $A^{(1)}$ *is currently in service at servers 1 and 2, and job* $B^{(1)}$ *is currently in service at server 3. From Theorem 3.1, the limiting probability of this state is*

$$\pi_{(A,B,A)} = \mathcal{C}\left(\frac{\lambda_A}{\mu_1+\mu_2}\right)\left(\frac{\lambda_B}{\mu_1+\mu_2+\mu_3}\right)\left(\frac{\lambda_A}{\mu_1+\mu_2+\mu_3}\right).$$

Note that the denominator for the first class-A *job is* $\mu_1+\mu_2$ *whereas the denominator for the second class-*A *job is* $\mu_1+\mu_2+\mu_3$*. The denominator for the* i*th job in the queue depends on the classes of all jobs ahead of it, and so the limiting probability cannot be written as a product of per-class terms or as a product of per-server terms.*

In Sections 3.3, 3.4, and 3.5, we use the result of Theorem 3.1 to study the $\mathbb{N}$, $\mathbb{W}$, and $\mathbb{M}$ models, defined below.

$\mathbb{N}$ Model

The $\mathbb{N}$ model is the simplest non-trivial example of a redundancy system where there are both redundant and non-redundant classes. In an $\mathbb{N}$ model there are two servers running at rates μ_1 and μ_2 and two classes of jobs (see Figure 3.2(a)). Class A jobs are non-redundant; they arrive with rate λ_A and join the queue at server 2 only ($S_A = \{2\}$). Class R jobs are redundant; they arrive with rate λ_R and join the queue at both servers ($S_R = \{1,\ 2\}$).

$\mathbb{W}$ Model

Consider a two-server, two-class system in which each class of jobs has its own dedicated server (no redundancy). Now suppose that a third class of jobs enters the system, where this new class is redundant at both servers. The $\mathbb{W}$ model helps us understand how the presence of this redundant class affects the existing non-redundant classes. In a $\mathbb{W}$ model, there are two servers running at rates μ_1 and μ_2 and three classes of jobs (see Figure 3.2(b)). Class A jobs arrive with rate λ_A and join the queue at server 1 only ($S_A = \{1\}$), class B jobs arrive with rate λ_B and join the queue at server 2 only ($S_B = \{2\}$), and class R jobs arrive with rate λ_R and join the queue at both servers ($S_R = \{1,\ 2\}$).

$\mathbb{M}$ Model

Again consider the two-server, two-class system in which each class of jobs has its own dedicated server. Suppose that a new server is added to the system and all jobs issue redundant requests at this server. The $\mathbb{M}$ model helps us understand how best to use the new server. In an $\mathbb{M}$ model, there are three servers with rates μ_1, μ_2, and μ_3 and two job classes (see Figure 3.2(c)). Class R_1 jobs arrive with rate λ_{R_1} and join the queue at servers 1 and 2 ($S_{R_1} = \{1,\ 2\}$), and class R_2 jobs arrive with rate λ_{R_2} and have $S_{R_2} = \{2,\ 3\}$.

3.2.1 Proof of Theorem 3.1

Proof. **[Theorem 3.1]** The stability condition can be seen as a generalization of Hall's Theorem [37], where the proof follows from max flow-min cut. To prove the form of the limiting probabilities, we begin by writing local balance equations for our states. The local balance equations are:

$$A \equiv \begin{array}{l}\text{Rate entering state } (c_n, \ldots, c_1) \\ \text{due to an arrival}\end{array} = \begin{array}{l}\text{Rate leaving state } (c_n, \ldots, c_1) \\ \text{due to a departure}\end{array} \equiv A'$$

$$B_c \equiv \begin{array}{l}\text{Rate entering state } (c_n, \ldots, c_1) \\ \text{due to a departure of class } c\end{array} = \begin{array}{l}\text{Rate leaving state } (c_n, \ldots, c_1) \\ \text{due to an arrival of class } c\end{array} \equiv B'_c.$$

For an empty system, the state is $(-)$. It is not possible to enter state $(-)$ due to an arrival or to leave due to a departure, so we only have one local balance equation of the form $B_c = B'_c$:

$$\pi_{(-)}\lambda_c = \pi_{(c)} \sum_{m \in S_c} \mu_m. \tag{3.1}$$

For any other state $(c_n,\ c_{n-1}, \ldots,\ c_1)$, we have local balance equations of the form:

$$A = \pi_{(c_{n-1},\ldots,c_1)}\lambda_{c_n} = \pi_{(c_n,\ldots,c_1)} \sum_{m \in \bigcup_{j \le n} S_{c_j}} \mu_m = A' \tag{3.2}$$

$$B_c = \sum_{i=0}^{n} \sum_{\substack{m \in S_c, \\ m \notin S_{c_j}, \\ 1 \le j \le i}} \pi_{(c_n,\ldots,c_{i+1},c,c_i,\ldots,c_1)}\mu_m = \pi_{(c_n,\ldots,c_1)}\lambda_c = B'_c. \tag{3.3}$$

We guess the following form for $\pi_{(c_n,\ldots,c_1)}$:

$$\pi_{(c_n,\ldots,c_1)} = \mathcal{C} \prod_{i=1}^{n} \frac{\lambda_{c_i}}{\sum\limits_{m \in \bigcup\limits_{j \le i} S_{c_j}} \mu_m}. \tag{3.4}$$

We will prove inductively that our guess satisfies the balance equations. The base case is state $(-)$. Substituting the guess from (3.4) into the left-hand side of (3.1), we get:

$$\begin{aligned} \pi_{(c)} \sum_{m \in S_c} \mu_m &= \mathcal{C} \frac{\lambda_c}{\sum\limits_{m \in S_c} \mu_m} \sum_{m \in S_c} \mu_m \\ &= \mathcal{C}\lambda_c \\ &= \pi_{(-)}\lambda_c, \end{aligned}$$

which is exactly the right-hand side of (3.1), letting $C = \pi_{(-)}$.

Now, assume that (3.2) and (3.3) hold for some $n-1 \geq 0$. We will show that both hold for n.

1. $\mathbf{A} = \mathbf{A}'$. From (3.2), we have:

$$\begin{aligned} A = \pi_{(c_{n-1},\ldots,c_1)}\lambda_{c_n} &= C\prod_{i=1}^{n-1}\frac{\lambda_{c_i}}{\sum\limits_{m\in\bigcup\limits_{j\leq i}S_{c_j}}\mu_m}\lambda_{c_n} \\ &= \pi_{(c_n,\ldots,c_1)}\frac{\sum_{m\in\bigcup\limits_{j\leq n}S_{c_j}}\mu_m}{\lambda_{c_n}}\lambda_{c_n} \\ &= \pi_{(c_n,\ldots,c_1)}\sum_{m\in\bigcup\limits_{j\leq n}S_{c_j}}\mu_m \\ &= A'. \end{aligned}$$

2. $\mathbf{B_c} = \mathbf{B}'_{\mathbf{c}}$. From (3.3), we have:

$$\begin{aligned} B_c =& \sum_{i=0}^{n}\sum_{\substack{m\in S_c,\\ m\notin S_{c_j},\\ 1\leq j\leq i}}\pi_{(c_n,\ldots,c_{i+1},c,c_i,\ldots,c_1)}\mu_m \\ =& \sum_{i=0}^{n-1}\sum_{m\in S_c\setminus\bigcup\limits_{j\leq i}S_{c_j}}\pi_{(c_n,\ldots,c_{i+1},c,c_i,\ldots,c_1)}\mu_m + \sum_{m\in S_c\setminus\bigcup\limits_{j\leq n}S_{c_j}}\pi_{(c,c_n,\ldots,c_1)}\mu_m \\ =& \sum_{i=0}^{n-1}\sum_{m\in S_c\setminus\bigcup\limits_{j\leq i}S_{c_j}}\frac{\lambda_{c_n}\pi_{(c_{n-1},\ldots,c_{i+1},c,c_i,\ldots,c_1)}}{\sum_{t\in\bigcup\limits_{j\leq n}S_{c_j}\bigcup S_c}\mu_t}\mu_m + \mathcal{C}\frac{\lambda_c\sum_{m\in S_c\setminus\bigcup\limits_{j\leq n}S_{c_j}}\mu_m}{\sum_{m\in\bigcup\limits_{j\leq n}S_{c_j}\bigcup S_c}\mu_m}\prod_{i=1}^{n}\frac{\lambda_{c_i}}{\sum\limits_{m\in\bigcup\limits_{j\leq i}S_{c_j}}\mu_m} \\ =& \frac{\lambda_{c_n}}{\sum\limits_{m\in\bigcup\limits_{j\leq n}S_{c_j}\bigcup S_c}\mu_m}\sum_{i=0}^{n-1}\sum_{m\in S_c\setminus\bigcup\limits_{j\leq i}S_{c_j}}\pi_{(c_{n-1},\ldots,c_{i+1},c,c_i,\ldots,c_1)}\mu_m + \lambda_c\pi_{(c_n,\ldots,c_1)}\frac{\sum_{m\in S_c\setminus\bigcup\limits_{j\leq n}S_{c_j}}\mu_m}{\sum_{m\in\bigcup\limits_{j\leq n}S_{c_j}\bigcup S_c}\mu_m} \\ =& \frac{\lambda_{c_n}}{\sum_{m\in\bigcup_{j\leq n}S_{c_j}\bigcup S_c}\mu_m}\pi_{(c_{n-1},\ldots,c_1)}\lambda_c + \lambda_c\pi_{(c_n,\ldots,c_1)}\frac{\sum_{m\in S_c\setminus\bigcup_{j\leq n}S_{c_j}}\mu_m}{\sum_{m\in\bigcup_{j\leq n}S_{c_j}\bigcup S_c}\mu_m} \end{aligned}$$

$$= \mathcal{C}\lambda_c \frac{\lambda_{c_n}}{\sum_{m \in \bigcup_{j \le n} S_{c_j} \bigcup S_c} \mu_m} \prod_{i=1}^{n-1} \frac{\lambda_{c_i}}{\sum_{m \in \bigcup_{j \le i} S_{c_j}} \mu_m} + \lambda_c \pi_{(c_n,\ldots,c_1)} \frac{\sum_{m \in S_c \setminus \bigcup_{j \le n} S_{c_j}} \mu_m}{\sum_{m \in \bigcup_{j \le n} S_{c_j} \bigcup S_c} \mu_m}$$

$$= \lambda_c \pi_{(c_n,\ldots,c_1)} \frac{\sum_{m \in \bigcup_{j \le n} S_{c_j}} \mu_m}{\sum_{m \in \bigcup_{j \le n} S_{c_j} \bigcup S_c} \mu_m} + \lambda_c \pi_{(c_n,\ldots,c_1)} \frac{\sum_{m \in S_c \setminus \bigcup_{j \le n} S_{c_j}} \mu_m}{\sum_{m \in \bigcup_{j \le n} S_{c_j} \bigcup S_c} \mu_m}$$

$$= \lambda_c \pi_{(c_n,\ldots,c_1)}$$

$$= B'_c.$$

Hence the local balance equations hold for all n, and so the guess for the limiting probabilities from (3.4) is correct. □

3.3 The $\mathbb{N}$ Model

We first turn our attention to the $\mathbb{N}$ model (Figure 3.2(a)). An immediate consequence of Theorem 3.1 is Lemma 3.1, which gives the limiting distribution of the $\mathbb{N}$ model.

Lemma 3.1. *In the $\mathbb{N}$ model, the limiting probability of being in state $(c_n,\ c_{n-1}, \ldots,\ c_1)$ is:*

$$\pi_{(c_n,\ldots,c_1)} = \mathcal{C}_{\mathbb{N}} \left(\frac{\lambda_A}{\mu_2}\right)^{a_0} \left(\frac{\lambda_R}{\mu_1+\mu_2}\right)^{r} \left(\frac{\lambda_A}{\mu_1+\mu_2}\right)^{a_1},$$

where a_0 is the number of class A jobs before the first class R job, a_1 is the number of class A jobs after the first class R job, r is the total number of class R jobs in the queue (note that $a_0 + a_1 + r = n$), and $\mathcal{C}_{\mathbb{N}} = \left(1 - \frac{\lambda_A}{\mu_2}\right)\left(1 - \frac{\lambda_R}{\mu_1+\mu_2-\lambda_A}\right)$ is a normalizing constant.

We use this result to prove Theorem 3.2, which shows that for the redundant class (class R), response time is exponentially distributed, which is pleasantly surprising because the system is not an M/M/1. Specifically, the distribution of response time is the same as that in an M/M/1 with arrival rate λ_R and service rate $\mu' = \mu_1 + \mu_2 - \lambda_A$. Note that μ' can be viewed as giving the class-R jobs the full μ_1, and the portion of μ_2 that is not appropriated for the class-A jobs, $\mu_2 - \lambda_A$. Equivalently, this is the response time in an M/M/1 with arrival rate $\lambda_A + \lambda_R$ and service rate $\mu_1 + \mu_2$. This is counterintuitive because as we will see in Lemma 3.3, the distribution of response time for class R does not depend on whether class A is redundant or non-redundant.

Theorem 3.2. *In the $\mathbb{N}$ model,*

1. *The number of class-R jobs in the system, N_R, is distributed $\mathrm{Geo}_0\,(1-\rho)$, where $\rho = \frac{\lambda_R}{\mu_1+\mu_2-\lambda_A}$ and $\mathrm{Geo}_0\,(p)$ is the geometric distribution supported on $\{0, 1, 2, \ldots\}$ with success probability p.*
2. *The response time of class R jobs, T_R, is distributed $\mathrm{Exp}(\mu_1 + \mu_2 - \lambda_A - \lambda_R)$.*

Proof. This is a special case of the more general result in Theorem 3.11, which is proved in Section 3.6.1. □

In Theorem 3.3, we find that the response time for the non-redundant class, T_A, follows a generalized hyperexponential distribution[1]. We can view the mean response time of class A jobs as that of an M/M/1 with arrival rate λ_A and service rate μ_2, plus a penalty term that captures the extent to which the redundant jobs hurt the A's (Equation 3.7). In Section 3.3.2, we give an alternative interpretation for T_A which provides intuition for this penalty term.

Theorem 3.3. *In the N model,*

1. *The number of class-A jobs in the system, N_A, has p.m.f.*

$$\mathbf{P}\{N_A = n_A\} = \zeta_{\mathbb{N}1}\left(\frac{\lambda_A}{\mu_2}\right)^{n_A} + \zeta_{\mathbb{N}2}\left(\frac{\lambda_A}{\mu_1+\mu_2-\lambda_R}\right)^{n_A} \tag{3.5}$$

   where

$$\zeta_{\mathbb{N}1} = \mathcal{C}_{\mathbb{N}}\left(\frac{\mu_1}{\mu_1-\lambda_R}\right),$$
$$\zeta_{\mathbb{N}2} = \mathcal{C}_{\mathbb{N}}\left(\frac{\lambda_R}{\mu_1+\mu_2-\lambda_R} - \frac{\lambda_R}{\mu_1-\lambda_R}\right),$$

   and $\mathcal{C}_{\mathbb{N}}$ is as in Lemma 3.1.

2. *The distribution of response time of class A jobs is*

$$T_A \sim H_2(\nu_{\mathbb{N}1}, \nu_{\mathbb{N}2}, \omega_{\mathbb{N}}),$$

   where

$$\begin{aligned}\nu_{\mathbb{N}1} &= \mu_2 - \lambda_A\\ \nu_{\mathbb{N}2} &= \mu_1+\mu_2-\lambda_A-\lambda_R\\ \nu_{\mathbb{N}3} &= \mu_1+\mu_2-\lambda_A\\ \omega_{\mathbb{N}} &= \frac{\lambda_R\nu_{\mathbb{N}1}}{(\mu_1-\lambda_R)\nu_{\mathbb{N}3}}.\end{aligned} \tag{3.6}$$

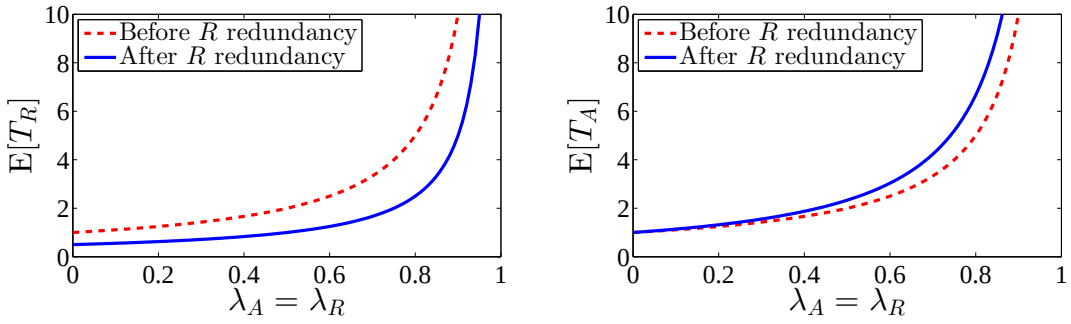
   The mean response time of class-A jobs is

$$\mathbf{E}[T_A] = \underbrace{\frac{1}{\nu_{\mathbb{N}1}}}_{\text{M/M/1}} + \underbrace{\frac{1}{\nu_{\mathbb{N}2}} - \frac{1}{\nu_{\mathbb{N}3}}}_{\text{penalty}}. \tag{3.7}$$

Proof. Deferred to the end of the section. □

Figure 3.4 compares mean response time before class-R jobs become redundant (each class sees its own independent M/M/1), and after class-R jobs become redundant. We hold $\mu_1 = \mu_2 = 1$ and vary the load by increasing $\lambda_R = \lambda_A$. We find redundancy helps class-R jobs by a factor of two (Figure 3.4(a)), but can hurt class-A by up to 50% (Figure 3.4(b)).

[1] A generalized hyperexponential, $H_2(\nu_1, \nu_2, \omega)$ is defined as the weighted mixture of two exponentials with rates ν_1 and ν_2, where the first exponential is given weight $1+\omega$ and the second is given weight $-\omega$. Note that ω can be any real number; it need not be a probability [16].



(a) Class R (b) Class A

Figure 3.4: Comparing mean response time before and after class R becomes redundant when $\mu_1 = \mu_2 = 1$ and $\lambda_A = \lambda_R$ for (a) class R, and (b) class A. The mean response time for the overall system is the weighted average of these two classes.

In Lemma 3.3, we ask what happens if class-A jobs decide they too should be redundant. That is, all jobs can be served at both servers—the system is *fully redundant*. This transforms the system into an M/M/1 with arrival rate $\lambda_A + \lambda_R$ and service rate $\mu_1 + \mu_2$ (Lemma 3.2). Surprisingly, class R is *immune* to pain when class A also becomes redundant: as Lemma 3.3 shows, the distribution of response time for class R is the same before and after class A becomes redundant. Of course, when the A's become redundant, they receive the benefit of having two servers.

Lemma 3.2. *The fully redundant system, in which all jobs issue redundant requests at both servers, is equivalent to an M/M/1 with arrival rate $\lambda_A + \lambda_R$ and service rate $\mu_1 + \mu_2$.*

Proof. In the fully redundant model, all jobs enter the FCFS queue at both servers and depart from both servers immediately upon completion at either server. This is exactly an M/M/1 with arrival rate $\lambda_A + \lambda_B$ and service rate $\mu_1 + \mu_2$. □

Lemma 3.3. *With respect to response time, both classes of jobs do at least as well in the fully redundant model as in the $\mathbb{N}$ model. In particular,*

1. $\mathbf{E}\left[T_A\right]^{\text{Fully Redundant}} \leq \mathbf{E}\left[T_A\right]^{\text{Redundant}}$
2. $T_R^{\text{Fully Redundant}} \stackrel{d}{=} T_R^{\text{Redundant}}$.

Proof. From Theorem 3.2, in the $\mathbb{N}$ model, $T_R \sim \text{Exp}(\mu_1 + \mu_2 - \lambda_A - \lambda_B)$, which is the response time distribution in an M/M/1 with arrival rate $\lambda_A + \lambda_R$ and service rate $\mu_2 + \mu_2$. From Theorem 3.3, in the $\mathbb{N}$ model,

$$\mathbf{E}\left[T_A\right] = \frac{1}{\mu_1 + \mu_2 - \lambda_A - \lambda_R} + \frac{1}{\mu_2 - \lambda_A} - \frac{1}{\mu_1 + \mu_2 - \lambda_A},$$

which is at least the mean response time in an M/M/1 with arrival rate $\lambda_A + \lambda_R$ and service rate $\mu_1 + \mu_2$ since $\frac{1}{\mu_2 - \lambda_A} - \frac{1}{\mu_1 + \mu_2 - \lambda_A}$ is nonnegative. □

Going back to the $\mathbb{N}$ model with only one redundant class, redundancy clearly helps the redundant class considerably. But there are alternative latency-reducing strategies. For example, each redundant class could optimally probabilistically split jobs among all allowable servers (Opt-Split), or join the shortest queue among allowable servers (JSQ).

Definition 3.1. *Under Opt-Split, p fraction of class R jobs go to server 2, and $1-p$ fraction go to server 1, where p is chosen to minimize $\mathbb{E}[T]$. The mean response times under Opt-Split in the $\mathbb{N}$ model for class R jobs, class A jobs, and the system are respectively:*

$$\mathbf{E}\left[T_R\right]^{\text{Opt}-\text{Split}} = \frac{1-p}{\mu_1-(1-p)\lambda_R} + \frac{p}{\mu_2-\lambda_A-p\lambda_R}$$

$$\mathbf{E}\left[T_A\right]^{\text{Opt}-\text{Split}} = \frac{1}{\mu_2-\lambda_A-p\lambda_R}$$

$$\mathbf{E}\left[T\right]^{\text{Opt}-\text{Split}} = \frac{\lambda_A}{\lambda_A+\lambda_R}\mathbf{E}\left[T_A\right]^{\text{Opt}-\text{Split}} + \frac{\lambda_R}{\lambda_A+\lambda_R}\mathbf{E}\left[T_R\right]^{\text{Opt}-\text{Split}}.$$

Definition 3.2. *Under JSQ in the $\mathbb{N}$ model, each arriving class-R job joins the queue at the server with the fewest jobs in queue (including both class-A and class-R jobs, and including the job in service if there is one). Ties are broken in favor of the faster server, or randomly if the servers have the same rate. Class-A jobs join the queue at server 2 only.*

In Figure 3.5, we compare Opt-Split and JSQ to redundancy for the $\mathbb{N}$ model, where the mean response times under Opt-Split are derived analytically (Definition 3.1), but JSQ is simulated. We find that, for the redundant class R, redundancy beats JSQ, which beats Opt-Split. Redundancy often is not much better than JSQ, yet they can differ by a factor of 2, depending on the load of class R and the relative server speeds.

Intuitively, it might seem that the non-redundant class-A jobs should prefer class-R jobs to be dispatched using Opt-Split or JSQ rather than redundancy, since under Opt-Split and JSQ only some of the class-R jobs join the queue at server 2, whereas under redundancy all class-R jobs join the queue at server 2. Surprisingly, however, class-A jobs often prefer redundancy of the other class to Opt-Split or JSQ. This is because the non-redundant class wants the redundant class to spend as little time as possible blocking the A jobs at server 2. Redundancy helps achieve this because all jobs that are in the queue at server 2 have an opportunity to complete service faster at server 1.

Note that under Opt-Split we see an inflection point in mean response time for both class R and class A. For example, in Figures 3.5(a) and (b), there is an inflection point at $\lambda_R = 0.6$, when $\lambda_R = \lambda_A$. This phase change occurs because when $\lambda_R < \lambda_A$, no class R jobs go to server 2 under Opt-Split, but when $\lambda_R > \lambda_A$ the R's compete with the A's. Also observe that $\mathbf{E}\left[T\right]$ is not monotonically increasing; this is because as λ_R increases, the redundant class contributes more to the weighted average.

From the overall system's perspective, redundancy is always preferable to Opt-Split and JSQ because it optimizes overall server utilization.

When $\mu_1 = \mu_2$, Theorem 3.4 tells us that even when non-redundant jobs prefer Opt-Split, redundancy is never more than 50% worse than Opt-Split for the non-redundant jobs.

Theorem 3.4. *If $\mu_1 = \mu_2$, then the following are true:*

1. $\frac{1}{2} \leq \frac{\mathbf{E}[T_R]^{\text{Redundant}}}{\mathbf{E}[T_R]^{\text{Opt}-\text{Split}}} \leq 1$. *If $\lambda_R > \lambda_A$, then* $\frac{\mathbf{E}[T_R]^{\text{Redundant}}}{\mathbf{E}[T_R]^{\text{Opt}-\text{Split}}} = \frac{1}{2}$.
2. $\frac{\mathbf{E}[T_A]^{\text{Redundant}}}{\mathbf{E}[T_A]^{\text{Opt}-\text{Split}}} \leq \frac{3}{2}$.
3. $\frac{1}{2} \leq \frac{\mathbf{E}[T]^{\text{Redundant}}}{\mathbf{E}[T]^{\text{Opt}-\text{Split}}} \leq 1$.

Proof. Deferred to the end of the section. □

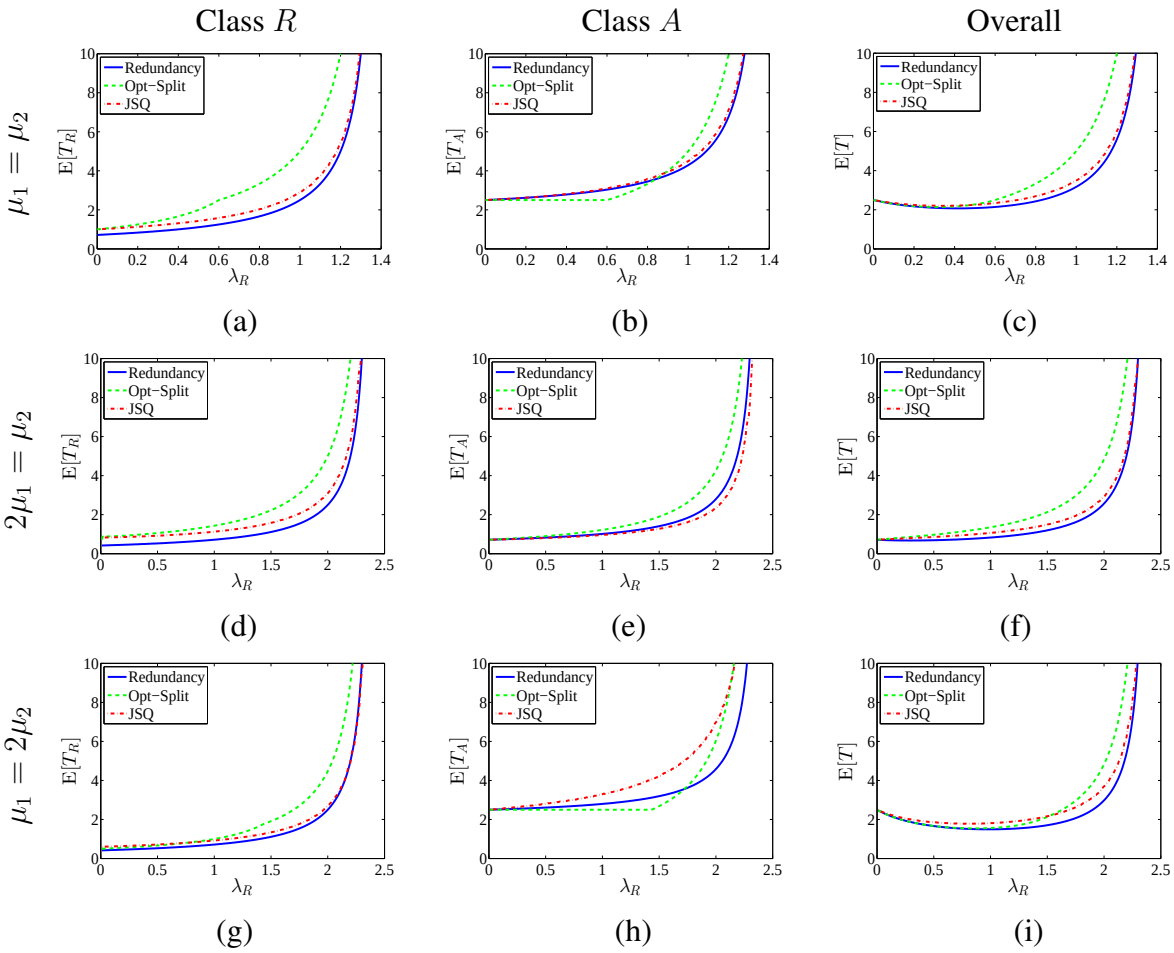


Figure 3.5: Comparing redundancy (solid blue), Opt-Split (dashed green), and JSQ (dashed red) for the $\mathbb{N}$ model as λ_R increases with $\lambda_A = 0.6$. We plot mean response time for the redundant R class (left column), the non-redundant A class (middle column), and the overall system (right column). Rows represent different ratios of server speeds.

3.3.1 Why Redundancy Helps

The analysis presented above demonstrates that redundancy leads to a significant response time improvement for both redundant class R jobs and non-redundant class A jobs. Redundancy helps because it takes advantage of two sources of variability in the system.

First, queueing times can be very different at different servers. Sending redundant requests enables a job to wait in multiple queues and therefore to obtain the shortest possible queueing time. The queueing time experienced under redundancy is even better than that under JSQ because JSQ considers only the number of jobs in the queue, not the actual duration of work in the queue. In fact, redundancy can be seen as obtaining the same queueing time benefit as the Least Work Left (LWL) dispatching policy, which assumes that job sizes are known in advance and an arriving job is dispatched to the queue with the least work. Importantly, redundancy achieves the LWL queueing time benefit without having to know job sizes.

The second source of variability that redundancy leverages is variability in the same job's runtimes on different servers. If a job is in service at multiple servers at the same time, it may experience very different runtimes on these different servers. The job departs the system as soon as its first copy completes service, so it receives the minimum runtime across many servers. The benefits of inter-server variability are unique to redundancy. Policies such as JSQ and LWL, which dispatch only one copy of each job, do not take advantage of the potential response time gains from running multiple copies of the same job. Hence even though JSQ and LWL load balance successfully to overcome queueing time variability, redundancy provides even lower response times by also leveraging runtime variability.

3.3.2 Alternative Interpretation of T_A

From Theorem 3.3, we know that T_A follows a generalized hyperexponential distribution. The Laplace transform of T_A is

$$\widetilde{T}_A(s) = (1+\omega_{\mathbb{N}})\frac{\nu_{\mathbb{N}1}}{\nu_{\mathbb{N}1}+s} - \omega_{\mathbb{N}}\frac{\nu_{\mathbb{N}2}}{\nu_{\mathbb{N}2}+s},$$

where $\nu_{\mathbb{N}1}$, $\nu_{\mathbb{N}2}$, and $\omega_{\mathbb{N}}$ are as defined in (3.6). An alternative way of writing the transform is

$$\widetilde{T}_A(s) = \left(\frac{\mu_1+\mu_2-\lambda_A-\lambda_R}{\mu_1+\mu_2-\lambda_A-\lambda_R+s}\right)\left(\frac{\mu_1+\mu_2-\lambda_A+s}{\mu_1+\mu_2-\lambda_A}\right)\left(\frac{\mu_2-\lambda_A}{\mu_2-\lambda_A+s}\right).$$

Note that the Laplace transform of response time in an M/M/1 with arrival rate λ and service rate μ is

$$\widetilde{T}^{M/M/1}(s) = \frac{\mu-\lambda}{\mu-\lambda+s},$$

and the Laplace transform of queueing time in such an M/M/1 is

$$\widetilde{T}_Q^{M/M/1}(s) = \left(\frac{\mu-\lambda}{\mu-\lambda+s}\right)\left(\frac{\mu+s}{\mu}\right).$$

Hence we can interpret the response time for class-A jobs in the $\mathbb{N}$ model as consisting of two components:

1. $\left(\frac{\mu_1+\mu_2-\lambda_A-\lambda_R}{\mu_1+\mu_2-\lambda_A-\lambda_R+s}\right)\left(\frac{\mu_1+\mu_2-\lambda_A+s}{\mu_1+\mu_2-\lambda_A}\right)$ is the transform of the queueing time in an M/M/1 with arrival rate λ_R and service rate $\mu_1+\mu_2-\lambda_A$.
2. $\left(\frac{\mu_2-\lambda_A}{\mu_2-\lambda_A+s}\right)$ is the transform of the response time in an M/M/1 with arrival rate λ_R and service rate μ_2.

The intuition behind this result is as follows. Consider a tagged class-A job arriving to the $\mathbb{N}$ system. The tagged A sees both class-A jobs and class-R jobs ahead of it in the queue. We first consider the work made up of class-R jobs. Before the class-A job can enter service, all class-R jobs ahead of it must depart from the system. Thus an arriving A job has to wait behind all work in its queue made up of class-R jobs. That total work is the same as the queueing time that a class-R arrival would experience. Hence the first thing the class-A job experiences is the queueing time for a class-R job. From Theorem 3.2, we know that class-R jobs experience the

same response time distribution as in an M/M/1 with arrival rate λ_R and service rate $\mu_1 + \mu_2 - \lambda_A$. Hence the class-$A$ job first experiences the queueing time in such an M/M/1.

After all of the class-R jobs ahead of the tagged class-A job depart, there may still be other A's ahead of the tagged job. Hence the tagged A then experiences the response time in a dedicated class-A M/M/1 with arrival rate λ_A and service rate μ_2.

3.3.3 Burke's Theorem for Class-R Jobs

Our goal in this section is to understand the departure process for the class-R jobs. In Theorem 3.5, we will see that not only do class-R jobs have the same response time distribution as that in an M/M/1, but from class R's perspective, the evolution of the system is stochastically equivalent to an M/M/1. This is surprising because class-A jobs interrupt the service that class-R jobs experience on server 2, and it is difficult to see intuitively why these interruptions should lead to M/M/1 behavior. An immediate consequence of this result is Corollary 3.1, which states that the class-R jobs have a Poisson departure process—giving us a version of Burke's Theorem for the $\mathbb{N}$ model.

Theorem 3.5. *The evolution of class-R jobs is stochastically equivalent to an M/M/1 with arrival rate λ_R and service rate $\mu_1 + \mu_2 - \lambda_A$.*

Proof. We define a class-R job's *effective runtime on server j* to be the time from when it becomes the first class-R job in the queue until it completes service on server j. The effective runtime on server 1 is $S_1 \sim \text{Exp}(\mu_1)$, since the first class R job in the queue must be in service on server 1.

The effective runtime on server 2 is $S_2 + \sum_{i=1}^{N_A} S_2^{(i)}$, where each $S_2^{(i)}$ is an i.i.d. instance of $S_2 \sim \text{Exp}\,(\mu_2)$, and N_A is the number of class-A jobs ahead of the class-R job when the class-R job becomes the first R in the queue. The first term is due to the class-R job's own runtime on server 2, and the sum is due to the runtimes of the N_A class A jobs for which it must wait before entering service. Lemma 3.4 tells us that $N_A \sim \text{Geo}_0\left(1 - \frac{\lambda_A}{\mu_2}\right)$. Hence we have

$$S_2 + \sum_{i=1}^{N_A} S_2^{(i)} = \sum_{i=1}^{N_A+1} S_2^{(i)},$$

which is the sum of a geometric number of exponential random variables, and hence is distributed $\text{Exp}\left(\mu_2(1 - \frac{\lambda_A}{\mu_2})\right) = \text{Exp}\,(\mu_2 - \lambda_A)$.

Thus, when a class-R job becomes the first R in the queue, the time until it leaves the system is the min of its effective runtime on server 1 and its effective runtime on server 2:

$$\min\{\text{Exp}(\mu_1),\ \text{Exp}(\mu_2 - \lambda_A)\} = \text{Exp}(\mu_1 + \mu_2 - \lambda_A).$$

Note that the effective runtime on server 1 is simply the job's runtime on server 1, which is drawn independently from the state at server 2.

Since the time from when the class-R job becomes the first R (i.e., the time when it enters service on some server) until it departs the system is exponentially distributed, from class R's perspective, the system behaves identically to an M/M/1 with arrival rate λ_R and service rate $\mu_1 + \mu_2 - \lambda_A$. □

Corollary 3.1. *In the $\mathbb{N}$ model, the class-R jobs have a Poisson departure process.*

Lemma 3.4. *When a class-R job becomes the first R in the queue, the number of class A jobs in front of it, N_A, is distributed* $\mathrm{Geo}_0\left(1-\frac{\lambda_A}{\mu_2}\right)$.

Proof. We want to determine $\mathbf{P}\{j\ A'\text{s in front of first } R \text{ when it becomes first } R\}$. We will do this by finding the rate at which an R becomes the first R and has j A's ahead of it, and dividing this by the total rate at which an R becomes the first R.

There are three ways in which a class R job can become the first R in the system.
1. The job arrives, and there are no R's already in the queue. Then the rate at which an R becomes the first R and has j A's ahead of it is

$$\pi_{(j)}\lambda_R = \mathcal{C}\left(\frac{\lambda_A}{\mu_2}\right)^j \lambda_R. \tag{3.8}$$

2. An R departs and leaves behind another R, where the departing R was running on both servers (at rate $\mu_1+\mu_2$). Then the rate of having j A's ahead of the first R is

$$\begin{aligned}\sum_X \pi_{(X,j,0)}(\mu_1+\mu_2) &= \mathcal{C}\left(\sum_{n_R=0}^{\infty}\left(\frac{\lambda_R}{\mu_1+\mu_2}\right)^{n_R}\right)\left(\sum_{n_A=0}^{\infty}\left(\frac{\lambda_A}{\mu_1+\mu_2}\right)^{n_A}\right)\\ &\qquad\cdot\left(\frac{\lambda_R}{\mu_1+\mu_2}\right)^2\left(\frac{\lambda_A}{\mu_1+\mu_2}\right)^j(\mu_1+\mu_2)\\ &= \mathcal{C}\left(\frac{\mu_1+\mu_2}{\mu_1+\mu_2-\lambda_R}\right)\left(\frac{\mu_1+\mu_2}{\mu_1+\mu_2-\lambda_A}\right)\left(\frac{\lambda_R}{\mu_1+\mu_2}\right)^2\left(\frac{\lambda_A}{\mu_1+\mu_2}\right)^j(\mu_1+\mu_2)\end{aligned} \tag{3.9}$$

3. An R departs and leaves behind another R, where the departing R was running only on server 1 (at rate μ_1). Note that this case only applies when $j\geq 1$. Then the rate of having j A's ahead of the first R is

$$\begin{aligned}\sum_X\sum_{i=1}^{j}\pi_{(X,j-i,i)}\mu_1 &= \mathcal{C}\left(\sum_{n_R=0}^{\infty}\left(\frac{\lambda_R}{\mu_1+\mu_2}\right)^{n_R}\right)\left(\sum_{n_A=0}^{\infty}\left(\frac{\lambda_A}{\mu_1+\mu_2}\right)^{n_A}\right)\left(\frac{\lambda_R}{\mu_1+\mu_2}\right)^2\\ &\qquad\cdot\mu_1\sum_{i=1}^{j}\left(\frac{\lambda_A}{\mu_2}\right)^i\left(\frac{\lambda_A}{\mu_1+\mu_2}\right)^{j-i}\\ &= \mathcal{C}\left(\frac{\mu_1+\mu_2}{\mu_1+\mu_2-\lambda_R}\right)\left(\frac{\mu_1+\mu_2}{\mu_1+\mu_2-\lambda_A}\right)\left(\frac{\lambda_R}{\mu_1+\mu_2}\right)^2\\ &\qquad\cdot\left(\left(\frac{\lambda_A}{\mu_2}\right)^j-\left(\frac{\lambda_A}{\mu_1+\mu_2}\right)^j\right)(\mu_1+\mu_2)\end{aligned} \tag{3.10}$$

Summing up (3.8), (3.9), and (for $j\geq 1$) (3.10) gives the total rate at which an R becomes the first R in the queue and has j A's in front of it. Summing over all j gives the total rate at which an R becomes the first R in the queue. Hence we have

$$\mathbf{P}\{0\ A'\text{s in front of first } R \text{ when it becomes first } R\} = \frac{(3.8)+(3.9)}{\sum_{j=0}^{\infty}(3.8)+(3.9)+(3.10)}$$

$$= 1 - \frac{\lambda_A}{\mu_2}$$

$$\begin{aligned}\mathbf{P}\left\{n > 0\ A'\text{s in front of first } R \text{ when it becomes first } R\right\} &= \frac{(3.8)+(3.9)+(3.10)}{\sum_{j=0}^{\infty}(3.8)+(3.9)+(3.10)} \\ &= \left(\frac{\lambda_A}{\mu_2}\right)\left(1-\frac{\lambda_A}{\mu_2}\right).\end{aligned}$$

Hence $N_A \sim \text{Geo}_0\left(1-\frac{\lambda_A}{\mu_2}\right)$. □

3.3.4 Proofs for the $\mathbb{N}$ Model

Proof. **[Theorem 3.3]** We first consider the case $n_A = 0$, noting that there can be any number of R jobs in the system.

$$\mathbf{P}\left\{N_A = 0\right\} = \mathcal{C}_{\mathbb{N}} \sum_{i=0}^{\infty} \left(\frac{\lambda_R}{\mu_1+\mu_2}\right)^i = \mathcal{C}_{\mathbb{N}} \frac{\mu_1+\mu_2}{\mu_1+\mu_2-\lambda_R}.$$

Now assume $n_A > 0$. We consider three cases:
1. There are no R jobs in the system. Then

$$\mathbf{P}\left\{N_A = n_A \text{ and no } R \text{ jobs in system}\right\} = \mathcal{C}_{\mathbb{N}} \left(\frac{\lambda_A}{\mu_2}\right)^{n_A}. \tag{3.11}$$

2. There are both R and A jobs in the system, and there is an R at the head of the queue. Let r_0+1 be the number of R jobs before the first A, and r_i be the number of R jobs following the ith A, $i > 0$. Then

$$\begin{aligned}\mathbf{P}\left\{\begin{matrix} N_A = n_a \\ R \text{ at head}\end{matrix}\right\} &= \sum_{r_0=0}^{\infty}\sum_{r_1=0}^{\infty}\cdots\sum_{r_{n_A}=0}^{\infty} \mathcal{C}_{\mathbb{N}} \left(\frac{\lambda_R}{\mu_1+\mu_2}\right)^{1+r_0+\cdots+r_{n_A}} \left(\frac{\lambda_A}{\mu_1+\mu_2}\right)^{n_A} \\ &= \mathcal{C}_{\mathbb{N}} \left(\frac{\lambda_A}{\mu_1+\mu_2}\right)^{n_A} \left(\sum_{r_0=0}^{\infty}\left(\frac{\lambda_R}{\mu_1+\mu_2}\right)^{r_0}\right)\cdots\left(\sum_{r_{n_A}=0}^{\infty}\left(\frac{\lambda_R}{\mu_1+\mu_2}\right)^{r_{n_A}}\right)\left(\frac{\lambda_R}{\mu_1+\mu_2}\right) \\ &= \mathcal{C}_{\mathbb{N}} \left(\frac{\lambda_A}{\mu_1+\mu_2}\right)^{n_A} \left(\frac{1}{1-\frac{\lambda_R}{\mu_1+\mu_2}}\right)^{n_A+1} \left(\frac{\lambda_R}{\mu_1+\mu_2}\right) \\ &= \mathcal{C}_{\mathbb{N}} \left(\frac{\lambda_A}{\mu_1+\mu_2}\right)^{n_A} \left(\frac{\mu_1+\mu_2}{\mu_1+\mu_2-\lambda_R}\right)^{n_A+1} \left(\frac{\lambda_R}{\mu_1+\mu_2}\right) \\ &= \mathcal{C}_{\mathbb{N}} \left(\frac{\lambda_R}{\mu_1+\mu_2-\lambda_R}\right)\left(\frac{\lambda_A}{\mu_1+\mu_2-\lambda_R}\right)^{n_A}\end{aligned} \tag{3.12}$$

3. There are both R and A jobs in the system, and there is an A at the head of the queue. Let $j+1 = a_0$ be the number of A jobs before the first R, let r_1+1 be the number of R's following

these initial A's, and let r_i be the number of R's following the $(j+i)$th A, $i > 1$. Then

$$\mathbf{P}\left\{\begin{matrix} N_A = n_a \\ A \text{ at head}\end{matrix}\right\} = \sum_{j=0}^{n_A-1}\sum_{r_1=0}^{\infty}\cdots\sum_{r_{n_A-j}=0}^{\infty} \mathcal{C}_{\mathbb{N}}\left(\frac{\lambda_A}{\mu_2}\right)^{j+1}\left(\frac{\lambda_R}{\mu_1+\mu_2}\right)^{1+r_1+\cdots+r_{n_A-j}}\left(\frac{\lambda_A}{\mu_1+\mu_2}\right)^{n_A-j-1}$$

$$= \mathcal{C}_{\mathbb{N}}\left(\frac{\lambda_R}{\mu_2}\right)\sum_{j=0}^{n_A-1}\left(\frac{\lambda_A}{\mu_2}\right)^{j}\left(\frac{\lambda_A}{\mu_1+\mu_2}\right)^{n_A-j}\left(\sum_{r_1=0}^{\infty}\left(\frac{\lambda_R}{\mu_1+\mu_2}\right)^{r_1}\right)\cdots\left(\sum_{r_{n_A-j}=0}^{\infty}\left(\frac{\lambda_R}{\mu_1+\mu_2}\right)^{r_{n_A-j}}\right)$$

$$= \mathcal{C}_{\mathbb{N}}\left(\frac{\lambda_R}{\mu_2}\right)\sum_{j=0}^{n_A-1}\left(\frac{\lambda_A}{\mu_2}\right)^{j}\left(\frac{\lambda_A}{\mu_1+\mu_2}\right)^{n_A-j}\left(\frac{1}{1-\frac{\lambda_R}{\mu_1+\mu_2}}\right)^{n_A-j}$$

$$= \mathcal{C}_{\mathbb{N}}\left(\frac{\lambda_R}{\mu_2}\right)\sum_{j=0}^{n_A-1}\left(\frac{\lambda_A}{\mu_2}\right)^{j}\left(\frac{\lambda_A}{\mu_1+\mu_2-\lambda_R}\right)^{n_A-j}$$

$$= \mathcal{C}_{\mathbb{N}}\left(\frac{\lambda_R}{\mu_2}\right)\frac{\mu_2\left[\left(\frac{\lambda_A}{\mu_2}\right)^{n_A}-\left(\frac{\lambda_A}{\mu_1+\mu_2-\lambda_R}\right)^{n_A}\right]}{\mu_1-\lambda_R}$$

$$= \mathcal{C}_{\mathbb{N}}\left(\frac{\lambda_R}{\mu_1-\lambda_R}\right)\left[\left(\frac{\lambda_A}{\mu_2}\right)^{n_A}-\left(\frac{\lambda_A}{\mu_1+\mu_2-\lambda_R}\right)^{n_A}\right] \tag{3.13}$$

Finally, we add (3.11), (3.12), and (3.13) to get the result in (3.5).

We now obtain the Laplace transform of the response time for class-A jobs, $\widetilde{T}_A(s)$, via distributional Little's Law [45]. First, we find the z-transform of the number of class-A jobs in the system, $\widehat{N}_A(z)$:

$$\begin{aligned}\widehat{N}_A(z) &= \sum_{n_A=0}^{\infty}\mathbf{P}\left\{N_A = n_A\right\}z^{n_A}\\ &= \zeta_{\mathbb{N}1}\sum_{n_A=0}^{\infty}\left(\frac{\lambda_A}{\mu_2}\right)^{n_A}z^{n_A} + \zeta_{\mathbb{N}2}\sum_{n_A=0}^{\infty}\left(\frac{\lambda_A}{\mu_1+\mu_2-\lambda_R}\right)^{n_A}z^{n_A}\\ &= \frac{\zeta_{\mathbb{N}1}\mu_2}{\mu_2-\lambda_A z} + \frac{\zeta_{\mathbb{N}2}(\mu_1+\mu_2-\lambda_R)}{\mu_1+\mu_2-\lambda_R-\lambda_A z}.\end{aligned}$$

Observe that class-A jobs depart the system in the same order in which they arrive, so A_{T_A}, the number of class-A arrivals during a class-A response time, is equivalent to N_A, the number of class-A jobs seen by an A departure. Then since $\widehat{A}_{T_A}(z) = \widetilde{T}_A(\lambda_A - \lambda_A z)$, we have, by Distributional Little's Law:

$$\begin{aligned}\widetilde{T}_A(\lambda_A-\lambda_A z) &= \widehat{N}_A(z)\\ &= \frac{\zeta_{\mathbb{N}1}\mu_2}{\mu_2-\lambda_A z} + \frac{\zeta_{\mathbb{N}2}(\mu_1+\mu_2-\lambda_R)}{\mu_1+\mu_2-\lambda_R-\lambda_A z}.\end{aligned}$$

Let $s = \lambda_A - \lambda_A z$, so $z = 1 - \frac{s}{\lambda_A}$. Then we have

$$\begin{aligned}\widetilde{T}_A(s) &= \frac{\zeta_{\mathbb{N}1}\mu_2}{\mu_2 - \lambda_A(1-\frac{s}{\lambda_A})} + \frac{\zeta_{\mathbb{N}2}(\mu_1+\mu_2-\lambda_R)}{\mu_1+\mu_2-\lambda_R-\lambda_A(1-\frac{s}{\lambda_A})} \\ &= (1+\omega_{\mathbb{N}})\frac{\nu_{\mathbb{N}1}}{\nu_{\mathbb{N}1}+s} - \omega_{\mathbb{N}}\frac{\nu_{\mathbb{N}2}}{\nu_{\mathbb{N}2}+s}.\end{aligned}$$

This is the transform of a generalized hyperexponential distribution, $H_2(\nu_{\mathbb{N}1}, \nu_{\mathbb{N}2}, \omega_{\mathbb{N}})$. Finally,

$$\begin{aligned}\mathbf{E}[T_A] &= -\widetilde{T}_A'(s)|_{s=0} \\ &= -\left[(1+\omega_{\mathbb{N}})\frac{-\nu_{\mathbb{N}1}}{(\nu_{\mathbb{N}1}+s)^2} + \omega_{\mathbb{N}}\frac{\nu_{\mathbb{N}2}}{(\nu_{\mathbb{N}2}+s)^2}\right]\Bigg|_{s=0} \\ &= \frac{1}{\nu_{\mathbb{N}1}} + \frac{1}{\nu_{\mathbb{N}2}} - \frac{1}{\nu_{\mathbb{N}3}}.\end{aligned}$$

□

Proof. **[Theorem 3.4]** Definition 3.1 gives us $\mathbf{E}[T_R]^{\text{Opt}-\text{Split}}$, $\mathbf{E}[T_A]^{\text{Opt}-\text{Split}}$, and $\mathbf{E}[T]^{\text{Opt}-\text{Split}}$. We know $\mathbf{E}[T_A]^{\text{Redundant}}$ from Theorem 3.3. Theorem 3.2 tells us that

$$T_R^{\text{Redundant}} \sim \text{Exp}(\mu_1 + \mu_2 - \lambda_A - \lambda_R),$$

so we know that $\mathbf{E}[T_R]^{\text{Redundant}} = \frac{1}{\mu_1+\mu_2-\lambda_A-\lambda_R}$. Finally,

$$\mathbf{E}[T]^{\text{Redundant}} = \frac{\lambda_R}{\lambda_A+\lambda_R}\mathbf{E}[T_R]^{\text{Redundant}} + \frac{\lambda_A}{\lambda_A+\lambda_R}\mathbf{E}[T_A]^{\text{Redundant}}.$$

Thus, $\frac{\mathbf{E}[T_R]^{\text{Redundant}}}{\mathbf{E}[T_R]^{\text{Opt}-\text{Split}}}$, $\frac{\mathbf{E}[T_A]^{\text{Redundant}}}{\mathbf{E}[T_A]^{\text{Opt}-\text{Split}}}$, and $\frac{\mathbf{E}[T]^{\text{Redundant}}}{\mathbf{E}[T]^{\text{Opt}-\text{Split}}}$, and the desired results follow after some minor algebra. □

3.4 The $\mathbb{W}$ Model

We now consider the $\mathbb{W}$ model (see Figure 3.2(b)). The $\mathbb{W}$ model has two non-redundant classes, A and B, each with its own server. A third class, R, enters the system and issues redundant requests at both servers. We study how this redundant class affects the performance of the system.

An immediate consequence of Theorem 3.1 is Lemma 3.5, which gives the limiting distribution of the $\mathbb{W}$ model.

Lemma 3.5. *In the $\mathbb{W}$ model, the limiting probability of being in state $(c_n,\ c_{n-1}, \ldots,\ c_1)$ depends on c_1, as follows:*

$$\begin{aligned}\pi_{(c_n,\ldots,A)} &= \mathcal{C}_{\mathbb{W}}\left(\frac{\lambda_A}{\mu_1}\right)^{a_0}\left(\frac{\lambda_A}{\mu_1+\mu_2}\right)^{a_1}\left(\frac{\lambda_B}{\mu_1+\mu_2}\right)^{b_1}\left(\frac{\lambda_R}{\mu_1+\mu_2}\right)^{r} \\ \pi_{(c_n,\ldots,B)} &= \mathcal{C}_{\mathbb{W}}\left(\frac{\lambda_B}{\mu_2}\right)^{b_0}\left(\frac{\lambda_A}{\mu_1+\mu_2}\right)^{a_1}\left(\frac{\lambda_B}{\mu_1+\mu_2}\right)^{b_1}\left(\frac{\lambda_R}{\mu_1+\mu_2}\right)^{r}\end{aligned}$$

$$\pi_{(c_n,\ldots,R)} = \mathcal{C}_{\mathbb{W}} \left(\frac{\lambda_A}{\mu_1+\mu_2}\right)^{a_1} \left(\frac{\lambda_B}{\mu_1+\mu_2}\right)^{b_1} \left(\frac{\lambda_R}{\mu_1+\mu_2}\right)^{r},$$

where a_0 is the number of class A jobs before the first class B or R job, b_0 is the number of class B jobs before the first class A or R job, a_1 (respectively, b_1) is the number of class A (class B) jobs after the first job of class R or B (A), r is the total number of class R jobs (note $a_0 + a_1 + b_0 + b_1 + r = n$), and

$$\mathcal{C}_{\mathbb{W}} = \left(1 - \frac{\lambda_A}{\mu_1}\right)\left(1 - \frac{\lambda_B}{\mu_2}\right)\left(1 - \frac{\lambda_R}{\mu_1+\mu_2-\lambda_A-\lambda_B}\right)$$

is a normalizing constant.

Like in the $\mathbb{N}$ model, the redundant class (class R) has an exponentially distributed response time (Theorem 3.6). This is again surprising because the system is not an M/M/1. Nonetheless, the response time for the redundant class is stochastically equivalent to the response time in an M/M/1 with arrival rate λ_R and service rate $\mu' = \mu_1 + \mu_2 - \lambda_A - \lambda_B$. We can interpret μ' as the remaining service capacity in the system after λ_A and λ_B have been apportioned to classes A and B respectively. Alternatively, we can view the response time for the redundant class as that in an M/M/1 with arrival rate $\lambda_A + \lambda_B + \lambda_R$ and service rate $\mu_1 + \mu_2$.

Theorem 3.6. *In the $\mathbb{W}$ model,*

1. *The number of class-R jobs in the system, N_R, is distributed $\mathrm{Geo}_0(1-\rho)$, where $\rho = \frac{\lambda_R}{\mu_1+\mu_2-\lambda_A-\lambda_B}$.*
2. *The response time of class-R jobs, T_R, is distributed $\mathrm{Exp}(\mu_1 + \mu_2 - \lambda_A - \lambda_B - \lambda_R)$.*

Proof. The proof follows the same approach as that of Theorem 3.11, and is omitted. □

In Theorem 3.7, we derive the distribution of response time for the non-redundant class A (class B is symmetric).

Theorem 3.7. *In the $\mathbb{W}$ model,*

1. *The number of class-A jobs in the system, N_A, has p.m.f.*

$$\mathbf{P}\{N_A = n_A\} = \zeta_{\mathbb{W}1}\left(\frac{\lambda_A}{\mu_1}\right)^{n_A} + \zeta_{\mathbb{W}2}\left(\frac{\lambda_A}{\mu_1+\mu_2-\lambda_B-\lambda_R}\right)^{n_A},$$

where

$$\zeta_{\mathbb{W}1} = \frac{(\mu_1-\lambda_A)(\mu_2-\lambda_B)(\mu_1+\mu_2-\lambda_A-\lambda_B-\lambda_R)}{\mu_1(\mu_2-\lambda_B-\lambda_R)(\mu_1+\mu_2-\lambda_A-\lambda_B)}$$
$$\zeta_{\mathbb{W}2} = \frac{-\lambda_R(\mu_1-\lambda_A)(\mu_1+\mu_2-\lambda_A-\lambda_B-\lambda_R)}{(\mu_2-\lambda_B-\lambda_R)(\mu_1+\mu_2-\lambda_A-\lambda_B)}.$$

2. *The distribution of response time of class A jobs is*

$$T_A \sim H_2(\nu_{\mathbb{W}1}, \nu_{\mathbb{W}2}, \omega_{\mathbb{W}}),$$

where

$$\nu_{\mathbb{W}1} = \mu_1 - \lambda_A$$

$$\nu_{\mathbb{W}2} = \mu_1 + \mu_2 - \lambda_A - \lambda_B - \lambda_R$$
$$\omega_{\mathbb{W}} = \frac{(\mu_2 - \lambda_B)\nu_{\mathbb{W}2}}{(\mu_2 - \lambda_B - \lambda_R)\nu_{\mathbb{W}3}}.$$

The mean response time of class A jobs is

$$\mathbf{E}[T_A] = \underbrace{\frac{1}{\nu_{\mathbb{W}1}}}_{\text{M/M/1}} + \underbrace{\frac{1}{\nu_{\mathbb{W}2}} - \frac{1}{\nu_{\mathbb{W}3}}}_{\text{penalty}}, \tag{3.14}$$

where

$$\nu_{\mathbb{W}3} = \mu_1 + \mu_2 - \lambda_A - \lambda_B.$$

Proof. The proof follows the same approach as that of Theorem 3.3, and is omitted. □

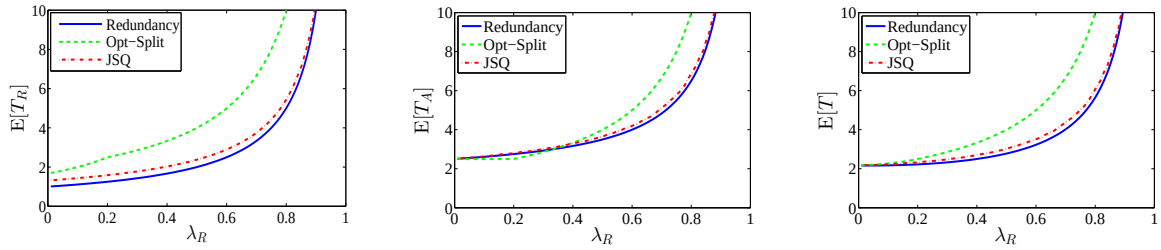
Like in the $\mathbb{N}$ model, we find that T_A follows a generalized hyperexponential distribution. The mean response time of class-A (or class-B) jobs can be interpreted as that in an M/M/1 with arrival rate λ_A (respectively, λ_B) and service rate μ_1 (respectively, μ_2), plus a penalty term that captures the extent to which the redundant class hurts the A's (or B's) (Equation 3.14). Surprisingly, this penalty is the same for class A and class B even if they have different loads: the pain caused by the redundant class is shared equally among the non-redundant classes. Like in the $\mathbb{N}$ model, the distribution of response time for class A can be rewritten as the distribution of time in queue in an M/M/1 with arrival rate λ_R and service rate $\mu_1 + \mu_2 - \lambda_A - \lambda_B - \lambda_R$, plus the response time in an M/M/1 with arrival rate λ_A and service rate μ_1 (class B is symmetric). That is, both classes experience the queueing time in an M/M/1 with arrival rate λ_R and service rate $\mu_1 + \mu_2 - \lambda_A - \lambda_B - \lambda_R$: the mean of this distribution is exactly the penalty term incurred by both class A and class B.

The introduction of a new redundant class clearly hurts the existing non-redundant classes, because the new redundant jobs compete for service with the non-redundant jobs. We now ask what would happen if class R chose which queue(s) to join according to some alternative policy, for example, Opt-Split or JSQ.

Definition 3.3. *Under Opt-Split, p fraction of class R jobs go to server 1, and $1-p$ fraction go to server 2, where p is chosen to minimize $\mathbf{E}[T]$. The mean response times under Opt-Split in the $\mathbb{W}$ model for class-R jobs, class-A jobs, and the overall system are:*

$$\mathbf{E}[T_R]^{\text{Opt-Split}} = \frac{p}{\mu_1 - \lambda_A - p\lambda_R} + \frac{1-p}{\mu_2 - \lambda_B - (1-p)\lambda_R}$$
$$\mathbf{E}[T_A]^{\text{Opt-Split}} = \frac{1}{\mu_1 - \lambda_A - p\lambda_R}$$
$$\mathbf{E}[T]^{\text{Opt-Split}} = \frac{\lambda_A}{\lambda_A + \lambda_B + \lambda_R}\mathbf{E}[T_A]^{\text{Opt-Split}} + \frac{\lambda_B}{\lambda_A + \lambda_B + \lambda_R}\mathbf{E}[T_B]^{\text{Opt-Split}} + \frac{\lambda_R}{\lambda_A + \lambda_B + \lambda_R}\mathbf{E}[T_R]^{\text{Opt-Split}}.$$

The mean response time for class-B jobs is symmetric to that of class-A jobs.



(a) $\mathbf{E}[T_R]$ (b) $\mathbf{E}[T_A]$ (c) $\mathbf{E}[T]$

Figure 3.6: Comparing redundancy (solid blue), Opt-Split (dashed green), and JSQ (dashed red) for the $\mathbb{W}$ model as λ_R increases, for $\mu_1 = \mu_2 = 1$, $\lambda_A = 0.6$, and $\lambda_B = 0.4$. Lines shown include mean response time for (a) class R, (b) class A, and (c) the system. Results for other values of μ_1 and μ_2 are similar.

Definition 3.4. *Under JSQ in the $\mathbb{W}$ model, each arriving class-R job joins the queue at the server with the fewest jobs in the queue (including both class-A, class-B, and class-R jobs, and including the job in service if there is one). Ties are broken in favor of the faster server, or randomly if the servers have the same rate. Class-A jobs join the queue at server 1 only, and class-B jobs join the queue at server 2 only.*

In Figure 3.6, we compare Opt-Split and JSQ to redundancy, where the mean response time under Opt-Split is derived analytically (Definition 3.3), but JSQ is simulated. We find that for the redundant class, redundancy outperforms JSQ, which in turn outperforms Opt-Split (Figure 3.6(a)).

For the non-redundant class-A jobs (Figure 3.6(b)), mean response time is often lower under redundancy than under Opt-Split or JSQ, particularly under higher loads of redundant jobs. This is because even though a larger number of R jobs compete with class A at server 1, some of these R jobs depart the system without ever using server 1 (they complete service at server 2 before entering service at server 1), and some of these R jobs receive service on both servers at once, thus departing the system faster. As in the $\mathbb{N}$ model, redundancy is always better for the overall system (Figure 3.6(c)).

When the servers are homogeneous, in the few cases in which mean response time of class A or B is lower under Opt-Split than under redundancy, we show that redundancy is never more than 50% worse for the A or B jobs.

Theorem 3.8. *If $\mu_1 = \mu_2$, then the following are true:*

1. $\frac{1}{2} \leq \frac{\mathbf{E}[T_R]^{\text{Redundant}}}{\mathbf{E}[T_R]^{\text{Opt-Split}}} \leq 1$. *If $\lambda_R \geq |\lambda_A - \lambda_B|$, then* $\frac{\mathbf{E}[T_R]^{\text{Redundant}}}{\mathbf{E}[T_R]^{\text{Opt-Split}}} = \frac{1}{2}$.
2. $\frac{1}{2} \leq \frac{\mathbf{E}[T_A]^{\text{Redundant}}}{\mathbf{E}[T_A]^{\text{Opt-Split}}} \leq \frac{3}{2}$. *The same inequality holds for class B.*
3. $\frac{1}{2} \leq \frac{\mathbf{E}[T]^{\text{Redundant}}}{\mathbf{E}[T]^{\text{Opt-Split}}} \leq 1$.

Proof. We have $\mathbf{E}[T_R]^{\text{Opt-Split}}$, $\mathbf{E}[T_A]^{\text{Opt-Split}}$, and $\mathbf{E}[T]^{\text{Opt-Split}}$ from Definition 3.3. We also know $\mathbf{E}[T_A]^{\text{Redundant}}$ from Theorem 3.7. Theorem 3.6 tells us that

$$T_R^{\text{Redundant}} \sim \text{Exp}(\mu_1 + \mu_2 - \lambda_A - \lambda_B - \lambda_R),$$

so

$$\mathbf{E}\left[T_R\right]^{\text{Redundant}} = \frac{1}{\mu_1 + \mu_2 - \lambda_A - \lambda_B - \lambda_R}.$$

Finally,

$$\mathbf{E}\left[T\right]^{\text{Redundant}} = \frac{\lambda_R}{\lambda_A + \lambda_B + \lambda_R}\mathbf{E}\left[T_R\right]^{\text{Redundant}} + \frac{\lambda_A}{\lambda_A + \lambda_B + \lambda_R}\mathbf{E}\left[T_A\right]^{\text{Redundant}} + \frac{\lambda_B}{\lambda_A + \lambda_B + \lambda_R}\mathbf{E}\left[T_B\right]^{\text{Redundant}}.$$

We use these expressions to find $\frac{\mathbf{E}[T_R]^{\text{Redundant}}}{\mathbf{E}[T_R]^{\text{Opt-Split}}}$, $\frac{\mathbf{E}[T_A]^{\text{Redundant}}}{\mathbf{E}[T_A]^{\text{Opt-Split}}}$, and $\frac{\mathbf{E}[T]^{\text{Redundant}}}{\mathbf{E}[T]^{\text{Opt-Split}}}$, and the desired results follow after some minor algebra. □

3.5 The $\mathbb{M}$ Model

Finally, we consider the $\mathbb{M}$ model (Figure 3.2(c)). Unlike the $\mathbb{N}$ and $\mathbb{W}$ models, there are two redundant classes in an $\mathbb{M}$ model, classes R_1 and R_2. We study how to best use a shared server to which both classes issue redundant requests. For convenience, throughout the remainder of this section we use the notation

$$\begin{aligned}\mu_{1,2,3} &= \mu_1 + \mu_2 + \mu_3\\ \mu_{1,2} &= \mu_1 + \mu_2\\ \mu_{2,3} &= \mu_2 + \mu_3.\end{aligned}$$

An immediate consequence of Theorem 3.1 is Lemma 3.6, which gives the limiting distribution of the $\mathbb{M}$ model.

Lemma 3.6. *In the $\mathbb{M}$ model, the limiting probability of being in state $(c_n,\ c_{n-1}, \ldots,\ c_1)$ depends on c_1, as follows:*

$$\pi_{(c_n,\ldots,R_1)} = C_{\mathbb{M}} \left(\frac{\lambda_{R_1}}{\mu_{1,2}}\right)^{r_{1,0}} \left(\frac{\lambda_{R_1}}{\mu_{1,2,3}}\right)^{r_{1,1}} \left(\frac{\lambda_{R_2}}{\mu_{1,2,3}}\right)^{r_{2,1}}$$
$$\pi_{(c_n,\ldots,R_2)} = C_{\mathbb{M}} \left(\frac{\lambda_{R_2}}{\mu_{2,3}}\right)^{r_{2,0}} \left(\frac{\lambda_{R_1}}{\mu_{1,2,3}}\right)^{r_{1,1}} \left(\frac{\lambda_{R_2}}{\mu_{1,2,3}}\right)^{r_{2,1}}$$

where $r_{1,0}$ (respectively, $r_{2,0}$) is the number of class R_1 (R_2) jobs before the first class R_2 (R_1) job, $r_{1,1}$ (respectively, $r_{2,1}$) is the number of class R_1 (R_2) jobs after the first R_2 (R_1) job (note that $r_{1,0} + r_{2,0} + r_{1,1} + r_{2,1} = n$), and

$$\mathcal{C}_{\mathbb{M}} = \frac{(\mu_{1,2} - \lambda_{R_1})(\mu_{1,2,3} - \lambda_{R_1} - \lambda_{R_2})(\mu_{2,3} - \lambda_{R_2})}{\mu_{1,2}\mu_{2,3}(\mu_{1,2,3} - \lambda_{R_1} - \lambda_{R_2}) + \lambda_{R_1}\lambda_{R_2}\mu_2}$$

is a normalizing constant.

In Theorem 3.9, we derive the distribution of response time for class R_1 (class R_2 is symmetric). The response time for class R_1 follows a generalized hyperexponential distribution.

Note that in the $\mathbb{N}$ and $\mathbb{W}$ models, the redundant class had an exponentially distributed response time and the response time distribution for *non-redundant* classes was a generalized hyperexponential, whereas in the $\mathbb{M}$ model, the *redundant* class has a generalized hyperexponential response time distribution. We hypothesize that the response time distribution is related to the degree of redundancy: fully redundant classes see exponentially distributed response time, and partially redundant or non-redundant classes see generalized hyperexponentially distributed response times.

Theorem 3.9. *In the $\mathbb{M}$ model,*

1. *The number of class-R_1 jobs, N_{R_1} has p.m.f.*

$$\mathbf{P}\left\{N_{R_1} = n_{R_1}\right\} = \zeta_{\mathbb{M}1}\left(\frac{\lambda_{R_1}}{\mu_{1,2}}\right)^{n_{R_1}} + \zeta_{\mathbb{M}2}\left(\frac{\lambda_{R_1}}{\mu_{1,2,3} - \lambda_{R_2}}\right)^{n_{R_1}},$$

where

$$\zeta_{\mathbb{M}1} = C_{\mathbb{M}}\frac{\mu_3}{\mu_3 - \lambda_{R_2}}$$

$$\zeta_{\mathbb{M}2} = C_{\mathbb{M}}\left(\frac{\lambda_{R_2}}{\mu_{2,3} - \lambda_{R_2}} - \frac{\lambda_{R_1}}{\mu_{1,2,3} - \lambda_{R_2}}\right).$$

2. *The distribution of response time of class-R_1 jobs is*

$$T_{R_1} \sim H_2(\nu_{\mathbb{M}1}, \nu_{\mathbb{M}2}, \omega_{\mathbb{M}}),$$

where

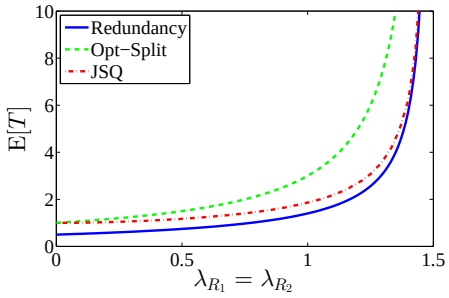
$$\nu_{\mathbb{M}1} = \mu_{1,2} - \lambda_{R_1}$$
$$\nu_{\mathbb{M}2} = \mu_{1,2,3} - \lambda_{R_1} - \lambda_{R_2}$$
$$\omega_{\mathbb{M}} = \zeta_{\mathbb{M}1}\frac{\mu_{1,2}}{\mu_{1,2} - \lambda_{R_1}}$$

Proof. The proof follows the same approach as that of Theorem 3.3, and is omitted. □

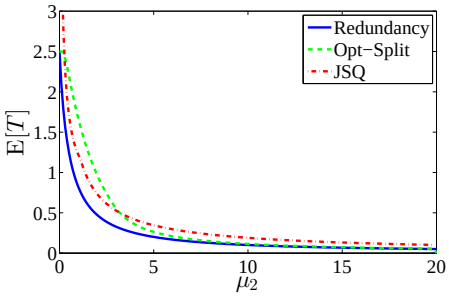
Both classes obviously benefit from issuing redundant requests on a shared server rather than each class having a single dedicated server. However, one might wonder whether mean response time could be further reduced by using some other policy, like Opt-Split or JSQ, instead of redundancy. In Figure 3.7 we investigate the relative performance of these alternative policies. Mean response time under Opt-Split is derived analytically (Definition 3.5); JSQ is simulated.

Definition 3.5. *Under Opt-Split in the $\mathbb{M}$ model, p fraction of class R_1 jobs go to server 2, and $1-p$ fraction go to server 1; and q fraction of class R_2 jobs go to server 2, and $1-q$ fraction go to server 3. We choose p and q to minimize the overall mean response time, given by*

$$\mathbf{E}\left[T\right]^{\text{Opt}-\text{Split}} = \frac{(1-p)\lambda_{R_1}}{\lambda_{R_1}+\lambda_{R_2}} \cdot \frac{1}{\mu_1 - (1-p)\lambda_{R_1}} + \frac{p\lambda_{R_1} + q\lambda_{R_2}}{\lambda_{R_1}+\lambda_{R_2}} \cdot \frac{1}{\mu_2 - p\lambda_{R_1} - q\lambda_{R_2}} + \frac{(1-q)\lambda_{R_2}}{\lambda_{R_1}+\lambda_{R_2}} \cdot \frac{1}{\mu_3 - (1-q)\lambda_{R_2}}.$$



(a) $\mu_1 = \mu_2 = \mu_3 = 1$



(b) $\mu_1 = \mu_3 = 1$, $\lambda_{R_1} = \lambda_{R_2} = 0.6$.

Figure 3.7: Comparing redundancy, Opt-Split, and JSQ for the $\mathbb{M}$ model. Lines shown include mean response time for the overall system under redundancy (solid blue), Opt-Split (dashed green), and JSQ with tiebreaking in favor of the faster server (dashed red). Mean response time as a function of (a) increasing $\lambda_{R_1} = \lambda_{R_2}$, (b) increasing μ_2.

Definition 3.6. *Under JSQ in the $\mathbb{M}$ model, all class-R_1 jobs join the shorter queue between servers 1 and 2, and all class-R_2 jobs join the shorter queue between servers 2 and 3 (including both class-R_1 and class-R_2 jobs at server 2, and including the job in service if there is one). Ties are broken in favor of the faster server, or randomly if the servers have the same rate.*

In all cases, redundancy outperforms both Opt-Split and JSQ. For homogeneous servers (Figure 3.7(a)), mean response time under JSQ approaches that under redundancy at high load, but at low load, redundancy is better by a factor of 2. For heterogeneous servers (Figure 3.7(b)), as the service rate of the shared server increases, mean response time under Opt-Split approaches that under redundancy (Theorem 3.10), but JSQ is worse by a factor of 2. As the system is symmetric, the response times of the individual classes are the same as that of the overall system, and thus are not shown.

We analytically prove performance bounds for the $\mathbb{M}$ model:

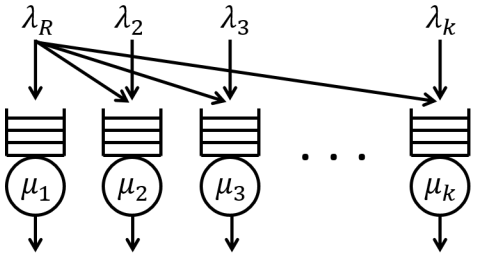
Theorem 3.10. *In the $\mathbb{M}$ model, for any μ_1, μ_2, μ_3, λ_{R_1}, and λ_{R_2} such that the system is stable,*

1. *If $\mu_1 = \mu_2 = \mu_3$ and $\lambda_{R_1} = \lambda_{R_2}$, $\frac{1}{3} \leq \frac{\mathbf{E}[T]^{\text{Redundant}}}{\mathbf{E}[T]^{\text{Opt-Split}}} \leq \frac{1}{2}$.*
2. *For finite μ_1 and μ_3, $\lim_{\mu_2 \to \infty} \frac{\mathbf{E}[T]^{\text{Redundant}}}{\mathbf{E}[T]^{\text{Opt-Split}}} = 1$.*
3. *If $\mu_1 = \mu_3$, $\frac{\mathbf{E}[T]^{\text{Redundant}}}{\mathbf{E}[T]^{\text{Opt-Split}}} \leq 1$.*

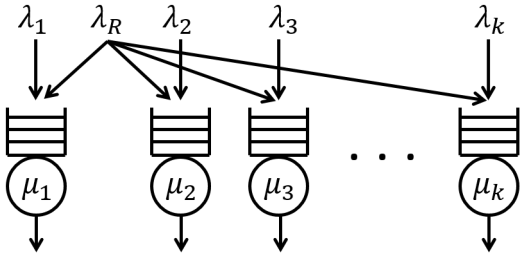
Proof. We know $\mathbf{E}\left[T\right]^{\text{Opt-Split}}$ from Definition 3.5, and

$$\mathbf{E}\left[T\right]^{\text{Redundant}} = \frac{\lambda_{R_1}}{\lambda_{R_1} + \lambda_{R_2}} \mathbf{E}\left[T_{R_1}\right]^{\text{Redundant}} + \frac{\lambda_{R_2}}{\lambda_{R_1} + \lambda_{R_2}} \mathbf{E}\left[T_{R_2}\right]^{\text{Redundant}},$$

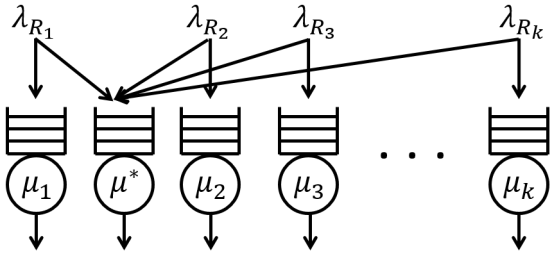
where we know $\mathbf{E}\left[T_{R_1}\right]^{\text{Redundant}}$ and $\mathbf{E}\left[T_{R_2}\right]^{\text{Redundant}}$ from Theorem 3.9. We can then write $\frac{\mathbf{E}[T]^{\text{Redundant}}}{\mathbf{E}\left[T^{\text{Opt-Split}}\right]}$, and the desired results follow after some minor algebra. □



(a) Scaled $\mathbb{N}$ model



(b) Scaled $\mathbb{W}$ model



(c) Scaled $\mathbb{M}$ model

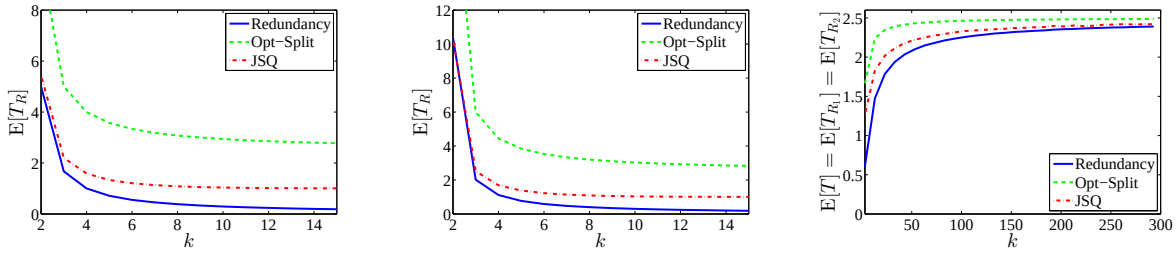
Figure 3.8: Scaled versions of (a) the $\mathbb{N}$ model, (b) the $\mathbb{W}$ model, and (c) the $\mathbb{M}$ model.

3.6 Scale

Thus far, we only have considered systems with two servers (the $\mathbb{N}$ and $\mathbb{W}$ models) and three servers (the $\mathbb{M}$ model). We now turn our attention to the question of scale.

The scaled $\mathbb{N}$, $\mathbb{W}$, and $\mathbb{M}$ models are shown in Figure 3.8. In the scaled $\mathbb{N}$ model, there are k servers and k classes of jobs (see Figure 3.8(a)). Class-R jobs replicate at all servers, while class-i jobs join only the queue at server i for $2 \leq i \leq k$. The scaled $\mathbb{W}$ model is similar; there are k servers and $k+1$ classes of jobs, with class-R replicating at all servers, and class-i jobs going only to server i, $1 \leq i \leq k$ (see Figure 3.8(b)). In the scaled $\mathbb{M}$ model, each class R_i, $1 \leq i < k$, joins the queue at its own dedicated server and at a single server shared by all classes (see Figure 3.8(c)).

The limiting probabilities derived in Theorem 3.1 for the general redundancy system apply to the scaled $\mathbb{N}$, $\mathbb{W}$, and $\mathbb{M}$ models. In Theorem 3.11, we use this result to find that in both the scaled



(a) Scaled $\mathbb{N}$ (b) Scaled $\mathbb{W}$ (c) Scaled $\mathbb{M}$

Figure 3.9: Comparing $\mathbf{E}[T_R]$ under redundancy (solid blue), Opt-Split (dashed green), and JSQ (dashed red) in scaled systems with homogeneous servers, all with rate 1. (a) The scaled $\mathbb{N}$ model with $\lambda_{C_i} = 0.6$ for all non-redundant classes, and $\lambda_R = 1.2$. (b) The scaled $\mathbb{W}$ model with $\lambda_{C_i} = 0.6$ for all non-redundant classes, and $\lambda_R = 0.7$. (c) The scaled $\mathbb{M}$ model with $\lambda_{R_i} = 0.6$ for all classes. In all three scaled systems, as k increases mean response time under all three policies converges to a constant; this convergence is slower in the scaled $\mathbb{M}$ model than in the scaled $\mathbb{N}$ and $\mathbb{W}$ models.

$\mathbb{N}$ and $\mathbb{W}$ models, response time for class R is exponentially distributed, extending the results of Theorem 3.2 and Theorem 3.6 respectively.

Theorem 3.11.

1. *In the scaled $\mathbb{N}$ model, the distribution of the number of class-R jobs in the system is*

$$N_R \sim \text{Geo}_0\left(1 - \frac{\lambda_R}{\sum_{i=1}^{k}\mu_i - \sum_{i=2}^{k}\lambda_i}\right),$$

and the distribution of the response time of class-R jobs is

$$T_R \sim \text{Exp}\left(\sum_{i=1}^{k}\mu_i - \sum_{i=2}^{k}\lambda_i - \lambda_R\right).$$

2. *In the scaled $\mathbb{W}$ model, the distribution of the number of class-R jobs in the system is*

$$N_R \sim \text{Geo}_0\left(1 - \frac{\lambda_R}{\sum_{i=1}^{k}\mu_i - \sum_{i=1}^{k}\lambda_i}\right),$$

and the distribution of the response time of class-R jobs is

$$T_R \sim \text{Exp}\left(\sum_{i=1}^{k}\mu_i - \sum_{i=1}^{k}\lambda_i - \lambda_R\right).$$

Proof. Deferred to the end of the section. □

For the $\mathbb{M}$ model and for the non-redundant classes in the $\mathbb{N}$ and $\mathbb{W}$ models, while the result from Theorem 3.1 applies, it is not straightforward to aggregate states to yield a closed form

expression for mean response time in the scaled models. The results discussed in the remainder of this section for these classes are obtained via simulation.

For the two-server $\mathbb{N}$ and $\mathbb{W}$ models, we saw that the redundant class had lower mean response time under redundancy than under both JSQ and Opt-Split, but often JSQ was very close to redundancy. Here, for scaled models, we investigate whether redundancy enjoys a greater advantage over JSQ and Opt-Split as the number of servers increases.

Indeed, we find that the redundant class sees a much greater benefit under redundancy than under Opt-Split and JSQ as k increases for the scaled $\mathbb{N}$ and $\mathbb{W}$ models (see Figures 3.9(a) and (b)). In fact, as k increases, the benefit grows unboundedly because when a class-R job enters a system with high k, it tends to see many idle servers. Under Opt-Split, this job may not be routed to one of the the idle servers. Under JSQ, the job goes to a *single* idle server i and receives mean response time $\frac{1}{\mu_i}$. Under redundancy, the job gets to use *all* of the idle servers, thereby receiving mean response time $\frac{1}{\sum_i \mu_i}$.

In the two-server $\mathbb{N}$ and $\mathbb{W}$ models, we saw that the benefit that class R received from redundancy came at a cost to the non-redundant class A. In the scaled $\mathbb{N}$ and $\mathbb{W}$ models, this cost approaches 0 because the pain caused by the redundant class is spread among all non-redundant classes, so the effect on any one of these classes is minimal; the response time for each non-redundant class C_i approaches that of an M/M/1 with arrival rate λ_{C_i} and service rate μ_i.

In the three-server version of the $\mathbb{M}$ model (Section 3.5), we saw that redundancy significantly outperformed Opt-Split and JSQ. In Figure 3.9(c), we look at the relative performance of the three policies as k increases. In the scaled $\mathbb{M}$ model, at low k, redundancy indeed gives a lower mean response time than Opt-Split and JSQ. However, as k increases, response time becomes the same under all three policies. As the load on the shared server becomes high, no class benefits from this server; each class experiences an independent M/M/1. Convergence to k independent M/M/1 queues is slow; for example, at $k = 200$, redundancy still provides a 5% lower mean response time than independent M/M/1 queues.

3.6.1 Proof of Theorem 3.11

Proof. **[Theorem 3.11]** To find $\mathbf{P}\{N_R = n_R\}$ in the $\mathbb{N}$ model, we will consider the non-R jobs in the queue as being split into two pieces: the non-R jobs before the first R in the queue, and the non-R jobs after the first R in the queue. We sum over all possible lengths of these two pieces, and all possible classes of these non-R jobs. Let x_0 be the number of non-R jobs before the first R in the queue, and let x_1 be the number of non-R jobs after the first R in the queue. Then we have:

$$\begin{aligned}\mathbf{P}\{N_R = n_R\} &= \sum_{x_0=0}^{\infty}\sum_{x_1=0}^{\infty} \mathcal{C}\cdot\eta_R^{n_R}\cdot X_0\binom{x_1+n_R-1}{x_1}\prod_{\substack{j\geq x_0\\ c_j\neq R}} X_1 \\ &= \mathcal{C}\cdot\eta_R^{n_R}\left(\sum_{x_0=0}^{\infty} X_0\right)\left(\sum_{x_1=0}^{\infty} X_1^{x_1}\binom{x_1+n_R-1}{x_1}\right),\end{aligned}$$

where

$$\eta_R = \frac{\lambda_R}{\sum_{m=1}^{k}\mu_m},$$

$$X_0 = \prod_{j=1}^{x_0} \frac{\sum_{c_i \neq R} \lambda_{c_i}}{\sum_{m \in \bigcup_{t \leq j} S_t} \mu_m},$$

$$X_1 = \frac{\sum_{c_i \neq R} \lambda_{c_i}}{\sum_{m=1}^{k} \mu_m}.$$

The sums in the numerators of X_0 and X_1 take into account all of the possible combinations of classes making up the x_0 and x_1 jobs, respectively.

Now let $\mathcal{C}_1 = \sum_{x_0=0}^{\infty} X_0$ (note that this is a constant with respect to n_R). Using the identity $\sum_{i=0}^{\infty} p^i \binom{i+n-1}{i} = \left(\frac{1}{1-p}\right)^n$ for $|p| < 1$, we have:

$$\begin{aligned} \mathbf{P}\left\{N_R = n_R\right\} &= \mathcal{C}\mathcal{C}_1 \eta_R^{n_R} \left(\frac{1}{1 - \frac{\sum_{c_i \neq R} \lambda_{c_i}}{\sum_m \mu_m}}\right)^{n_R} \\ &= \mathcal{C}\mathcal{C}_1 \left(\frac{\lambda_R}{\sum_m \mu_m - \sum_{c_i \neq R} \lambda_{c_i}}\right)^{n_R}. \end{aligned}$$

Using the normalization equation

$$\sum_{n_R=0}^{\infty} \mathbf{P}\left\{N_R = n_R\right\} = 1,$$

we find

$$\mathcal{C}\mathcal{C}_1 = 1 - \frac{\lambda_R}{\sum_m \mu_m - \sum_{c_i \neq R} \lambda_{c_i}}.$$

Hence $N_R \sim \mathrm{Geo}_0\left(1 - \frac{\lambda_R}{\sum_m \mu_m - \sum_{c_i \neq R} \lambda_{c_i}}\right)$.

Next, we obtain the Laplace transform of the response time for class R jobs, $\widetilde{T}_R(s)$, via distributional Little's Law. First, we consider the z-transform of the number of class-R Poisson arrivals during T, $\widehat{A}_{T_R}(z) = \widetilde{T}_R(\lambda_R - \lambda_R z)$. Class-$R$ jobs depart the system in the same order in which they arrive, so A_{T_R} is equivalent to N_R, the number of jobs seen by an R departure. Hence

$$\widetilde{T}_R(\lambda_R - \lambda_R z) = \widehat{N}_R(z).$$

We know that N_R is distributed $\mathrm{Geo}_0\left(p\right)$, where $p = 1 - \frac{\lambda_R}{\sum_m \mu_m - \sum_{c_i \neq R} \lambda_{c_i}}$. Hence we have

$$\widetilde{T}_R(\lambda_R - \lambda_R z) = \widehat{N}_R(z) = \frac{p}{1 - z(1-p)}.$$

Let $s = \lambda_R - \lambda_R z$, so that $z = 1 - s/\lambda_R$. Then we have

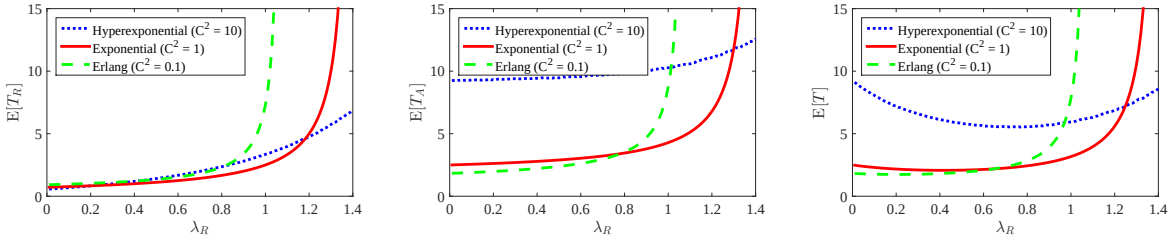
$$\widetilde{T}_R(s) = \frac{p}{1 - (1 - \frac{s}{\lambda_R})(1-p)},$$

which after some simplification gives

$$\widetilde{T}_R(s) = \frac{\sum_m \mu_m - \sum_{c_i \neq R} \lambda_{c_i} - \lambda_R}{\sum_m \mu_m - \sum_{c_i \neq R} \lambda_{c_i} - \lambda_R + s}.$$

Hence $T_R \sim \mathrm{Exp}(\sum_m \mu_m - \sum_{c_i \neq R} \lambda_{c_i} - \lambda_R)$.

The derivation for the $\mathbb{W}$ model is very similar, and is omitted here. $\square$



(a) Class R (b) Class A (c) Overall

Figure 3.10: Mean response time in the $\mathbb{N}$ model for (a) redundant jobs, (b) non-redundant jobs, and (c) the overall system as a function of λ_R when $\lambda_A = 0.6$ and mean runtime is 1. Lines shown include exponential runtimes (solid red line), hyperexponential runtimes with $C^2 = 10$ (dotted blue line), and Erlang runtimes with $C^2 = 0.1$ (dashed green line).

3.7 Relaxing Assumptions

Thus far, we have assumed the IR model and that runtimes are exponentially distributed. The IR model is appropriate in situations in which server-dependent factors such as network congestion, background load, and disk seek times dominate a job's actual computation time. For example, suppose that each server is actually a virtual machine (VM) running in the cloud, serving web queries. A VM running on one physical machine is completely independent from a VM running on a different physical machine. While a VM running on one physical machine might experience slowdown factors up to 27 due to other traffic on its physical machine [76], this does not affect a VM running on a different physical machine, which may experience a slowdown factor of 1. In such cases, if a job consists of only a small amount of work, then the job's runtime depends primarily on the physical machine on which it runs, rather than on the job itself.

It is less clear that runtimes should be exponentially distributed. If the dominating component of runtime is a disk seek time, for example, then the runtimes will have low variability. On the other hand, factors such as network congestion can lead to highly variable runtimes. In this section we consider the impact of redundancy when runtimes are more or less variable than the exponential, but still independent across servers. Unfortunately our exact analysis does not apply to non-exponential runtimes, hence we study this question via simulation.

The effect of runtime variability on mean response time is very complicated. Recall that redundancy helps for two reasons: by allowing jobs to experience the minimum queueing time across servers, and by allowing jobs to experience the minimum runtime across servers. Changing the runtime variability reveals a tradeoff between these two factors. When runtime variability is high, the redundant jobs experience a bigger runtime benefit (which likewise helps the non-redundant jobs), but queueing times increase because a large non-redundant job can block a server for a long period of time. While redundancy helps mitigate the negative effect of high variability on queueing time, in general when runtime variability is high there is a larger probability that a job
much lower, but redundant jobs experience less of a runtime benefit and can even waste server capacity, causing the system to become unstable at lower loads.

Figure 3.10 shows mean response time in the $\mathbb{N}$ model for the redundant jobs, non-redundant

jobs, and overall system as λ_R increases. For the non-redundant jobs (Figure 3.10(b)), at low load the queueing time effect is most pronounced: higher variability runtimes lead to higher mean response times. This is because non-redundant jobs can experience very long queueing times due to waiting behind other non-redundant jobs with long running times; the benefit of clearing the redundant jobs out of the system more quickly is insufficient to overcome this effect. For the redundant jobs (Figure 3.10(a)), at most values of λ_R mean response time is lowest under exponentially distributed runtimes. This is because the exponential runtimes provide the right balance between reducing queueing time and reducing runtime: as runtime variability increases, the runtime benefit of redundancy increases, but queueing time also increases. When load becomes very high the runtime benefit is enough to increase the system's stability region, benefiting both the redundant and non-redundant jobs.

3.8 Discussion and Conclusion

In this chapter we present the first exact analysis of systems with redundancy, deriving the limiting distribution of the queue state. Our state space is very complex and furthermore yields a non-standard product form, and non-obvious, limiting distribution. Nonetheless, we find very clean, simple results for response time distributions for both redundant and non-redundant classes in small systems. For large systems, we derive the response time distribution of a fully redundant class. Many of our results are counterintuitive:

1. The redundant class experiences a response time distribution identical to that in an M/M/1, even though the system is not an M/M/1 ($\mathbb{N}$ and $\mathbb{W}$ models).
2. Once a class is fully redundant, it is *immune* to additional classes becoming redundant: the distribution of its response time does not change ($\mathbb{N}$ and $\mathbb{W}$ model).
3. The non-redundant class often prefers the other class to be redundant as opposed to routing the other class according to Opt-Split or JSQ ($\mathbb{N}$ and $\mathbb{W}$ models).
4. Given two classes of jobs, A and B, each with its own queue, if class R is redundant at both queues, the pain caused to class A is equal to that caused to class B, even though A's and B's respective arrival rates and service rates may be different ($\mathbb{W}$ model).
5. When multiple classes share a single server, redundancy can improve mean response time relative to Opt-Split and JSQ by a factor of 2 ($\mathbb{M}$ model).
6. As the number of servers increases, redundancy gives an even greater benefit to the redundant class while causing less pain to the non-redundant classes (scaled $\mathbb{N}$ and scaled $\mathbb{W}$ models).
7. When runtimes are highly variable, redundancy yields an even bigger improvement, even increasing the system's stability region.

The work presented in this chapter gives rise to a number of follow-up questions about the behavior of redundancy systems. In this chapter, we compare redundancy to other load-balancing strategies such as Opt-Split and JSQ. We observe empirically that while often both the redundant and non-redundant classes of jobs prefer redundancy to JSQ, there are some cases in which non-redundant jobs prefer the redundant jobs to use JSQ instead of redundancy. Nageswaran and Scheller-Wolf formalize this observation, proving that the non-redundant jobs experience

lower mean response time under JSQ in a symmetric system (e.g., the $\mathbb{W}$ model with $\lambda_A = \lambda_B$), but that redundancy can be better in an asymmetric system [56]. Their goal is to understand whether redundancy is fair, or whether there are alternative dispatching policies that perform better with respect to fairness. In Chapter 4, we consider the question of fairness from a *scheduling* perspective.

Another question relates to the time required to cancel the extra copies of a job when the first copy completes service. Here we assume that the cancellation time is negligible, but what if it is not? In 2016, Lee et al. presented an exact analysis in a system with exponentially distributed cancellation times [50]. Like our model, the model in [50] assumes independent, exponential runtimes; unlike our model, the authors require each job to be fully redundant, sending copies to all servers. They find that when cancellation times are high enough, it is optimal to run redundant copies of a job only when the queue is sufficiently small. We revisit the observation that state-aware policies can lead to better performance in Chapter 6, in which we introduce a new dispatching policy for redundancy systems.

One can also ask what would happen if we had networks in which each node in the network was itself a redundancy system. Our result that in the $\mathbb{N}$ model the fully redundant class has a Poisson departure process means that our limiting distribution and response time distribution results will hold for networks of $\mathbb{N}$ models in which the fully redundant class can be routed between nodes, but non-redundant classes can only visit a single node. Bonald and Comte extend this result to general networks of redundancy systems, proving that the form we derive for the limiting probabilities holds for any general redundancy network in which all job classes can be routed through the network, regardless of their redundancy degrees [14]. Unfortunately, they are unable to aggregate the limiting probabilities to find per-class response time distributions in networks of redundancy systems.

Perhaps the largest question left open by the work presented in this chapter is what happens at scale when the number of servers, k, grows large. While the two and three-server systems we study here are useful for providing initial insights about the behavior of redundancy systems, real data centers or server farms typically consist of hundreds or thousands of servers. In the remaining chapters of this thesis we turn to the analysis of larger systems.

Chapter 4

Scheduling in Redundancy Systems

4.1 Introduction

In Chapter 3 we focused primarily on the use of redundancy to reduce overall system response time. But one of the major concerns about redundancy is that its potential benefits are not felt equally across different classes of jobs. In particular, as we saw in Chapter 3, jobs that can become redundant are likely to benefit from being redundant, whereas their redundancy might hurt some non-redundant jobs. This observation, which is particularly important in applications such as organ transplant waitlists where patients' ability to become redundant correlates with their socioeconomic status, leads us to a second goal: in addition to reducing overall mean response time, we want response time gains to be experienced *fairly* across classes.

There are many possible definitions of "fairness." One popular metric used to evaluate fairness is *slowdown*, defined as the ratio of a job's response time to its runtime [11]. Using the slowdown metric, a scheduling policy is viewed as fair if all jobs experience the same slowdown, regardless of their size; hence processor sharing is a fair policy. Policies that are fair with respect to slowdown force jobs with a longer runtime to experience a longer queueing time. Hence slowdown is not an appropriate metric to evaluate fairness in redundancy systems, in which a job's runtime depends largely on server-dependent factors. Intuitively, it does not make sense to punish jobs that happen to run on servers that provide longer runtimes. Nageswaran and Scheller-Wolf [56] propose an alternative definition of fairness for redundancy systems, defining policy $\mathcal{X}$ as being "more fair" than policy $\mathcal{Y}$ if the non-redundant jobs experience lower response time under policy $\mathcal{X}$ than under policy $\mathcal{Y}$. This concept of fairness allows us to more closely match an intuitive notion of what fairness should mean in systems with redundancy: under fair policies, redundancy should not hurt the non-redundant jobs.

In this chapter we use a similar definition of fairness: we say that a scheduling policy $\mathcal{X}$ is fair if, under policy $\mathcal{X}$, no job class experiences a higher mean response time than under a baseline policy $\mathcal{Y}$, where our baseline policy is first-come first-served with no redundancy. To understand our definition of fairness, consider two systems. In the first system, no jobs are redundant; each job is dispatched to a single server, where this server is determined by the job's class, and each server works on the jobs in its queue in FCFS order. In the second system, some jobs become redundant. The redundant jobs are dispatched to their original server as well as to one or more

additional servers, where the additional servers again are determined by the job's new class. Our goal is to improve efficiency, i.e., overall response time, while ensuring that each job class experiences a mean response time in the second (redundant) system that is no worse than what that job class experiences in the first (non-redundant) system. This idea of defining fairness based on comparison to a baseline policy has been used to develop fair variations on the Shortest Remaining Processing Time policy [27].

While there are many ways in which one could attempt to control the effects of job redundancy, in this paper we focus on server-side scheduling policies. We assume a class-based system structure in which each job class replicates itself to a particular subset of the servers, where this subset is determined by, e.g., data locality constraints. Given this replication, we ask in what order each server should work on the jobs in its queue in order to achieve our dual goals of (1) achieving low overall mean response time (high system efficiency), and (2) maintaining per-class fairness. Throughout, we assume that job sizes are not known, hence we focus on non-size-based scheduling policies. We also assume that jobs are preemptible.

We focus on a particular system structure called a *nested* structure. In a nested redundancy system, if two classes of jobs share a server, then one of the class's servers must be a subset of the other class's servers. The nesting structure is common in operating systems such as Multics and UNIX for organizing access to resources via hierarchical protection domains, known as "nested protection rings." The most privileged classes of users have access to all resources. Less privileged classes of users have access to smaller and smaller subsets of the resources. Classes are nested so that, for any two user classes, A and B, that share access to some resource, either A's set of resources is a subset of B's set, or vice versa [63, 67]. In this chapter we think of nested structures as existing not at the operating system level, but at the level of an entire data center, where data may be replicated across servers on the same rack, or across multiple racks.

Our contributions in this chapter are as follows:

Exact analysis of nested redundancy systems under FCFS-R scheduling. We start with a system that has no redundancy in which each server schedules jobs in first-come first-served (FCFS) order, and ask what happens when some jobs become redundant and the servers continue to use first-come first-served scheduling (FCFS-R). We provide an exact analysis of the full distribution of response time for each class. While FCFS-R can yield significant response time improvements relative to FCFS (with no redundancy), our analysis demonstrates that FCFS-R is not always fair.

A new scheduling policy that minimizes response time. We propose a new policy, Least Redundant First (LRF), which gives less redundant jobs preemptive priority over more redundant jobs. We prove that LRF is *optimal* in a very strong sense. It stochastically maximizes the cumulative job departure process, and hence it minimizes overall mean response time. While response time under LRF is not analytically tractable, we derive upper and lower bounds on per-class and overall system mean response time. Surprisingly, mean response time under FCFS-R is nearly as low as that under the optimal LRF policy.

A new fair scheduling policy. The observation that unsophisticated policies like FCFS-R achieve near-optimal performance has important implications for where to focus our efforts in system design, because unfortunately, both FCFS-R and LRF fail to achieve our second goal: maintaining per-class fairness. Although FCFS-R always helps the jobs that become redundant, the non-redundant jobs can be hurt. On the other hand, LRF is so conservative about protecting non-redundant jobs that the redundant jobs can actually suffer and experience higher response times than they would if they were not redundant at all. To balance out the benefits of LRF, under which non-redundant jobs are isolated from competing redundant jobs, and FCFS-R, under which redundant jobs get to retain their place in line, we propose the Primaries First (PF) policy. Under PF, each job has a single primary copy which joins a queue and is served in FCFS order. If the job is redundant all new copies are designated secondaries and are given lowest preemptive priority. Like FCFS-R, PF provides overall mean response times that are nearly indistinguishable from the optimal LRF, thereby achieving our first goal. Furthermore, we prove that all classes of jobs are at least as well off under PF as under non-redundant FCFS, thereby achieving our second goal.

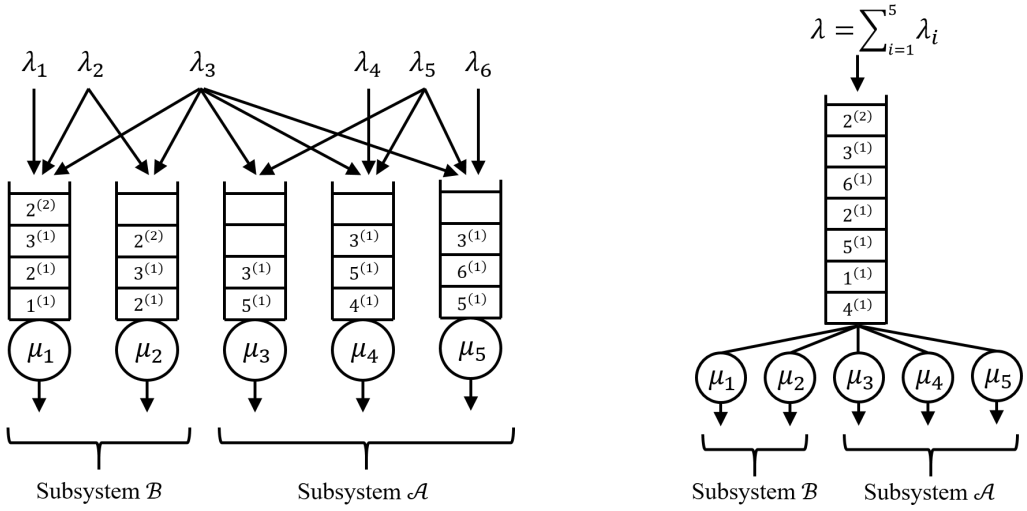
The remainder of this chapter is organized as follows. In Section 4.2 we present the model and notation used throughout the chapter. In Section 4.3 we study the FCFS-R scheduling policy, deriving exact expressions for per-class mean response time. In Section 4.4 we introduce the LRF policy and prove its optimality with respect to response time. In Section 4.5 we introduce the PF policy and prove that it is fair in a very general setting. Finally, in Section 4.6 we conclude. This chapter is based on joint work with Mor Harchol-Balter, Esa Hyytiä, and Rhonda Righter and appears in the following paper:

- Kristen Gardner, Mor Harchol-Balter, Esa Hyytiä, and Rhonda Righter. "Scheduling for Efficiency and Fairness in Systems with Redundancy." Under submission to *Performance Evaluation*. [29]

4.2 Model

We consider a general multi-server system consisting of k servers and ℓ classes of jobs, as shown in Figure 4.1. Jobs arrive to the system with average rate λ. Arrivals form an exogenous renewal process; in some cases we restrict ourselves to a Poisson arrival process. Each job belongs to class j independently with probability p_j; the arrival rate of class j is $\lambda_j = \lambda \cdot p_j$. Each job of class j replicates itself upon arrival by joining the queues at a fixed subset of the servers $S_j = \{s \mid \text{server } s \text{ can serve class } j\}$. A job is allowed to be in service at multiple servers simultaneously. Each job departs the system immediately as soon as its first replica completes service, at which time all remaining replicas are immediately cancelled.

The k servers are heterogeneous, where each server s provides exponential runtimes with rate μ_s. The system as a whole has total average service rate $\mu = \sum_{s=1}^{k} \mu_s$. We assume the IR model, hence if a job is in service at both servers s and r, its remaining time is distributed as $\min\{\text{Exp}(\mu_s), \text{Exp}(\mu_r)\} = \text{Exp}(\mu_s + \mu_r)$. We denote the set of job classes that a particular server s can serve by $J_s = \{j \mid s \in S_j\}$. Note that in our model, if we have multiple identical servers (identical in the sense that they can serve the same classes of jobs, not necessarily in service rates), we can think of these servers as a single fast server. This is because a single job will



(a) A nested redundancy structure (b) Central queue system representation

Figure 4.1: (a) A nested redundancy structure with $k = 5$ servers and $\ell = 6$ job classes. The most redundant class, in this case class 3, replicates itself to subsystem $\mathcal{A}$, which consists of servers 3, 4, and 5 and job classes 4, 5, and 6; and subsystem $\mathcal{B}$, which consists of servers 1 and 2 and job classes 1 and 2. The two subsystems are disjoint: they have no servers or job classes in common except for class 3, which replicates to all servers in both subsystems. In this example, the jobs arrived in the following order: $4^{(1)}, 1^{(1)}, 5^{(1)}, 2^{(1)}, 6^{(1)}, 3^{(1)}, 2^{(2)}$, where $i^{(j)}$ denotes the jth arrival of class-i, illustrated in the central queue shown in (b).

occupy all the identical servers, releasing them all at the same time for the next job to be assigned.

We restrict our attention to a specific type of redundancy structure called a *nested* system.

Definition 4.1. *In a nested system, for all job classes i and j, we have either (1) $S_i \subset S_j$, (2) $S_i \supset S_j$, or (3) $S_i \cap S_j = \emptyset$.*

Figure 4.1 shows an example of a nested system with 5 servers and 6 classes of jobs. As examples of the above cases, we see that $S_4 \subset S_5$ (Case 1), $S_3 \supset S_6$ (Case 2), and $S_1 \cap S_5 = \emptyset$ (Case 3).

In Lemmas 4.1 and 4.2 we note two properties of nested systems that we will use in the remainder of this chapter.

Lemma 4.1. *In a nested redundancy system with k servers, either there exists a class R such that $S_R = \{1, \ldots, k\}$ (i.e., class-R jobs replicate to all k servers), or the system can be separated into two or more independent subsystems, each of which is a nested system.*

Proof. If there is some class of jobs that replicates to all k servers, we are done. So suppose this is not the case. Let class i be the most redundant class, i.e., $|S_j| \leq |S_i|$ for all classes $j \neq i$, and let set $\mathcal{S} = \{s : \ s \notin S_i\}$ denote the set of servers to which class-i jobs do not replicate. For every job class j that shares a server with class i, it must be the case that $S_j \subset S_i$, and so there does not exist a server s such that $s \in S_j$ and $s \in \mathcal{S}$. Hence S_i and $\mathcal{S}$ do not have any servers or job classes in common, so they form two independent subsystems. □

From Lemma 4.1 we assume without loss of generality that the most redundant class, R, replicates to all k servers. In Lemma 4.2 we make the observation that the fully redundant class-R

jobs can be viewed as replicating themselves to two independent subsystems, which we will call subsystems $\mathcal{A}$ and $\mathcal{B}$ (see Figure 4.1).

Lemma 4.2. *Let class-R be the most redundant job class in the nested system (i.e., class-R jobs replicate to all k servers). Then the remaining $\ell - 1$ job classes can be partitioned into two nested subsystems, denoted $\mathcal{A}$ and $\mathcal{B}$, such that for all classes $i \in \mathcal{A}$, $j \in \mathcal{B}$, $S_i \cap S_j = \emptyset$. That is, none of the classes in $\mathcal{A}$ have any servers in common with any of the classes in $\mathcal{B}$.*

Proof. Assume the job classes are sorted with nonincreasing $|S_j|$, so that $|S_j| \geq |S_{j+1}|$ for $j = 2, \ldots, \ell - 1$. We will construct $\mathcal{A}$ and $\mathcal{B}$ as follows. Begin by initializing $\mathcal{A} = \{1\}$ and $\mathcal{B} = \emptyset$. For each class $j = 2, \ldots, \ell - 1$,

1. If there exists a class $i \in \mathcal{A}$ such that $S_j \subset S_i$, add class j to $\mathcal{A}$.
2. Otherwise, by the nesting property and the fact that all classes i already in $\mathcal{A}$ have $|S_i| \geq |S_j|$, we know that for all classes $i \in \mathcal{A}$ we have $S_i \cap S_j = \emptyset$. Hence we add class j to $\mathcal{B}$.

After each addition of a class to either set $\mathcal{A}$ or $\mathcal{B}$, we continue to have the property that for all classes $i \in \mathcal{A}$, $j \in \mathcal{B}$, $S_i \cap S_j = \emptyset$. □

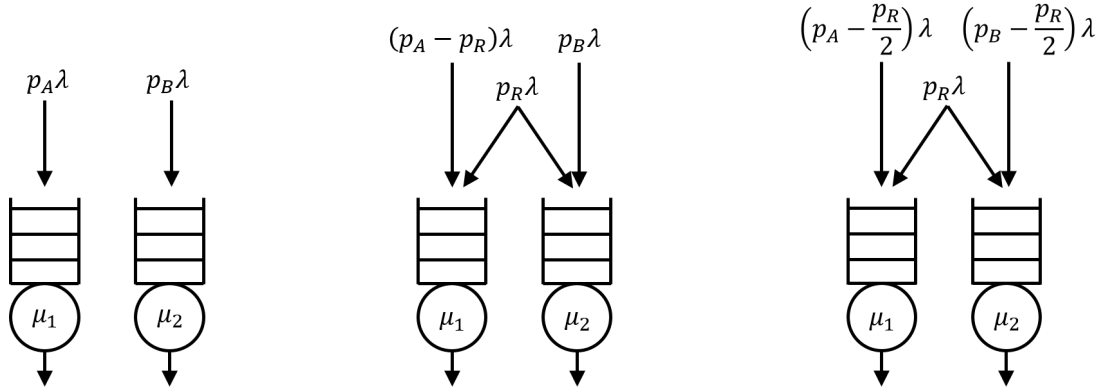
Example 4.1. *In Figure 4.1, class 3 jobs are fully redundant. We sort the remaining job classes in nonincreasing order of redundancy degree: 5, 2, 1, 4, 6. Starting at the beginning of this list, we set $\mathcal{A} = \{5\}$ and $\mathcal{B} = \emptyset$. Since $S_2 \cap S_5 = \emptyset$, we add class 2 to $\mathcal{B}$; we then add class 1 to $\mathcal{B}$ for the same reason. Since $S_4 \subset S_5$, we add class 4 to $\mathcal{A}$. Finally, we add class 6 to $\mathcal{A}$. Observe that $\mathcal{A} = \{5, 4, 6\}$ and $\mathcal{B} = \{2, 1\}$ are disjoint.*

Throughout much of the remainder of this chapter, we use a specific nested system called the $\mathbb{W}$ model as a case study to illuminate the effects of different scheduling policies on different classes of jobs. We begin with a system in which there are two classes of jobs, each with its own server (see Figure 4.2(a)). Class-A jobs are non-redundant and join the queue at server 1 only. Class-B jobs are non-redundant and join the queue at server 2 only. We let p_A and p_B denote the fraction of jobs that are class-A and class-B respectively, where $p_A + p_B = 1$. To understand the impact that different redundancy-based scheduling policies have on response time, we compare this system to the $\mathbb{W}$ model, in which some jobs become redundant class-R jobs, which join the queues at both servers 1 and 2. We consider two cases: the case in which p_R fraction of the jobs become redundant, where originally all of these jobs were class-A (Figure 4.2(b)), and the case in which p_R fraction of the jobs become redundant, where originally half of these jobs were class-A and half were class-B (Figure 4.2(c)).

For notational simplicity, we let λ_A, λ_B and λ_R denote the arrival rates of class-A, B, and R jobs respectively, where λ_i is defined differently in terms of λ, p_A, p_B, and p_R for each of the systems shown in Figure 4.2. For example, in Figure 4.2(a) $\lambda_A = p_A\lambda$, whereas in Figure 4.2(b) $\lambda_A = (p_A - p_R)\lambda$ and in Figure 4.2(c) $\lambda_A = (p_A - \frac{p_R}{2})\lambda$.

4.3 First-Come First-Served with Redundancy (FCFS-R)

We first consider redundancy systems using the standard first-come first-served scheduling policy, which we call FCFS-R to distinguish this policy from FCFS scheduling in a system in which no jobs are redundant. Under FCFS-R, each server works on the jobs in its queue in FCFS order; a



(a) No redundancy

(b) Some class-A jobs become class-R jobs

(c) Some class-A jobs and some class-B jobs become class-R jobs

Figure 4.2: The $\mathbb{W}$ model. Class-A jobs join the queue at server 1 only, class-B jobs join the queue at server 2 only, and class-R jobs join both queues. Throughout this chapter, we compare (a) The system in which there are no redundant jobs ($p_R = 0$) to (b) The system in which some class-A jobs become class-R jobs, and (c) The system in which some class-A jobs and some class-B jobs become class-R jobs.

$(2,3,6,5,4,2,1)$	$(2,3,6,5,2,4,1)$	$(2,3,6,5,2,1,4)$	$(2,3,6,2,5,4,1)$	$(2,3,6,2,5,1,4)$
$(2,3,6,2,1,5,4)$	$(2,3,2,6,5,4,1)$	$(2,3,2,6,5,1,4)$	$(2,3,2,6,1,5,4)$	$(2,3,2,1,6,5,4)$

Table 4.1: If we did not know the order in which the jobs arrived, we would not be able to distinguish between these ten states just from looking at the snapshot of the system shown in Figure 4.1(a).

job may be in service at multiple servers at the same time. Throughout this section we assume that arrivals form a Poisson process.

4.3.1 Response Time Analysis

Our goal in this section is to derive the distribution of response time—defined as the time from when a job arrives to the system until its first copy completes service—in nested redundancy systems under FCFS-R scheduling. We do so using a Markov chain approach. We use the same state space as in Chapter 3, which tracks all of the jobs in the system in the order in which they arrived; this can be viewed as a single central queue, as shown in Figure 4.1(b). We denote the system state by $(c_n, c_{n-1}, \ldots, c_1)$, where there are n jobs in the system and c_i is the class of the ith job in the (central) queue; the head of the queue is at the right. In the example shown in Figure 4.1 the system state is $(2,3,6,2,5,1,4)$. Note that without knowing the arrival order of the jobs shown in Figure 4.1(a), we do not have enough information to distinguish between the states shown in Table 4.1. While we know that the oldest class-2 job arrived after the class-1 job, and the class-6 job arrived after the class-5 job, which in turn arrived after the class-4 job, we do

not know how the class-1 and class-2 jobs are interleaved with the class-4, class-5, and class-6 jobs because the two sets of jobs have no servers in common. However, all of the states listed in Table 4.1 are equivalent in terms of the future evolution of the system.

Theorem 3.1 in Chapter 3 tells us that the limiting probability of being in state $(c_n, c_{n-1}, \ldots, c_1)$ is

$$\pi_{(c_n,\ldots,c_1)} = \mathcal{C} \prod_{i=1}^{n} \frac{\lambda_{c_i}}{\mu_{\mathcal{I}_{1,\ldots,i}}}, \tag{4.1}$$

where $\mathcal{I}_{1,\ldots,i} = \{s | s \in \bigcup_{j \leq i} S_{c_j}\}$ is the set of all servers that can serve at least one job in positions 1 to i in the queue, and $\mathcal{C}$ is a normalizing constant. For example, the limiting probability of the state shown in Figure 4.1(b) is

$$\pi_{(2,3,6,2,5,1,4)} = \mathcal{C} \cdot \left(\frac{\lambda_2}{\sum_{i=1}^{5} \mu_i}\right) \cdot \left(\frac{\lambda_3}{\sum_{i=1}^{5} \mu_i}\right) \cdot \left(\frac{\lambda_6}{\sum_{i=1}^{5} \mu_i}\right) \cdot \left(\frac{\lambda_2}{\sum_{i=1}^{5} \mu_i}\right) \cdot \left(\frac{\lambda_5}{\mu_1 + \mu_3 + \mu_4 + \mu_5}\right) \cdot \left(\frac{\lambda_1}{\mu_1 + \mu_4}\right) \cdot \left(\frac{\lambda_4}{\mu_4}\right).$$

As noted in Chapter 3, this form is quite unusual: the limiting probabilities cannot be written in general as a product of per-class terms or of per-server terms.

Unfortunately, knowing an explicit expression for the limiting distribution of our state space does not immediately yield results for the distribution of response time for each class. The problem is that the state space is very detailed, and it is not obvious how to aggregate the states to find the per-class distribution of the number in system, from which we can find the distribution of response time via Distributional Little's Law (note that within a class, jobs leave the system in the order in which they arrived, so Distributional Little's Law applies per-class, but not for the overall system).

In this section, we derive the distribution of response time for each job class in any nested redundancy structure. We perform a state aggregation that allows us to express the limiting probabilities in terms of the limiting probabilities in the left and right subsystems. Our new aggregated states are still sufficiently detailed for us to determine the steady-state response time distribution for each job class.

Let class-R denote the most redundant job class in the system (i.e., class-R jobs replicate to all k servers). We define the aggregated state $\left(R, n_R, \begin{smallmatrix}(\mathcal{A})\\(\mathcal{B})\end{smallmatrix}\right)$ to be the state in which subsystems $\mathcal{A}$ and $\mathcal{B}$ have states $(\mathcal{A})$ and $(\mathcal{B})$ respectively (including only those jobs that are ahead of the first class-R job in the queue), the R denotes the first class-R job in the system, and n_R is the number of jobs in the queue behind the first class-R job (note that these jobs may be of any class). If there are no class-R jobs in the system, we set $n_R =$ '-', so the state is $\left(R, -, \begin{smallmatrix}(\mathcal{A})\\(\mathcal{B})\end{smallmatrix}\right)$. If the system is empty the state is defined to be $\emptyset$. Our new state aggregates over two things. First, we aggregate over all possible interleavings of the jobs in subsystems $\mathcal{A}$ and $\mathcal{B}$. Second, we track only the number of jobs in the queue behind the first class-R job rather than the specific classes of each of these jobs. We can recursively apply our aggregation so that in states $(\mathcal{A})$ and $(\mathcal{B})$ we only track the number of jobs behind the first job that is fully redundant in subsystems $\mathcal{A}$ and $\mathcal{B}$ respectively.

Example 4.2. *We rewrite the state in Figure 4.1 as* $(3, n_3 = 1, {}^{(6,5,4)}_{(2,1)})$*, where here* $(\mathcal{A}) = (6, 5, 4)$ *and* $(\mathcal{B}) = (2, 1)$*. Note that substates* $(\mathcal{A})$ *and* $(\mathcal{B})$ *only include the jobs that appear in the queue ahead of the first class-*3 *job, which is fully redundant. We then recursively aggregate substates* $(\mathcal{A})$ *and* $(\mathcal{B})$ *to get the state*

$$\left(3, n_3 = 1, {}^{\left(5, n_5=1, {}^{(4, n_4=0)}_{(6, n_6=-)}\right)}_{\left(2, n_2=0, {}^{(1, n_1=0)}_{(-)}\right)}\right).$$

In this new aggregated state, we know that there is a class-3 job in the system with one job in the queue behind it, but we do not know the class of this job (it could be any class). Similarly, we know that there is a class-5 job in the system with one job in the queue behind it but in front of the class-3 job. This job in the queue could be class-4, 5, or 6.

Effectively, aggregating states in this manner defers the point at which we determine a job's class. In our system description in Section 4.2, we assign each job's class upon arrival. However, because job classes are assigned independently according to fixed probabilities, there is no need to realize a job's class until a server becomes available and we need to know whether the server is capable of working on the job. We can interpret our aggregated states as being the state of an alternative (but equivalent) system in which we only discover information about a job's class at the moment when we must determine whether the job will enter service on an idle server. This state description also aggregates states that are indistinguishable in terms of arrival order. For example, all of the states listed in Table 4.1 are included in the aggregated state in the above example.

Let $\mathcal{I}_j$ denote the subsystem in which class-j is fully redundant. That is, the classes in subsystem $\mathcal{I}_j$ are all classes i such that $S_i \subseteq S_j$, and the servers in subsystem $\mathcal{I}_j$ are the servers in S_j. In our example, $\mathcal{A} = \mathcal{I}_5$ and $\mathcal{B} = \mathcal{I}_2$. We use $\mu_{\mathcal{I}_j}$ to denote the total service rate of all servers in $\mathcal{I}_j$, and $\lambda_{\mathcal{I}_j}$ to denote the total arrival rate of all job classes in $\mathcal{I}_j$, and define $\mu_{\mathcal{A}} = \sum_{s \in S_{\mathcal{A}}} \mu_s$, $\mu_{\mathcal{B}} = \sum_{s \in S_{\mathcal{B}}} \mu_s$, $\lambda_{\mathcal{A}}$, and $\lambda_{\mathcal{B}}$ similarly.

Lemma 4.1.[1] *The limiting probability of being in state* $\left(R, n_R, {}^{(\mathcal{A})}_{(\mathcal{B})}\right)$*, for* $n_R \neq -$*, is*

$$\pi_{\left(R, n_R, {}^{(\mathcal{A})}_{(\mathcal{B})}\right)} = \mathcal{C} \cdot P\left(R, n_R, {}^{(\mathcal{A})}_{(\mathcal{B})}\right), \tag{4.2}$$

where $\mathcal{C}$ *is a normalizing constant and* P *satisfies*

$$P\left(R, n_R, {}^{(\mathcal{A})}_{(\mathcal{B})}\right) = \left(\frac{\lambda_{\mathcal{A}} + \lambda_{\mathcal{B}} + \lambda_R}{\mu_{\mathcal{A}} + \mu_{\mathcal{B}}}\right)^{n_R} \left(\frac{\lambda_R}{\mu_{\mathcal{A}} + \mu_{\mathcal{B}}}\right) \cdot P(\mathcal{A}) \cdot P(\mathcal{B}) \tag{4.3}$$

and $P(\emptyset) = 1$*. The limiting probability of being in state* $\left(R, -, {}^{(\mathcal{A})}_{(\mathcal{B})}\right)$ *is*

$$\pi_{\left(R, -, {}^{(\mathcal{A})}_{(\mathcal{B})}\right)} = \mathcal{C} \cdot P\left(R, -, {}^{(\mathcal{A})}_{(\mathcal{B})}\right),$$

[1]Thanks to John Wright for proving what I call the "John Wright Lemma," a less general version of this lemma that involved only classes A and B, rather than entire subsystems $\mathcal{A}$ and $\mathcal{B}$.

where $P\left(R, -, \binom{(\mathcal{A})}{(\mathcal{B})}\right)$ *satisfies*

$$P\left(R, -, \binom{(\mathcal{A})}{(\mathcal{B})}\right) = P(\mathcal{A}) \cdot P(\mathcal{B}).$$

Proof. We defer the detailed proof to the end of the section and here give a proof sketch for an example of the $\mathbb{W}$ model. Consider state (A, R, B, A). From (4.1) we know that the limiting probability of being in this state is

$$\pi_{A,B,R,A} = \mathcal{C}\left(\frac{\lambda_A}{\mu} \cdot \frac{\lambda_R}{\mu} \cdot \frac{\lambda_B}{\mu} \cdot \frac{\lambda_A}{\mu_1}\right).$$

We first aggregate over the different arrival orders that are indistinguishable from looking at a snapshot of the system. In particular, we cannot distinguish between states (A, R, B, A) and (A, R, A, B). We have:

$$\begin{aligned}
\pi_{\left(A,R,\frac{A}{B}\right)} &= \pi_{(A,R,B,A)} + \pi_{(A,R,A,B)} \\
&= \mathcal{C}\left(\frac{\lambda_A}{\mu} \cdot \frac{\lambda_R}{\mu} \cdot \frac{\lambda_B}{\mu} \cdot \frac{\lambda_A}{\mu_1} + \frac{\lambda_A}{\mu} \cdot \frac{\lambda_R}{\mu} \cdot \frac{\lambda_A}{\mu} \cdot \frac{\lambda_B}{\mu_2}\right) \\
&= \mathcal{C}\left(\frac{\lambda_A}{\mu} \cdot \frac{\lambda_R}{\mu} \cdot \frac{\lambda_A}{\mu_1} \cdot \frac{\lambda_B}{\mu_2}\right).
\end{aligned}$$

We then aggregate over all job classes that could appear in the queue after the first class-R job. That is, we aggregate states $(A, R, \frac{A}{B})$, $(B, R, \frac{A}{B})$, and $(R, R, \frac{A}{B})$:

$$\begin{aligned}
\pi_{\left(R,1,\frac{A}{B}\right)} &= \pi_{\left(A,R,\frac{A}{B}\right)} + \pi_{\left(B,R,\frac{A}{B}\right)} + \pi_{\left(R,R,\frac{A}{B}\right)} \\
&= \mathcal{C}\left(\frac{\lambda_A}{\mu} \cdot \frac{\lambda_R}{\mu} \cdot \frac{\lambda_A}{\mu_1} \cdot \frac{\lambda_B}{\mu_2} + \frac{\lambda_B}{\mu} \cdot \frac{\lambda_R}{\mu} \cdot \frac{\lambda_A}{\mu_1} \cdot \frac{\lambda_B}{\mu_2} + \frac{\lambda_R}{\mu} \cdot \frac{\lambda_R}{\mu} \cdot \frac{\lambda_A}{\mu_1} \cdot \frac{\lambda_B}{\mu_2}\right) \\
&= \mathcal{C}\left(\frac{\lambda}{\mu} \cdot \frac{\lambda_R}{\mu} \cdot \frac{\lambda_A}{\mu_1} \cdot \frac{\lambda_B}{\mu_2}\right),
\end{aligned}$$

which is the form given in 4.3. The full proof, given in Section 4.3.3, is a generalization of this example. □

Note that Lemma 4.1 can be applied recursively to the subsystems $\mathcal{A}$ and $\mathcal{B}$.

Let $\rho_i = \frac{\lambda_i}{\mu_{\mathcal{I}_i} - \lambda_{\mathcal{I}_i} + \lambda_i}$. We can interpret $\mu_{\mathcal{I}_i} - \lambda_{\mathcal{I}_i} + \lambda_i$ as being the service capacity in subsystem $\mathcal{I}_i$ that is available for class-i after capacity is allocated to the remaining classes in subsystem $\mathcal{I}_i$. Then ρ_i is the fraction of time that this available capacity is used to process class-i jobs. We will show later that ρ_i is also the probability that there is at least one class-i job in the system.

Lemma 4.3. *In a nested redundancy system with ℓ job classes, the normalizing constant $\mathcal{C}$ is*

$$\mathcal{C} = \prod_{i=1}^{\ell} (1 - \rho_i) \tag{4.4}$$

Proof. Deferred to Section 4.3.4. □

Example 4.3. *In the system shown in Figure 4.1, the normalizing constant is*

$$\begin{aligned}\mathcal{C} &= \prod_{i=1}^{6}(1-\rho_i) \\ &= \left(1-\frac{\lambda_1}{\mu_1}\right)\cdot\left(1-\frac{\lambda_2}{\mu_1+\mu_2-\lambda_1}\right)\cdot\left(1-\frac{\lambda_3}{\sum_{s=1}^{5}\mu_s-\sum_{i=1}^{5}\lambda_i+\lambda_3}\right) \\ &\quad\cdot\left(1-\frac{\lambda_4}{\mu_4}\right)\cdot\left(1-\frac{\lambda_5}{\mu_3+\mu_4+\mu_5-\lambda_4-\lambda_6}\right)\cdot\left(1-\frac{\lambda_6}{\mu_6}\right)\end{aligned}$$

Theorem 4.1. *In a nested redundancy system with ℓ job classes, the response time for class i, T_i, can be expressed as*

$$T_i = T(\lambda_{\mathcal{I}_i}, \mu_{\mathcal{I}_i}) + \sum_{j:S_i\subset S_j} T_Q(\lambda_{\mathcal{I}_j}, \mu_{\mathcal{I}_j}),$$

where $T(\lambda,\mu)$ and $T_Q(\lambda,\mu)$ represent the response time and queueing time respectively in an M/M/1 with arrival rate λ and service rate μ. That is, the Laplace transform of response time for class i is

$$\widetilde{T}^{(i)}(s) = \left(\frac{\mu_{\mathcal{I}_i}-\lambda_{\mathcal{I}_i}}{\mu_{\mathcal{I}_i}-\lambda_{\mathcal{I}_i}+s}\right)\cdot\prod_{j:S_i\subset S_j}\left(\frac{\mu_{\mathcal{I}_j}-\lambda_{\mathcal{I}_j}}{\mu_{\mathcal{I}_j}-\lambda_{\mathcal{I}_j}+s}\right)\left(\frac{\mu_{\mathcal{I}_j}-\lambda_{\mathcal{I}_j}+\lambda_j+s}{\mu_{\mathcal{I}_j}-\lambda_{\mathcal{I}_j}+\lambda_j}\right) \tag{4.5}$$

To interpret the form of the Laplace transform of response time given in (4.5), observe that the Laplace transform of time in queue in an M/M/1 with arrival rate λ and service rate μ is

$$\widetilde{T}_Q^{M/M/1} = \left(\frac{\mu-\lambda}{\mu-\lambda+s}\right)\cdot\left(\frac{\mu+s}{\mu}\right).$$

Hence we can interpret the class-i response time as the queueing time in multiple successive M/M/1s, followed by the response time in a final M/M/1. Consider the example shown in Figure 4.1(a) for class $i=4$. We imagine that when a class-4 job arrives to the system it behaves like a fully redundant class-3 job and joins the queue at all five servers. Now suppose that our class-4 job reaches the head of the queue at server 2, which cannot serve class-4 jobs. At this point the class-4 job has experienced the queueing time of a class-3 job, but it cannot yet enter service. We now imagine the class-4 job remaining in only the queues in subsystem $\mathcal{A}$ and repeat the above reasoning to see that the class-4 job will now experience the queueing time of a class-5 job (which is fully redundant in subsystem $\mathcal{A}$). Hence we can view class-i jobs as moving through an M/M/1 queue for each successive nested subsystem, until they are finally served in the "innermost" subsystem $\mathcal{I}_i$.

We now turn to the formal proof of Theorem 4.1.

Proof. Consider $\mathcal{I}_i$, the subsystem consisting only of those servers in S_i and the job classes r such that $S_r \subseteq S_i$. From Lemma 4.1 we know that the state of subsystem $\mathcal{I}_i$ can be written in the form $\left(i, n_i, \binom{(\mathcal{A}_i)}{(\mathcal{B}_i)}\right)$, where $(\mathcal{A}_i)$ and $(\mathcal{B}_i)$ represent the substates of the smaller subsystems into which subsystem $\mathcal{I}$ decomposes, as in Lemma 4.2. Note that we exclude jobs that appear in the

queue behind the first class-i' job, where i' is the class such that $S_{i'}$ is the smallest set that strictly contains S_i.

Now consider some class j such that $S_j \supseteq S_{i'}$. We can write the state of subsystem $\mathcal{I}_j$ in the form $\left(j, n_j, \begin{smallmatrix}(\mathcal{A}_j)\\(\mathcal{B}_j)\end{smallmatrix}\right)$, where $(\mathcal{A}_j)$ and $(\mathcal{B}_j)$ represent the states of the smaller subsystems into which subsystem $\mathcal{I}_j$ decomposes. It must be the case that either $\mathcal{I}_i \subset \mathcal{A}_j$ or $\mathcal{I}_i \subset \mathcal{B}_j$; without loss of generality assume that $\mathcal{I}_i \subset \mathcal{A}_j$.

When deriving the distribution of the number of class-i jobs in the system, we can immediately aggregate over all possible $(\mathcal{B})$ states because none of the $\mathcal{B}$ subsystems contain any class-i jobs. Hence the overall system states we care about are of the form $\left(R, n_R, \begin{smallmatrix}(\mathcal{A}_R)\\(*)\end{smallmatrix}\right)$. Expanding all $(\mathcal{A})$ substates until we reach subsystem $\mathcal{I}_i$ and dropping the $(*)$ substates from our notation, we obtain states of the form $\left(C_x, n_x, C_{x-1}, n_{x-1}, \ldots, i', n_{i'}, i, n_i, \begin{smallmatrix}(\mathcal{A}_i)\\(\mathcal{B}_i)\end{smallmatrix}\right)$, where job classes $C_{i+1=i'}, \ldots, C_x = R$ are all the job classes j with $S_j \supset S_i$. Note that due to the nested structure, there is only one order in which these classes can appear in this description of the system state. In our state, if there is not a class-j job in the system we set $n_j = -$; otherwise n_j is the number of jobs behind the first class-j job in the queue and in front of the first class-$(j+1)$ job. In the example shown in Figure 4.1, we can write the system state for class $i = 4$ as $\left(3, n_3 = 1, 5, n_5 = 2, 4, n_4 = 1, \begin{smallmatrix}(-)\\(-)\end{smallmatrix}\right)$.

From Lemma 4.1, the limiting probability of being in state $\left(C_x, n_x, C_{x-1}, n_{x-1}, \ldots, i, n_i, \begin{smallmatrix}(*)\\(*)\end{smallmatrix}\right)$ is

$$\pi_{\left(C_x, n_x, C_{x-1}, n_{x-1}, \ldots, i, n_i, \begin{smallmatrix}(*)\\(*)\end{smallmatrix}\right)} = \prod_{\substack{j=i \\ n_j \neq -}}^{x} \left(\frac{\lambda_j}{\mu_{\mathcal{I}_j}}\right) \cdot \left(\frac{\lambda_{\mathcal{I}_j}}{\mu_{\mathcal{I}_j}}\right)^{n_j} \cdot (1 - \rho_j).$$

Now we are ready to find the z-transform of the number of class-i jobs in the system. We will do this by conditioning on there being a class-a job in the system, where $S_a \supset S_i$. Aggregating over all possible values of n_j (including $n_j = -$) for all $j \neq a$, we find

$$\Pr\{\text{at least one class-}a\text{ job in system}\} = \frac{\lambda_a}{\mu_{\mathcal{I}_a} - \lambda_{\mathcal{I}_a} + \lambda_a} = \rho_a.$$

Conditioned on there being a class-a job in the system, we have:

$$\pi_{\left(C_x, n_x, C_{x-1}, n_{x-1}, \ldots, i, n_i, \begin{smallmatrix}(*)\\(*)\end{smallmatrix}\right) \Big| n_a \neq -} = \left(\frac{\lambda_a}{\mu_{\mathcal{I}_a}}\right) \cdot \left(\frac{\lambda_{\mathcal{I}_a}}{\mu_{\mathcal{I}_a}}\right)^{n_a} \cdot \left(1 - \frac{\lambda_{\mathcal{I}_a}}{\mu_{\mathcal{I}_a}}\right) \cdot \prod_{\substack{j=i \\ j \neq a \\ n_j \neq -}}^{x} \left(\frac{\lambda_j}{\mu_{\mathcal{I}_j}}\right) \cdot \left(\frac{\lambda_{\mathcal{I}_j}}{\mu_{\mathcal{I}_j}}\right)^{n_j} \cdot (1 - \rho_j).$$

Observe that the number of jobs in the queue behind the class-a job (but in front of the first class-$(a+1)$ job), $N^{(a)}$, is geometrically distributed with parameter $1 - \frac{\lambda_{\mathcal{I}_a}}{\mu_{\mathcal{I}_a}}$. Each of these jobs is a class-i job with probability $\frac{\lambda_i}{\lambda_{\mathcal{I}_a}}$. Let $N_{i|a}$ be the number of class-i jobs behind the first class-a

job (but in front of the first class-$(a+1)$ job), given that there is a class-a job. Conditioning on the value of $N^{(a)}$, we find that the z-transform of $N_{i|a}$ is

$$\begin{aligned}
\widehat{N}_{i|a}(z) &= \sum_{n^{(a)}=0}^{\infty} \left(\frac{\lambda_{\mathcal{I}_a}}{\mu_{\mathcal{I}_a}}\right)^{n^{(a)}} \left(1-\frac{\lambda_{\mathcal{I}_a}}{\mu_{\mathcal{I}_a}}\right) \sum_{n_i^{(a)}=0}^{n^{(a)}} \binom{n^{(a)}}{n_i^{(a)}} z^{n_i^{(a)}} \left(\frac{\lambda_i}{\lambda_{\mathcal{I}_a}}\right)^{n_i^{(a)}} \left(1-\frac{\lambda_i}{\lambda_{\mathcal{I}_a}}\right) \\
&= \sum_{n^{(a)}=0}^{\infty} \left(\frac{\lambda_{\mathcal{I}_a}}{\mu_{\mathcal{I}_a}}\right)^{n^{(a)}} \left(1-\frac{\lambda_{\mathcal{I}_a}}{\mu_{\mathcal{I}_a}}\right) \left(1-\frac{\lambda_i}{\lambda_{\mathcal{I}_a}}(1-z)\right)^{n^{(a)}} \\
&= \frac{1-\frac{\lambda_{\mathcal{I}_a}}{\mu_{\mathcal{I}_a}}}{1-\frac{\lambda_{\mathcal{I}_a}}{\mu_{\mathcal{I}_a}}\left(1-\frac{\lambda_i}{\lambda_{\mathcal{I}_a}}(1-z)\right)} \\
&= \frac{\mu_{\mathcal{I}_a}-\lambda_{\mathcal{I}_a}}{\mu_{\mathcal{I}_a}-\lambda_{\mathcal{I}_a}+\lambda_i-\lambda_i z} \\
&= \frac{\frac{\mu_{\mathcal{I}_a}-\lambda_{\mathcal{I}_a}}{\mu_{\mathcal{I}_a}-\lambda_{\mathcal{I}_a}+\lambda_i}}{1-\frac{\lambda_i}{\mu_{\mathcal{I}_a}-\lambda_{\mathcal{I}_a}+\lambda_i}z}.
\end{aligned}$$

Note that this is the transform of a geometric random variable with parameter

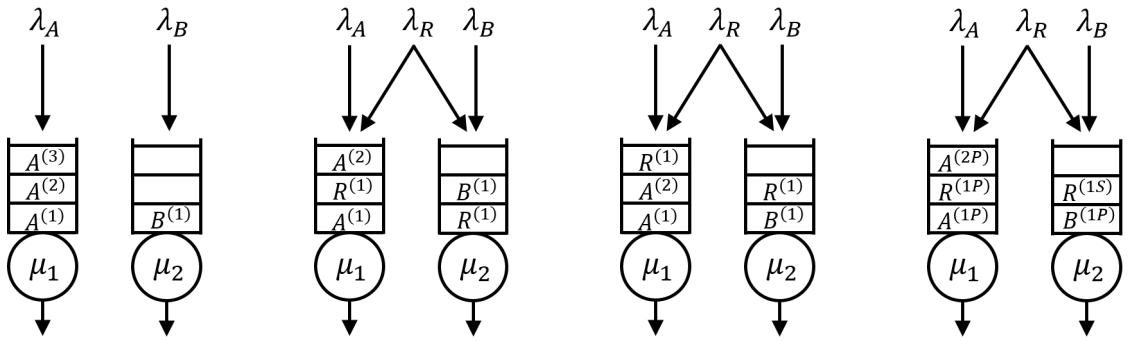
$$p_{i|a} = 1 - \frac{\lambda_i}{\mu_{\mathcal{I}_a}-\lambda_{\mathcal{I}_a}+\lambda_i}.$$

We now condition on whether there is a class-a job in the system to find $\widehat{N}_{i,a}(z)$, the z-transform of the number of class-i jobs in the queue immediately behind the class-a job (if there is one):

$$\begin{aligned}
\widehat{N}_{i,a}(z) &= \Pr\{\text{at least one class-}a\text{ job in system}\}\widehat{N}_{i|a}(z) + \Pr\{\text{no class-}a\text{ job in system}\}z^0 \\
&= \rho_a \cdot \left(\frac{\mu_{\mathcal{I}_a}-\lambda_{\mathcal{I}_a}}{\mu_{\mathcal{I}_a}-\lambda_{\mathcal{I}_a}+\lambda_i-\lambda_i z}\right) + (1-\rho_a) \\
&= \left(\frac{\mu_{\mathcal{I}_a}-\lambda_{\mathcal{I}_a}}{\mu_{\mathcal{I}_a}-\lambda_{\mathcal{I}_a}+\lambda_a}\right) \cdot \left(1+\frac{\lambda_a}{\mu_{\mathcal{I}_a}-\lambda_{\mathcal{I}_a}+\lambda_i-\lambda_i z}\right) \\
&= \left(\frac{\mu_{\mathcal{I}_a}-\lambda_{\mathcal{I}_a}}{\mu_{\mathcal{I}_a}-\lambda_{\mathcal{I}_a}+\lambda_a}\right) \cdot \left(\frac{\mu_{\mathcal{I}_a}-\lambda_{\mathcal{I}_a}+\lambda_i+\lambda_a-\lambda_i z}{\mu_{\mathcal{I}_a}-\lambda_{\mathcal{I}_a}+\lambda_i-\lambda_i z}\right).
\end{aligned}$$

The above tells us that we can view the queue as $x-i$ independent pieces, each with a geometrically distributed number of class-i jobs. We can thus express the total number of class-i jobs in the system as the sum of the $x-i$ geometrically distributed components. The z-transform of the number of class-i jobs in the system is thus:

$$\begin{aligned}
\widehat{N}_i(z) = &\left(\frac{\mu_{\mathcal{I}_i}-\lambda_{\mathcal{I}_i}}{\mu_{\mathcal{I}_i}-\lambda_{\mathcal{I}_i}+\lambda_i}\right) \cdot \left(\frac{\mu_{\mathcal{I}_i}-\lambda_{\mathcal{I}_i}+\lambda_i}{\mu_{\mathcal{I}_i}-\lambda_{\mathcal{I}_i}+\lambda_i-\lambda_i z}\right) \\
&\cdot \prod_{a=i+1}^{x} \left(\frac{\mu_{\mathcal{I}_a}-\lambda_{\mathcal{I}_a}}{\mu_{\mathcal{I}_a}-\lambda_{\mathcal{I}_a}+\lambda_a}\right) \cdot \left(\frac{\mu_{\mathcal{I}_a}-\lambda_{\mathcal{I}_a}+\lambda_i+\lambda_a-\lambda_i z}{\mu_{\mathcal{I}_a}-\lambda_{\mathcal{I}_a}+\lambda_i-\lambda_i z}\right),
\end{aligned}$$



(a) non-redundant FCFS (b) FCFS-R (c) LRF (d) PF

Figure 4.3: The $\mathbb{W}$ model with four different scheduling policies: (a) FCFS in a system with no redundancy, (b) FCFS in a system where class-R jobs are redundant (FCFS-R, Section 4.3), (c) Least Redundant First (Section 4.4), and (d) Primaries First, where P and S denote a job's primary and secondary copies respectively (Section 4.5).

where the first term is different because if $n_i \neq -$ we have one additional class-i job in the system.

Finally, to find the Laplace transform of response time for class-i jobs, note that class-i jobs depart the system in the order in which they arrived, so we can apply Distributional Little's Law. Let $z = \frac{\lambda_i + s}{\lambda_i}$. Then we have

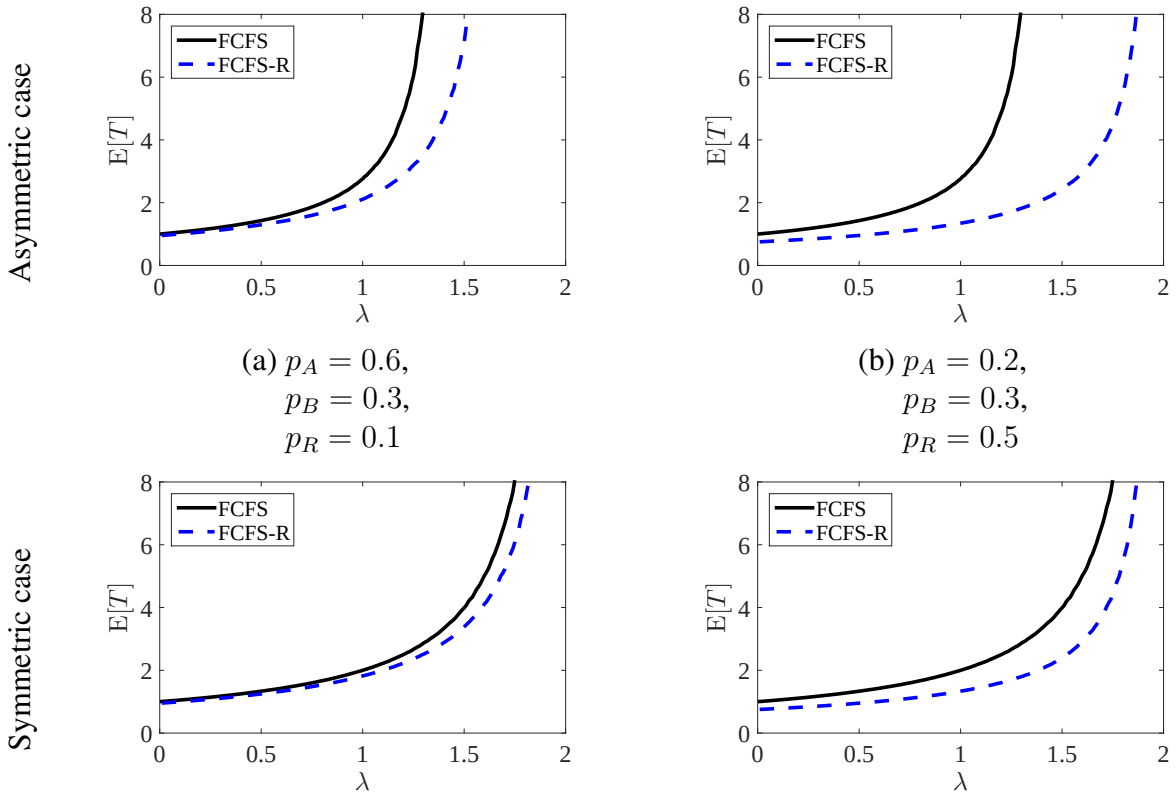
$$\widetilde{T}^{(i)}(s) = \left(\frac{\mu_{\mathcal{I}_i} - \lambda_{\mathcal{I}_i}}{\mu_{\mathcal{I}_i} - \lambda_{\mathcal{I}_i} + s}\right) \cdot \prod_{j:S_i \subset S_j} \left(\frac{\mu_{\mathcal{I}_j} - \lambda_{\mathcal{I}_j}}{\mu_{\mathcal{I}_j} - \lambda_{\mathcal{I}_j} + s}\right)\left(\frac{\mu_{\mathcal{I}_j} - \lambda_{\mathcal{I}_j} + \lambda_j + s}{\mu_{\mathcal{I}_j} - \lambda_{\mathcal{I}_j} + \lambda_j}\right)$$

as desired. □

4.3.2 Performance

Recall that our two goals are (1) to achieve low overall system mean response time, and (2) to preserve fairness across job classes, meaning that no class of jobs should have a higher mean response time in the redundancy system than in the original system with no redundancy under FCFS scheduling. In this section we evaluate the extent to which FCFS-R meets these goals. We look specifically at the $\mathbb{W}$ model; Figure 4.3(a) shows the non-redundant system, and Figure 4.3(b) shows the redundancy system with FCFS scheduling. For our performance analysis, we assume the service rate is the same at both servers ($\mu_1 = \mu_2 = 1$). Figures 4.3(c) and (d) show the scheduling policies that we study in Section 4.4 and 4.5.

As described in Section 4.2, we consider two different settings. In the first setting, we start with an *asymmetric* system in which $p_A = 0.7$ and $p_B = 0.3$; we then look at the effect of having some jobs switch from being class-A to class-R by decreasing p_A while increasing p_R (Figure 4.2(b)). In the second setting, we start with a *symmetric* system (i.e., $p_A = p_B = 0.5$) and look at the effect of having some become redundant while the system remains symmetric. That is, we hold $\lambda_A = \lambda_B$ while increasing p_R (Figure 4.2(c)).



(a) $p_A = 0.6$, $p_B = 0.3$, $p_R = 0.1$

(b) $p_A = 0.2$, $p_B = 0.3$, $p_R = 0.5$

(c) $p_A = 0.45$, $p_B = 0.45$, $p_R = 0.1$

(d) $p_A = 0.25$, $p_B = 0.25$, $p_R = 0.5$

Figure 4.4: Comparing mean response time for the overall system under FCFS-R (dashed blue line) and under non-redundant FCFS (solid black line) in the asymmetric (top row) and symmetric (bottom row) cases.

Objective 1: Low Overall Mean Response Time

We first evaluate the response time benefit achieved by the overall system from using redundancy. In the asymmetric case ($p_A = 0.7$, $p_B = 0.3$), allowing some class-A jobs to become redundant (i.e., increasing p_R and decreasing p_A) dramatically reduces mean response time for the overall system, and can even increase the system's stability region. The improvement is larger as the fraction of jobs that are redundant increases (see Figure 4.4(a),(b)). When the system initially is symmetric ($p_A = p_B = 0.5$), mean response time can be as much as 30% lower under FCFS-R than under non-redundant FCFS, but the advantage is not as great as in the asymmetric case (see Figure 4.4(c),(d)).

The difference in the magnitude of improvement in the two system configurations occurs because one of the primary benefits of redundancy is that it helps with load balancing. In the asymmetric system, allowing some class-A jobs to become redundant shifts some of the class-A load away from server 1 and onto server 2, which helps alleviate long queueing times experienced by the jobs that are served at server 1. In contrast, the symmetric system does not have an

initial load imbalance, so there is not as much room for improvement in queueing time. While waiting in both queues can help the redundant jobs to experience shorter queueing times, in the symmetric case the runtime reduction becomes a more significant component of the benefit offered by redundancy. When both class-A and class-B jobs become redundant and the system remains symmetric as p_R increases, an increasing number of jobs end up in service on both servers simultaneously and thus get to experience the minimum runtime across the two servers.

Objective 2: Fairness Across Classes

Here we consider the per-class mean response times; our goal is for each class to perform at least as well in the redundancy system as under the initial non-redundant FCFS.

We begin by considering the asymmetric case, in which in the nonredundant system 70% of the jobs are class-A and 30% are class-B, and in the redundant system some of the class-A jobs become redundant. It is easy to see by a sample-path argument that jobs that switch from class-A to class-R will benefit from the switch (Figures 4.5 and 4.6). Under FCFS-R, each class-R job retains the spot in first-come first-served order in the queue at server 1 that it held before it became redundant, while adding an opportunity to complete earlier on server 2. Note that there is a kink in the curve for class-R jobs under FCFS-R at $\lambda = 1.67$. This is because at $\lambda = 1.67$, the class-A jobs become overloaded ($\lambda \cdot p_A = 1 = \mu$), so none of the class-$R$ jobs end up in service on server 1. Instead, all class-R jobs effectively only get to join the queue at server 2, which becomes an M/M/1 with arrival rate $\lambda_B + \lambda_R$ and service rate μ. Thus at $\lambda = 1.67$ the mean response time for class-R jobs shifts from $\frac{1}{2\mu - \lambda_A - \lambda_B - \lambda_R}$ to $\frac{1}{\mu - \lambda_B - \lambda_R}$.

Class-A jobs also benefit from allowing some jobs to become redundant class-R jobs because the class-R jobs may leave the queue at server 1 without ever entering service. Unfortunately, the story is the opposite for the class-B jobs. The class-B jobs can be hurt by redundancy because the redundant jobs, which initially were class-A jobs, now join the queue at server 2 and take away some capacity from the class-B jobs. For example, when $p_R = 0.5$ and $\lambda = 1.4$ mean response time for the class-B jobs is 32% higher under FCFS-R than under non-redundant FCFS.

In the symmetric case, in which in the nonredundant system $p_A = p_B = 0.5$, and in the redundant system both class-A jobs and class-B jobs become redundant, the effects of redundancy are much less pronounced (see Figure 4.6). Here all three classes experience lower mean response time under FCFS-R than under non-redundant FCFS. But for the class-A and class-R jobs, the improvement is far greater in the asymmetric case. Unlike in the asymmetric case, the class-B jobs are not hurt by redundancy in the symmetric case because here some class-B jobs get to become redundant, thereby benefiting from waiting in both queues and potentially running on both servers.

We see, from the pain experienced by the class-B customers in the asymmetric case, that, in general, FCFS-R *fails to achieve our second objective* of ensuring that all classes perform at least as well in the redundant system as in the non-redundant system.

4.3.3 Proof of Lemma 4.1

Proof. Our proof is by induction on the number of jobs in subsystems $\mathcal{A}$ and $\mathcal{B}$. As our base case when $n_R \neq -$, assume there is one job in each subsystem, and call these jobs a_1 and b_1. Then

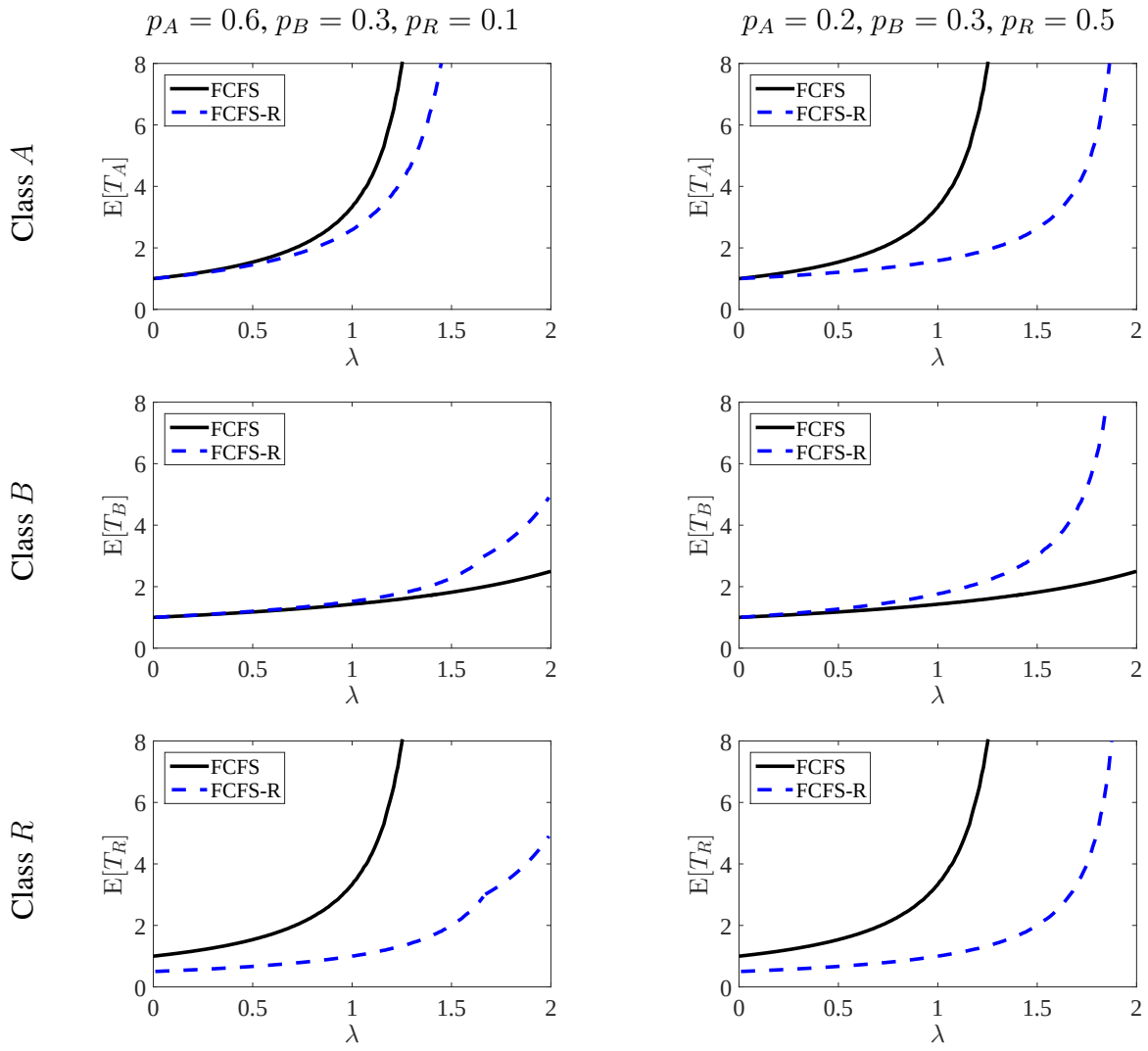


Figure 4.5: Comparing mean response time under FCFS-R (dashed blue line) and FCFS (solid black line) in the asymmetric case for class-A (top row), class-B (middle row), and class-R (bottom row) jobs. Note the kink at $\lambda = 1.67$ for class-B and class-R jobs when $p_A = 0.6$; this occurs because at $\lambda = 1.67$ the class-A jobs become unstable.

there are two possible ways to interleave jobs a_1 and b_1, so we have

$$\begin{aligned} P(R, n_R, \binom{a_1}{b_1}) &= \sum_{c_1,\ldots,c_{n_R}} \left(P(c_{n_R}, \ldots, c_1, R, a_1, b_1) + P(c_{n_R}, \ldots, c_1, R, b_1, a_1)\right) \\ &= \sum_{c_1,\ldots,c_{n_R}} \left[\left(\frac{\lambda_{c_1}}{\mu}\right) \cdots \left(\frac{\lambda_{c_{n_R}}}{\mu}\right) \cdot \left(\frac{\lambda_R}{\mu}\right) \cdot \left(\frac{\lambda_{a_1}}{\mu_{\mathcal{I}_{a_1}} + \mu_{\mathcal{I}_{b_1}}}\right) \cdot \left(\frac{\lambda_{b_1}}{\mu_{\mathcal{I}_{b_1}}}\right) \right. \\ &\qquad \left. + \left(\frac{\lambda_{c_1}}{\mu}\right) \cdots \left(\frac{\lambda_{c_{n_R}}}{\mu}\right) \cdot \left(\frac{\lambda_R}{\mu}\right) \cdot \left(\frac{\lambda_{b_1}}{\mu_{\mathcal{I}_1} + \mu_{\mathcal{I}_{b_1}}}\right) \cdot \left(\frac{\lambda_{a_1}}{\mu_{\mathcal{I}_{a_1}}}\right)\right] \end{aligned}$$

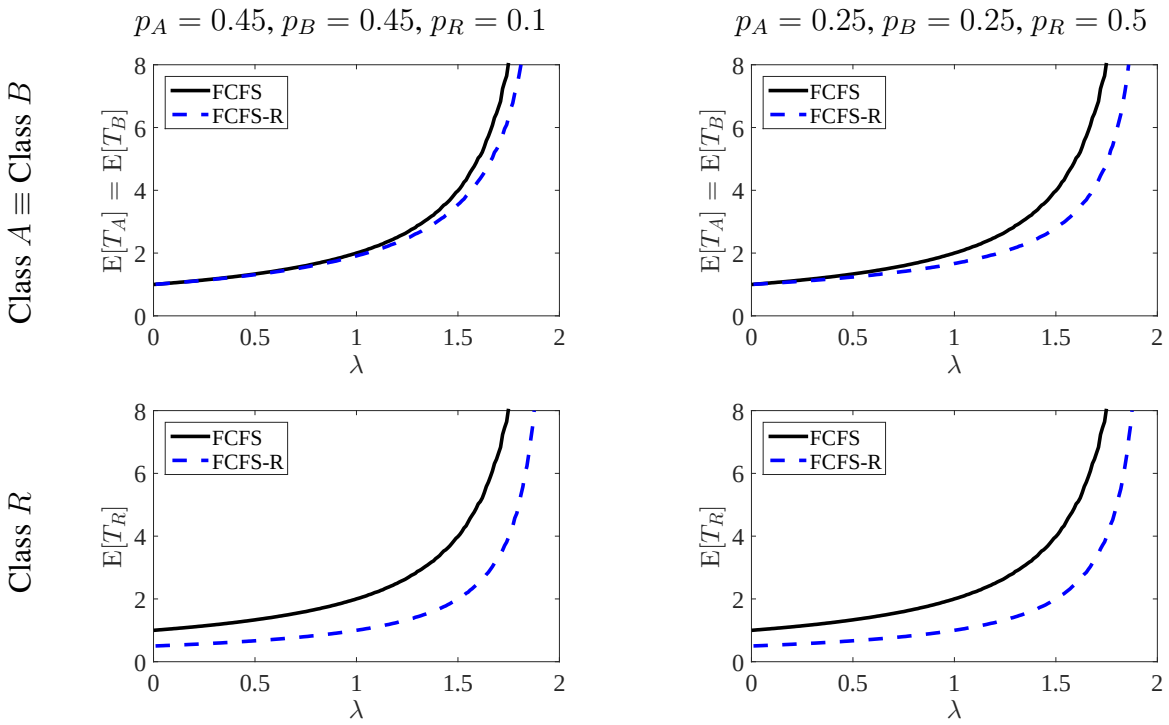


Figure 4.6: Comparing mean response time under FCFS-R (dashed blue line) and FCFS (solid black line) in the symmetric case for class-A and class-B (top row, classes A and B experience the same performance in the symmetric case), and class-R (bottom row).

$$
\begin{aligned}
&= \left(\frac{\lambda}{\mu}\right)^{n_R} \cdot \left(\frac{\lambda_R}{\mu}\right) \cdot \frac{\lambda_{a_1}\lambda_{b_1}(\mu_{\mathcal{I}_{a_1}} + \mu_{\mathcal{I}_{b_1}})}{(\mu_{\mathcal{I}_{a_1}} + \mu_{\mathcal{I}_{b_1}})\mu_{\mathcal{I}_{a_1}}\mu_{\mathcal{I}_{b_1}}} \\
&= \left(\frac{\lambda}{\mu}\right)^{n_R} \cdot \left(\frac{\lambda_R}{\mu}\right) \cdot \left(\frac{\lambda_{a_1}}{\mu_{\mathcal{I}_{a_1}}}\right) \cdot \left(\frac{\lambda_{b_1}}{\mu_{\mathcal{I}_{b_1}}}\right) \\
&= \left(\frac{\lambda}{\mu}\right)^{n_R} \cdot \left(\frac{\lambda_R}{\mu}\right) \cdot P(\mathcal{A}) \cdot P(\mathcal{B}),
\end{aligned}
$$

where $P(\mathcal{A}) = \frac{\lambda_{a_1}}{\mu_{\mathcal{I}_{a_1}}}$ (respectively, $P(\mathcal{B}) = \frac{\lambda_{b_1}}{\mu_{\mathcal{I}_{b_1}}}$) gives the (unnormalized) limiting probability of being in state (a_1) (respectively, (b_1)) in a system consisting of only the job classes and servers in subsystem $\mathcal{I}_{a_1}$ (respectively, $\mathcal{I}_{b_1}$).

Now suppose inductively that (4.3) holds for all $i < \ell_{\mathcal{A}}$, $j < \ell_{\mathcal{B}}$ jobs in subsystems $\mathcal{A}$ and $\mathcal{B}$. Then with $\ell_{\mathcal{A}} + 1$ jobs in subsystem $\mathcal{A}$, we have:

$$
\begin{aligned}
P\left(R, n_R, {}^{(a_{\ell_{\mathcal{A}}},\ldots,a_1)}_{(b_{\ell_{\mathcal{B}}},\ldots,b_1)}\right) = \sum_{c_1,\ldots,c_{n_R}} \Big[& P\left(c_{n_R},\ldots,c_1,R,a_{\ell_{\mathrm{A}}+1}, {}^{(a_{\ell_{\mathcal{A}}},\ldots,a_1)}_{(b_{\ell_{\mathcal{B}}},\ldots,b_1)}\right) \\
& + P\left(c_{n_R},\ldots,c_1,R,b_{\ell_{\mathrm{B}}},a_{\ell_{\mathrm{A}}+1}, {}^{(a_{\ell_{\mathcal{A}}},\ldots,a_1)}_{(b_{\ell_{\mathcal{B}}-1},\ldots,b_1)}\right) \\
& + \cdots + P\left(c_{n_R},\ldots,c_1,R,b_{\ell_{\mathrm{B}}},b_{\ell_{\mathcal{B}}-1},\ldots,b_1,a_{\ell_{\mathrm{A}}+1}, {}^{(a_{\ell_{\mathcal{A}}},\ldots,a_1)}_{(-)}\right)\Big]
\end{aligned}
$$

$$
\begin{aligned}
&= \left(\frac{\lambda}{\mu}\right)^{n_R} \cdot \left(\frac{\lambda_R}{\mu}\right) \cdot \Bigg[\left(\frac{\lambda_{a_{\ell_{\mathcal{A}}+1}}}{\mu_{\bigcup_{i=1}^{\ell_{\mathcal{A}}+1} \mathcal{I}_{a_i}} + \mu_{\bigcup_{i=1}^{\ell_{\mathcal{B}}} \mathcal{I}_{b_i}}} \right) \cdot \left(\prod_{j=1}^{\ell_{\mathcal{A}}} \frac{\lambda_{a_j}}{\mu_{\bigcup_{i=1}^{j} \mathcal{I}_{a_i}}} \right) \cdot \left(\prod_{j=1}^{\ell_{\mathcal{B}}} \frac{\lambda_{b_j}}{\mu_{\bigcup_{i=1}^{j} \mathcal{I}_{b_i}}} \right) \\
&+ \left(\frac{\lambda_{b_{\ell_{\mathcal{B}}}}}{\mu_{\bigcup_{i=1}^{\ell_{\mathcal{A}}+1} \mathcal{I}_{a_i}} + \mu_{\bigcup_{i=1}^{\ell_{\mathcal{B}}} \mathcal{I}_{b_i}}} \right) \cdot \left(\frac{\lambda_{a_{\ell_{\mathcal{A}}+1}}}{\mu_{\bigcup_{i=1}^{\ell_{\mathcal{A}}+1} \mathcal{I}_{a_i}} + \mu_{\bigcup_{i=1}^{\ell_{\mathcal{B}}-1} \mathcal{I}_{b_i}}} \right) \cdot \left(\prod_{j=1}^{\ell_{\mathcal{A}}} \frac{\lambda_{a_j}}{\mu_{\bigcup_{i=1}^{j} \mathcal{I}_{a_i}}} \right) \cdot \left(\prod_{j=1}^{\ell_{\mathcal{B}}-1} \frac{\lambda_{b_j}}{\mu_{\bigcup_{i=1}^{j} \mathcal{I}_{b_i}}} \right) \\
&+ \cdots + \left(\frac{\lambda_{b_{\ell_{\mathcal{B}}}}}{\mu_{\bigcup_{i=1}^{\ell_{\mathcal{A}}+1} \mathcal{I}_{a_i}} + \mu_{\bigcup_{i=1}^{\ell_{\mathcal{B}}} \mathcal{I}_{b_i}}} \right) \cdots \left(\frac{\lambda_{b_1}}{\mu_{\bigcup_{i=1}^{\ell_{\mathcal{A}}+1} \mathcal{I}_{a_i}} + \mu_{\mathcal{I}_{b_1}}} \right) \cdot \left(\frac{\lambda_{a_{\ell_{\mathcal{A}}+1}}}{\mu_{\bigcup_{i=1}^{\ell_{\mathcal{A}}+1} \mathcal{I}_{a_i}}} \right) \cdot \left(\prod_{j=1}^{\ell_{\mathcal{A}}} \frac{\lambda_{a_j}}{\mu_{\bigcup_{i=1}^{j} \mathcal{I}_{a_i}}} \right) \Bigg] \\
&= \left(\frac{\lambda}{\mu}\right)^{n_R} \cdot \left(\frac{\lambda_R}{\mu}\right) \cdot \left(\prod_{j=1}^{\ell_{\mathcal{A}}+1} \frac{\lambda_{a_j}}{\mu_{\bigcup_{i=1}^{j} \mathcal{I}_{a_i}}} \right) \cdot \left(\prod_{j=1}^{\ell_{\mathcal{B}}} \frac{\lambda_{b_j}}{\mu_{\bigcup_{i=1}^{j} \mathcal{I}_{b_i}}} \right),
\end{aligned}
$$

where the first equality results from the inductive hypothesis and the form of the limiting probabilities of the detailed state space given in (4.1), and the second equality results from combining the summed terms using a common denominator, and then simplifying. We thus have exactly the form given in (4.3).

A similar argument follows for the case when $n_R = -$. □

4.3.4 Proof of Lemma 4.3

Proof. We find the normalizing constant via induction on the number of job classes. Our base case is the $\mathbb{W}$ model, which has three classes and two servers (note that we can obtain systems with two or one classes by simply setting the arrival rates for unwanted classes to be 0). From 3, we know that the normalizing constant in the $\mathbb{W}$ model is

$$\mathcal{C}_{\mathbb{W}} = (1 - \rho_A)(1 - \rho_B)(1 - \rho_R).$$

Now assume that the normalizing constant follows the form given in (4.4) for all systems with at most ℓ job classes. Consider a system with $\ell + 1$ job classes, and let class R be the most redundant class (i.e., class-R jobs replicate to all servers). By Lemma 4.1, we know that the limiting probability of being in state $\left(R, n_R, \binom{\mathcal{A}}{\mathcal{B}}\right)$ is

$$\pi_{\left(R, n_R, \binom{\mathcal{A}}{\mathcal{B}}\right)} = \left(\frac{\lambda}{\mu_{\mathcal{A}} + \mu_{\mathcal{B}}}\right)^{n_R} \left(\frac{\lambda_R}{\mu_{\mathcal{A}} + \mu_{\mathcal{B}}}\right) \cdot \pi_{\mathcal{A}} \cdot \pi_{\mathcal{B}},$$

up to the normalizing constant, where there are $i \leq m$ job classes in the left subsystem and $m - i$ job classes in the right subsystem.

Aggregating over all possible states for the left and the right subsystems, we find, for a constant ξ to be determined below,

$$\pi_{\left(R, n_R, \binom{*}{*}\right)} = \xi \left(\frac{\lambda}{\mu_{\mathcal{A}} + \mu_{\mathcal{B}}}\right)^{n_R} \left(\frac{\lambda_R}{\mu_{\mathcal{A}} + \mu_{\mathcal{B}}}\right),$$

where, letting $A_1, \ldots, A_i$ and $B_1, \ldots, B_{n-i}$ denote the job classes in subsystems $\mathcal{A}$ and $\mathcal{B}$ respectively,

$$\mathcal{C}_{\mathcal{A}} = \prod_{y=1}^{i}(1 - \rho_{A_y})$$

and

$$\mathcal{C}_{\mathcal{B}} = \prod_{y=1}^{\ell-i}(1 - \rho_{B_y})$$

follow from the inductive hypothesis, and $\mathcal{C} = \xi \cdot \mathcal{C}_{\mathcal{A}} \cdot \mathcal{C}_{\mathcal{B}}$.

Finally, we sum over all values of n_R, as well as the state in which there are no class-R jobs in the system; this sum must be equal to 1:

$$\begin{aligned}
1 &= \pi_{\left(R,-,\substack{(*)\\(*)}\right)} + \sum_{n_R=0}^{\infty} \pi_{\left(R,n_R,\substack{(*)\\(*)}\right)} \\
1 &= \xi + \sum_{n_R=0}^{\infty} \xi \left(\frac{\lambda}{\mu_{\mathcal{A}} + \mu_{\mathcal{B}}}\right)^{n_R} \left(\frac{\lambda_R}{\mu_{\mathcal{A}} + \mu_{\mathcal{B}}}\right) \\
1 &= \xi \left(1 + \left(\frac{\lambda_R}{\mu_{\mathcal{A}} + \mu_{\mathcal{B}}}\right)\left(\frac{1}{1 - \frac{\lambda}{\mu_{\mathcal{A}}+\mu_{\mathcal{B}}}}\right)\right) \\
1 &= \xi \left(\frac{\mu_{\mathcal{A}} + \mu_{\mathcal{B}} - \lambda + \lambda_R}{\mu_{\mathcal{A}} + \mu_{\mathcal{B}} - \lambda}\right),
\end{aligned}$$

so we have

$$\begin{aligned}
\xi = \pi_{\left(R,-,\substack{(*)\\(*)}\right)} &= \frac{\mu_{\mathcal{A}} + \mu_{\mathcal{B}} - \lambda}{\mu_{\mathcal{A}} + \mu_{\mathcal{B}} - \lambda + \lambda_R} \\
&= 1 - \rho_R
\end{aligned}$$

and hence

$$\begin{aligned}
\mathcal{C} &= \xi \cdot \mathcal{C}_{\mathcal{A}} \cdot \mathcal{C}_{\mathcal{B}} \\
&= \prod_{i=1}^{\ell}(1 - \rho_i)
\end{aligned}$$

as desired. □

4.4 Least Redundant First (LRF)

Our study of the FCFS-R policy reveals that using redundancy can lead to significant response time reductions relative to a system in which no jobs are redundant. In this section, we ask what is the *most* benefit we can achieve with respect to efficiency, i.e., reduction in overall system

response time. Is FCFS-R the best we can do? Or is there some other policy that allows us to achieve even lower response times?

Intuitively, one important reason why redundancy helps is that it reduces the likelihood that servers are idle while there is still work in the system. While FCFS-R helps with load balancing, it is possible for all of the redundant jobs to complete service, leaving behind a potentially unbalanced number of non-redundant jobs at each server. If this imbalance is large enough, one server can idle while others still have many jobs in the queue. We can alleviate this problem by preemptively prioritizing less-redundant jobs so that we preserve redundancy to prevent idling in the future. We call this idea the Least Redundant First (LRF) scheduling policy. We will show that LRF minimizes overall response time for any general exogenous arrival process.

4.4.1 Policy and Optimality

Under LRF, at all times each server works on the job in its queue that is redundant at the fewest queues. Formally, for an arbitrary server s, let $J_s = \{i : \ s \in S_i\}$ be the set of job classes that replicate to server s, as defined in Section 4.2.

Definition 4.2. *Under Least Redundant First (LRF), server s gives preemptive priority to jobs of class i over jobs of class $j \neq i$ if $i, j \in J_s$ and $|S_i| < |S_j|$.*

Note that this policy is well defined due to the nested structure: it is not possible to have $i, j \in J_s$ and $|S_i| = |S_j|$ unless $i = j$. For the purpose of our analysis, we require that jobs are *resumed*, rather than restarted, when they re-enter service after being preempted.

Intuitively, from the overall system's perspective LRF achieves low mean response time by preserving redundant jobs as long as possible (without idling) to maximize system efficiency. Under LRF, a redundant job only enters service at a particular server if there are no less-redundant jobs in the queue at that server, meaning that the jobs that stay in the system the longest are those that have the most options. This makes it much less likely under LRF than under non-redundant FCFS or even under FCFS-R that servers will idle while there is still work in the system. Indeed, we prove that LRF is optimal with respect to overall mean response time among all preemptive policies (Theorem 4.2).

Our proof relies on tracking the number of jobs of each class in the system at all times t for a given sample path. We show that for all job classes i and at all times t, if we count up the number of jobs that are class i or that share servers with but are less redundant than class i, this number is smaller under LRF than under any other policy. This tells us that LRF stochastically maximizes the number of departures from the system by time t, for all t, and so also minimizes mean response time (Corollary 4.1). Let $N_i(t)$ be the number of class-i jobs in the system at time t, and let $N^i(t) = \sum_{j:S_j \subseteq S_i} N_j(t)$ be the total number of class-i jobs plus those jobs that have priority over class-i jobs at any server (i.e., the jobs that are less redundant than class-i jobs and share a server with class-i jobs). Finally, let $\vec{N}(t) = (N^1(t), N^2(t), \ldots, N^\ell(t))$, where ℓ is the number of job classes. Unlike our analysis of FCFS-R, the proof of optimality for LRF does not require Poisson arrivals; we simply assume that arrivals are an arbitrary exogenous process.

Theorem 4.2. *The preemptive non-idling LRF policy stochastically minimizes $\{\vec{N}(t)\}_{t=0}^{\infty}$, among all preemptive, possibly idling, policies when runtimes are exponential, the arrivals form a general exogenous process, and the redundancy structure is nested.*

Proof. We first show that idling is non-optimal. Suppose that some arbitrary scheduling policy π idles server s at time 0 when there is a job of class i in the system such that $s \in S(i)$. Let an alternative scheduling policy π' serve the job of class i on server s and otherwise agree with π at time 0. Let $\delta > 0$ be the first time an event (arrival, service completion, or preemption) occurs under π, and let $Y \sim \text{Exp}(\mu_s)$ be the next completion time on server s under π' (at which time our class-i job will depart).

Case 1: $Y > \delta$**.** Let π' agree with π for all scheduling decisions beginning at time δ, and couple all random variables under π and π'. Then $\{\vec{N}'(t)\}_{t=0}^{\infty} = \{\vec{N}(t)\}_{t=0}^{\infty}$ almost surely.

Case 2: $Y = \delta$**.** From time Y on, whenever any server serves our class-i job under π, let the corresponding server idle under π', and let π' otherwise agree with π, again coupling all random variables under π and π'. Then our class-i job completes earlier under π' than under π, while all other jobs complete at the same time, so $\{\vec{N}'(t)\}_{t=0}^{\infty} \leq \{\vec{N}(t)\}_{t=0}^{\infty}$ almost surely.

We can repeat this argument at each time at which a server idles.

We next show that LRF is optimal among all non-idling policies. Suppose that some arbitrary scheduling policy π serves a class-j job on server s at time 0 when there is a class-i job in the system such that $s \in S(i) \subset S(j)$. Let an alternative scheduling policy π' serve the class-i job on server s and otherwise agree with π at time 0. As before, let δ be the first time an event occurs under π, and let Y be the next completion time on server s (note that under π this completion is of a class-j job, whereas under π' it is a class-i job).

Case 1: $Y > \delta$**.** Let π' agree with π for all scheduling decisions beginning at time δ, and couple all random variables under π and π'. Then $\{\vec{N}'(t)\}_{t=0}^{\infty} = \{\vec{N}(t)\}_{t=0}^{\infty}$ almost surely.

Case 2: $Y = \delta$**.** From time Y on, whenever any server serves our class-i job under π (note that this job has departed under π'), let the corresponding server serve our class-j job under π' (which is possible because $S(i) \subset S(j)$, and let π' otherwise agree with π. Then $N'^i(Y) < N^i(Y)$ and $N'^j(Y) = N^j(Y)$ (recall that $N^j(Y)$ counts both class-j jobs and jobs that have priority over class-j and share a server with class-j). More generally, repeating the argument at each time at which a server serves a more-redundant job instead of a less-redundant job that it could serve, we find that $\{\vec{N}'(t)\}_{t=0}^{\infty} \leq \{\vec{N}(t)\}_{t=0}^{\infty}$ almost surely, and hence it is optimal for each server to always serve the least redundant job among those it can serve. This policy is exactly LRF. □

Corollary 4.1. *LRF stochastically maximizes the number of departures (job completions) by time t, for all t, among all preemptive, possibly idling, policies, and therefore also minimizes mean response time.*

While our focus is on nested systems, it is worth noting that our optimality proof for LRF holds for any nested substructure, even in general, non-nested systems. For example, non-redundant jobs are always nested relative to all other job classes, so they should be given highest priority.

Corollary 4.2. *For* any *redundancy system (not just nested systems), any scheduling policy that minimizes mean response time must give highest preemptive priority to job classes i with $|S_i| = 1$, i.e., non-redundant job classes.*

4.4.2 Analysis

In general, analyzing per-class and overall system response time under LRF seems to be intractable. For non-redundant classes, analysis is possible because these classes are isolated from all other

jobs in the system; we provide exact expressions for the response time distribution for all non-redundant classes under LRF scheduling.

Lemma 4.4. *Under LRF, any non-redundant class i that runs only on server j has response time* $T_i \sim \text{Exp}(\mu_j - \lambda_i)$.

Proof. The non-redundant class-i jobs have highest preemptive priority among all jobs that share server j. Hence these jobs see an M/M/1 with arrival rate λ_i and service rate μ_j. □

Redundant classes and the overall system are more difficult to analyze exactly because a redundant job's response time is the minimum of the time it would experience across multiple queues, and these queues are not independent. Instead, we derive bounds on mean response time for redundant jobs and the overall system. Here we consider the $\mathbb{W}$ model, but our approach extends to other nested systems.

Theorem 4.3. *Under LRF scheduling in the $\mathbb{W}$ model, the overall system mean response time,* $\mathbf{E}\left[T^{\text{LRF}}\right]$*, is bounded by:*

$$\max\left\{\frac{1}{\mu-\lambda}, \frac{1}{\lambda}\left(\frac{\lambda_R}{\mu-\lambda} + \frac{\lambda_A}{\mu_1-\lambda_A} + \frac{\lambda_B}{\mu_2-\lambda_B}\right)\right\}$$
$$\leq \mathbf{E}\left[T^{\text{LRF}}\right] \leq \frac{1}{\lambda}\left(\frac{\lambda}{\mu-\lambda} + \frac{\lambda_A}{\mu_1-\lambda_A} + \frac{\lambda_B}{\mu_2-\lambda_B} - \frac{\lambda_A+\lambda_B}{\mu-\lambda_A-\lambda_B}\right),$$

where $\mu = \mu_1 + \mu_2$ and $\lambda = \lambda_A + \lambda_B + \lambda_R$.

Proof. Mean response time is minimized when all jobs are fully redundant (see, e.g., [48]). If all jobs are fully redundant, the system is equivalent to a single M/M/1 with arrival rate λ and service rate $\mu_1 + \mu_2$. This gives us the first term in our lower bound. The second term results from individual per-class bounds: the response time for class-i must be at least that in an M/M/1 with arrival rate λ_i and service rate $\sum_{s\in S_i}\mu_s$.

The upper bound is the overall system mean response time under FCFS-R (derived in Theorem 4.1); that it is an upper bound follows from the optimality of LRF (Theorem 4.2). □

Combining Lemma 4.4 and Theorem 4.3 gives us Theorem 4.4, which provides bounds on the class-R mean response time under LRF.

Theorem 4.4. *Under LRF in the $\mathbb{W}$ model, the mean response time for class-R,* $\mathbf{E}\left[T_R^{\text{LRF}}\right]$*, is bounded by*

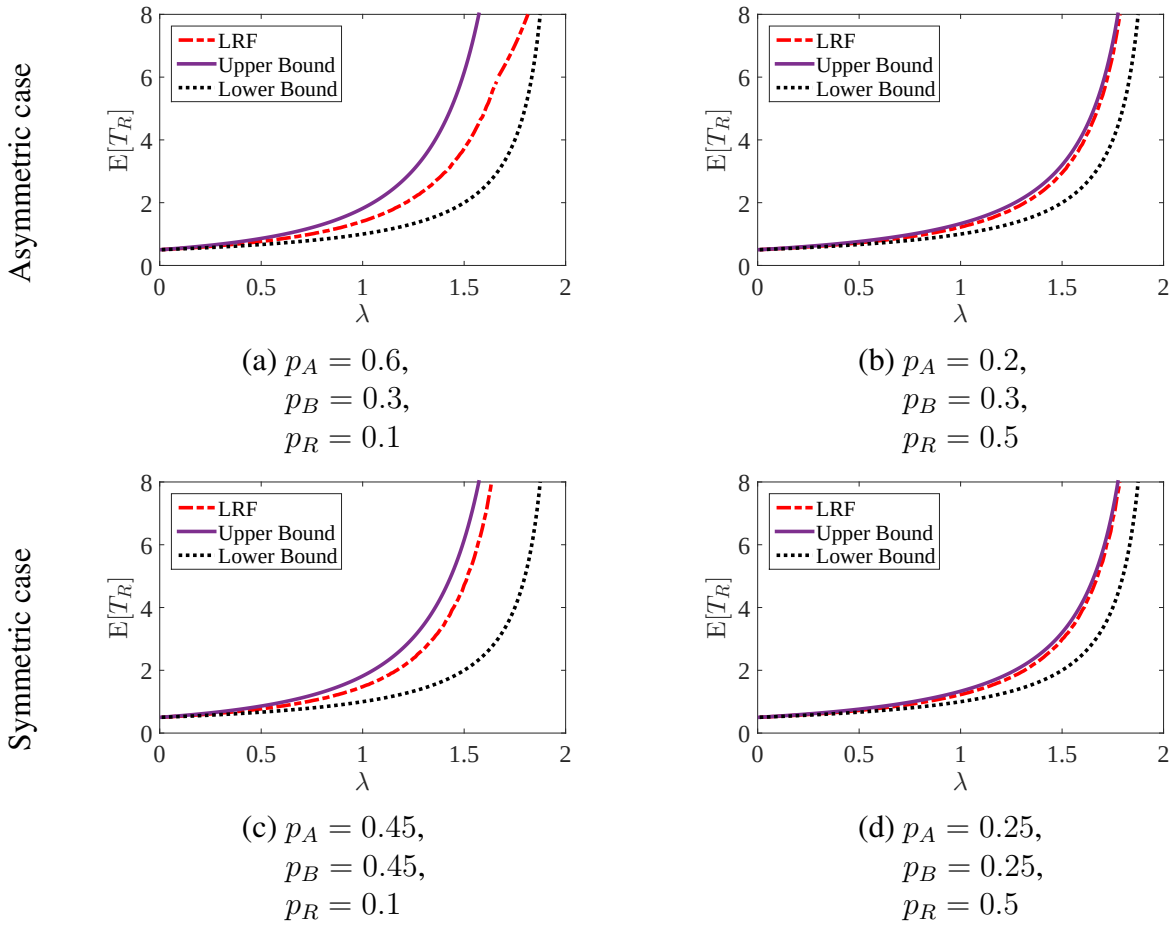
$$\frac{1}{\mu-\lambda} \leq \mathbf{E}\left[T_R^{\text{LRF}}\right] \leq \frac{1}{\lambda_R}\left(\frac{\lambda}{\mu-\lambda} - \frac{\lambda_A+\lambda_B}{\mu-\lambda_A-\lambda_B}\right).$$

Proof. We know that

$$\mathbf{E}\left[T^{\text{LRF}}\right] = \frac{\lambda_A}{\lambda}\mathbf{E}\left[T_A^{\text{LRF}}\right] + \frac{\lambda_B}{\lambda}\mathbf{E}\left[T_B^{\text{LRF}}\right] + \frac{\lambda_R}{\lambda}\mathbf{E}\left[T_R^{\text{LRF}}\right].$$

Combining this with the exact forms for $\mathbf{E}\left[T_A^{\text{LRF}}\right]$ and $\mathbf{E}\left[T_B^{\text{LRF}}\right]$ given in Lemma 4.4 and the upper bound for $\mathbf{E}\left[T^{\text{LRF}}\right]$ given in Theorem 4.3 immediately yields the upper bound on $\mathbf{E}\left[T_R^{\text{LRF}}\right]$. The lower bound is the response time for class-R jobs under FCFS-R, and it is clear that giving class-R jobs lowest priority can only increase their response time. □

Figure 4.7 shows the quality of the bounds on $\mathbf{E}\left[T_R\right]^{\text{LRF}}$ in both the asymmetric and symmetric cases described in Section 4.2. The upper bound becomes increasingly tight as a larger fraction of jobs become redundant.



(a) $p_A = 0.6$, $p_B = 0.3$, $p_R = 0.1$

(b) $p_A = 0.2$, $p_B = 0.3$, $p_R = 0.5$

(c) $p_A = 0.45$, $p_B = 0.45$, $p_R = 0.1$

(d) $p_A = 0.25$, $p_B = 0.25$, $p_R = 0.5$

Figure 4.7: Bounds on class-R mean response time under LRF in the asymmetric (top row) and symmetric (bottom row) cases. As the fraction of class-R jobs increases, the upper bound becomes increasingly tight.

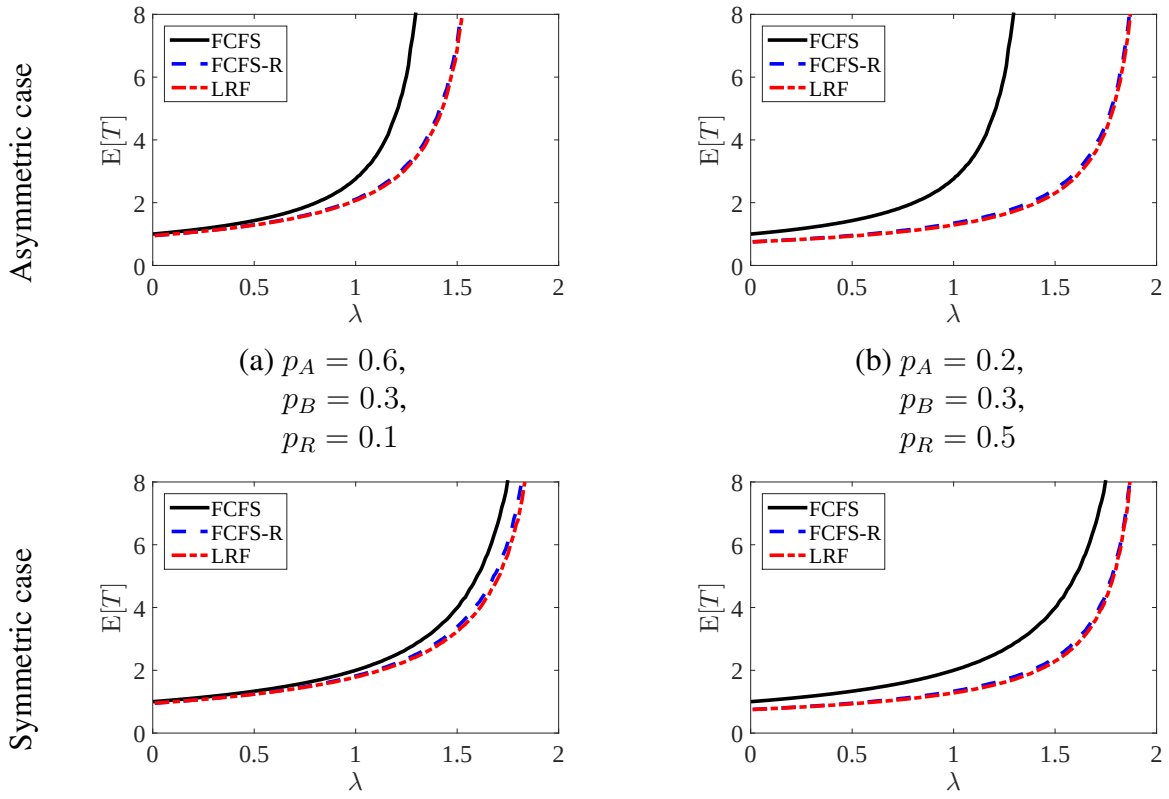
4.4.3 Performance

To evaluate the performance of LRF, we again turn to the $\mathbb{W}$ model. As before, we consider two system configurations: the asymmetric system where only class-A jobs become redundant, and the symmetric system where both class-A and class-B jobs become redundant.

Objective 1: Low Overall Mean Response Time

Given that LRF is optimal with respect to overall mean response time (Theorem 4.2), it is unsurprising that Figure 4.8(d) shows that LRF outperforms both non-redundant FCFS and FCFS-R. What is surprising is that the gap is so small between FCFS-R and LRF for all of the system configurations.

To understand why, we turn to the bounds on mean response time under LRF (Theorem 4.3). Figure 4.9 shows that for nearly all parameter settings, the upper bound (which is equivalent to FCFS-R) is extremely close to the lower bound. We see very little difference in overall mean



(a) $p_A = 0.6$, $p_B = 0.3$, $p_R = 0.1$

(b) $p_A = 0.2$, $p_B = 0.3$, $p_R = 0.5$

(c) $p_A = 0.45$, $p_B = 0.45$, $p_R = 0.1$

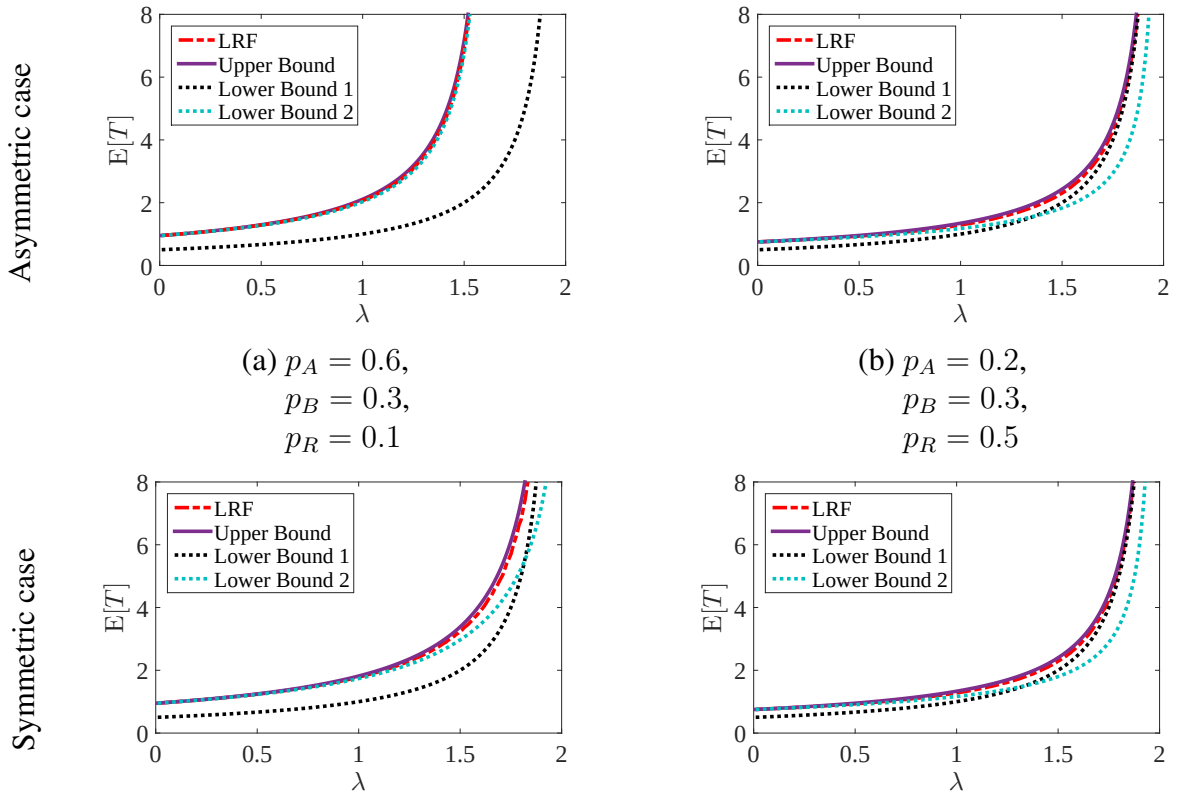
(d) $p_A = 0.25$, $p_B = 0.25$, $p_R = 0.5$

Figure 4.8: Comparing mean response time for the overall system under LRF (dot-dashed red line) to that under FCFS-R (dashed blue line) and non-redundant FCFS (solid black line) in the asymmetric (top row) and symmetric (bottom row) cases.

response time from using different redundancy-based scheduling policies because *redundancy itself* is extremely powerful. The benefit of simply allowing some jobs to be redundant, thereby preventing servers from idling, is much greater than any additional benefit that can be achieved via sophisticated scheduling policies.

Objective 2: Fairness Across Classes

Under LRF, in the asymmetric case all three classes of jobs are better off than they were under FCFS-R (see Figure 4.10). The class-A jobs see a much more pronounced improvement relative to the non-redundant system under LRF than under FCFS-R, particularly as the fraction of class-R jobs increases. Under FCFS-R the class-A jobs approach instability as λ approaches 2, whereas the class-A mean response time remains very low under LRF even as λ gets high because class-A jobs have preemptive priority over class-R jobs. Similarly, under LRF the class-B jobs are not hurt by the class-R jobs. Class-B jobs experience exactly the same mean response time under LRF



(a) $p_A = 0.6$, $p_B = 0.3$, $p_R = 0.1$

(b) $p_A = 0.2$, $p_B = 0.3$, $p_R = 0.5$

(c) $p_A = 0.45$, $p_B = 0.45$, $p_R = 0.1$

(d) $p_A = 0.25$, $p_B = 0.25$, $p_R = 0.5$

Figure 4.9: Upper and lower bounds for overall system mean response time under LRF in the asymmetric (top row) and symmetric (bottom row) cases. Here we show both terms in the lower bound given in Theorem 4.4.

as under non-redundant FCFS because under both policies, there are no class-R jobs competing with the class-B jobs, which have preemptive priority over class-R jobs.

The class-R jobs, too, benefit significantly from getting to wait in both queues. Even though the class-R jobs have lowest priority in both queues, they are likely to be preempted by fewer class-B jobs at server 2 than class-A jobs at server 1. When λ_B is sufficiently low, the class-R jobs are better off having low priority behind the class-B jobs than they would have been waiting in FCFS order among the class-A jobs. Note that, as was the case for FCFS-R, there is a kink in the curve for class-R jobs at $\lambda = 1.67$. Again, this is because at $\lambda = 1.67$ the class-A jobs become unstable, so the class-R jobs effectively only join the queue at server 2. Server 2 then behaves like a priority queue in which class-B jobs have preemptive priority over class-R jobs. Mean response time for the low-priority class-R jobs in this system is

$$\mathbf{E}[T_R] = \frac{1}{\mu - \lambda_B} + \frac{\lambda_B + \lambda_R}{(\mu - \lambda_B)(\mu - \lambda_B - \lambda_R)}.$$

Unfortunately, the symmetric case reveals LRF's inability to satisfy our fairness objective in

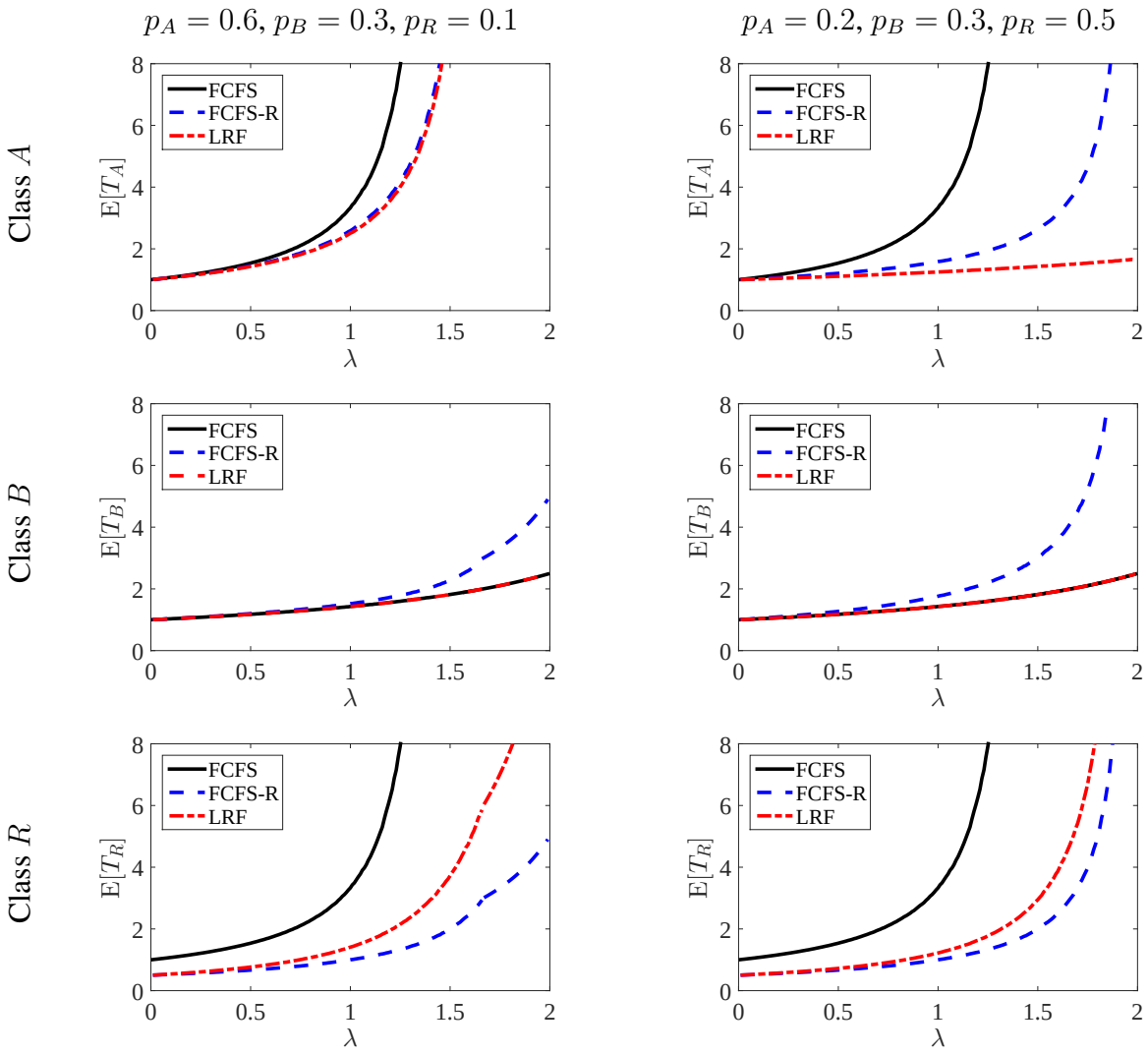


Figure 4.10: Comparing mean response time under LRF (dot-dashed red line) to that under FCFS-R (dashed blue line) and FCFS (solid black line) in the asymmetric case for class-A (top row), class-B (middle row), and class-R (bottom row) jobs. Note that there is a kink at $\lambda = 1.67$ for the class-R jobs when $p_A = 0.6$ because at this point the class-A jobs become unstable.

general (see Figure 4.11). LRF is able to achieve optimal overall mean response time and avoid hurting the non-redundant jobs by forcing the redundant jobs to have lowest preemptive priority. This means that redundant jobs only receive service when there are no non-redundant jobs at the server. When the system load is sufficiently high, both the class-A and class-B busy periods can be very long. Hence the redundant jobs can be starved and experience higher mean response times than they would have if they had remained class-A jobs under non-redundant FCFS. For example, in the symmetric case, when $p_R = 0.1$ and $\lambda = 1.6$, mean response time for the class-R jobs is 36% higher under LRF than under non-redundant FCFS. Thus like FCFS-R, LRF is unable to achieve our goal of fairness across job classes for all parameter values.

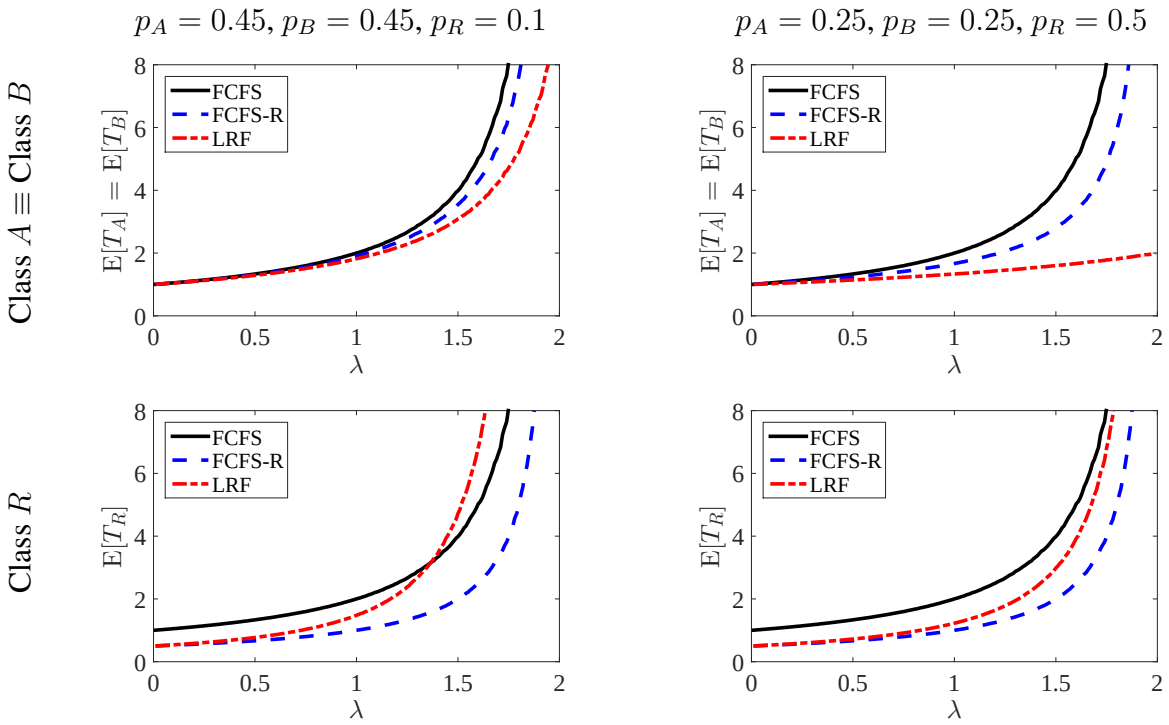


Figure 4.11: Comparing mean response time under LRF (dot-dashed red line) to that under FCFS-R (dashed blue line) and FCFS (solid black line) in the symmetric case for class-A and class-B (top row, classes A and B experience the same performance in the symmetric case), and class-R (bottom row).

4.5 Primaries First (PF)

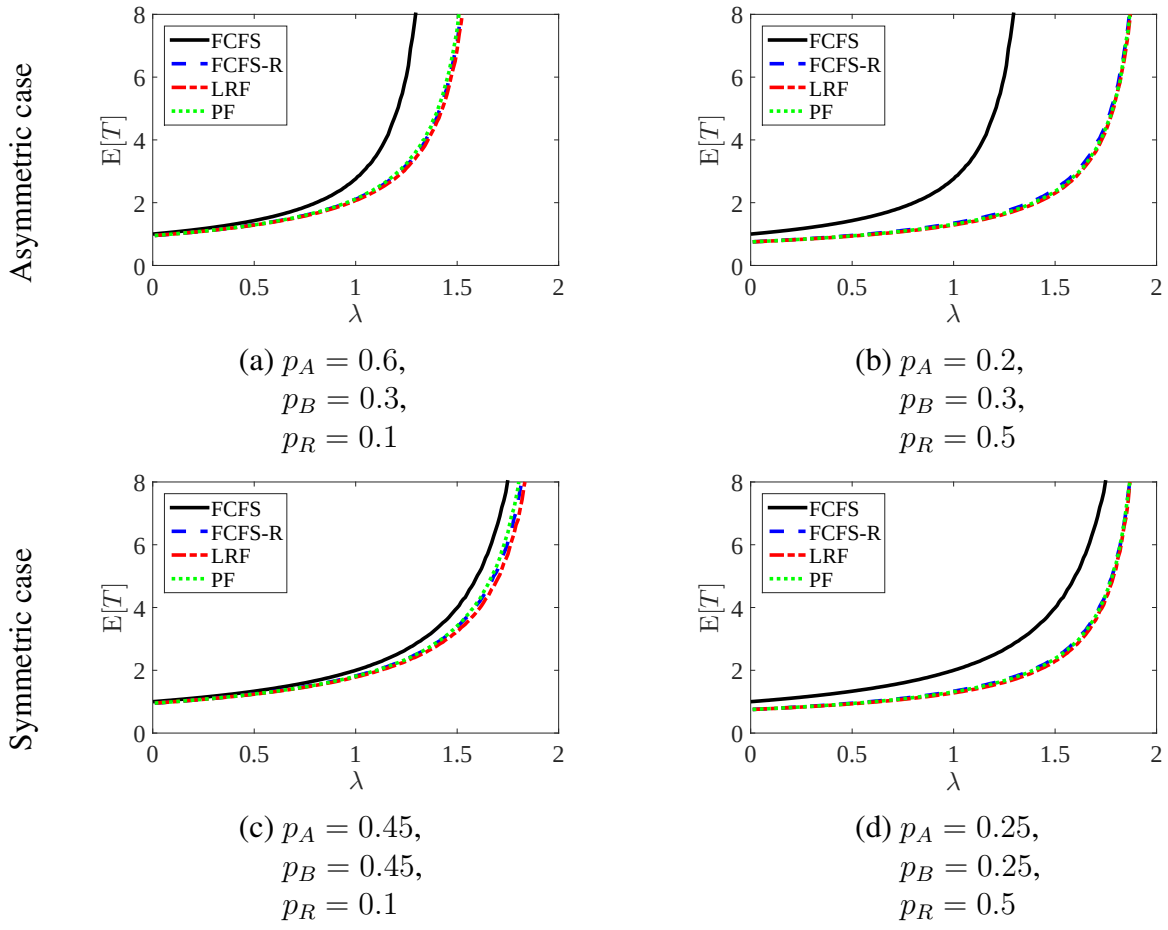
In Sections 4.3 and 4.4, we saw that although FCFS-R and LRF achieve near-optimal and optimal performance respectively with respect to overall mean response time, they both fail to achieve our second objective of maintaining fairness across classes. Specifically, FCFS-R may penalize the class-B jobs by forcing them to wait behind the class-R jobs that become redundant. LRF tries to compensate for this by prioritizing the non-redundant jobs. However, this can cause the class-R jobs to experience higher mean response times than if they were not redundant.

The Primaries First (PF) policy is designed to balance the strengths of both LRF and FCFS.

Definition 4.3. *Under PF, each arriving job designates a single* primary *copy, and any additional replicas are designated* secondary *copies. At each server, primaries have preemptive priority over secondaries, regardless of job class. Within the primaries (respectively, secondaries) jobs are served in FCFS order.*

When comparing PF with a system in which no jobs are redundant, we assume that jobs that become redundant designate their primary to be the copy that joins the queue at the server that the job would have gone to in the corresponding non-redundant system (for example, in the asymmetric $\mathbb{W}$ model, class-R jobs have their primary copy on server 1).

Intuitively, PF is successful at balancing the dual goals of achieving good overall performance



(a) $p_A = 0.6$, $p_B = 0.3$, $p_R = 0.1$

(b) $p_A = 0.2$, $p_B = 0.3$, $p_R = 0.5$

(c) $p_A = 0.45$, $p_B = 0.45$, $p_R = 0.1$

(d) $p_A = 0.25$, $p_B = 0.25$, $p_R = 0.5$

Figure 4.12: Comparing mean response time for the overall system under PF (dotted green line) to that under LRF (dot-dashed red line), FCFS-R (dashed blue line) and FCFS (solid black line) for both the asymmetric (top row) and symmetric (bottom row) cases.

and preserving fairness among classes because PF represents a compromise between LRF, which achieves optimal overall performance at the expense of the redundant jobs, and FCFS-R, which tries to be fair to the redundant jobs but consequently can hurt the non-redundant jobs and the overall system. Like LRF, PF only allows extra copies to run when there is spare server capacity. This prevents PF from hurting non-redundant jobs. Like FCFS, PF allows one copy of each redundant job to hold its place in the system-wide FCFS ordering. This prevents PF from hurting redundant jobs. In Theorem 4.5 we formalize this intuition by proving that under PF, every job class has response time at least as low as its response time under (non-redundant) FCFS. Unlike our results for FCFS-R and LRF, the proof of Theorem 4.5 is quite general: runtimes can be generally distributed, runtimes for the same job can be correlated across servers, the arrival process can be any exogenous process, and the redundancy structure need not be nested.

Theorem 4.5. *For all job classes i, $T^{(i)}_{\text{PF}} \leq_{st} T^{(i)}_{\text{FCFS}}$ for any general runtimes and any exogenous arrival process.*

Proof. On any sample path, if PF never schedules any secondaries, then all jobs complete at the

same time under PF as in the system with no redundancy. Suppose a secondary copy of job i is scheduled at some time under PF, say on server s. This means that server s is empty of primaries. If a different copy of job i completes on a different server before completing on server s, then job i and all other jobs continue to have the same completion times under PF as in the system with no redundancy. If job i completes on server s, then all of its copies disappear immediately from the system (including, in particular, its primary copy, which we will say disappeared from server s'). Then server s' now has one fewer job under PF than under non-redundant FCFS, while all other queues are the same length under both policies (ignoring all other jobs' secondaries). Hence the jobs at server s' will all complete earlier under PF than under non-redundant FCFS.

The result follows from repeating this argument each time PF schedules a secondary copy.□

4.5.1 Performance

We again consider the $\mathbb{W}$ model, shown in Figure 4.3(d) with redundancy and PF scheduling.

Objective 1: Low Overall Mean Response Time

PF successfully meets our first objective of achieving low overall mean response time (Figure 4.12): like FCFS-R and LRF, PF always outperforms non-redundant FCFS. One might think that since PF uses FCFS for primary copies, yet gives secondary copies lowest priority as in LRF, its mean response time should lie between that under FCFS-R and LRF. Unfortunately, this is not the case, particularly when load is high and the fraction of redundant jobs is relatively small (see Figure 4.12(c)). PF load balances by dispatching each job to a single server, and then adding extra low-priority secondary copies of the redundant jobs at other servers. When the arrival rate is high, it is fairly unlikely that the secondary copies get to run, so most redundant jobs experience the same performance as they did when they were class-A jobs in the original non-redundant system. In contrast, under FCFS-R a redundant job enters service at whichever server has less work in the queue when the job arrives, which reduces queueing time relative to the original system. As we increase the proportion of redundant jobs or decrease load, we increase the likelihood that the secondary copies actually enter service under PF, and then PF starts doing better than FCFS-R.

However, we observe empirically that mean response time under PF is quite similar to that under the optimal LRF policy. Again, this is because redundancy inherently is so powerful at keeping servers from idling that the particular scheduling policy does not play a significant role in further reducing mean response time. More specifically, when load is very low, a redundant job is likely to enter service on both servers under any scheduling policy, thereby receiving the same (low) response time regardless of the scheduling policy. When load is high, a redundant job is not likely to enter service on both servers. Instead, redundancy reduces overall system response time because of its ability to balance the system load.

Objective 2: Fairness Across Classes

PF successfully meets our second objective of preserving fairness across job classes in both the asymmetric case (Figure 4.13) and the symmetric case (Figure 4.14): as we saw in Theorem 4.5, all classes experience mean response times that are no worse than that under non-redundant FCFS.

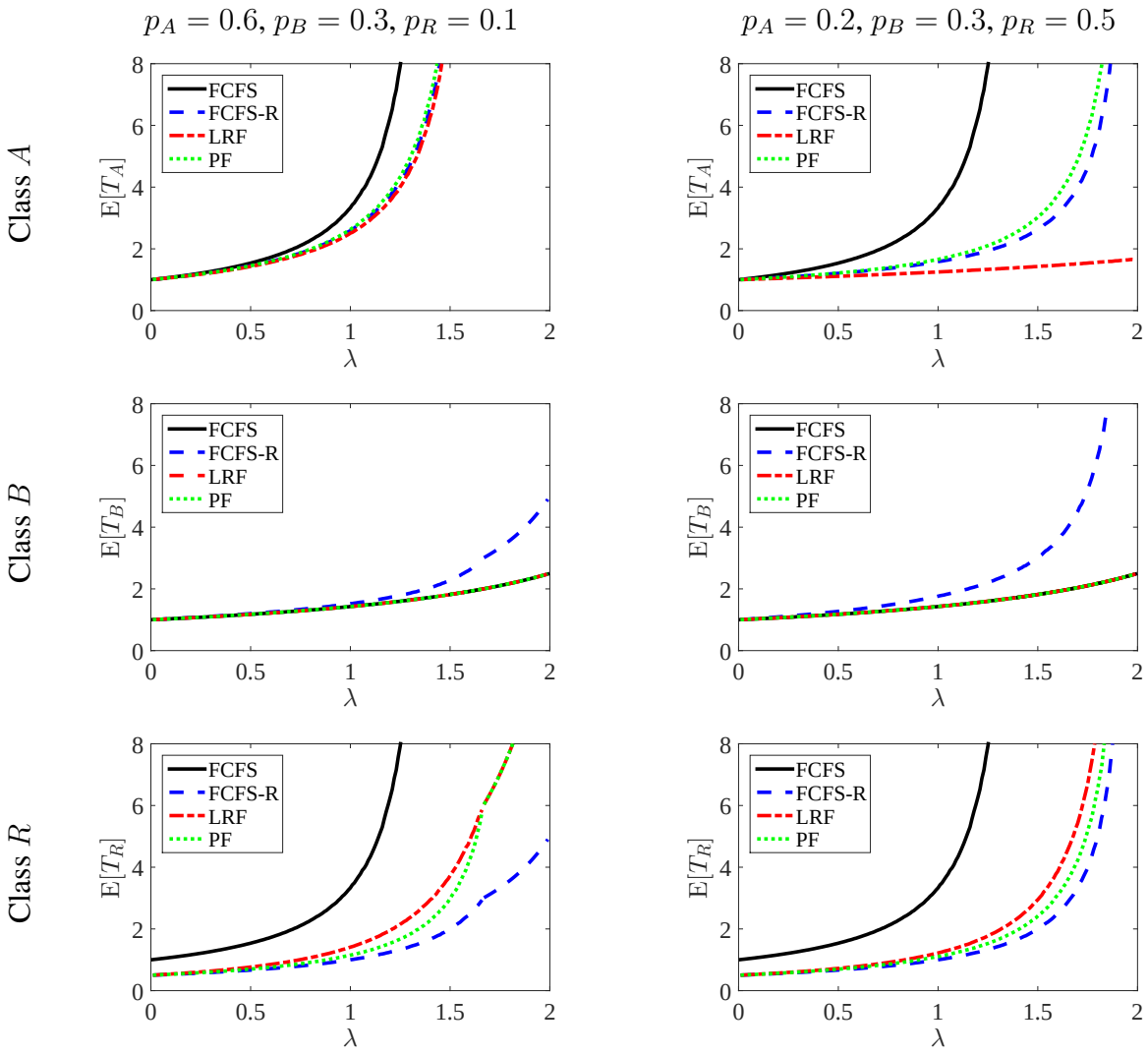


Figure 4.13: Comparing mean response time under PF (dotted green line) to that under LRF (dot-dashed red line), FCFS-R (dashed blue line) and FCFS (solid black line) in the asymmetric case for class-A (top row), class-B (middle row), and class-R (bottom row) jobs.

4.6 Discussion and Conclusion

Our goal in this chapter is to design scheduling policies that simultaneously satisfy the two objectives of (1) *efficiency*, i.e., low overall mean response time and (2) *fairness* across classes, i.e., each job class should perform at least as well under our redundancy-based scheduling policies as in a system in which no jobs are redundant. The second objective is motivated by systems such as organ transplant waitlists, in which a job's ability to become redundant correlates with the job's socioeconomic status; here it is important to ensure that we do not improve response time for the system as a whole at the expense of jobs that cannot become redundant. We restrict our attention to a particular system structure called a *nested* system, in which for every pair of job classes that have servers in common, one class's servers is a subset of the other's. Our first scheduling

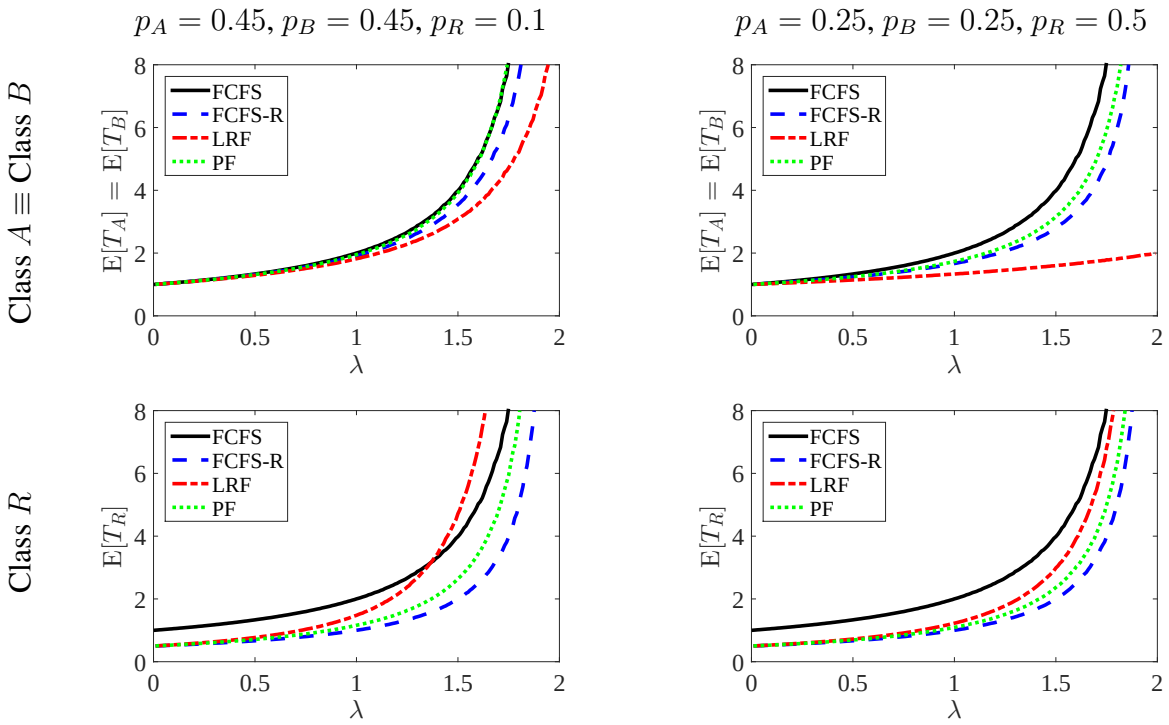


Figure 4.14: Comparing mean response time under PF (dotted green line) to that under LRF (dot-dashed red line), FCFS-R (dashed blue line) and FCFS (solid black line) in the symmetric case for class-A and class-B (top row, classes A and B experience the same performance in the symmetric case), and class-R (bottom row).

policy is First-Come, First-Served with redundancy (FCFS-R). We derive exact, closed-form expressions for the per-class distribution of response time, which we use to show that FCFS-R is highly effective at reducing the overall system mean response time. But it is possible to do even better: we introduce the Least Redundant First (LRF) policy, which we prove is optimal with respect to minimizing overall mean response time. LRF prevents servers from idling while there is still work in the system, thereby maximizing the system efficiency. But surprisingly, FCFS-R achieves nearly the optimal overall mean response time. This suggests that the benefits offered by redundancy itself are so substantial that there is limited room for the particular scheduling policy to greatly affect mean response time.

On the other hand, scheduling plays a crucial role in achieving our second objective of fairness across classes. Under FCFS-R non-redundant jobs can be hurt when some jobs become redundant due to the extra load added by the redundant jobs. LRF prevents the non-redundant jobs from being hurt by giving them full preemptive priority, but this can cause the redundant jobs to suffer. To overcome these weaknesses, we introduce the Primaries First (PF) policy, under which each job designates one copy as its primary and all other copies as secondaries; primaries have preemptive priority over secondaries on all servers. We prove that PF successfully maintains fairness across all job classes. Furthermore, PF accomplishes this while still providing low overall mean response time, thereby achieving our first goal as well.

Our analyses of FCFS-R and LRF rely on several assumptions, including that runtimes are exponentially distributed and independent across servers, and that the system has a nested structure. These assumptions may not hold in practice, and under different assumptions policies like LRF may not be optimal, and can even be poorly defined. For example, consider the $\mathbb{M}$ model introduced in Chapter 3. In the $\mathbb{M}$ model, there are two job classes, each with its own dedicated server, and a third server that is shared by the two classes. Under LRF, the shared server does not prioritize either of the classes because they have the same degree of redundancy. However, it may be possible to achieve lower overall mean response time by, for example, always serving the class with more jobs in the queue.

Fortunately, our fairness results under PF do *not* require any assumptions about nesting, exponential runtimes, or independent runtimes. PF continues to satisfy our desired fairness property when runtimes are generally distributed and correlated across servers, when the arrival process is non-Poisson, and when the system is not nested. This makes PF an excellent candidate for implementation in real systems. Indeed, recent work on implementing a redundancy-aware network stack uses a policy similar to PF, in which each job creates one primary copy and one or more "duplicates," and at each server primaries are given strict priority over duplicates [42]. The authors demonstrate that Duplicate-Aware Scheduling (their version of PF) can reduce mean response time by 75% relative to using no redundancy. While the experimental results presented in [42] do not allow for multiple classes of jobs, the results presented in this chapter suggest that PF will remain an effective policy even in more complex systems.

Chapter 5

Redundancy-d: Scaling to Large Systems

5.1 Introduction

Thus far, we have focused on systems in which each job has a class that specifies the servers to which it sends its copies, perhaps due to data locality constraints. However, such data locality constraints may not always exist. In some systems, it is possible that each job is capable of running on all of the machines. Here, we are interested in how much redundancy is needed in order to achieve a significant reduction in response time.

In the two-server $\mathbb{N}$ model that we studied in Chapter 3, we saw that allowing all jobs to be fully redundant leads to substantial response time improvements. This suggests that more redundancy is better. But real systems can consist of many more than two servers: data centers can have hundreds or thousands of servers. Do the messages we learn in small systems continue to hold as the number of servers scales up? How much redundancy is needed in order to achieve an appreciable response time improvement? How much faster does a job depart the system if it joins the queue at two servers rather than one? Is there further improvement from waiting in three queues rather than two? What about five? Ten?

We study these questions by introducing and analyzing a dispatching policy called Redundancy-$\mathbf{d}$ (see Figure 5.1). We consider a theoretical model consisting of k servers, each with its own queue. Under the Redundancy-$\mathbf{d}$ policy, each arriving job joins the queue at $\mathbf{d}$ of these servers, chosen uniformly at random. Here $\mathbf{d}$ is a constant that does not depend on k, and typically is small relative to k.

Our contributions in this chapter are as follows:

Exact analysis of mean response time under Redundancy-$\mathbf{d}$. We derive an exact closed-form expression for mean response time by modeling the system as a Markov chain. We use the state space introduced in Chapter 3, which tracks the location of all copies of all jobs in the system. The difficulty in finding mean response time for this system lies in aggregating the stationary probabilities for our detailed states, which is necessary to find the distribution of the number of jobs in the system. We present a novel state aggregation approach to accomplish this. We then use generating functions to derive mean response time under Redundancy-$\mathbf{d}$ (Section 5.3).

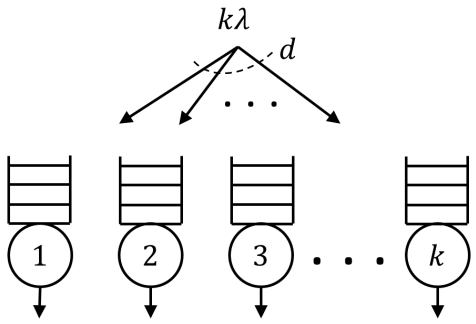


Figure 5.1: The system consists of k servers, each providing exponential runtimes with rate μ. Jobs arrive to the system as a Poisson process with rate $k\lambda$. Under the Redundancy-$\mathbf{d}$ policy, each job sends copies to $\mathbf{d}$ servers chosen uniformly at random. A job is considered complete as soon as the first of its copies completes service.

Analysis of the full distribution of response time. Unfortunately, our Markov chain approach does not allow us to derive the response time distribution under Redundancy-$\mathbf{d}$. Instead, we introduce a different approach. We consider the system in the limit as the number of servers k approaches infinity. We make the further assumption that in this asymptotic regime, the work in different queues is independent; such independence has been shown to hold under related policies, for example, Join-the-Shortest-Queue dispatching [55, 72, 77]. Under these assumptions, we formulate a system of differential equations that describes the evolution of the system. Coming up with the right differential equations is not straightforward because the system has a very complicated departure process: each service completion results in the removal of $\mathbf{d}$ copies from different servers. We use our differential equations to derive an asymptotically exact expression for the distribution of response time (Section 5.4).

Understanding the power of $\mathbf{d}$ choices for redundancy. We use our analytical results to investigate the effect of $\mathbf{d}$ on response time under Redundancy-$\mathbf{d}$ (Section 6.5). This problem is reminiscent of the power-of-$\mathbf{d}$ results which exist in the literature for Join-the-Shortest-Queue dispatching (with no redundancy) [55, 72]. While the trends we observe are what one might expect, our exact analysis allows us to quantify the magnitude of these trends for the first time. As $\mathbf{d}$ increases, mean response time decreases, and the biggest improvement comes from adding just a single extra copy of each job ($\mathbf{d} = 2$). For example, at high load, setting $\mathbf{d} = 2$ reduces mean response time by a factor of 6. Our results support the empirical observation that the improvement is even more pronounced in the tail: at high load, setting $\mathbf{d} = 2$ reduces the 95th percentile of response time by a factor of 8. We further show that when $\mathbf{d}$ is high, mean response time drops in proportion to $\frac{1}{\mathbf{d}}$. Leveraging the fact that the largest benefit comes from having a single extra replica ($\mathbf{d} = 2$), we introduce the idea of "fractional $\mathbf{d}$" redundancy, in which each job makes on average between one and two copies. We find that even with fewer than two copies on average, redundancy still provides a significant response time improvement.

Understanding response time when runtimes are generally distributed. The closed-form results we derive via our Markov chain analysis and asymptotic analysis assume that runtimes are exponentially distributed. However, some relaxation of our model's assumptions are possible *numerically*. In particular, the analytical approach we present in Section 5.4 applies even when runtimes are generally distributed. We develop a numerical extension to our analytical approach, which allows us to study the effect of $\mathbf{d}$ on response time under non-exponential runtime distributions (Section 5.5.2).

The remainder of this chapter is organized as follows. In Section 5.2 we formalize our theoretical model and the Redundancy-$\mathbf{d}$ dispatching policy. Sections 5.3 and 5.4 present our analytical results for the mean and distribution of response time respectively. In Section 5.5 we use our analysis to investigate the impact of the choice of $\mathbf{d}$ on response time. Finally, in Section 5.6 we conclude. This chapter is based on joint work with Mor Harchol-Balter, Alan Scheller-Wolf, Mark Velednitsky, and Samuel Zbarsky and appears in the following papers:

- Kristen Gardner, Samuel Zbarsky, Mark Velednitsky, Mor Harchol-Balter, Alan Scheller-Wolf. "Understanding Response Time in the Redundancy-$\mathbf{d}$ System." *Workshop on Mathematical Performance Modeling and Analysis* (MAMA), June 2016. [36]
- Kristen Gardner, Samuel Zbarsky, Mor Harchol-Balter, Alan Scheller-Wolf. "The Power of $\mathbf{d}$ Choices for Redundancy." Poster, *SIGMETRICS/Performance*, June 2016. [35]
- Kristen Gardner, Mor Harchol-Balter, Alan Scheller-Wolf, Mark Velednitsky, and Samuel Zbarsky. "Redundancy-$\mathbf{d}$: The Power of $\mathbf{d}$ Choices for Redundancy." *Operations Research* 2017. doi: 10.1287/opre.2016.1582. [32]

5.2 Model

We consider a k-server system, shown in Figure 5.1. Jobs arrive to the system as a Poisson process with rate $k\lambda$. Each server provides exponential runtimes with rate μ and works on the jobs in its queue in first-come first-served order. In accordance with the IR model, a job's runtimes are i.i.d. across servers. A job may be in service at multiple servers at the same time, in which case it experiences the minimum runtime among all servers at which it is in service. A job is considered complete as soon as its first copy completes, at which time all remaining copies disappear from the system regardless of whether they are in service or in the queue.

We define the system load to be $\rho = \frac{\lambda}{\mu}$. This is the total arrival rate to the system ($k\lambda$) divided by the maximum service rate of the system ($k\mu$). The system is stable as long as $\rho < 1$ (see Section 5.3).

Upon arrival, each job is dispatched to servers according to the Redundancy-$\mathbf{d}$ policy.

Definition 5.1. *Under Redundancy-$\mathbf{d}$, upon arrival each job sends a copy of itself to $\mathbf{d}$ servers chosen uniformly at random without replacement.*

Our goal is to analyze response time, T, under Redundancy-$\mathbf{d}$ as a function of the arrival rate λ, the service rate μ, the number of servers k, and the degree of redundancy $\mathbf{d}$, to help us understand the role redundancy can play in reducing response time.

5.3 Markov Chain Analysis

The purpose of this section is to prove Theorem 5.1, which gives a simple expression for the mean response time under Redundancy-$\mathbf{d}$ in a system with k servers.

Theorem 5.1. *The mean response time under Redundancy-$\mathbf{d}$ in a system with k servers is*

$$\mathbf{E}[T] = \sum_{i=\mathbf{d}}^{k} \frac{1}{k\mu \frac{\binom{k-1}{\mathbf{d}-1}}{\binom{i-1}{\mathbf{d}-1}} - k\lambda}. \tag{5.1}$$

The remainder of this section is devoted to proving the above result.

5.3.1 Alternative System View: Class-Based Redundancy

In Chapters 3 and 4, we considered constrained redundancy systems in which each job has a class that determines the subset of servers on which the job can run. Our current system model assumes a flexible structure in which each job is capable of running on any server, but chooses to replicate itself at only $\mathbf{d}$ randomly selected servers. We can view the flexible system as being a class-based system—allowing us to leverage the analysis developed in Chapter 3—by defining a job's *class* as the set of $\mathbf{d}$ particular servers to which the job sends copies. There are $\binom{k}{\mathbf{d}}$ possible classes; all classes are equally likely since each job chooses its servers uniformly at random. Let λ_{class} denote the arrival rate of any class, where

$$\lambda_{\text{class}} = \frac{k\lambda}{\binom{k}{\mathbf{d}}}.$$

As in Chapter 3, we model the system as a Markov chain, where the system state is a list of all jobs in the system in the order in which they arrived and we track the class of each job. We write the state as $(c_n,\ c_{n-1}, \ldots,\ c_1)$, denoting that there are n jobs in the system, c_1 is the class of the oldest job in the system (the first of the n jobs to arrive), and c_i is the class of the ith job in the system in order of arrival. Since the state tracks all jobs in the system in order of arrival, the state information implicitly tracks which jobs are in service at which servers. For example, the oldest job in the system, which has class c_1, must be in service at all $\mathbf{d}$ of its servers.

Once we have defined the notion of a job class and written the system state as defined above, we obtain the following result for the stationary distribution of the state space:

Theorem 5.2. *Under Redundancy-$\mathbf{d}$, the stationary probability of being in state $(c_n,\ c_{n-1}, \ldots,\ c_1)$ is*

$$\pi_{(c_n, c_{n-1}, \ldots, c_1)} = \mathcal{C} \prod_{j=1}^{n} \frac{\lambda_{\text{class}}}{|S_j|\mu}, \tag{5.2}$$

where S_j is the set of all servers working on jobs $1, \ldots, j$ ($|S_j|$ is the number of servers in this set) and

$$\mathcal{C} = \prod_{i=d}^{k} \left(1 - \frac{\binom{i-1}{d-1}\lambda}{\binom{k-1}{d-1}\mu}\right)$$

is a normalizing constant representing the probability that all servers are idle.

Proof. The general form of the stationary probabilities given in (5.2) is an immediate consequence of Theorem 3.1 in Chapter 3. However the normalizing constant $\mathcal{C}$ is not derived there, and this is the heart of our proof.

Let π_n be the stationary probability that there are n jobs in the system (note that π_n results from aggregating states $(c_n, \cdots, c_1)$ over all possible classes $c_1, \ldots, c_n$). If we number our servers as 1 through k, then combining the normalizing equation

$$\sum_{n=0}^{\infty} \pi_n = 1,$$

with the form given in (5.2), we see that $\Pr\{\text{all servers are idle}\} = \mathcal{C}$. We then derive $\mathcal{C}$ as follows:

$$\begin{aligned}
\mathcal{C} &= \mathbf{P}\left\{\text{all servers are idle}\right\} \\
&= \mathbf{P}\left\{\text{server } k \text{ idle}\right\} \cdot \mathbf{P}\left\{\text{server } k-1 \text{ idle} \mid \text{server } k \text{ idle}\right\} \\
&\qquad \cdots \mathbf{P}\left\{\text{server 1 idle} \mid \text{servers } 2, \ldots, k \text{ idle}\right\} \\
&= \mathbf{P}\left\{\text{server } k \text{ idle}\right\} \cdot \mathbf{P}\left\{\text{server } k-1 \text{ idle} \mid \text{server } k \text{ idle}\right\} \\
&\qquad \cdots \mathbf{P}\left\{\text{server } \mathbf{d} \text{ idle} \mid \text{servers } \mathbf{d}+1, \ldots, k \text{ idle}\right\}, \qquad (5.3)
\end{aligned}$$

where the last line is due to the fact that if fewer than $\mathbf{d}$ servers are busy then no jobs can be present.

First we will find $\mathbf{P}\left\{\text{server } k \text{ idle}\right\}$. Since the system is symmetric in permuting the servers, each server, including server k, has probability $1-\rho$ of being idle.

To find $\mathbf{P}\left\{\text{server } k-i \text{ idle} \mid \text{servers } k-i+1, \ldots, k \text{ idle}\right\}$, we will consider a sequence of systems of smaller and smaller size. We begin by rewriting the stationary probability given in (5.2) conditioning on servers $k-i+1, \ldots, k$ being idle:

$$\begin{aligned}
&\mathbf{P}\left\{\text{system in state } c_n, \ldots, c_1 \mid \text{servers } k-i+1, \ldots, k \text{ idle}\right\} \\
&\qquad = \begin{cases} 0 & \text{if } s \in S_n \text{ for some } k-i+1 \le s \le k \\ \dfrac{\mathcal{C}}{P_i} \cdot \displaystyle\prod_{j=1}^{n} \frac{\lambda_{\text{class}}}{|S_j|\mu} & \text{otherwise,} \end{cases} \qquad (5.4)
\end{aligned}$$

where $P_i = \mathbf{P}\left\{\text{servers } k-i+1, \ldots, k \text{ are idle}\right\}$. Now consider a system that consists of only servers $1, \ldots, k-i$ and only the $\binom{k-i}{\mathbf{d}}$ classes of jobs that go to servers $1, \ldots, k-i$ in the original system. The stationary probability of being in state $c_n, \ldots, c_1$ in this system is exactly that given in (5.4). That is, the stationary probability of any state in our original system given that servers $k-i+1, \ldots, k$ are idle is the same as the stationary probability of the same state in the $(k-i)$-server system. In particular, the time-average fraction of time any given server is busy is the same in the two systems. In our $(k-i)$-server system, the total arrival rate is $\binom{k-i}{\mathbf{d}} \cdot k\lambda$ and the total service rate is $(k-i)\mu$. Hence the time-average fraction of time a given server is busy in the $(k-i)$-server system is

$$\rho_{k-i} = \frac{\frac{\binom{k-i}{\mathbf{d}}}{\binom{k}{\mathbf{d}}} \cdot k\lambda}{(k-i)\mu} = \frac{\binom{k-i}{\mathbf{d}}}{\binom{k}{\mathbf{d}}} \cdot \frac{\lambda}{\mu} \cdot \frac{k}{k-i}.$$

The probability that any server, and in particular server $k-i$, is idle in this system—and hence in the original k-server system given that servers $k-i+1, \ldots, k$ are idle—is $1-\rho_{k-i}$.

Going back to (5.3), we have

$$\begin{aligned}
\mathcal{C} &= \mathbf{P}\left\{\text{server } k \text{ idle}\right\} \cdot \mathbf{P}\left\{\text{server } k-1 \text{ idle} \mid \text{server } k \text{ idle}\right\} \\
&\qquad \cdots \mathbf{P}\left\{\text{server } \mathbf{d} \text{ idle} \mid \text{servers } \mathbf{d}+1, \ldots, k \text{ idle}\right\} \\
&= \prod_{i=0}^{k-\mathbf{d}} (1-\rho_{k-i}) \\
&= \prod_{i=0}^{k-\mathbf{d}} \left(1 - \frac{\binom{k-i}{\mathbf{d}}}{\binom{k}{\mathbf{d}}} \cdot \frac{\lambda}{\mu} \cdot \frac{k}{k-i}\right) \\
&= \prod_{i=0}^{k-\mathbf{d}} \left(1 - \frac{\binom{k-i-1}{\mathbf{d}-1}}{\binom{k-1}{\mathbf{d}-1}} \cdot \frac{\lambda}{\mu}\right) \\
&= \prod_{i=\mathbf{d}}^{k} \left(1 - \frac{\binom{i-1}{\mathbf{d}-1}\lambda}{\binom{k-1}{\mathbf{d}-1}\mu}\right).
\end{aligned}$$

□

As noted in Chapter 3, the form of the stationary probabilities given in (5.2) is quite unusual. Although it looks like a product form, it cannot be written as a product of per-class terms or as a product of per-server terms. Example 5.1 illustrates the particular form of the limiting probabilities under Redundancy-$\mathbf{d}$.

Example 5.1. *Consider a system with $k=4$ servers and $\mathbf{d}=2$ copies per job. Suppose that there are currently four jobs in the system. The first job has class A and its copies are at servers 1 and 2. The second job has class B and its copies are at servers 2 and 4. The third job has class C and its copies are at servers 3 and 4. The fourth job has class A (the same as the first job) and its copies are at servers 1 and 2. Then the state of the system is (A, C, B, A) and the stationary probability of being in this state is*

$$\pi_{(A,C,B,A)} = \left(\frac{\lambda_{\text{class}}}{4\mu}\right)\left(\frac{\lambda_{\text{class}}}{4\mu}\right)\left(\frac{\lambda_{\text{class}}}{3\mu}\right)\left(\frac{\lambda_{\text{class}}}{2\mu}\right),$$

where the rightmost term is the contribution of the first A arrival and the leftmost term is the contribution of the last A arrival, and where $\lambda_{\text{class}} = \frac{2}{3}\lambda$. Note that the stationary probability is not simply a product of per-class terms or of per-server terms since the denominators depend on the order of all jobs in the system.

Theorem 5.3. *Under Redundancy-$\mathbf{d}$, the system is stable when $\rho = \frac{\lambda}{\mu} < 1$.*

Proof. The proof of Theorem 6.2 relies on the state aggregation approach we present in Section 5.3.2, so we defer the proof to the end of the section. □

5.3.2 State Aggregation

One might think that $\mathbf{E}[T]$ follows immediately from the limiting distribution on the state space given in Theorem 5.2. Unfortunately, knowing the stationary distribution on the state space does

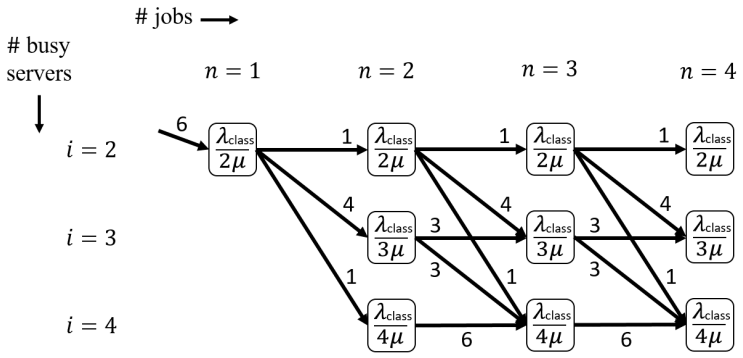


Figure 5.2: Aggregating states in the $k = 4$, $\mathbf{d} = 2$ system. Horizontally we track the number of jobs in the system and vertically we track the number of busy servers. The value at node (i, n) gives the contribution of the job at position n to the limiting probability. An edge from node (i, n) to node $(j, n + 1)$ has weight equal to the number of classes the job in position $n + 1$ could be in order for there to be j servers busy working on the first $n + 1$ jobs when there were i servers busy working on the first n jobs.

not immediately yield results for mean number in system and mean response time. This is because to find mean response time, we must first find

$$\pi_n = \mathbf{P}\left\{n \text{ jobs in system}\right\}.$$

To do this, we need to sum π values over all $\binom{k}{\mathbf{d}}$ possible classes for each queue position j, $1 \leq j \leq n$. This is not straightforward because the denominators in the stationary probabilities depend on the order of all jobs in the system: $\pi_{(c_n,\ldots,c_1)}$ depends on the particular choices of $c_1, \ldots, c_n$.

The key observation that helps us aggregate states is that we only need to track the denominator contributed to the stationary probability by the job in each queue position j—*not* the specific class c_j of the job. This is equivalent to tracking the number of servers that are busy working on the first j jobs in the queue. We leverage this observation by collapsing our state space so that instead of $\binom{k}{\mathbf{d}}$ possible classes for each position in the queue, we now have at most $k - \mathbf{d}$ possible denominators. In addition, not all denominators are possible for each position; for example, position 1 must contribute denominator $\mathbf{d}\mu$, and if position j contributes denominator $i\mu$ then position $j + 1$ must contribute denominator $\geq i\mu$.

We define $P(i, n)$ to be the stationary probability that there are n jobs in the system and i busy servers, disregarding a normalization constant. To find π_n, we need to compute $P(i, n)$ for all $\mathbf{d} \leq i \leq k$.

At a high level, our approach takes the following steps:

1. Write recurrences for $P(i, n)$, the (unnormalized) stationary probability that there are i servers busy and n jobs in the system (Section 5.3.2).
2. Define a generating function for our recurrences and use this generating function to find $\mathbf{E}[N]$ and $\mathbf{E}[T]$ (Section 5.3.2).

Throughout the remainder of this section we refer to Figure 5.2, which provides a running example of our approach in the case where $k = 4$ and $\mathbf{d} = 2$.

Formulating Recurrences $P(i, n)$

Our goal in this section is to write recurrences for $P(i, n)$, the (unnormalized) stationary probability that the system has i busy servers and n jobs in the system.

Theorem 5.4. *For $n > 1$, $P(i, n)$, the unnormalized stationary probability that there are n jobs in the system and i busy servers, satisfies*

$$P(i,n) = \frac{\lambda_{\text{class}}}{\mu i} \cdot \sum_{y=0}^{\mathbf{d}} \binom{i-y}{\mathbf{d}-y} \cdot \binom{k-(i-y)}{y} \cdot P(i-y, n-1). \tag{5.5}$$

For $n = 1$, we have the initial conditions

$$P(i,1) = \begin{cases} \binom{k}{\mathbf{d}} \frac{\lambda_{\text{class}}}{\mu \mathbf{d}}, & i = \mathbf{d} \\ 0, & \mathbf{d} < i \leq k. \end{cases} \tag{5.6}$$

For $n = 0$, we have the initial conditions

$$P(i,0) = \begin{cases} 1, & i = 0 \\ 0, & i > 0 \end{cases} \tag{5.7}$$

Proof. We first consider the case $n = 1$. Here there is a single job in the system, so the system state is (c_1). Regardless of the specific class c_1, there are always $\mathbf{d}$ servers busy working on this job and the arrival rate of class c_1 is always λ_{class}, so from Theorem 5.2 the stationary probability of this state is $\pi_{(c_1)} = \mathcal{C} \cdot \frac{\lambda_{\text{class}}}{\mu \mathbf{d}}$. The job could belong to any class, so there are $\binom{k}{\mathbf{d}}$ states in which $n = 1$. Hence the total probability that there is one job in the system is

$$\pi_1 = \binom{k}{\mathbf{d}} \cdot \mathcal{C} \cdot \frac{\lambda_{\text{class}}}{\mu \mathbf{d}} = \mathcal{C} \cdot P(\mathbf{d}, 1).$$

For any value of $i \neq \mathbf{d}$ it is not possible to have i servers working on only $n = 1$ job, so $P(i, 1) = 0$. This gives the initial conditions in (5.6) (recall that we omit the normalizing constant).

When $n = 2$, the system state is $(c_2,\ c_1)$. The number of busy servers can range from $\mathbf{d}$ (if both jobs are of the same class and therefore share all $\mathbf{d}$ servers) to $2\mathbf{d}$ (if the two jobs do not share any servers). Hence we need to find expressions for $P(\mathbf{d}, 2),\ P(\mathbf{d}+1, 2), \ldots,\ P(2\mathbf{d}, 2)$.

To find $P(\mathbf{d}, 2)$, observe that the first job in the system, which has class c_1, contributes a factor of $\frac{\lambda_{\text{class}}}{\mu \mathbf{d}}$ to the stationary probability, and there are $\binom{k}{\mathbf{d}}$ ways of choosing class c_1, so its total contribution to $P(\mathbf{d}, 2)$ is $\binom{k}{\mathbf{d}} \frac{\lambda_{\text{class}}}{\mu \mathbf{d}}$. This is exactly $P(\mathbf{d}, 1)$ (up to the normalizing constant). The second job also contributes a factor of $\frac{\lambda_{\text{class}}}{\mu \mathbf{d}}$ and there is only one way of choosing the second job's class so that it shares all $\mathbf{d}$ servers with the first job, namely $c_2 = c_1$. Hence we find $P(\mathbf{d}, 2) = \frac{\lambda_{\text{class}}}{\mu \mathbf{d}} \cdot P(\mathbf{d}, 1)$. For example, when $k = 4$ and $\mathbf{d} = 2$ (see Figure 5.2), we find that $P(2, 2) = \frac{\lambda_{\text{class}}}{2\mu} \cdot P(2, 1)$.

Similarly, to find $P(\mathbf{d}+1, 2)$, we first consider the contribution of the first job to the stationary probability. Again, the first job contributes a factor of

$$\binom{k}{\mathbf{d}} \frac{\lambda_{\text{class}}}{\mu \mathbf{d}} = P(\mathbf{d}, 1) \tag{5.8}$$

since class c_1 has arrival rate λ_{class}, $\mathbf{d}$ servers are busy working on this job, and there are $\binom{k}{\mathbf{d}}$ possible choices for the specific class c_1. The second job contributes a factor of

$$\frac{\lambda_{\text{class}}}{\mu(\mathbf{d}+1)} \cdot \begin{pmatrix} \#\text{ ways to choose 2nd job} \\ \text{so it shares } \mathbf{d}-1 \text{ servers} \\ \text{with first job} \end{pmatrix} = \frac{\lambda_{\text{class}}}{\mu(\mathbf{d}+1)} \cdot \binom{\mathbf{d}}{\mathbf{d}-1} \cdot \binom{k-\mathbf{d}}{1}, \tag{5.9}$$

where the $\binom{\mathbf{d}}{\mathbf{d}-1}$ term gives the number of ways that the second job can choose $\mathbf{d}-1$ servers in common with the first job and the $\binom{k-\mathbf{d}}{1}$ term gives the number of ways for the second job to choose one server that is different from all $\mathbf{d}$ of the first job's servers. Combining (5.8) and (5.9),

$$P(\mathbf{d}+1, 2) = \frac{\lambda_{\text{class}}}{\mu(\mathbf{d}+1)} \cdot \binom{\mathbf{d}}{\mathbf{d}-1} \cdot \binom{k-\mathbf{d}}{1} \cdot P(\mathbf{d}, 1).$$

In the case where $k = 4$ and $\mathbf{d} = 2$, the graph in Figure 5.2 tells us that there are four ways in which the second job can choose servers such that it shares one server with the first job (that is, $\binom{\mathbf{d}}{\mathbf{d}-1} \cdot \binom{k-\mathbf{d}}{1} = \binom{2}{1} \cdot \binom{4-2}{1} = 4$). Thus the recurrence for $P(3, 2)$ in the $k = 4$, $\mathbf{d} = 2$ system is

$$P(3, 2) = \frac{\lambda_{\text{class}}}{3\mu} \cdot 4 \cdot P(2, 1).$$

In general, when writing a recurrence for $P(i, n)$ we consider all possible values of y, the number of new servers busy working on the nth job. Equivalently, $i - y$ servers must be busy working on the first $n-1$ servers. Except in edge cases where there are already at least $k - \mathbf{d} + 1$ servers working on the first $n-1$ jobs, the value of y can range from 0 to $\mathbf{d}$.

Given that $i - y$ servers are busy working on the first $n-1$ jobs, the contribution of the first $n-1$ jobs is $P(i-y, n-1)$. The nth job contributes

$$\frac{\lambda_{\text{class}}}{\mu i} \cdot \begin{pmatrix} \#\text{ ways to choose } n\text{th job} \\ \text{so it shares } \mathbf{d}-y \text{ servers} \\ \text{with first } n-1 \text{ jobs} \end{pmatrix} = \frac{\lambda_{\text{class}}}{\mu i} \cdot \binom{i-y}{\mathbf{d}-y} \cdot \binom{k-(i-y)}{y}$$

to $P(i, n)$, where the term $\binom{i-y}{\mathbf{d}-y}$ gives the number of ways to choose the $\mathbf{d}-y$ servers that the nth job shares with the first $n-1$ jobs from among the $i-y$ servers busy working on the first $n-1$ jobs, and the term $\binom{k-(i-y)}{y}$ gives the number of ways to choose y new servers from the remaining $k-(i-y)$ servers.

Finally, we condition on the number of new servers y to obtain the general form of $P(i, n)$ given in (5.5). □

Finding Mean Response Time

Now that we have a form for $P(i,n)$, we can imagine finding the mean number in system $\mathbf{E}[N]$ by summing over all possible numbers of busy servers and all possible numbers of jobs in the system:

$$\mathbf{E}[N] = \sum_{i=\mathbf{d}}^{k} \sum_{n=1}^{\infty} n \cdot P(i,n) \cdot \mathcal{C},$$

where $\mathcal{C}$ is our normalizing constant. Unfortunately, computing these sums would require having an explicit form for $P(i,n)$, which is difficult to compute. Instead, we will find $\mathbf{E}[N]$ using generating functions.

We begin by rewriting our recurrences $P(i,n)$ in a form that eliminates the dependency on k. Starting with the expression given in (5.5), we substitute $\lambda_{\text{class}} = \frac{k\lambda}{\binom{k}{\mathbf{d}}}$ and rearrange the combinatorial terms to obtain:

$$P(i,n) = \frac{k\lambda}{\mu i} \sum_{y=0}^{\mathbf{d}} \frac{\binom{\mathbf{d}}{y}\binom{k-\mathbf{d}}{i-\mathbf{d}}}{\binom{k}{i-y}} \cdot P(i-y, n-1).$$

Our next step is to eliminate the $\binom{k}{i-y}$ term in the denominator. Let $Q(i,n) = \frac{1}{\binom{k}{i}} \cdot P(i,n)$. We then have

$$\begin{aligned}
Q(i,n) \cdot \binom{k}{i} &= P(i,n) \\
&= \frac{k\lambda}{\mu i} \sum_{y=0}^{\mathbf{d}} \frac{\binom{\mathbf{d}}{y}\binom{k-\mathbf{d}}{i-\mathbf{d}}}{\binom{k}{i-y}} \cdot P(i-y, n-1) \\
&= \frac{k\lambda}{\mu i} \sum_{y=0}^{\mathbf{d}} \frac{\binom{\mathbf{d}}{y}\binom{k-\mathbf{d}}{i-\mathbf{d}}}{\binom{k}{i-y}} \cdot \binom{k}{i-y} Q(i-y, n-1) \\
&= \frac{k\lambda}{\mu i} \cdot \binom{k-\mathbf{d}}{i-\mathbf{d}} \sum_{y=0}^{\mathbf{d}} \binom{\mathbf{d}}{y} Q(i-y, n-1).
\end{aligned}$$

Multiplying both sides by $\frac{i}{k}$, we get

$$Q(i,n) \cdot \binom{k-1}{i-1} = \frac{\lambda}{\mu} \cdot \binom{k-\mathbf{d}}{i-\mathbf{d}} \sum_{y=0}^{\mathbf{d}} \binom{\mathbf{d}}{y} Q(i-y, n-1).$$

Next, we will eliminate the $\frac{\lambda}{\mu}$ term from the recurrence. Let $R(i,n) = \left(\frac{\mu}{\lambda}\right)^n \cdot Q(i,n)$. Then we have

$$R(i,n) \cdot \binom{k-1}{i-1} = \binom{k-\mathbf{d}}{i-\mathbf{d}} \sum_{y=0}^{\mathbf{d}} \binom{\mathbf{d}}{y} R(i-y, -1).$$

Finally, to eliminate the dependency on k, we let $S(i,n) = \binom{k-1}{\mathbf{d}-1}^n \cdot R(i,n)$ and obtain

$$S(i,n)\binom{k-1}{i-1} = \binom{k-1}{\mathbf{d}-1}\binom{k-\mathbf{d}}{i-\mathbf{d}}\sum_{y=0}^{\mathbf{d}}\binom{\mathbf{d}}{y}S(i-y,n-1)$$

$$S(i,n) = \binom{i-1}{\mathbf{d}-1}\sum_{y=0}^{\mathbf{d}}\binom{\mathbf{d}}{y}S(i-y,n-1).$$

Note that $S(i,n)$ relates to our original recurrence $P(i,n)$ as follows:

$$P(i,n) = \frac{\binom{k}{i}\left(\frac{\lambda}{\mu}\right)^n}{\binom{k-1}{\mathbf{d}-1}^n} \cdot S(i,n).$$

We will now define a generating function for $S(i,n)$:

$$G_i(x) = \sum_{n=1}^{\infty} S(i,n)x^n.$$

Taking the derivative of this generating function, we obtain

$$\begin{aligned} G_i'(x) &= \sum_{n=1}^{\infty} n \cdot S(i,n)x^{n-1} \\ xG_i'(x) &= \sum_{n=1}^{\infty} n \cdot S(i,n)x^n \\ \sum_{i=\mathbf{d}}^{k}\binom{k}{i}xG_i'(x) &= \sum_{i=\mathbf{d}}^{k}\sum_{n=1}^{\infty}\binom{k}{i}n \cdot S(i,n)x^n, \end{aligned} \tag{5.10}$$

where the second line results from multiplying both sides of the equation by x and the third line results from multiplying both sides of the equation by $\binom{k}{i}$ and summing over all $\mathbf{d} \leq i \leq k$. Evaluating (5.10) at $x_0 = \frac{\lambda/\mu}{\binom{k-1}{\mathbf{d}-1}}$ we have

$$\sum_{i=\mathbf{d}}^{k}\binom{k}{i}x_0G_i'(x_0) = \sum_{i=\mathbf{d}}^{k}\sum_{n=1}^{\infty} n \cdot P(i,n) = \frac{\mathbf{E}[N]}{\mathcal{C}},$$

which is exactly what we want, noting that we already know $\mathcal{C}$ from Theorem 5.2.

All we need to do is find $G_i'(x)$. Observe that if we evaluate $G_i(x)$ at $x_0 = \frac{\lambda/\mu}{\binom{k-1}{\mathbf{d}-1}}$ we get

$$\binom{k}{i}G_i(x_0) = \binom{k}{i}\sum_{n=1}^{\infty}S(i,n)x_0^n = \sum_{n=1}^{\infty}P(i,n) = \frac{p_i}{\mathcal{C}}, \tag{5.11}$$

where p_i is the stationary probability that i servers are busy. Furthermore, since the stationary probabilities have to sum to 1, we have the normalization equation

$$\frac{1}{\mathcal{C}} = 1 + \sum_{i=\mathbf{d}}^{k} \frac{p_i}{\mathcal{C}}. \tag{5.12}$$

We define the function $C(x)$:

$$C(x) = \prod_{i=\mathbf{d}}^{k} \left(1 - \binom{i-1}{\mathbf{d}-1} x\right).$$

Note that $C(x_0) = \mathcal{C}$ at $x_0 = \frac{\lambda/\mu}{\binom{k-1}{\mathbf{d}-1}}$.

Combining (5.12) and (5.11), we have

$$\frac{1}{C(x_0)} = 1 + \sum_{i=\mathbf{d}}^{k} \binom{k}{i} G_i(x_0). \tag{5.13}$$

Since $\frac{\lambda}{\mu}$ can range from 0 to 1, x_0 can take on any value from 0 to $\frac{1}{\binom{k-1}{\mathbf{d}-1}}$, so (5.13) holds for all $x \in (0, \frac{1}{\binom{k-1}{\mathbf{d}-1}})$. This allows us to differentiate both sides of (5.13), to get

$$\begin{aligned} \frac{d}{dx}\frac{1}{C(x)} &= \sum_{i=\mathbf{d}}^{k} \binom{k}{i} G_i'(x) \\ x\frac{d}{dx}\frac{1}{C(x)} &= \sum_{i=\mathbf{d}}^{k} \binom{k}{i} x G_i'(x). \end{aligned} \tag{5.14}$$

Note that when evaluated at $x_0 = \frac{\lambda/\mu}{\binom{k-1}{\mathbf{d}-1}}$, the right-hand side of (5.14) is equal to $\frac{\mathbf{E}[N]}{\mathcal{C}}$.

So we have

$$\mathbf{E}[N] = C(x_0) \cdot x_0 \cdot \left(\frac{d}{dx}\frac{1}{C(x)}\right)\Big|_{x=x_0} = \sum_{i=\mathbf{d}}^{k} \frac{\lambda}{\mu\frac{\binom{k-1}{\mathbf{d}-1}}{\binom{i-1}{\mathbf{d}-1}} - \lambda}, \tag{5.15}$$

where the final equality results from taking the derivative of $\frac{1}{C(x)}$.

Finally, by Little's Law we have $\mathbf{E}[T] = \frac{\mathbf{E}[N]}{k\lambda}$, which gives us the form for $\mathbf{E}[T]$ given in (5.1). This completes the proof of Theorem 5.1.

5.3.3 Intuition for $\mathbf{E}[N]$

It is not immediately obvious why $\mathbf{E}[N]$ and $\mathbf{E}[T]$ follow the forms derived above. To understand this, consider the state space we use for our system, described in Section 5.3.1.

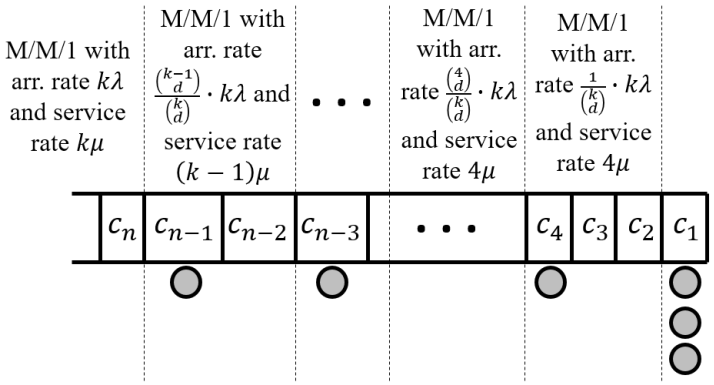


Figure 5.3: Illustration of intuition behind the form of $\mathbf{E}[N]$, equation (5.15). We track all jobs in the system in the order in which they arrived, where the most recent arrival is at the left. Circles below a job indicate the number of servers currently working on that job. In this example, the oldest job in the system is in service at all $\mathbf{d} = 3$ of its servers. With respect to the mean number of jobs in the system, we can view the system as decoupling into $k - \mathbf{d}$ separate M/M/1 queues, where the separation points are indicated by dashed lines.

Our state space maintains one list of all jobs in the system in the order in which they arrived. This list includes both the jobs in service at one or more servers and jobs that are waiting in the queue at all $\mathbf{d}$ of their servers. Now suppose we annotate this list with the locations of all k servers. For example, in Figure 5.3 there are 3 servers working on the oldest job in the system and 1 server working on the second-newest job in the system.

Now imagine starting at the left of the queue (at job c_n) and moving to the right, partitioning the system at each server. From the perspective of any job to the left of all k servers (e.g., job c_n in Figure 5.3), all k servers are pooled together to help the job move forward in the queue. All jobs arrive to the back of the queue (with total rate $k\lambda$), and the pooled servers work at combined rate $k\mu$. Hence the left-most section of the system looks like an M/M/1 with arrival rate $k\lambda$ and service rate $k\mu$.

Once we have moved to the right of the left-most server (e.g., to job c_{n-2} in Figure 5.3), the remaining $k-1$ servers pool together to help the remaining jobs move forward in the queue. But only the jobs that cannot be served by the left-most server "arrive" to this subsystem. There are $\binom{k-1}{\mathbf{d}}$ such classes of jobs. Hence the second section of the system looks like an M/M/1 with arrival rate $\frac{\binom{k-1}{\mathbf{d}}}{\binom{k}{\mathbf{d}}} \cdot k\lambda$ and service rate $(k-1)\mu$.

Proceeding in this manner, we can view the system as decomposing into $k - \mathbf{d}$ separate M/M/1s. The total number of jobs in this system is then

$$\mathbf{E}[N] = \sum_{i=\mathbf{d}}^{k} \frac{\frac{\binom{i}{\mathbf{d}}}{\binom{k}{\mathbf{d}}} \cdot k\lambda}{\frac{i}{k} \cdot k\mu - \frac{\binom{i}{\mathbf{d}}}{\binom{k}{\mathbf{d}}} \cdot k\lambda},$$

which we can obtain from rearranging the terms in equation (5.15).

5.3.4 Proof of Theorem 3

Theorem 5.3. *Under Redundancy-*$\mathbf{d}$*, the system is stable when* $\rho = \frac{\lambda}{\mu} < 1$.

Proof. Consider the system as the number of jobs $n \to \infty$. For any given number of busy servers $i < k$, the probability of increasing the number of busy servers when going from n to $n+1$ jobs is greater than $1/\binom{k}{\mathbf{d}} > 0$ and is independent of n. Hence as $n \to \infty$, the probability that all k servers are busy approaches 1 no slower than the c.d.f. of a geometric random variable with parameter $1/\binom{k}{\mathbf{d}}$. Thus as $n \to \infty$, $P(i,n) \to 0$ for all $i < k$, and so $\pi_n \to P(k,n)$. Looking at the recurrence for $P(k,n)$ given in (5.5), since $P(i,n) \to 0$ for all $i < k$, the tail terms of $P(k,n)$ are all of the form

$$\frac{\binom{k}{\mathbf{d}}\lambda_{\text{class}}}{k\mu}P(k,n-1) = \frac{\binom{k}{\mathbf{d}}\frac{k\lambda}{\binom{k}{\mathbf{d}}}}{k\mu}P(k,n-1) = \frac{\lambda}{\mu}P(k,n-1).$$

When $\lambda < \mu$, this term is less than 1 and so the $P(k,n)$'s form a geometric sequence. Hence the series $\sum_{n=0}^{\infty}\pi_n$ converges if and only if $\frac{\lambda}{\mu} < 1$. Since the series converges, there is some constant $\mathcal{C}$ such that the π_n's sum to 1. □

5.4 Large System Limit Analysis

In Section 5.3 we derived exact expressions for mean response time under Redundancy-$\mathbf{d}$ for any specific k and $\mathbf{d}$ using a Markov chain approach. Even though the Markov chain approach gives us the full distribution of the number of jobs in the system, we cannot apply Distributional Little's Law to find the distribution of response time because jobs need not leave the system in the order in which they arrived. In this section we provide an alternative approach to analyzing Redundancy-$\mathbf{d}$ that yields a closed-form expression for the distribution of response time. Our result is exact in the limiting regime in which $k \to \infty$, under the assumption that the queues are *asymptotically independent*.

To understand what we mean by asymptotic independence, we first define a job's "non-redundant response time" on a server i to be the response time that the job would experience if it arrived to the system and sent only one copy to a randomly chosen server i. The queues are $\mathbf{d}$-wise asymptotically independent if knowing a job's non-redundant response time on servers $i_1, \ldots, i_{\mathbf{d}-1}$ does not tell us anything about the job's non-redundant response time on server $i_{\mathbf{d}}$. Assumption 5.1 formalizes this notion of asymptotic independence.

Assumption 5.1. *Under Redundancy-*$\mathbf{d}$*, as* $k \to \infty$*, the queues are* $\mathbf{d}$*-wise asymptotically independent. That is,* $\mathbf{P}\left\{T_{i_{\mathbf{d}}} > t \mid T_{i_1}, \ldots, T_{i_{\mathbf{d}-1}}\right\} = \mathbf{P}\left\{T_{i_{\mathbf{d}}} > t\right\}$ *for all* $i_{\mathbf{d}} \neq i_1, \ldots, i_{\mathbf{d}-1}$*, where* T_i *is a job's non-redundant response time at server* i*.*

Theorem 5.6. *Under Assumption 5.1, as* $k \to \infty$*, the response time under Redundancy-*$\mathbf{d}$ *with* $\mathbf{d} > 1$ *has tail distribution*

$$\mathbf{P}\left\{T > t\right\} = \bar{F}_T(t) = \left(\frac{1}{\rho + (1-\rho)e^{t\mu(\mathbf{d}-1)}}\right)^{\frac{\mathbf{d}}{\mathbf{d}-1}}, \tag{5.16}$$

where $\rho = \frac{\lambda}{\mu}$.

Conjecture 5.1. *Assumption 5.1 holds.*

Remark The analogue of Conjecture 5.1 has been proved in a wide range of settings: asymptotic independence of queues was shown under the JSQ-$\mathbf{d}$ policy in [72] for exponential runtimes, and extended to general runtimes in [18]. In [77], a similar result was shown for a variety of dispatching policies in a system with batch arrivals. Unfortunately, the proofs presented in the above work do not extend to the Redundancy-$\mathbf{d}$ policy, thus we consign proving Conjecture 5.1 to future work. In Section 5.4.2 we compare our analytical results to simulation and see that the results converge, supporting Conjecture 5.1.

We now turn to the proof of Theorem 5.6.

Proof. **[Theorem 5.6]** We consider a tagged arrival to the system, which we assume without loss of generality arrived at time 0 to a system that is stationary. We want to find

$$\bar{F}_T(t) = \mathbf{P}\left\{\text{tagged arrival is not complete by time } t\right\}.$$

Denote by T_i the non-redundant response time of a job on server i, i.e., the time from when a job arrives at server i to when it would complete on server i if it had no other copies; note that T_i might be longer than response time T since T is the min of $T_1, \ldots, T_{\mathbf{d}}$. Throughout this section, T will always represent the response time in a system using the Redundancy-$\mathbf{d}$ policy, whereas T_i represents the non-redundant response time at server i.

We can express T in terms of T_i as follows:

$$\begin{aligned}
\bar{F}_T(t) = \Pr\{T > t\} &= \mathbf{P}\left\{T_1 > t \;\&\; T_2 > t \;\&\; \cdots \;\&\; T_{\mathbf{d}} > t\right\} \\
&= \mathbf{P}\left\{T_1 > t\right\} \cdot \mathbf{P}\left\{T_2 > t\right\} \cdots \mathbf{P}\left\{T_{\mathbf{d}} > t\right\} \\
&= \left(\mathbf{P}\left\{T_i > t\right\}\right)^{\mathbf{d}} \\
&= \bar{F}_{T_i}(t)^{\mathbf{d}}, \qquad (5.17)
\end{aligned}$$

where the second line is due to the asymptotic independence assumption. Thus in order to find $\bar{F}_T(t)$, we need to understand $\bar{F}_{T_i}(t)$.

To understand $\bar{F}_{T_i}(t)$, observe that there are two ways in which a tagged arrival could have not completed service at server i by time t (assuming the tagged job has no other copies). First, the tagged job could have size larger than t at server i. Second, even if the tagged job has size $S_i < t$, it will not complete at server i by time t if it does not enter service at server i by time $t - S_i$, that is, if its non-redundant time in queue at server i, T_i^Q, exceeds $t - S_i$. We thus have

$$\begin{aligned}
\bar{F}_{T_i}(t) &= \mathbf{P}\left\{T_i > t\right\} \\
&= \mathbf{P}\left\{S_i > t\right\} + \mathbf{P}\left\{0 < S_i < t \;\wedge\; T_i^Q > t - S_i\right\} \\
&= \bar{F}_S(t) + \int_0^t f_S(x) \cdot \bar{F}_{T_i^Q}(t-x)dx \qquad (5.18) \\
&= e^{-\mu t} + \int_0^t \mu e^{-\mu x} \cdot \bar{F}_{T_i^Q}(t-x)dx \\
&= e^{-\mu t} + \int_0^t \mu e^{-\mu(t-y)} \cdot \bar{F}_{T_i^Q}(y)dy,
\end{aligned}$$

where the integral is due to conditioning on the value of S_i.

Next we need to understand $\bar{F}_{T_i^Q}(t)$, the probability that the tagged job has not entered service at server i by time t (assuming the tagged job has no other copies). To do this, we look back in time to the most recent arrival to server i before the tagged job arrived. Call this most recent arrival job A. Suppose job A arrived at time $t - Y < 0$. The tagged job will not enter service by time t if and only if either

1. There is still some other job ahead of job A at server i. This is equivalent to saying that for job A, $T_i^Q > Y$, recalling that T_i^Q is the time that job A would spend in the queue at server i if it had no other copies.
2. Job A is in service at server i at time t. That is, job A has not departed from server i or from any of its other $\mathbf{d} - 1$ servers by time t.

We thus have

$$\begin{aligned}
\mathbf{P}\left\{\begin{array}{c}\text{tagged job not}\\ \text{in service by}\\ \text{time } t\end{array}\right\} &= \mathbf{P}\left\{\begin{array}{c}\text{job } A \text{ cannot have}\\ \text{entered service at}\\ \text{server } i \text{ by time } t\end{array}\right\} + \mathbf{P}\left\{\begin{array}{c}\text{job } A \text{ is in service}\\ \text{at server } i \text{ by time } t\\ \text{but has not departed}\\ \text{any server by time } t\end{array}\right\}\\
&= \mathbf{P}\left\{T_i^Q > Y\right\} + \mathbf{P}\left\{T_i^Q < Y \ \wedge \ T > Y\right\}\\
&= \bar{F}_{T_i^Q}(Y) + \Pr\left\{T_i^Q < Y \ \wedge \ T_1 > Y \wedge \cdots \wedge T_{\mathbf{d}} > Y\right\}\\
&= \bar{F}_{T_i^Q}(Y) + \Pr\left\{T_i^Q < Y \wedge T_i > Y\right\} \cdot \bar{F}_{T_i}(Y)^{\mathbf{d}-1}\\
&= \bar{F}_{T_i^Q}(Y) + (\bar{F}_{T_i}(Y) - \bar{F}_{T_i^Q}(Y))\bar{F}_{T_i}(Y)^{\mathbf{d}-1},
\end{aligned}$$

where again we assume that the $\mathbf{d}$ queues are independent.

Now to find $\bar{F}_{T_i^Q}(t)$, we integrate over all possible values of Y such that job A could have arrived at time $t - Y$, noting that the interarrival times to a particular queue are exponentially distributed with rate $\lambda\mathbf{d}$:

$$\bar{F}_{T_i^Q}(t) = \int_t^\infty \lambda\mathbf{d}e^{\lambda\mathbf{d}(t-y)}\left(\bar{F}_{T_i^Q}(y) + (\bar{F}_{T_i}(y) - \bar{F}_{T_i^Q}(y))\bar{F}_{T_i}(y)^{\mathbf{d}-1}\right)dy.$$

Note that $\bar{F}_{T_i}(t)$ and $\bar{F}_{T_i^Q}(t)$ are defined recursively in terms of each other.

We thus have a system of two differential equations:

$$\bar{F}_{T_i}(t) = e^{-\mu t} + \int_0^t \mu e^{-\mu(t-y)} \cdot \bar{F}_{T_i^Q}(y)dy \tag{5.19}$$

$$\bar{F}_{T_i^Q}(t) = \int_t^\infty \lambda\mathbf{d}e^{\lambda\mathbf{d}(t-y)}\left(\bar{F}_{T_i^Q}(y) + (\bar{F}_{T_i}(y) - \bar{F}_{T_i^Q}(y))\bar{F}_{T_i}(y)^{\mathbf{d}-1}\right)dy. \tag{5.20}$$

To solve the system, we begin by taking the derivative of (5.19), using the general Leibniz's rule to differentiate the function of two variables under the integral:

$$\bar{F}'_{T_i}(t) = -\mu e^{-\mu t} + \int_{y=0}^t \frac{d}{dt}\left(\mu e^{-\mu(t-y)}\bar{F}_{T_i^Q}(y)\right)dy$$

$$
\begin{aligned}
&\quad + \mu e^{-\mu(t-t)} \bar{F}_{T_i^Q}(t) \cdot \frac{d}{dt} t - \mu e^{-\mu(t-0)} \bar{F}_{T_i^Q}(0) \cdot \frac{d}{dt} 0 \\
&= -\mu e^{-\mu t} + \int_0^t -\mu^2 e^{-\mu(t-y)} \bar{F}_{T_i^Q}(y) dy + \mu \bar{F}_{T_i^Q}(t) \\
&= -\mu \left(e^{-\mu t} + \int_0^t \mu e^{-\mu(t-y)} \bar{F}_{T_i^Q}(y) dy - \bar{F}_{T_i^Q}(t) \right) \\
&= \mu \left(\bar{F}_{T_i^Q}(t) - \bar{F}_{T_i}(t) \right), \qquad (5.21)
\end{aligned}
$$

where the last line results from substituting (5.19). Taking the derivative of (5.20) in a similar manner,

$$
\begin{aligned}
\bar{F}'_{T_i^Q}(t) &= \lambda \mathbf{d} \bar{F}_{T_i}(t)^{\mathbf{d}-1} \left(\bar{F}_{T_i^Q}(t) - \bar{F}_{T_i}(t) \right) \\
&= \frac{\lambda \mathbf{d}}{\mu} \bar{F}_{T_i}(t)^{\mathbf{d}-1} \cdot \bar{F}'_{T_i}(t), \qquad (5.22)
\end{aligned}
$$

where the last line results from substituting (5.21). Taking the derivative of (5.21), we find

$$
\begin{aligned}
\bar{F}''_{T_i}(t) &= \mu \left(\bar{F}'_{T_i^Q}(t) - \bar{F}'_{T_i}(t) \right) \\
&= \lambda \mathbf{d} \bar{F}_{T_i}(t)^{\mathbf{d}-1} \cdot \bar{F}'_{T_i}(t) - \mu \bar{F}'_{T_i}(t) + \eta, \qquad (5.23)
\end{aligned}
$$

where the last line results from substituting (5.22) and η is a constant which is equal to 0 by Lemma 5.1 in Section 5.4.3. Integrating (5.23), we get

$$
\bar{F}'_{T_i}(t) = \lambda \bar{F}_{T_i}(t)^{\mathbf{d}} - \mu \bar{F}_{T_i}(t).
$$

Now we have a single differential equation for $\bar{F}_{T_i}(t)$, which we solve to get

$$
\bar{F}_{T_i}(t) = \left(\frac{\mu}{\lambda + \alpha e^{t\mu(d-1)}} \right)^{\frac{1}{\mathbf{d}-1}},
$$

where α is a constant. Note that solving this differential equation is the only place where we needed $\mathbf{d} > 1$. We know that $\bar{F}_{T_i}(0) = 1$, so we can solve for α, yielding $\alpha = \mu - \lambda$. So we have

$$
\bar{F}_{T_i}(t) = \left(\frac{\mu}{\lambda + (\mu - \lambda) e^{t\mu(\mathbf{d}-1)}} \right)^{\frac{1}{\mathbf{d}-1}}.
$$

Finally, we need $\bar{F}_T(t)$, which from (5.17) is

$$
\bar{F}_T(t) = \bar{F}_{T_i}(t)^{\mathbf{d}} = \left(\frac{\mu}{\lambda + (\mu - \lambda) e^{t\mu(\mathbf{d}-1)}} \right)^{\frac{\mathbf{d}}{\mathbf{d}-1}}.
$$

An alternative way of writing this is

$$
\bar{F}_T(t) = \left(\frac{1}{\rho + (1 - \rho) e^{t\mu(\mathbf{d}-1)}} \right)^{\frac{\mathbf{d}}{\mathbf{d}-1}}. \qquad \square
$$

Once we have the c.c.d.f. of response time, we can integrate this over all values of t to find mean response time, $\mathbf{E}[T]$. In Theorem 5.7, we see that $\mathbf{E}[T]$ can be expressed in terms of the hypergeometric function. When $\mathbf{d} = 2$, $\mathbf{E}[T]$ has a simple closed form.

Theorem 5.7. *The mean response time under Redundancy-$\mathbf{d}$ in the infinite-server system is*

$$\mathbf{E}[T] = \frac{{}_2F_1(1,\ 1;\ 1 + \frac{\mathbf{d}}{\mathbf{d}-1};\ \frac{-\rho}{1-\rho})}{\mu\mathbf{d}(\rho - 1)}, \tag{5.24}$$

where

$${}_2F_1(a, b; c; z) = \sum_{n=0}^{\infty} \frac{a^{(n)}b^{(n)}}{c^{(n)}} \frac{z^n}{n!}$$

is the hypergeometric function and

$$x^{(n)} = \begin{cases} 1 & n = 0 \\ x(x+1)\cdots(x+n-1) & n > 0 \end{cases}$$

is the rising Pochammer symbol.

In the case $\mathbf{d} = 2$, this is equivalent to

$$\mathbf{E}[T_{d=2}] = \frac{\mu \ln\left(\frac{\mu}{\mu-\lambda}\right) - \lambda}{\lambda^2}. \tag{5.25}$$

Proof. This follows directly from Theorem 5.6 by integrating $\bar{F}_T(t)$ given in (5.17) over all values of t. □

It is worth comparing the expression in (5.25) to the mean response time under JSQ-2, in which each arriving job polls 2 servers and joins only that queue which is the shorter of the two. From [55] and [72] it is known that when $\mu = 1$ and $\lambda \to 1$ the mean response time under JSQ-2 is given by

$$\mathbf{E}[T] = \frac{\ln\left(\frac{1}{1-\lambda}\right)}{\lambda \ln(2\lambda)} + O(1). \tag{5.26}$$

Thus for $\mathbf{d} = 2$ as $\lambda \to \mu = 1$ the mean response times in both systems contain the term $\ln(\frac{1}{1-\lambda})$.

5.4.1 Insights

Theorem 5.6 tells us the distribution of response time under Redundancy-$\mathbf{d}$ in the infinite-server system. Here we discuss the characteristics of system behavior that follow from the form of this distribution.

Theorem 5.8. *The response time under Redundancy-$\mathbf{d}$ in the infinite-server system has increasing failure rate.*

Proof. We first find the failure rate

$$r_T(t) = \frac{f_T(t)}{\bar{F}_T(t)}$$

$$= \frac{(1-\rho)\mu\mathbf{d}e^{t\mu(\mathbf{d}-1)}}{(1-\rho)e^{t\mu(\mathbf{d}-1)}+\rho}.$$

Now we find the derivative of $r_T(t)$ as follows:

$$r_T'(t) = \frac{\mathbf{d}(\mathbf{d}-1)\rho(1-\rho)\mu^2 e^{t\mu(\mathbf{d}-1)}}{(\rho+(1-\rho)e^{t\mu(\mathbf{d}-1)})^2},$$

which is positive since the denominator is positive and all terms in the numerator are positive. Hence the response time distribution has increasing failure rate. □

The intuition behind this result is that as time passes, a job is likely to be in service at more and more servers, so its probability of completing ("failing") increases.

Theorem 5.9 also tells us that although the response time distribution has increasing failure rate, as $t \to \infty$ the failure rate approaches $\mu\mathbf{d}$. This is because once a job has been in the system for a very long time, it has entered service at all $\mathbf{d}$ of its servers. At this point, the remaining time to completion is simply the minimum of $\mathbf{d}$ exponentials with rate μ, which is an exponential with rate $\mu\mathbf{d}$.

Theorem 5.9. *As $t \to \infty$, the failure rate of the response time distribution under Redundancy-$\mathbf{d}$ approaches $\mu\mathbf{d}$.*

Proof. From Theorem 5.8, we have that

$$r_T(t) = \frac{(1-\rho)\mu\mathbf{d}e^{t\mu(\mathbf{d}-1)}}{(1-\rho)e^{t\mu(\mathbf{d}-1)}+\rho}.$$

Taking the limiting as $t \to \infty$, we find

$$\lim_{t\to\infty} r_T(t) = \lim_{t\to\infty} \frac{(1-\rho)\mu\mathbf{d}e^{t\mu(\mathbf{d}-1)}}{(1-\rho)e^{t\mu(\mathbf{d}-1)}+\rho}$$
$$= \lim_{t\to\infty} \frac{\mu\mathbf{d}}{1+\frac{\rho}{(1-\rho)e^{t\mu(\mathbf{d}-1)}}} = \mu\mathbf{d}.$$ □

Theorem 5.9 addresses what happens to a job's remaining response time given that it has been in the system for a long time. In Theorem 5.10, we look at the effect of $\mathbf{d}$ on the response time distribution.

Theorem 5.10. *As $\mathbf{d} \to \infty$, mean response time scales as $\frac{1}{\mathbf{d}}$.*

Proof. Theorem 5.6 tells us that

$$\bar{F}_T(t) = \left(\frac{1}{\rho+(1-\rho)e^{t\mu(\mathbf{d}-1)}}\right)^{\frac{\mathbf{d}}{\mathbf{d}-1}}.$$

As $\mathbf{d} \to \infty$, the exponent approaches 1, hence

$$\lim_{\mathbf{d}\to\infty} \bar{F}_T(t) = \frac{1}{\rho+(1-\rho)e^{t\mu(\mathbf{d}-1)}}.$$

Integrating over all t to find mean response time, we find

$$\mathbf{E}[T] = \frac{\ln(\frac{1}{1-\rho})}{\mathbf{d}\mu\rho}.$$ □

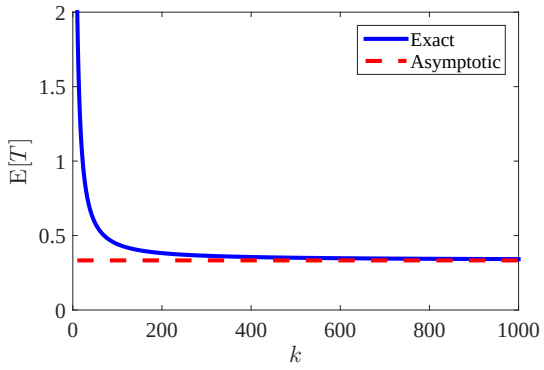


Figure 5.4: Convergence of the mean response time $\mathbf{E}[T]$ in the finite k-server system (solid line) to that in the infinite system (dashed line) when $\rho = 0.95$ and $\mathbf{d} = 10$. As k increases, $\mathbf{E}[T]$ in the finite system drops very steeply to meet that in the infinite system.

		ρ							
		0.2	**0.5**	**0.9**	**0.95**	**0.2**	**0.5**	**0.9**	**0.95**
d	**2**	3	7	41	73	10	31	192	346
	4	7	18	105	190	23	78	496	904
	6	10	28	168	305	36	125	794	1450
	10	17	49	293	534	63	216	1387	2538

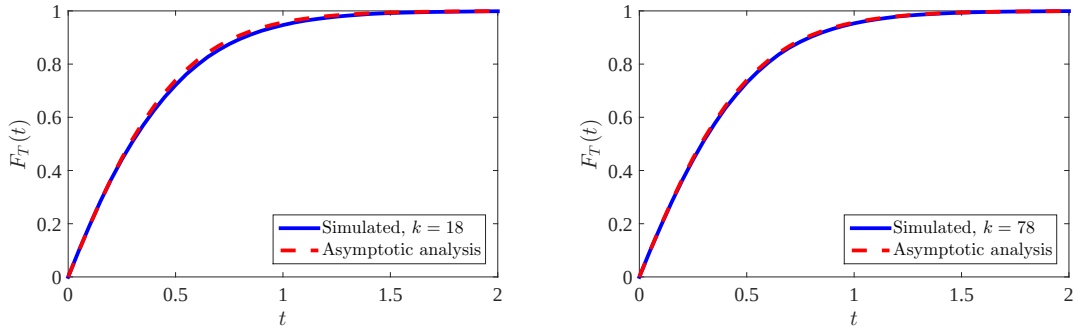
Table 5.1: Number of servers k at which the mean response time in the finite-k system is within 5% (left) and 1% (right) of the asymptotic mean response time.

Theorem 5.10 tells us that as $\mathbf{d}$ becomes large, we see a *diminishing marginal improvement* from further increasing $\mathbf{d}$. This makes sense: when a job is only running on one server, adding an additional server can make a big difference. But when a job is already running on many servers, one extra server does not add very much service capacity relative to what the job already is experiencing. This is important because it suggests that the biggest improvement in response time will come from moving from $\mathbf{d} = 1$ to $\mathbf{d} = 2$, that is, creating just one extra copy of each job. We further explore this phenomenon in Section 5.5.

5.4.2 Convergence

We now turn to the question of convergence: how high does k have to be for the asymptotic analysis to provide a good approximation for a finite k-server system?

In Figure 5.4 we consider the convergence of the mean response time in a finite k-server system to that in the infinite system in the case of $\rho = 0.95$ and $\mathbf{d} = 10$. We see that when k is very low, the mean response time given by our asymptotic analysis is up to a factor of 5 smaller than the exact mean response time given by our analysis in Section 5.3. However as k increases



(a) $k = 18$ (b) $k = 78$

Figure 5.5: Convergence of the full distribution in a system with $\mathbf{d} = 4$, $\lambda = 0.5$, and (a) $k = 18$ and (b) $k = 78$. The dashed line shows the c.d.f. of response time in the limiting system (Theorem 5.6, Section 5.4) and the solid line shows the simulated c.d.f. in the finite system (95% confidence intervals are within the line).

mean response time drops very quickly and ultimately converges to the asymptotic result. This suggests that the asymptotic independence assumption we make when proving Theorem 5.6 may be reasonable.

Table 5.1 shows the number of servers k required for the mean response time in the finite system to be within 5% (left) and within 1% (right) of that in the infinite system for different values of ρ and $\mathbf{d}$. We consider $\mathbf{d} = 2, 4, 6$, and 10; as we will see in Section 5.5, higher values of $\mathbf{d}$ do not yield appreciable response time improvements.

From Table 5.1 we see that the number of servers required for convergence increases in both ρ and $\mathbf{d}$. When load is low, only tens of servers are required for convergence at all values of $\mathbf{d}$ considered. But even at very high load ($\rho = 0.95$) and $\mathbf{d} = 10$, about 530 servers are sufficient for convergence within 5% and about 2500 servers for convergence within 1%. This indicates that mean response time in the limiting system approximates that in the finite system very well for many system sizes of practical interest. For example, typical data centers consist of hundreds or thousands of servers.

Thus far we have only considered convergence of the mean response time; we now turn to convergence of the response time distribution. Since our exact analysis of the finite system only gives mean response time, we use simulation to compare the response time distribution in the finite k-server system with our asymptotic expression. We consider one cell in Table 5.1: the case of $\mathbf{d} = 4$, $\rho = 0.5$. As shown in Table 5.1, 18 servers (respectively, 78 servers) suffice for convergence in the mean to 5% (respectively, 1%). Figure 5.5 shows convergence of the response time distribution for (a) $k = 18$ servers, and (b) $k = 78$ servers, where the biggest difference between the empirical and analytical c.d.f.s is only 0.002. While we show only one value of $\mathbf{d}$ and ρ here, similar results hold for all other parameter choices tested; the values of k corresponding to convergence of the mean to within 1% in Table 5.1 are typically high enough for the response time c.d.f. in the finite system to appear virtually the same as that in the infinite system. Thus the distributional results obtained for the limiting system also can be used to understand finite systems.

5.4.3 Additional Details for Proof of Theorem 5.6

Lemma 5.1.[1] *In the equation*

$$\bar{F}'_{T_i}(t) = \lambda \bar{F}_{T_i}(t)^d - \mu \bar{F}_{T_i}(t) + \eta,$$

$\eta = 0$.

Proof. We know that $\lim_{t\to\infty} \bar{F}_{T_i}(t) = 0$ since $\bar{F}_{T_i}(t)$ is a c.c.d.f. Hence it suffices to show that $\lim_{t\to\infty} \bar{F}'_{T_i}(t) = 0$.

We know that $\bar{F}_{T_i}(t)$ is bounded since it is a c.c.d.f.; hence $\bar{F}'_{T_i}(t)$ is also bounded since it is a polynomial in $\bar{F}_{T_i}(t)$, and $\bar{F}''_{T_i}(t)$ is bounded since it is a polynomial in $\bar{F}_{T_i}(t)$ and $\bar{F}'_{T_i}(t)$. Additionally, we know that $\bar{F}_{T_i}(t)$ is nonincreasing.

If it is not the case that $\lim_{t\to\infty} \bar{F}'_{T_i}(t) = 0$, then there exists $\epsilon > 0$ such that for all $t_0 > 0$, there exists $t > t_0$ such that $\bar{F}'_{T_i}(t) < -\epsilon$. Let M be a bound on $|\bar{F}''_{T_i}(t)|$. Choose $t_1, t_2, \ldots$ such that $t_1 > \frac{\epsilon}{2M}$, $t_{j+1} > t_j + \frac{\epsilon}{M}$, and $\bar{F}'_{T_i}(t_j) < -\epsilon$.

Then for all $t \in (t_j - \frac{\epsilon}{2M}, t_j + \frac{\epsilon}{2M})$, we have $\bar{F}'_{T_i}(t) < -\frac{\epsilon}{2}$, and so

$$\bar{F}_{T_i}\left(t_j + \frac{\epsilon}{2M}\right) - \bar{F}_{T_i}\left(t_j - \frac{\epsilon}{2M}\right) = \int_{t_j - \frac{\epsilon}{2M}}^{t_j + \frac{\epsilon}{2M}} \bar{F}'_{T_i}(t)dt < -\frac{\epsilon^2}{2M}.$$

We also have

$$\bar{F}_{T_i}\left(t_j - \frac{\epsilon}{2M}\right) \leq \bar{F}_{T_i}\left(t_{j-1} + \frac{\epsilon}{2M}\right),$$

since $t_j - \frac{\epsilon}{2M} > t_{j-1} + \frac{\epsilon}{2M}$, and finally

$$\bar{F}_{T_i}\left(t_j + \frac{\epsilon}{2M}\right) < \bar{F}_{T_i}\left(t_{j-1} + \frac{\epsilon}{2M}\right) - \frac{\epsilon^2}{2M}.$$

Then the sequence $\{\bar{F}(t_j + \frac{\epsilon}{2M})\}$ contradicts that $\bar{F}(t)$ is bounded. Hence $\lim_{t\to\infty} \bar{F}'_{T_i}(t) = 0$, and so $\eta = 0$. □

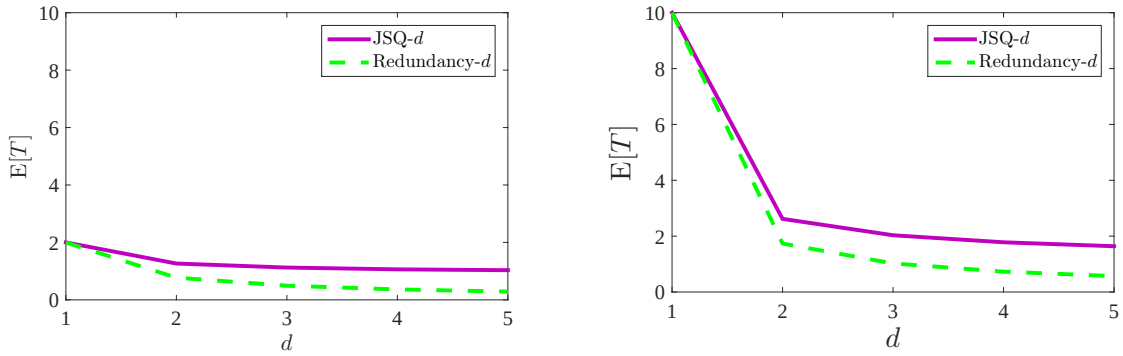
5.5 Power of d Choices

In this section we study the effect of increasing $\mathbf{d}$ on response time under Redundancy-$\mathbf{d}$. We assume that k is large enough to allow us to leverage our asymptotic analysis from Section 5.4. Throughout this section we assume the service rate at every server is $\mu = 1$.

Figure 5.6 compares mean response time as a function of $\mathbf{d}$ under Redundancy-$\mathbf{d}$ to that under the Join-the-Shortest-Queue-$\mathbf{d}$ (JSQ-$\mathbf{d}$, [55, 72]) dispatching policy when the system load, defined as $\rho = \frac{\lambda}{\mu}$, is $\rho = 0.5$ and $\rho = 0.9$. Under JSQ-$\mathbf{d}$, each arrival polls $\mathbf{d}$ servers chosen uniformly at random and joins the queue at the server with the fewest jobs in the queue. Note that jobs only join *one* queue under JSQ-$\mathbf{d}$; there is no redundancy. The Redundancy-$\mathbf{d}$ results are from our asymptotic analysis (Section 5.4); JSQ-$\mathbf{d}$ is simulated with $k = 1000$.

Like under JSQ-$\mathbf{d}$, we see that under Redundancy-$\mathbf{d}$ increasing $\mathbf{d}$ yields a substantial response time improvement relative to $\mathbf{d} = 1$ (no redundancy): both JSQ-$\mathbf{d}$ and Redundancy-$\mathbf{d}$ take

[1] Thanks to Evan Cavallo and Danny Zhu for providing the proof of Lemma 5.1.



(a) $\rho = 0.5$ **(b)** $\rho = 0.9$

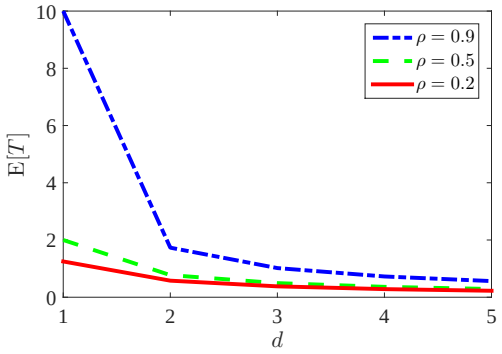
Figure 5.6: Comparing mean response time, $\mathbf{E}[T]$, under Redundancy-$\mathbf{d}$ (dashed line) and JSQ-$\mathbf{d}$ (solid line) as $\mathbf{d}$ increases when load is (a) $\rho = 0.5$ and (b) $\rho = 0.9$. Under both policies as $\mathbf{d}$ increases $\mathbf{E}[T]$ decreases; this improvement is much greater under Redundancy-$\mathbf{d}$. For JSQ-$\mathbf{d}$ (simulated), 95% confidence intervals are within the line.

advantage of queue length variability by allowing a job to wait in the shortest of $\mathbf{d}$ queues. But redundancy provides an additional benefit as well—the same job can be in service at multiple servers at the same time, in which case it experiences the minimum runtime across these servers. This allows Redundancy-$\mathbf{d}$ to provide much lower response times than JSQ-$\mathbf{d}$.

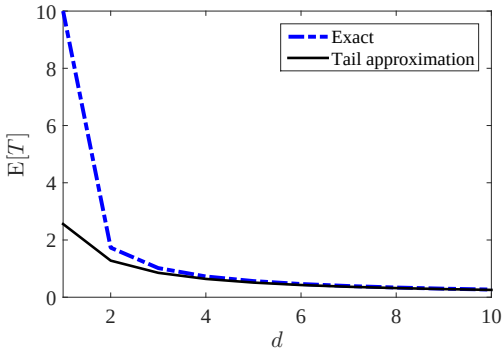
Mean response time under Redundancy-$\mathbf{d}$ exhibits these same trends under other loads: Figure 5.7(a) shows mean response time under Redundancy-$\mathbf{d}$ as a function of $\mathbf{d}$ for low, medium, and high load (again, we assume that k is large and thus show results from our asymptotic analysis). At all loads, as $\mathbf{d}$ increases, mean response time decreases, with this benefit being greatest under higher loads. When load is low, queueing times are low so the primary benefit of redundancy comes from a job receiving the minimum runtime on $\mathbf{d}$ servers. Queueing times increase at higher load so redundancy can now reduce queueing time as well as runtime.

At all loads, the most significant improvement occurs between $\mathbf{d} = 1$ and $\mathbf{d} = 2$. This improvement ranges from a factor of 2 at $\rho = 0.2$ to a factor of 6 at $\rho = 0.9$. As $\mathbf{d}$ grows large, Lemma 5.10 tells us that mean response time scales as $\frac{1}{\mathbf{d}}$. This is shown in Figure 5.7(b), which compares our analytical result for $\mathbf{E}[T]$ to the tail form given in Lemma 5.10 when $\rho = 0.9$. We see that $\mathbf{E}[T]$ converges to the predicted tail shape relatively quickly; by $\mathbf{d} = 6$ the lines are nearly indistinguishable.

Thus far we have discussed only the mean response time; however our asymptotic analysis provides the full response time distribution. Figure 5.8 shows the c.d.f. of response time under Redundancy-$\mathbf{d}$ with $\mathbf{d} = 2$ at low, medium, high, and very high load. When load is high, not only is $F_T(t) = \mathbf{P}\{T < t\}$ lower, but the shape of the c.d.f. actually is *convex* at low values of t. This convexity is due to the probability of queueing: when load is high, an arrival is likely to experience a non-zero queueing time at all $\mathbf{d}$ of its servers. Thus the probability that the job completes service by time t does not increase substantially until t is sufficiently high, making it more likely that the job has entered service at one or more servers. In contrast, when load is low, an arrival is likely to begin service immediately on at least one server, so its probability of



(a) Mean improvement



(b) $\frac{1}{\mathbf{d}}$ shape

Figure 5.7: (a) Mean response time, $\mathbf{E}[T]$, under Redundancy-$\mathbf{d}$ as a function of $\mathbf{d}$ under low ($\rho = 0.2$, solid line), medium ($\rho = 0.5$, dashed line), and high ($\rho = 0.9$, dot-dashed line) load. At all loads increasing $\mathbf{d}$ reduces $\mathbf{E}[T]$. The improvement in $\mathbf{E}[T]$ is greatest at high load. (b) As $\mathbf{d}$ grows large, $\mathbf{E}[T]$ scales in proportion to $\frac{1}{\mathbf{d}}$, in accordance with Lemma 5.10.

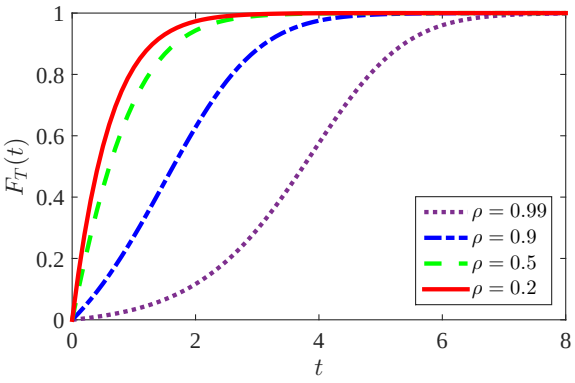


Figure 5.8: $\mathbf{P}\{T \leq t\}$ under Redundancy-$\mathbf{d}$ when $\mathbf{d} = 2$ under four different loads.

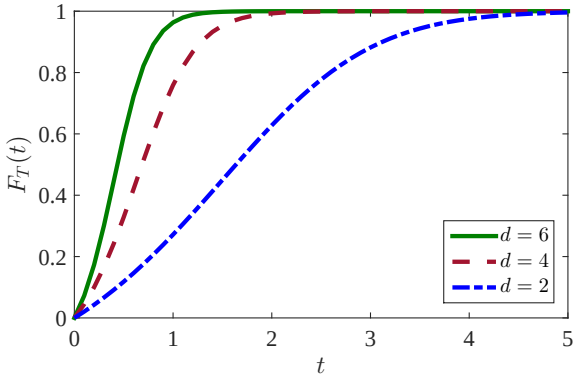


Figure 5.9: $\mathbf{P}\{T \leq t\}$ under Redundancy-$\mathbf{d}$ when $\rho = 0.9$ and $\mathbf{d} = 2$, 4, or 6.

completing service by time t resembles the probability that the runtime on a single server is less than t.

In Figure 5.9 we look at the c.d.f. of response time at high load ($\rho = 0.9$) as $\mathbf{d}$ increases from 2 to 6. When $\mathbf{d} = 2$, the c.d.f. is convex at low t (this is the same curve as shown in Figure 5.8). As $\mathbf{d}$ increases, the c.d.f. is still convex at low t but this is much less pronounced. At high $\mathbf{d}$, the c.d.f. approaches that of an exponential distribution because sending more copies means that a job is more likely to enter service immediately on at least one server. The convexity has disappeared, illustrating the dramatic effect redundancy can have on queueing time, and thus on system time. Looking at the tail of response time, we see from Figure 5.9 that $T_{95} = 3.58$ when $\mathbf{d} = 2$; this is $8\times$ smaller than $T_{95} = 29.9$ when $\mathbf{d} = 1$ (which corresponds to an M/M/1 system).

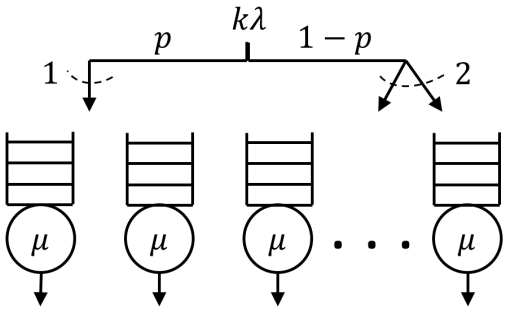


Figure 5.10: The fractional Redundancy-d policy. With probability p an arriving job sends a request to a single server chosen uniformly at random, and with probability $1-p$ an arriving job sends redundant requests to two servers chosen uniformly at random.

5.5.1 Fractional d

In Section 6.5 we saw that the largest improvement in mean response time occurred between $\mathbf{d} = 1$ (no redundancy) and $\mathbf{d} = 2$. Given the magnitude of this gap, we now explore the response time benefits offered by sending on average fewer than two copies of each job.

We define the fractional Redundancy-$\mathbf{d}$ policy as shown in Figure 5.10. When a job arrives to the system, with probability p it is non-redundant and joins the queue at a single server chosen uniformly at random. With probability $1-p$ the job joins the queue at two servers chosen uniformly at random. In this system, we define $\mathbf{d}$ to be the weighted average number of copies sent per job:

$$\mathbf{d} = p \cdot 1 + (1-p) \cdot 2 = 2 - p.$$

To analyze mean response time under fractional Redundancy-$\mathbf{d}$, we follow the Markov chain approach presented in Section 5.3, which extends easily to the fractional $\mathbf{d}$ case (unfortunately the asymptotic analysis in Section 5.4 does not extend to fractional $\mathbf{d}$). We obtain an exact closed-form expression for $\mathbf{E}[T]$, given in Theorem 5.11.

Theorem 5.11. *The mean response time under fractional Redundancy-$\mathbf{d}$ is*

$$\mathbf{E}[T] = \frac{1}{k} \sum_{i=1}^{k} \frac{(i-1) + (k-i)p}{(k-1)\mu - ((i-1) + (k-i)p)\lambda}. \tag{5.27}$$

Proof. At a high level, we begin by writing a system of recurrence equations for the limiting probability that there are i servers busy and m jobs in the system. We then use generating functions to find $\mathbf{E}[N]$. The derivation under fractional Redundancy-$\mathbf{d}$ follows the same approach as in the proof of Theorem 5.1 (see Section 5.3), hence we omit the details of the proof. □

Figure 5.11 shows mean response time, $\mathbf{E}[T]$, as a function of $\mathbf{d}$ for $1 \leq \mathbf{d} \leq 2$ for low, medium, and high load when $k = 1000$. Note that Figure 5.11 is the same setting as Figure 5.7, but zooms in on the range from $\mathbf{d} = 1$ to $\mathbf{d} = 2$. As $\mathbf{d}$ increases, mean response time decreases convexly; introducing even a small amount of redundancy to the system provides a substantial improvement. This is particularly pronounced at high load; at $\rho = 0.9$ setting $\mathbf{d} = 1.5$ (i.e., half

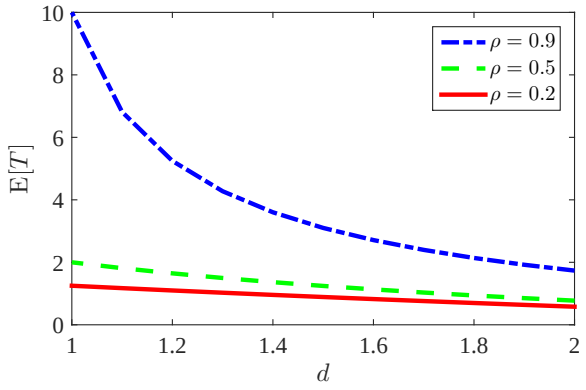


Figure 5.11: Mean response time, $\mathbf{E}[T]$, as a function of $\mathbf{d}$ under fractional Redundancy-$\mathbf{d}$ under low ($\rho = 0.2$, solid line), medium ($\rho = 0.5$, dashed line), and high ($\rho = 0.9$, dot-dashed line) load.

of the jobs are non-redundant) corresponds to a response time improvement of 69% relative to having no redundancy. Even at low load, $\mathbf{E}[T]$ is 29% lower when $\mathbf{d} = 1.5$ than when $\mathbf{d} = 1$.

Once again, very little redundancy is required to achieve significant performance gains. This result is encouraging for systems where there may be costs to redundancy, because it suggests that one can achieve response time benefits with only a limited amount of redundancy.

5.5.2 Non-Exponential Service Times

Thus far, we have assumed that runtimes are exponentially distributed. This assumption was necessary to obtain the closed-form results for mean response time given in Theorem 5.1 and for the distribution of response time given in Theorem 5.6. However, in real systems runtimes may not be exponential. For example, in computer systems network congestion can cause web query round trip times to be highly variable [76]. In this section we use our differential equations approach from Section 5.4 to study, numerically, what happens when runtimes are more or less variable than an exponential.

Returning to Section 5.4, our argument allows us to write equations (5.18) and (5.20) regardless of the particular runtime distribution S. In the case where $S \sim \text{Exp}(\mu)$ we are able to solve the system in closed form. For non-exponential runtimes, while we are unable to find a closed-form solution, we can solve our differential equations numerically.[2]

Figure 5.12 shows mean response time as a function of $\mathbf{d}$ when runtimes are more and less variable than an exponential and $\lambda = 0.5$. When runtimes are highly variable, increasing the value of $\mathbf{d}$ reduces mean response time even more than when runtimes are exponential. For example, when $C^2 = 10$, mean response time decreases by a factor of 17 (compared to a factor of 2.6 for exponentially distributed runtimes). The improvement is bigger under more highly variable runtimes for two reasons. First, when $\mathbf{d} = 1$ (i.e., there is no redundancy) queueing times can be extremely high when runtimes are highly variable, and waiting in multiple queues helps shorter

[2]Thanks to Brandon Lieberthal for providing code to solve the differential equations numerically.

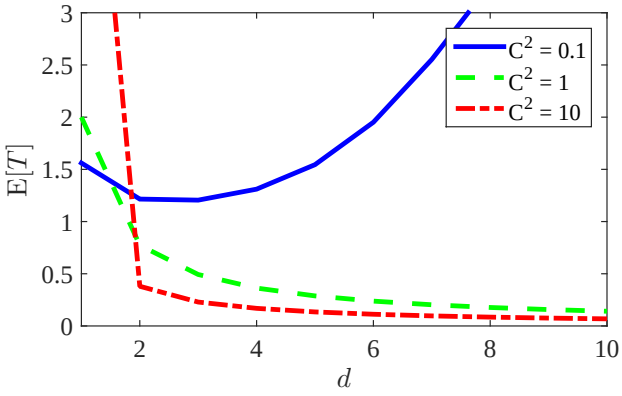


Figure 5.12: Mean response time as a function of $\mathbf{d}$ under Redundancy-$\mathbf{d}$ when runtimes are more or less variable than an exponential. Here $\lambda = 0.5$ and we assume k is large. Lines shown include $S \sim \mathrm{H}_2$ with $C^2 = 10$ (dot-dashed line), $S \sim \mathrm{Exp}$ (dashed line), and $S \sim \mathrm{Erlang}$ with $C^2 = 0.1$ (solid line). For all distributions $\mathbf{E}[S] = 1$.

jobs to avoid waiting behind very long jobs. Second, a job that runs on multiple servers benefits from seeing the minimum runtime across servers, and taking the minimum of multiple runtimes leads to a larger improvement when the runtimes are more variable.

The trend is very different when runtimes have low variability. Unlike for higher variability job size distributions, going from $\mathbf{d} = 1$ to $\mathbf{d} = 2$ yields only a small improvement in mean response time, and as $\mathbf{d}$ becomes higher mean response time actually *increases*. This is because queueing times are already quite low when runtimes have low variability, and rather than benefiting from running on multiple servers, adding multiple copies of similarly-sized jobs congests the system. Thus the "power-of-$\mathbf{d}$" depends crucially not only on load, but also on runtime variability.

5.6 Discussion and Conclusion

In this chapter we introduce a new dispatching policy called Redundancy-$\mathbf{d}$, under which each job that arrives to the system joins the queue at $\mathbf{d}$ servers chosen uniformly at random. Our goal is to understand response time as a function of $\mathbf{d}$. To do this, we provide the first exact analysis of response time under Redundancy-$\mathbf{d}$ using two different approaches.

We first model the system as a Markov chain, as in Chapter 3. In general, aggregating the state space to get the distribution of the number of jobs in the system is combinatorially challenging, especially as the number of servers in the system grows large. Our key insight is that we can derive π_n, the probability that there are n jobs in the system, by further conditioning on the probability that there are n jobs in the system and i servers busy working on these jobs. Expressing π_n in this manner yields a recursive structure that we leverage to find the distribution of the number of jobs in the system and the mean response time in systems with any number of servers k and any number of copies per job $\mathbf{d}$.

In our second analytical approach, we consider the system in the limit as the number of servers approaches infinity. In such a setting we are able to capture the system's behavior under

Redundancy-$\mathbf{d}$ via a system of differential equations that track the amount of work seen by a tagged arrival. We use these differential equations to derive asymptotic expressions for the distribution of response time that are exact under an asymptotic independence assumption. That is, we assume that knowing a job's "non-redundant" sojourn time at one server does not provide any information about what that same job's "non-redundant" sojourn time would be at a different server. This type of asymptotic independence is a very common precondition for the analysis of many related queueing systems. Unfortunately, the techniques typically used to prove asymptotic independence do not easily generalize to the Redundancy-$\mathbf{d}$ policy, again because the departure process under Redundancy-$\mathbf{d}$ is very complicated. We leave the asymptotic independence assumption as a strongly supported conjecture; proving it remains open for future work.

Our analysis allows us to answer questions about the benefits of redundancy that have important implications for real systems. For example, in Section 5.5 we saw that being redundant at only two queues is enough to give most of the response time benefit of redundancy. In systems such as organ transplant waitlists, this suggests that multiple listing in only a small number of regions may suffice. In Section 5.5.1 we saw that much of the response time benefit of redundancy can be achieved when only a small fraction of jobs are redundant. Many patients may be unable to multiple list because they cannot travel to alternative regions to receive a transplant, perhaps because of financial limitations. Our "fractional-$\mathbf{d}$" result suggests that the system as a whole benefits even if only a small proportion of patients multiple list.

The observation that a little redundancy goes a long way is also important in computer systems, particularly when there may be some cost to creating multiple copies of jobs. For example, sending the same request to multiple servers might add network overhead or load, or cancelling the extra copies once the first copy completes might take some amount of time. In Chapter 6 we examine our modeling assumptions in great detail to better understand how these costs affect the messages presented in this chapter.

Although Redundancy-$\mathbf{d}$ appears somewhat similar to dispatching policies such as JSQ-$\mathbf{d}$, our analysis of Redundancy-$\mathbf{d}$ is different. Dispatching policies such as JSQ-$\mathbf{d}$ and similar policies have only been studied in the limit as the number of servers approaches infinity using differential equations that typically track the fraction of queues with at least i jobs [55, 72, 77]. This is very different from our analysis, in which we track the amount of work in a queue as seen by a tagged arrival. Simply tracking the number of jobs in a queue is not powerful enough for analyzing the Redundancy-$\mathbf{d}$ system because the departure process is much more complicated: it includes not only departures due to service completions at that server, but also departures due to completions of jobs' copies at *other* servers.

Our analysis represents a first step towards solving several related queueing problems. For example, in an (n, k) system redundant copies of jobs are sent to multiple queues chosen at random, but more than one copy needs to complete service [43, 44]. The analysis of mean response time in the (n, k) system presented by Li et al. [51] is somewhat similar to the large-system asymptotic approach that we use. Like in our analysis, Li et al. analyze the system in the limit as the number of servers approaches infinity. However unlike our analysis, Li et al. uses a mean-field approach, which considers the interaction between an individual queue and the average system behavior. Mean-field theory is an increasingly common technique for understanding complex queueing systems, but does not apply in systems in which there are synchronous departures from different queues [13]. Hence the approach cannot be used to understand redundancy systems. We hope

that our approach will offer a useful alternative for systems that, like redundancy systems, have synchronous departures or other complex features that cannot be analyzed using the mean-field theory approach.

Chapter 6

S&X: A Better Model for Redundancy

6.1 Introduction

One of the main concerns when implementing redundancy-based policies in practice is that there can be substantial *costs* to redundancy. For example, consider what happens when one job runs on multiple servers at the same time. On one hand, the job itself benefits because it may experience a much shorter runtime on one server than on another. On the other hand, by using multiple servers at once, the job blocks other jobs from entering service. Is the benefit obtained from experiencing the shorter runtime enough to outweigh the additional delay potentially inflicted on subsequent jobs? Does running a job on multiple servers waste too much server capacity? Can the extra load even cause the system to become unstable?

Empirical computer systems results suggest that there are indeed risks to using redundancy [71], yet much of the existing theoretical work on redundancy, including the results presented in Chapters 3-5 of this thesis, suggests that more redundancy is always better. This gap between theoretical and empirical results emerges because the modeling assumptions we have made thus far do not necessarily match the characteristics of the computer systems in which redundancy is used. We assume, for example, that no time is required to cancel the extra copies of a job once the first copy completes service. Often, we assume that runtimes are exponentially distributed. We further assume that a single job's runtimes are independent across servers.

Unfortunately, these assumptions do not always hold in practice. Communication overhead or time required to reset the server state may cause non-negligible cancellation times. If cancelling a job involves shutting down a virtual machine, the cancellation time could be very long. Similarly, runtimes are known to typically follow non-exponential distributions; for example, UNIX job lifetimes [38] and web file sizes [21] both are known to typically follow a heavy-tailed distribution [38]. Much of the exact analysis presented in Chapters 3-5 of this thesis does not apply when we relax our assumptions to allow for cancellation times and general runtimes.

Most importantly, however, in real computer systems a job's runtimes are not independent across servers. While the commonly-assumed IR model makes sense in certain settings, it can be problematic in others. Consider for example a job that is very large, meaning that it comprises a large volume of computation. That job should appear to be large on all servers, possibly slowed down more on some servers than others. This does not happen in the IR model: the job is assigned

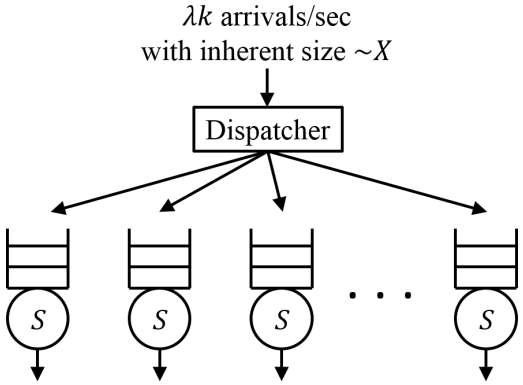


Figure 6.1: The $S\&X$ model. The system has k servers and jobs arrive as a Poisson process with rate λk. Each job has an inherent size X. When a job runs on a server it experiences slowdown S. A job's running time on a single server is $R(1) = X \cdot S$. When a job runs on multiple servers, its inherent size X is the same on all these servers and it experiences a different, independently drawn instance of S on each server.

a different, independent, runtime on different servers. When the job runs on multiple servers it experiences the minimum runtime of all those servers' independent runtimes. Thus an inherently large job can become arbitrarily small under the independence assumption. There is no concept of an inherently large job which remains large at every server.

Our goal in this chapter is to develop a new theoretical model for redundancy that allows us to understand the tradeoff between the benefits and costs associated with redundancy in computer systems. This will enable us to design policies that best leverage the potential performance improvements offered by redundancy without significantly sacrificing performance due to the corresponding costs. Such a model must eliminate many of the assumptions commonly used in analyzing redundancy systems, including in the preceding chapters of this thesis.

Our contributions in this chapter are as follows:

A new CR model for redundancy. We propose the $S\&X$ model (see Figure 6.1), which is the first theoretical model to explicitly decouple the server slowdown (represented by the random variable S) from the inherent job size (represented by the random variable X), thereby causing runtimes to be correlated across servers. The $S\&X$ model marks a departure from traditional queueing theory, which uses a single "service time" variable to jointly represent the server speed and job size. In the context of redundancy, we need to decouple the variables so that a job with a large X component (large job size) will have a large X component on every server.

A better dispatching policy for redundancy. The $S\&X$ model confirms the empirical observation that in computer systems, redundancy is *not* always necessarily a win and can in fact be dangerous. Consider for example, the Redundancy-$\mathbf{d}$ policy, which replicates every arriving job to $\mathbf{d}$ queues (see Chapter 5). Under the IR model, mean response time only decreases as we increase $\mathbf{d}$. By contrast, in the $S\&X$ model, mean response time under Redundancy-$\mathbf{d}$ can

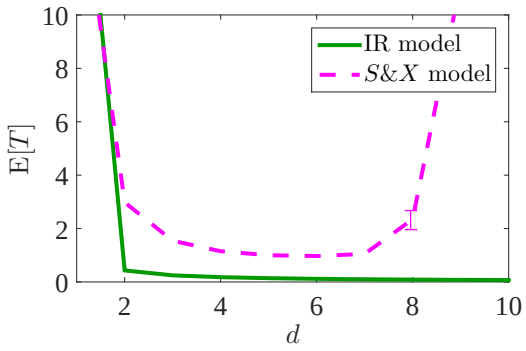


Figure 6.2: Under Redundancy-$\mathbf{d}$, each arriving job sends copies to $\mathbf{d}$ servers chosen uniformly at random. Here we show mean response time, $\mathbf{E}[T]$, as a function of $\mathbf{d}$ under Redundancy-$\mathbf{d}$ when the system has $k = 1000$ servers, the total arrival rate is λk where $\lambda = 0.7$, and inherent job sizes, X, follow a two-phase hyperexponential distribution with balanced means, $\mathbf{E}[X] = \frac{1}{4.7}$, and $C_X^2 = 10$. In the IR model (solid green line, from analysis in Chapter 5), each job draws an i.i.d. instance of X on each server. As $\mathbf{d}$ increases, mean response time decreases. In the $S\&X$ model (dashed pink line, simulated; 95% confidence intervals are within the line unless shown), a job draws a single instance of X which is the same on all servers, and an i.i.d. instance of S on each server. Here S is an empirically measured distribution described in Section 6.2. While in the $S\&X$ model mean response time initially decreases as a function of $\mathbf{d}$, as $\mathbf{d}$ becomes high the system eventually becomes unstable.

improve significantly as we add a small mount of redundancy, but the system (typically) eventually becomes unstable because of the increased load of replication, sending mean response time to infinity (see Figure 6.2). Unfortunately, in general we cannot determine the values of $\mathbf{d}$ that will lead to good performance or to instability, because providing a performance analysis of policies like Redundancy-$\mathbf{d}$ in the $S\&X$ model is an open problem that is likely very difficult (as we saw in Chapter 5, analyzing Redundancy-$\mathbf{d}$ even within the IR model requires a very complex state space).

The difficulty in tuning and potential instability of Redundancy-$\mathbf{d}$ in the $S\&X$ model motivate us to look for new redundancy policies which are less sensitive to $\mathbf{d}$, are robust in that they provably will not go into overload, and are analytically tractable within the $S\&X$ model. To this end, we introduce a new redundancy policy, called Redundant-to-Idle-Queue (RIQ). This policy is similar to Redundancy-$\mathbf{d}$ in that every arrival queries $\mathbf{d}$ servers. However replicas are only made to those servers which are idle. If no server is idle, then the job is sent to a random one of the $\mathbf{d}$ servers (with no additional replicas). We provide an analytical approximation for the transform of response time under RIQ as a function of $\mathbf{d}$. Our analysis allows for any distribution of S, any distribution of X, and any cancellation time. Our analysis matches simulation very well, provided that $\mathbf{d}$ is small relative to the total number of servers, which is certainly typical in practice. Most importantly, our analysis shows us that RIQ is extremely robust. While Redundancy-$\mathbf{d}$ can outperform RIQ at low $\mathbf{d}$, we derive an analytical upper bound on the response time of RIQ under any S and X for any $\mathbf{d}$ that shows the system does not go into overload as $\mathbf{d}$ gets high.

Table 6.1: The Dolly(1,12) empirical slowdown distribution. The server slowdown ranges from 1 to 12, with mean 4.7.

S	1	2	3	4	5	6	7	8	9	10	11	12
Prob.	0.23	0.14	0.09	0.03	0.08	0.10	0.04	0.14	0.12	0.021	0.007	0.002

Further improvements on redundancy policies. The RIQ policy demonstrates that it is possible to design provably robust and analytically understood redundancy policies within the $S\&X$ model. But there is still room for improvement. We consider several modifications to the baseline policies, motivated by strategies used in practice. One idea that further reduces response time is to use Join-the-Shortest-Queue dispatching for jobs that do not find idle servers. Another idea is to replicate only small jobs; unfortunately this hurts RIQ, which is already quite conservative. On the other hand, replicating only small jobs can prevent instability under Redundancy-$\mathbf{d}$, while still allowing Redundancy-$\mathbf{d}$ to achieve good performance at low $\mathbf{d}$. As a compromise between Redundancy-$\mathbf{d}$ and RIQ, we propose the THRESHOLD-n policy, under which each arriving job polls $\mathbf{d}$ servers and replicates itself to those servers with fewer than n jobs in the queue. Like under RIQ, the system is stable under THRESHOLD-n for all $\mathbf{d}$. Like under Redundancy-$\mathbf{d}$, mean response time under THRESHOLD-n can be quite low at the best choice of $\mathbf{d}$.

The remainder of this chapter is organized as follows. In Section 6.2 we introduce the $S\&X$ model. In Section 6.3 we discuss Redundancy-$\mathbf{d}$ and the difficulties it poses in the $S\&X$ model, and in Section 6.4 we introduce and analyze Redundant-to-Idle-Queue (RIQ). Sections 6.5 and 6.6 evaluate the performance of RIQ at low $\mathbf{d}$ and high $\mathbf{d}$ respectively. In Section 6.7 we discuss improvements upon Redundancy-$\mathbf{d}$ and RIQ. Finally, in Section 6.9 we conclude.

This chapter is based on joint work with Mor Harchol-Balter, Alan Scheller-Wolf, and Benny Van Houdt and appears in the following papers:

- Kristen Gardner, Mor Harchol-Balter, Alan Scheller-Wolf. "A Better Model for Job Redundancy: Decoupling Server Slowdown and Job Size." *MASCOTS*, September 2016. [30]
- Kristen Gardner, Mor Harchol-Balter, Alan Scheller-Wolf, and Benny Van Houdt. "A Better Model for Job Redundancy: Decoupling Server Slowdown and Job Size." Under revision at *Transactions on Networking*. [31]

6.2 The $S\&X$ Model

We consider a setting with k homogeneous servers (see Figure 6.1). Each server has its own queue and works on the jobs in its queue in first-come first-served order. Jobs arrive to the system as a Poisson process with rate $k\lambda$, where λ is a constant, and are dispatched immediately upon arrival. We assume that jobs are non-preemptible.

We introduce the $S\&X$ *model*, which captures the effects of both job-dependent and server-dependent factors on a job's runtime. Here S is a random variable denoting the *slowdown* that a job experiences at a server and X is the job's *inherent size*.

We consider two different models for how S and X interact to form a job's runtime on a single

server, which we denote by $R(1)$.

1. In the *multiplicative* slowdown model, a job that runs on one server has runtime $R(1) = X \cdot S$. Here S represents the factor by which the inherent size, X, is "stretched." We assume that $S \geq 1$, which captures the fact that the server slowdown cannot make the job's runtime shorter than the job's inherent size X.
2. In the *additive* slowdown model, a job that runs on one server has runtime $R(1) = X + S$. Here we assume that $S \geq 0$, again so that the server slowdown cannot make the job's runtime shorter than X.

Unlike in the IR model, in the $S\&X$ model runtimes are *not* independent across servers because a job's X component is the same on all servers.

We assume that every time a server begins working on a job, it draws a new independent instance of S. Thus consecutive jobs running on the same server may see different slowdowns. This reflects the fact that the server slowdown changes over time (due to factors such as garbage collection, background load, network interference, etc.) and is not fixed for a particular server. We can think of the particular value of S drawn for an individual job as representing the average slowdown the job experiences while it is in service.

If a job is dispatched to a single server j, it completes service after time equal to its queueing time, T_j^Q, plus its running time on that server, which in the multiplicative model is $R(1)_j = X \cdot S_j$ and in the additive model is $R(1)_j = X + S_j$, where S_j is the particular instance of S that the job experiences on server j. S_j is drawn independently across servers for a given job, and across jobs for a given server. If a job is dispatched to all servers in some set $\mathcal{S}$, its response time is

$$T = \min_{j \in \mathcal{S}} \{T_j^Q + X \cdot S_j\}.$$

When a job that is running on multiple servers completes service on one of these servers, all of its remaining copies must be cancelled. We model the time it takes to cancel a copy as taking time Z, where Z is a random variable.

We use ρ to denote the system load. Unlike in traditional queueing systems, in systems with redundancy load and stability depend not only on the arrival rate and mean runtime, but also on the particular dispatching and scheduling policies. Hence we defer a more detailed discussion of load and stability to Sections 6.3 and 6.4, which introduce specific dispatching policies: Redundancy-$\mathbf{d}$ and Redundant-to-Idle-Queue.

6.2.1 Examples of S Distributions

Several papers have measured the slowdown distributions that occur in real systems. In much of the remainder of this chapter, the results and graphs that we present use these empirical S distributions. Here we describe three empirical slowdown distributions, two of which are multiplicative and one of which is additive.

Multiplicative "Dolly" slowdown. The Dolly(1,12) slowdown distribution (see Table 6.1) was measured empirically in [4] by analyzing traces collected from Facebook's Hadoop cluster and

Microsoft Bing's Dryad cluster. Slowdown values range from 1 to 12, with mean 4.7 and variance 9.5. [1]

Multiplicative "Google" slowdown. The Google slowdown distribution was measured in an unspecified Google service [22]. Here the slowdown distribution can be approximated by a Bimodal distribution:

$$S \sim \begin{cases} 1 & \text{w.p. } 0.99 \\ 14 & \text{w.p. } 0.01 \end{cases}$$

The particular distribution measured here has mean 1.13 and variance 1.67. The Bimodal represents a family of slowdown distributions, the parameters of which can be set to achieve different means and variances.

Additive "Amazon" slowdown. The Amazon slowdown distribution was measured in the S3 storage service [52]. Job runtimes were observed to consist of a constant component that was the same for all jobs, plus a random component that was independent across jobs and could be closely approximated using an exponential. While [52] does not state the measured mean of the exponential component, we can view this as a family of slowdown distributions.

For much of the remainder of this chapter, the results we present use the Dolly$(1, 12)$ slowdown distribution. The qualitative trends we observe are the same when we use other multiplicative slowdown distributions, as well as when we use additive slowdown.

6.3 Redundancy-$\mathbf{d}$ in the $S\&X$ Model

A natural dispatching policy for systems with redundancy is *Redundancy*-$\mathbf{d}$, which we first introduced in Chapter 5.

Definition 6.1. *Under* ***Redundancy-*****d***, each arriving job creates* $\mathbf{d}$ *copies of itself and sends these copies to* $\mathbf{d}$ *servers chosen uniformly at random without replacement.*

Figure 6.2 compares mean response time, $\mathbf{E}[T]$, under Redundancy-$\mathbf{d}$ in the IR model (from the analysis presented in Chapter 5) to that in the $S\&X$ model (simulated; in the $S\&X$ model, we set $S \sim \text{Dolly}(1, 12)$). In the IR model, as $\mathbf{d}$ increases $\mathbf{E}[T]$ decreases provided runtimes are at least as variable as an exponential distribution. This is very different from what happens under the $S\&X$ model: while at first $\mathbf{E}[T]$ decreases as a function of $\mathbf{d}$, as $\mathbf{d}$ becomes higher $\mathbf{E}[T]$ starts to increase and ultimately the system becomes unstable.

While Figure 6.2 shows that the system can become unstable under Redundancy-$\mathbf{d}$ in the $S\&X$ model if $\mathbf{d}$ is too high, it is difficult to prove this analytically. Typically a system is stable as long as the system load, ρ, is less than 1. We can think of ρ as being the arrival rate, λ, multiplied by the expected server capacity used per job. But in the $S\&X$ model, deriving the expected server capacity used per job is not straightforward because it requires knowing the average number of

[1] The full Dolly(1,12) distribution does not appear in [4]; we obtained this from personal communication with the authors.

servers on which a job runs; this is not $\mathbf{d}$ because some copies may be cancelled while still in the queue. Furthermore, a job's copies can enter service at different times on different servers, so knowing the number of servers on which a job runs is not enough to determine the duration of time for which the job occupies each server.

Figure 6.2 highlights the importance of making the right modeling assumptions. The $S\&X$ model results tell a very different story from the IR model results. Unlike in the IR model, in the $S\&X$ model Redundancy-$\mathbf{d}$ is *not* robust to the choice of $\mathbf{d}$; choosing the wrong value of $\mathbf{d}$ can lead to unacceptably poor performance or even instability. Unfortunately, the analysis of Redundancy-$\mathbf{d}$ in the $S\&X$ model remains an open problem, meaning that it is very difficult to know how to choose a good $\mathbf{d}$.

The lack of robustness, difficulty in tuning, and potentially poor performance of Redundancy-$\mathbf{d}$ motivate the need for a better dispatching policy for the $S\&X$ model. In designing such a policy, it is important to avoid the factors that cause poor performance for Redundancy-$\mathbf{d}$. In particular, Redundancy-$\mathbf{d}$ is oblivious to the system state. It creates copies of jobs even when the system has no extra capacity with which to run these copies. Consequently, Redundancy-$\mathbf{d}$ can add too much load to the system, causing queue lengths to build up. An important consideration when designing dispatching policies for the $S\&X$ model therefore is to ensure that we do not add excessive load.

6.4 Redundant-to-Idle-Queue

In this section we introduce a new dispatching policy, Redundant-to-Idle-Queue (RIQ).[2] The RIQ policy is defined in Section 6.4.1 and is analyzed in the remaining subsections. For ease of explanation, we begin by assuming that the cancellation time $Z = 0$. We first approximately analyze mean response time under RIQ (Section 6.4.2). We then build on this analysis to derive an approximation for the transform of response time under RIQ (Section 6.4.3). In Section 6.4.4 we extend our analysis to allow for cancellation times. In Section 6.4.5 we evaluate the quality of our approximations by comparing our analytical results to simulation.

6.4.1 The Redundant-to-Idle-Queue Dispatching Policy

Definition 6.2. *Under* ***Redundant-to-Idle-Queue*** *(RIQ), each arriving job queries* $\mathbf{d}$ *servers chosen uniformly at random without replacement. If all* $\mathbf{d}$ *queried servers are busy, the job joins the queue at one of the* $\mathbf{d}$ *servers chosen at random. If* $0 < i \leq \mathbf{d}$ *queried servers are idle, the job enters service at all* i *idle servers, and its runtime is* $R(i) = X \cdot \min\{S_1, \ldots, S_i\}$.

Under RIQ, the system load is $\rho = \lambda \cdot \mathbf{E}[I \cdot R(I)]$, where I is a random variable denoting the number of servers on which a job runs. Here ρ is the average arrival rate multiplied by the average service capacity used per job. Forming an expression for load under RIQ is easier than for Redundancy-$\mathbf{d}$ because under RIQ any job that runs on multiple servers occupies all of its servers for the same duration. Nonetheless, it is not immediately obvious how to compute $\mathbf{E}[I \cdot R(I)]$; we address this at the end of Section 6.4.2.

[2] RIQ is motivated by the highly practical Join-Idle-Queue (JIQ) policy, under which each arrival is dispatched to a single idle queue, if one exists [53].

The analysis of RIQ relies on the Asymptotically Independent Idleness assumption, defined as follows:

Assumption 6.1. ***[Asymptotically Independent Idleness]*** *The servers are idle* $\mathbf{d}$*-wise independently; that is,* $\mathbf{P}\{\text{server } s_{\mathbf{d}} \text{ idle} \mid \text{servers } s_1, \ldots, s_{\mathbf{d}-1} \text{ idle}\} = \mathbf{P}\{\text{server } s_{\mathbf{d}} \text{ idle}\}$ *for all sets of* $\mathbf{d}$ *distinct servers* $s_1, \ldots, s_{\mathbf{d}}$.

In Section 6.4.5 we show that our analysis matches simulation when $\mathbf{d} \ll k$, indicating that using Assumption 6.1 leads to a reasonable approximation.

6.4.2 Analysis: Mean Response Time

Our goal in this section is to derive mean response time, $\mathbf{E}[T]$, under RIQ. We use Assumption 6.1 throughout our analysis.

We begin by conditioning on whether an arrival to the system finds any idle servers:

$$\begin{aligned}\mathbf{E}[T] &= \mathbf{P}\left\{\begin{matrix}\text{job finds}\\ \text{idle servers}\end{matrix}\right\} \cdot \mathbf{E}\left[T \,\middle|\, \begin{matrix}\text{job finds}\\ \text{idle servers}\end{matrix}\right] + \mathbf{P}\left\{\begin{matrix}\text{job finds no}\\ \text{idle servers}\end{matrix}\right\} \cdot \mathbf{E}\left[T \,\middle|\, \begin{matrix}\text{job finds no}\\ \text{idle servers}\end{matrix}\right] \\ &= (1-\rho^{\mathbf{d}})\mathbf{E}[T \mid \text{job finds idle servers}] + \rho^{\mathbf{d}}\mathbf{E}[T \mid \text{job finds no idle servers}], \end{aligned} \tag{6.1}$$

where the second line is due to Assumption 6.1. We defer our derivation of ρ to the end of the section.

We first find $\mathbf{E}[T \mid \text{job finds idle servers}]$ by conditioning on the number of idle servers a job finds, given that it finds at least one idle server:

$$\begin{aligned}\mathbf{E}\left[T \,\middle|\, \begin{matrix}\text{job finds}\\ \text{idle servers}\end{matrix}\right] &= \sum_{i=1}^{\mathbf{d}} \mathbf{P}\left\{\begin{matrix}\text{job finds } i\\ \text{idle servers}\end{matrix} \,\middle|\, \begin{matrix}\text{job finds}\\ \text{idle servers}\end{matrix}\right\} \cdot \mathbf{E}[R(i)] \\ &= \sum_{i=1}^{\mathbf{d}} \frac{(1-\rho)^i \rho^{\mathbf{d}-i}\binom{\mathbf{d}}{i}}{1-\rho^{\mathbf{d}}} \mathbf{E}[R(i)]. \end{aligned} \tag{6.2}$$

To find $\mathbf{E}[T \mid \text{job finds no idle servers}]$, we observe that when a job finds no idle servers it joins the queue at a single server. That server has the following properties:

1. When the server is idle, arrivals form a Poisson process with rate $\lambda\mathbf{d}$. This is because the total arrival rate is λk, each arrival polls the server with probability $\frac{\mathbf{d}}{k}$, and every job that polls the server while it is idle runs on it.
2. When the server is busy, arrivals form a Poisson process with rate $\lambda' = \lambda\rho^{\mathbf{d}-1}$. This is because the total arrival rate is λk, each arrival polls the server with probability $\frac{\mathbf{d}}{k}$, and every job that polls the server while it is busy runs on it if all $\mathbf{d}-1$ other servers that the job polls are also busy (probability $\rho^{\mathbf{d}-1}$) and then the job chooses our particular server (probability $\frac{1}{\mathbf{d}}$).
3. All jobs that find the server idle have runtime R_0, where

$$R_0 = R(i) \quad \text{w.p. } (1-\rho)^{i-1}\rho^{\mathbf{d}-i}\binom{\mathbf{d}-1}{i-1}, \;\; 1 \le i \le \mathbf{d}. \tag{6.3}$$

   Here we use the probability that the arrival found $i-1$ *other* idle servers among its $\mathbf{d}-1$ other servers.

4. All jobs that find the server busy have runtime $R(1)$.

We call the system described above an M*/G/1/efs. Here M* denotes that the arrival rate depends on whether the system is idle, and "efs" indicates that the system has an exceptional first service (i.e., the first job served during a busy period has a different runtime distribution than all other jobs.)

To find $\mathbf{E}\left[T \mid \text{job finds no idle servers}\right]$, we first observe:

$$\mathbf{E}\left[T \,\middle|\, \begin{matrix}\text{job finds no} \\ \text{idle servers}\end{matrix}\right] = \mathbf{E}\left[R(1)\right] + \mathbf{E}\left[T_Q \mid \text{queueing}\right]^{\text{M}^*/\text{G}/1/\text{efs}} \tag{6.4}$$

$$\mathbf{E}\left[T_Q \mid \text{queueing}\right]^{\text{M}^*/\text{G}/1/\text{efs}} = \frac{\mathbf{E}\left[T_Q\right]^{\text{M}^*/\text{G}/1/\text{efs}}}{P_Q^{\text{M}^*/\text{G}/1/\text{efs}}}, \tag{6.5}$$

where

$$P_Q^{\text{M}^*/\text{G}/1/\text{efs}} = \frac{\mathbf{E}\left[N_B\right]}{\mathbf{E}\left[N_B\right]+1} \tag{6.6}$$

is the probability that an arrival has to wait in the queue in an M*/G/1/efs and

$$\mathbf{E}\left[N_B\right] = \lambda' \cdot \frac{\mathbf{E}\left[R_0\right]}{1-\lambda'\mathbf{E}\left[R(1)\right]} \tag{6.7}$$

is the expected number of arrivals during an M*/G/1/efs busy period, and $\mathbf{E}\left[T_Q\right]^{\text{M}^*/\text{G}/1/\text{efs}}$ is given in Lemma 6.1.

Lemma 6.1. *The mean queueing time in an M*/G/1/efs where the first job experiences runtime R_0, all remaining jobs experience runtime $R(1)$, and the arrival rate while the system is busy is λ', is*

$$\begin{aligned}\mathbf{E}\left[T_Q\right]^{\text{M}^*/\text{G}/1/\text{efs}} &= \mathbf{E}\left[T_Q\right]^{\text{M}/\text{G}/1/\text{efs}} \\ &= \mathbf{E}\left[T\right]^{\text{M}/\text{G}/1/\text{efs}} - \mathbf{E}\left[S\right]^{\text{M}^*/\text{G}/1/\text{efs}},\end{aligned} \tag{6.8}$$

where

$$\mathbf{E}\left[T\right]^{\text{M}/\text{G}/1/\text{efs}} = \frac{(1-\lambda'\mathbf{E}\left[R(1)\right])(2\mathbf{E}\left[R_0\right]+\lambda'\mathbf{E}\left[R_0^2\right])+\lambda'^2\mathbf{E}\left[R_0\right]\mathbf{E}\left[R(1)^2\right]}{2(1-\lambda'\mathbf{E}\left[R(1)\right]+\lambda'\mathbf{E}\left[R_0\right])(1-\lambda'\mathbf{E}\left[R(1)\right])}$$

is derived in [74] and

$$\mathbf{E}\left[S\right]^{\text{M}^*/\text{G}/1/\text{efs}} = \frac{1}{\mathbf{E}\left[N_B\right]+1}\cdot\mathbf{E}\left[R_0\right] + \frac{\mathbf{E}\left[N_B\right]}{\mathbf{E}\left[N_B\right]+1}\cdot\mathbf{E}\left[R(1)\right],$$

where $\mathbf{E}\left[N_B\right]$ is given in (6.7).

Proof. The analysis used to find mean response time in an M/G/1/efs does not directly apply to the M*/G/1/efs because of our state-dependent arrival rate: the derivation of the transform of response time in an M/G/1/efs relies on PASTA, which does not apply in the M*/G/1/efs. However, from a *job's* perspective, the arrival rate while the system is idle does not affect response time. The arrival rate while the system is idle affects how close together busy periods occur, but does not affect any of the jobs during the busy period. Hence to derive response time, we can view the system as being an M/G/1/efs, the response time in which is derived in [74]. □

Combining equations (6.1)-(6.8) gives a closed-form expression for mean response time under RIQ in terms of ρ.

Our last step is to derive ρ, the probability that a server is busy. In Section 6.2 we defined $\rho = \lambda \cdot \mathbf{E}[I \cdot R(I)]$, where I is the number of servers on which a job runs. Using Assumption 6.1, we can write

$$\rho = \lambda \left(\sum_{i=1}^{\mathbf{d}} i \cdot (1-\rho)^i \rho^{\mathbf{d}-i} \binom{\mathbf{d}}{i} \mathbf{E}[R(i)] + \rho^{\mathbf{d}} \cdot \mathbf{E}[R(1)] \right). \tag{6.9}$$

Alternatively, we can compute ρ using renewal-reward theory, defining a cycle as the moment from when a server becomes idle until it is next idle. We have

$$\rho = \frac{\mathbf{E}[B]}{\mathbf{E}[B] + \frac{1}{\lambda \mathbf{d}}}, \tag{6.10}$$

where $\mathbf{E}[B] = \frac{\mathbf{E}[R_0]}{1-\lambda' \mathbf{E}[R(1)]}$ is the expected duration of a busy period and $\frac{1}{\lambda \mathbf{d}}$ is the mean interarrival time in the M*/G/1/efs when the server is idle.

Noting that $\mathbf{E}[R_0]$ is defined in terms of ρ, equation (6.10) gives us an expression for ρ in terms of itself. It is easy to verify algebraically that equations (6.9) and (6.10) are equivalent. When $\mathbf{d} = 2$, the equations are of degree 2 and can be solved exactly; for higher $\mathbf{d}$ we can solve for ρ numerically.

The system is stable as long as $\rho < 1$. However, without a closed-form expression for ρ, it is difficult to understand this stability condition intuitively. In Section 6.6 we derive an upper bound on mean response time under RIQ which gives us an alternative condition: the system is stable if $\lambda \cdot \mathbf{E}[R(1)] < 1$.

6.4.3 Analysis: Transform of Response Time

Our approach in Section 6.4.2 also allows us to find the Laplace transform of response time, $\widetilde{T}(s)$. As before, we begin by conditioning on whether an arrival to the system finds any idle servers:

$$\widetilde{T}(s) = (1-\rho^{\mathbf{d}}) \widetilde{T}\left(s \,\middle|\, \begin{matrix} \text{job finds} \\ \text{idle servers} \end{matrix} \right) + \rho^{\mathbf{d}} \widetilde{T}\left(s \,\middle|\, \begin{matrix} \text{job finds no} \\ \text{idle servers} \end{matrix} \right).$$

We first find $\widetilde{T}(s \mid \text{job finds idle servers})$ by conditioning on the number of idle servers a job finds, given that it finds at least one idle server:

$$\widetilde{T}\left(s \,\middle|\, \begin{matrix} \text{job finds} \\ \text{idle servers} \end{matrix} \right) = \sum_{i=1}^{\mathbf{d}} \frac{(1-\rho)^i \rho^{\mathbf{d}-i} \binom{\mathbf{d}}{i}}{1-\rho^{\mathbf{d}}} \widetilde{R(i)}(s).$$

Next we need to find $\widetilde{T}(s \mid \text{job finds no idle server})$. We do this using the M*/G/1/efs defined in Section 6.4.2:

$$\widetilde{T}\left(s \,\middle|\, \begin{matrix} \text{job finds no} \\ \text{idle servers} \end{matrix} \right) = \widetilde{R(1)}(s) \cdot \widetilde{T_Q}(s \mid \text{queueing})^{\text{M}^*/\text{G}/1/\text{efs}}$$

$$\widetilde{T_Q}(s \mid \text{queueing})^{\text{M}^*/\text{G}/1/\text{efs}} = \frac{\widetilde{T}(s)^{\text{M}^*/\text{G}/1/\text{efs}} - (1 - P_Q^{\text{M}^*/\text{G}/1/\text{efs}}) \cdot \widetilde{R_0}(s)}{P_Q^{\text{M}^*/\text{G}/1/\text{efs}} \cdot \widetilde{R(1)}(s)},$$

where $P_Q^{\text{M}^*/\text{G}/1/\text{efs}}$ is given in (6.6) and $\widetilde{T}(s)^{\text{M}^*/\text{G}/1/\text{efs}}$ is given in Lemma 6.2.

Lemma 6.2. *The transform of response time in an M*/G/1/efs where the first job experiences runtime R_0, all remaining jobs experience runtime $R(1)$, and the arrival rate while the server is busy is λ' is*

$$\widetilde{T}(s)^{\text{M}^*/\text{G}/1/\text{efs}} = \widetilde{T}(s)^{\text{M}/\text{G}/1/\text{efs}}$$
$$\overset{[74]}{=} \frac{1 - \lambda' \mathbf{E}[R(1)]}{1 - \lambda' \mathbf{E}[R(1)] + \lambda' \mathbf{E}[R_0]} \cdot \frac{(\lambda' - s)\widetilde{R_0}(s) - \lambda' \widetilde{R(1)}(s)}{\lambda' - s - \lambda' \widetilde{R(1)}(s)}.$$

Proof. The equivalence between response time in the M*/G/1/efs and in the M/G/1/efs follows the same reasoning as in the proof of Lemma 6.1. □

6.4.4 Cancellation Costs

When a job that is running on multiple servers completes service on one of these servers, all of its remaining copies must be cancelled. We model the time it takes to cancel a copy as taking deterministic time Z. In practice, if a job is running on i servers then only $i - 1$ of these servers need to incur a cancellation cost. To simplify the analysis slightly, we assume that all i servers incur the cancellation cost; this slightly increases our approximation for mean response time.

To include a cancellation time, we redefine R_0 from (6.3):

$$R_0 = \begin{cases} R(1) & \text{w.p. } \rho^{\mathbf{d}-1} \\ R(i) + Z & \text{w.p. } (1-\rho)^{i-1}\rho^{\mathbf{d}-1}\binom{\mathbf{d}-1}{i-1}, \quad 1 < i \leq \mathbf{d}. \end{cases}$$

The rest of the analysis for both the mean and the transform of response time remains the same as above. It is easy to make Z a non-deterministic random variable if desired.

6.4.5 Quality of Approximation

Our analysis relies on the assumption that the servers are idle independently. This is not true in general: if $\mathbf{d}$ is very close to k then a single job can cause nearly all of the servers to become busy. Nonetheless, by comparing our analytical approximation to simulation results, we observe that our approximation matches simulation quite well provided $\mathbf{d}$ is sufficiently small relative to k. Figure 6.3 compares mean response time obtained from our analysis to that obtained via simulation in a system where $\mathbf{d} = 20$, $\lambda = 0.7$, inherent job sizes are highly variable, and S follows the Dolly(1,12) server slowdown distribution. As k increases, our analysis becomes more accurate; when $k = 1000$ the analysis is within 1% of simulation. While we only show results for one set of parameters, when $\mathbf{d}$ and λ are lower the simulation results converge to the analytical results even more quickly.

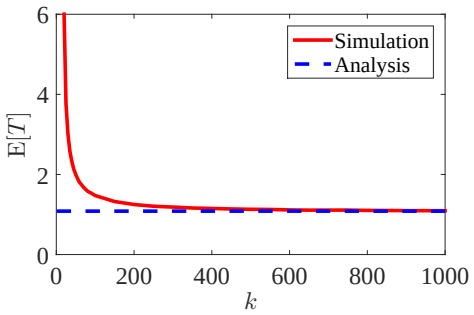


Figure 6.3: Mean response time under RIQ from analysis (dashed blue line) and simulation (solid red line, 99% confidence intervals are within the line) when $\lambda = 0.7$, $\mathbf{d} = 20$, $X \sim \mathrm{H}_2$ with $\mathbf{E}[X] = \frac{1}{4.7}$ and $C_X^2 = 10$, and $S \sim \mathrm{Dolly}(1, 12)$. When $k = 1000$, the analysis is within 1% of simulation.

6.5 Results: $\mathbf{d} \ll k$

In this section we evaluate the performance of RIQ using our analysis from Section 6.4. Throughout the section, $\mathbf{E}[R(1)] = 1$, $k = 1000$ servers, S follows the Dolly(1,12) distribution (Table 6.1, $\mathbf{E}[S] = 4.7$), and X follows a two-phase hyperexponential distribution (H_2) with balanced means:

$$X \sim \begin{cases} \mathrm{Exp}(\mu_1) & \text{w.p. } p \\ \mathrm{Exp}(\mu_2) & \text{w.p. } 1-p \end{cases},$$

where $\frac{p}{\mu_1} = \frac{1-p}{\mu_2}$.

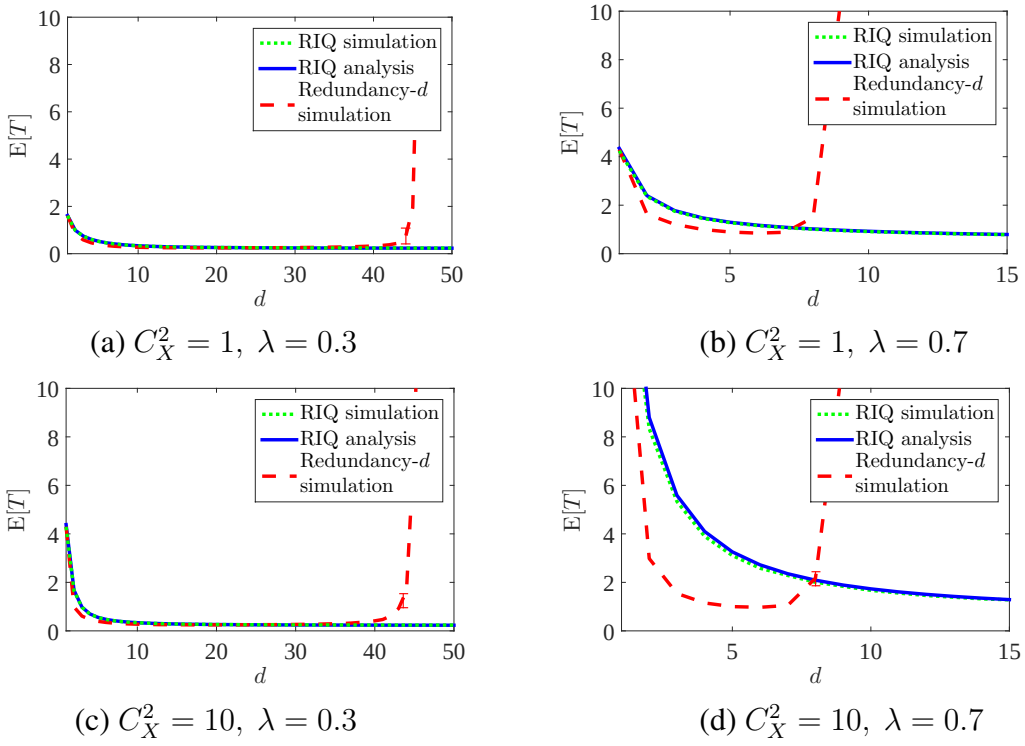
We consider only the $\mathbf{d} \ll k$ regime, in which our analysis provides a close approximation for performance under RIQ. In Section 6.6 we study what happens when $\mathbf{d}$ is close to k.

In Figure 6.4 we compare RIQ (our analysis in Section 6.4 matches simulation) to Redundancy-$\mathbf{d}$ (simulated) at both low ($\lambda = 0.3$) and high ($\lambda = 0.7$) arrival rate when the inherent job size X has different squared coefficients of variation, C_X^2, and the cancellation time is $Z = 0$. At low arrival rate and low $\mathbf{d}$ both policies perform similarly, which is unsurprising since under both policies many jobs find idle servers. At high arrival rate, when $\mathbf{d}$ is low Redundancy-$\mathbf{d}$ outperforms RIQ because Redundancy-$\mathbf{d}$ allows *all* jobs to benefit from creating multiple copies, not just jobs finding multiple idle servers, and this outweighs any pain caused by the extra load added by these copies. However, the system becomes unstable under Redundancy-$\mathbf{d}$ at high values of $\mathbf{d}$, whereas mean response time under RIQ continues to decrease as a function of $\mathbf{d}$.

Under both RIQ and Redundancy-$\mathbf{d}$, mean response time increases as the cancellation time Z increases (see Figure 6.5). Under Redundancy-$\mathbf{d}$, as Z increases the system becomes unstable at lower values of $\mathbf{d}$. Under RIQ, the system remains stable, indicating that RIQ is *more robust* than Redundancy-$\mathbf{d}$.

Results are more dramatic at the tail of response time, which in computer systems is often a more important metric than the mean. We numerically invert the transform derived analytically in Section 6.4 to obtain the c.d.f. of response time using the procedure described in [1].[3] As $\mathbf{d}$

[3]Thanks to Jan-Pieter Dorsman for help with the numerical inversion.



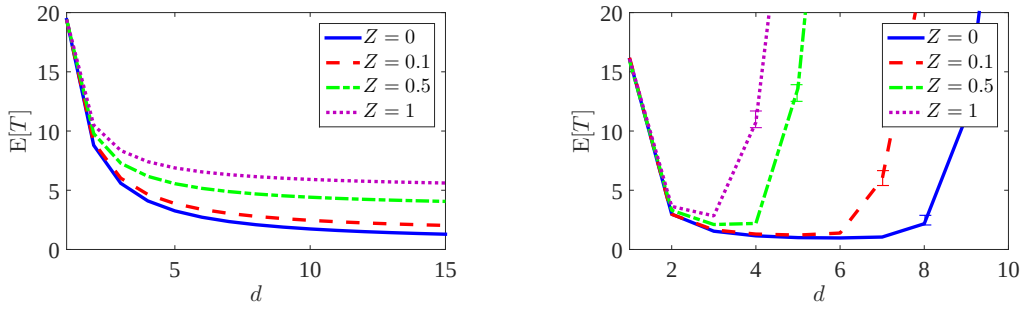
(a) $C_X^2 = 1,\ \lambda = 0.3$ (b) $C_X^2 = 1,\ \lambda = 0.7$

(c) $C_X^2 = 10,\ \lambda = 0.3$ (d) $C_X^2 = 10,\ \lambda = 0.7$

Figure 6.4: $\mathbf{E}[T]$ vs. $\mathbf{d}$ under Redundancy-$\mathbf{d}$ (from simulation) and RIQ (from both simulation and analysis) for $X \sim \mathrm{H}_2$ with $\mathbf{E}[X] = \frac{1}{4.7}$ and $C_X^2 = 1$ (top row) and $C_X^2 = 10$ (bottom row), where $\mathbf{E}[R(1)] = \mathbf{E}[X] \cdot \mathbf{E}[S] = 1$ and $S \sim$ Dolly(1,12), under low ($\lambda = 0.3$, left) and high ($\lambda = 0.7$, right) arrival rate. The simulations have $k = 1000$ servers; 95% confidence intervals are within the line except where shown. At low arrival rate, RIQ and Redundancy-$\mathbf{d}$ perform similarly, but at high arrival rate Redundancy-$\mathbf{d}$ becomes unstable when $\mathbf{d}$ is large, whereas RIQ continues to achieve low $\mathbf{E}[T]$.

increases, the 95th percentile of response time, T_{95}, drops much more steeply than $\mathbf{E}[T]$ (see Figure 6.6), indicating that RIQ successfully overcomes the negative effects of the variable server slowdowns. In fact, all percentiles of response time are much lower at $\mathbf{d} = 5$ than at $\mathbf{d} = 1$ (see Figure 6.7).

The trends are similar when the server slowdown follows distributions other than the Dolly$(1, 12)$ distribution. In Figure 6.8 we show mean response time as a function of $\mathbf{d}$ under Redundancy-$\mathbf{d}$ and RIQ when S follows the Google Bimodal distribution [22] (see Section 6.2). With Bimodal slowdown, Redundancy-$\mathbf{d}$ is unstable at lower values of $\mathbf{d}$ than with the more uniform Dolly$(1, 12)$ distribution, but $\mathbf{E}[T]$ under both RIQ and Redundancy-$\mathbf{d}$ follows the same qualitative shape regardless of the distribution of S.



(a) RIQ

(b) Redundancy-d

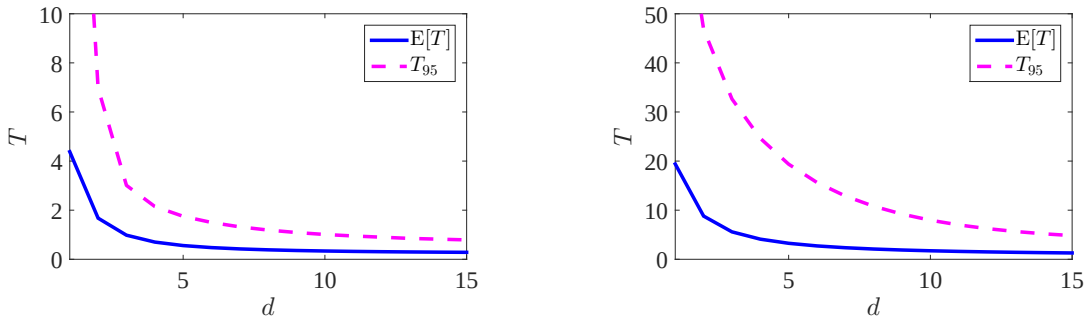
Figure 6.5: Mean response time as a function of $\mathbf{d}$ under (a) RIQ (from analysis) and (b) Redundancy-$\mathbf{d}$ (simulated with $k = 1000$ servers, 95% confidence intervals are within the line except where shown), $\lambda = 0.7$, job sizes are $X \sim \mathrm{H}_2$ with $\mathbf{E}[X] = \frac{1}{4.7}$ and $C_X^2 = 10$, and server slowdowns are $S \sim \mathrm{Dolly}(1, 12)$ for different constant cancellation times Z. As Z increases, mean response time increases under both RIQ and Redundancy-$\mathbf{d}$. Under Redundancy-$\mathbf{d}$, this leads to the system becoming unstable at lower values of $\mathbf{d}$, whereas RIQ remains stable for all values of Z.

6.6 Results: High $\mathbf{d}$

One might think that under RIQ mean response time should be lowest when $\mathbf{d} = k$, since in the $\mathbf{d} \ll k$ regime (Sections 6.4 and 6.5) we saw that $\mathbf{E}[T]$ decreases as $\mathbf{d}$ increases. However, this intuitively does not make sense. As a job runs on more servers, it achieves decreasing marginal runtime benefit from querying one more server (with respect to both minimizing server slowdown and increasing the likelihood of finding an idle server). Eventually, the job gets virtually no runtime benefit from increasing $\mathbf{d}$, and instead only adds load to the system. Hence as $\mathbf{d}$ gets high, we would expect to see a turning point at which mean response time will begin to increase because the extremely small runtime benefit to the replicated job is not enough to overcome the pain caused by the extra load.

While our analysis from Section 6.4 does not apply when $\mathbf{d}$ becomes high, in this section we demonstrate that, unlike Redundancy-$\mathbf{d}$, RIQ does not become unstable even as $\mathbf{d}$ gets large. We begin by discussing simulation results that show the worst-case behavior of RIQ (we omit Redundancy-$\mathbf{d}$ from our graphs in this section because we have already seen that Redundancy-$\mathbf{d}$ becomes unstable at relatively low $\mathbf{d}$). We then derive an analytical upper bound on mean response time under RIQ for any k, $\mathbf{d}$, S, X, and Z, which allows us to prove that RIQ does not become unstable as $\mathbf{d}$ gets large.

Figure 6.9 shows mean response time as a function of $\mathbf{d}$ when $k = 1000$, inherent job sizes are hyperexponential with $C_X^2 = 10$, and $S \sim \mathrm{Dolly}(1, 12)$. Surprisingly, we see that the system is never unstable, even when $\mathbf{d} = k$. In Section 6.6.1 we explain this intuitively, and we formalize the stability conditions under RIQ in Section 6.6.2.



(a) $\lambda = 0.3$ (b) $\lambda = 0.7$

Figure 6.6: Mean (solid blue line) and 95th percentile (dashed pink line) of response time, T, (via analysis) under RIQ when $X \sim \mathrm{H}_2$ with $\mathbf{E}[X] = \frac{1}{4.7}$ and $C_X^2 = 10$, $S \sim \mathrm{Dolly}(1, 12)$, $Z = 0$, and (a) $\lambda = 0.3$ and (b) $\lambda = 0.7$. As $\mathbf{d}$ increases, both the mean and the 95th percentile of response time decrease; the improvement is more pronounced in the tail.

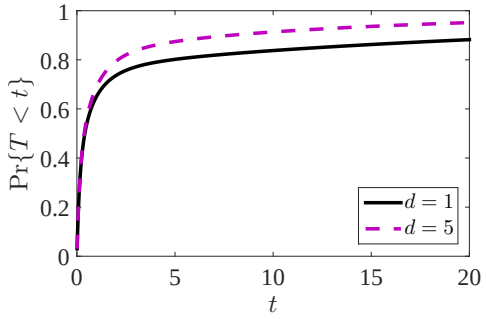


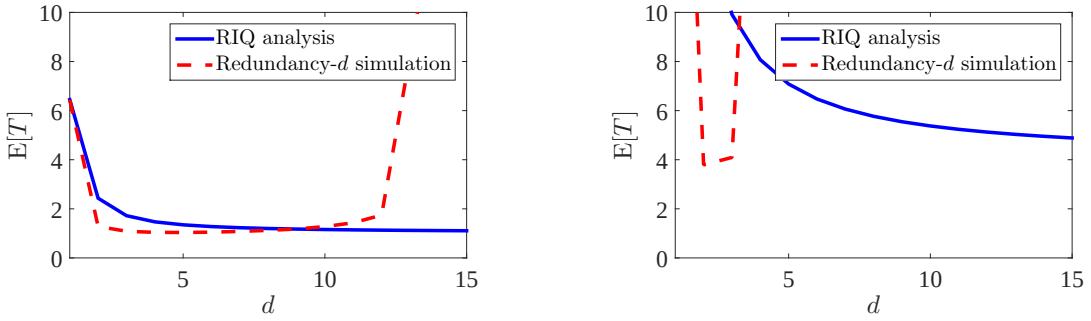
Figure 6.7: Distribution of response time under RIQ when $\mathbf{d} = 1$ (solid black line) and $\mathbf{d} = 5$ (dashed purple line) when $S \sim \mathrm{Dolly}(1, 12)$, $X \sim \mathrm{H}_2$ with $\mathbf{E}[X] = \frac{1}{4.7}$ and $C_X^2 = 10$, and $\lambda = 0.7$.

6.6.1 Intuition for why RIQ is Stable

One might think that as each job makes more copies, the added work causes the servers to be always busy, thereby driving queue lengths to infinity. Indeed, we see in Figure 6.9 that $\rho \to 1$ as $\mathbf{d} \to k$. Nonetheless, this does not cause the system to become unstable because RIQ only allows jobs to replicate when there are idle servers. Effectively, whenever a server goes idle we can think of it as being given some extra work to do, where this extra work is some job's replicated copy. The extra work causes the server to be always busy – sending ρ to 1 – but it affects the queue length like a vacation time, which cannot cause instability.

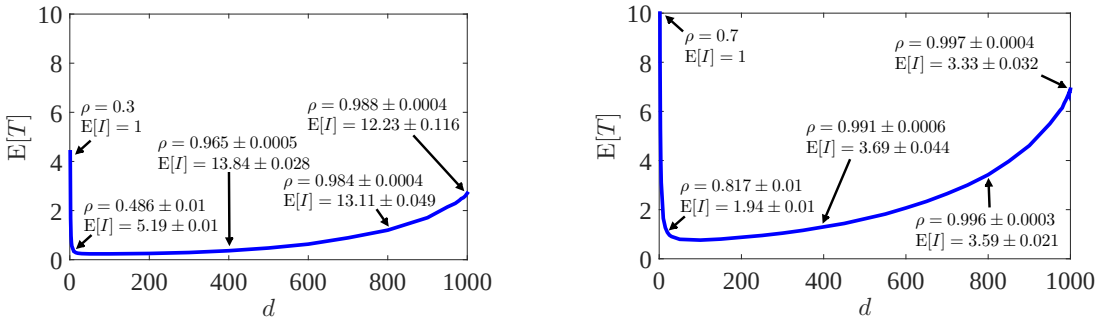
6.6.2 Formal Upper Bound on Mean Response Time under RIQ

Consider a system with k servers (unlike in the analysis in Section 6.4, here we do not need to assume k is large). Our dispatching policy is RIQ, and $\mathbf{d}$ may take any value $0 < \mathbf{d} \leq k$. We



(a) $\lambda = 0.3$ (b) $\lambda = 0.7$

Figure 6.8: Both RIQ and Redundancy-$\mathbf{d}$ exhibit similar trends to their performance when $S \sim \text{Dolly}(1, 12)$ under alternative S distributions. Here we show results for $S \sim \text{Bimodal}(1, 14; 0.99)$ ($\mathbf{E}[S] = 1.13$) when $X \sim \text{H}_2$ with $\mathbf{E}[X] = \frac{1}{1.13}$ and $C_X^2 = 10$, and $Z = 0$, for (a) $\lambda = 0.3$ and (b) $\lambda = 0.7$. RIQ results are from analysis; Redundancy-$\mathbf{d}$ is simulated (95% confidence intervals are within the line).



(a) $\lambda = 0.3$ (b) $\lambda = 0.7$

Figure 6.9: Mean response time under RIQ (simulated) as a function of $\mathbf{d}$ when $k = 1000$, $X \sim \text{H}_2$ with $\mathbf{E}[X] = \frac{1}{4.7}$ and $C_X^2 = 10$, $S \sim \text{Dolly}(1, 12)$, and (a) $\lambda = 0.3$, (b) $\lambda = 0.7$. While mean response time increases as $\mathbf{d}$ gets close to k, even when $\mathbf{d} = k$ mean response time is lower than when $\mathbf{d} = 1$. The lines are annotated with the values of ρ and $\mathbf{E}[I]$ (the expected number of copies per job) at several different values of $\mathbf{d}$. 99% confidence intervals are within the line for $\mathbf{E}[T]$ and are explicitly shown for ρ and $\mathbf{E}[I]$.

allow S and X to follow any distribution with finite first and second moments. As defined in Section 6.2, a job's runtime on a single server is $R(1) = S \cdot X$. The arrival rate λ can be anything such that $\lambda \cdot \mathbf{E}[R(1)] < 1$.

Theorem 6.1. *Assume that $\lambda \cdot \mathbf{E}[R(1)] < 1$. Then the response time under RIQ is upper bounded by the response time in an M/G/1/vacation queue with arrival rate λ, runtime $G = R(1) = S \cdot X$, and vacation time $R(1) + Z$:*

$$\begin{aligned}\mathbf{E}[T]^{\text{RIQ}} &\leq \mathbf{E}[T]^{\text{M/G/1/vacation}} \\ &= \mathbf{E}[T]^{\text{M/G/1}} + \mathbf{E}[(R(1) + Z)_e].\end{aligned}$$

where $(R(1) + Z)_e$ denotes the excess of $R(1) + Z$.

Proof. We will begin by expressing mean response time under RIQ by conditioning on whether a tagged arrival to the system finds any idle servers. We have

$$\begin{aligned}\mathbf{E}[T]^{\text{RIQ}} = &\mathbf{P}\left\{\begin{matrix}\text{tagged job finds}\\ \text{idle servers}\end{matrix}\right\} \cdot \mathbf{E}\left\{T \;\middle|\; \begin{matrix}\text{tagged job finds}\\ \text{idle servers}\end{matrix}\right\} \\ &+ \mathbf{P}\left\{\begin{matrix}\text{tagged job finds}\\ \text{no idle servers}\end{matrix}\right\} \cdot \mathbf{E}\left\{T \;\middle|\; \begin{matrix}\text{tagged job finds}\\ \text{no idle servers}\end{matrix}\right\}.\end{aligned}$$

When the tagged job arrives to the system, if it finds i idle servers it enters service on all of these servers. For all i, we have

$$\begin{aligned}\mathbf{E}\left[T \;\middle|\; \begin{matrix}\text{tagged job finds}\\ i \text{ idle servers}\end{matrix}\right] &= \mathbf{E}[R(i)] \\ &\leq \mathbf{E}[R(1)] \\ &\leq \mathbf{E}[R(1)] + \mathbf{E}\left[T^Q \;\middle|\; \begin{matrix}\text{tagged job finds}\\ \text{no idle servers}\end{matrix}\right] \\ &= \mathbf{E}\left[T \;\middle|\; \begin{matrix}\text{tagged job finds}\\ \text{no idle servers}\end{matrix}\right].\end{aligned}$$

Hence we have the following upper bound:

$$\mathbf{E}[T]^{\text{RIQ}} \leq \mathbf{E}[T|\text{tagged job finds no idle servers}]. \tag{6.11}$$

Given that a tagged arrival found no idle servers, it joins the queue at a randomly chosen server with the following properties:

1. When the server is busy, jobs arrive with rate $k\lambda \cdot \frac{d}{k} \cdot \mathbf{P}\{\text{job finds other } d-1 \text{ servers busy}\} \cdot \frac{1}{d} \leq \lambda$.
2. All jobs that arrive to the server while it is busy experience runtime $R(1)$.
3. A job that arrives to the server while it is idle, and finds $i-1$ other idle servers, experiences runtime $R(i) \leq_{st} R(1) + Z$.

Since (6.19) forces all jobs to have a non-zero queueing time in our upper bound, the first job in the busy period (the job that arrives to the server while it is idle) can be viewed as a "dummy" job that does not contribute to response time. Every time a server goes idle, we can imagine that

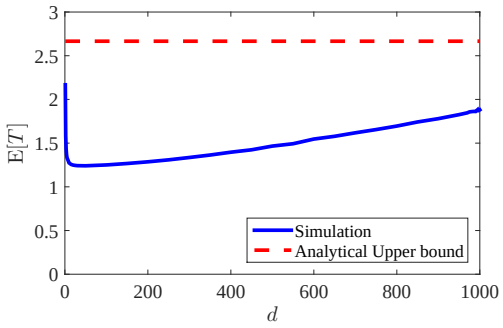


Figure 6.10: Solid line shows mean response time under RIQ (simulated, 99% confidence intervals are within the line) when for all i, $R(i) = S \cdot X = 1$ is deterministic and $\lambda = 0.7$; dashed line shows the analytical upper bound. While the bound is not tight, it shows that mean response time under RIQ cannot become extremely high. In particular, the upper bound tells us that RIQ does not become unstable as $\mathbf{d}$ grows large.

the server is forced to work on a "dummy" job with size $R(1) + Z$, so the server is always busy when the next real job arrives. This exactly describes an M/G/1/vacation system with arrival rate λ, runtime $R(1)$, and vacation time $R(1) + Z$. □

Theorem 6.2. *Under RIQ, the system is stable if* $\lambda \mathbf{E}[R(1)] < 1$.

Proof. The stability region for the M/G/1/vacation is the same as that for the M/G/1, namely $\lambda \cdot \mathbf{E}[R(1)] < 1$. If the M/G/1/vacation is stable, then RIQ is stable since the M/G/1/vacation gives an upper bound on RIQ. □

Figure 6.10 compares our upper bound to simulation results when $k = 1000$ and runtimes are deterministically equal to 1 and there is no cancellation cost ($Z = 0$). We consider deterministic S because, intuitively, this shows the worst possible case for redundancy: with no server slowdown variability, redundancy offers no possible runtime benefit. The only benefit of RIQ is therefore that it helps jobs find idle servers on which to run. While increasing $\mathbf{d}$ helps jobs find idle servers, it also means that jobs create many copies that *only* add load to the system. Hence we would expect deterministic server slowdowns to yield the worst performance as $\mathbf{d}$ increases. As we increase the variability of inherent job size X, the difference between actual mean response time and the upper bound increases. However, for all distributions of X, the difference between the exact mean response time and the upper bound at $\mathbf{d} = 1$ (i.e., random dispatching; all jobs see an M/G/1 queue) is exactly the mean excess of $R(1) + Z$. Hence RIQ can never perform more than an additive constant worse than random dispatching. We observe from simulation that RIQ actually does much better at high $\mathbf{d}$ than at $\mathbf{d} = 1$ for nearly all parameter settings.

6.7 Improving Redundancy Policies

In this section we study several policy variations designed to improve performance relative to the baseline RIQ and Redundancy-$\mathbf{d}$ policies. In some cases, the analysis presented in Section 6.4

extends easily to the variations on RIQ. In other cases, we develop new response time analysis. When analysis is not feasible, we study performance via simulation.

6.7.1 JSQ: Better Dispatching for Jobs that Find No Idle Servers

One weakness of RIQ is that when a job finds no idle servers it is dispatched to a single queue chosen at random, which is known to yield poor performance. A superior dispatching policy is Join-the-Shortest-Queue (JSQ), under which each arriving job joins the queue containing the fewest jobs. A practical variant of JSQ is JSQ-$\mathbf{d}$, where each arriving job polls $\mathbf{d}$ queues chosen at random and joins the queue containing the fewest jobs among the $\mathbf{d}$ polled queues. JSQ-$\mathbf{d}$ is motivated by the fact that polling all k queues can be expensive. The RIQ+JSQ policy combines RIQ with JSQ-$\mathbf{d}$ dispatching.

Definition 6.3. *Under **RIQ+JSQ**, each arriving job polls $\mathbf{d}$ servers chosen uniformly at random without replacement. If $1 \leq i \leq \mathbf{d}$ of the servers are idle, the job enters service at all i idle servers. If all $\mathbf{d}$ servers are busy, the job joins the queue with the fewest jobs among the $\mathbf{d}$ polled queues.*

Observation 6.1. *RIQ+JSQ has the same stability region as RIQ, since the only difference between the two policies is that under RIQ+JSQ we achieve better load balancing of the jobs that do not find idle servers. Hence under RIQ+JSQ, the system is stable as long as $\lambda \cdot \mathbf{E}[X \cdot S] < 1$.*

Response Time Analysis

Our analysis of mean response time under RIQ+JSQ follows the approach used in [3]. We write a system of equations that describes the evolution of the RIQ+JSQ system, then use numerical methods to approximate the steady-state behavior. It is easy to modify the analysis presented below to include a cancellation time.

Let $Y_{n,i}(t, t+r)$ be the fraction of servers with at least n jobs in the queue (including the job in service, if there is one) at time t such that the job in service at time t is still in service at time $t+r$ and is running on exactly i servers. Let $Y_n(t, t+r) = \sum_{i=1}^{\mathbf{d}} Y_{n,i}(t, t+r)$.

For $n, i > 1$, there are two ways in which a server can contribute to $Y_{n,i}(t, t+r)$:

1. The queue length was at least n at time 0 and the same job remains in service on i servers during time interval $(0, t+r)$.
2. At some time $u < t$ an arrival caused the queue length to go from $n-1$ to n and the same job is in service on i servers during the time interval $(u, t+r)$.

When $i = 1$ there is a third case: a job completed service at time $u \leq t$ and left behind at least n jobs, and the next job stays in service during time $(u, t+r)$. Hence for $n > 1$,

$$\begin{aligned} Y_{n,i}(t, t+r) = Y_{n,i}(0, t+r) \\ + \lambda \int_{u=0}^{t} (Y_{n-1}(u,u)^{\mathbf{d}} - Y_n(u,u)^{\mathbf{d}}) \cdot \frac{Y_{n-1,i}(u, t+r) - Y_{n,i}(u, t+r)}{Y_{n-1}(u,u) - Y_n(u,u)} du \\ + I_{i=1} \int_{u=0}^{t} (-\partial_r Y_{n+1}(u,u)) \bar{G}(t+r-u) du, \end{aligned} \tag{6.12}$$

where $\bar{G}(x)$ is the probability that a job's runtime exceeds x and

$$-\partial_r Y_{n+1}(u,u) = \lim_{r \to 0} \frac{Y_{n+1}(u,u) - Y_{n+1}(u, u+r)}{r}$$

is the service completion rate of jobs in a queue with length at least $n+1$ at time u.

When $n = 1$ the second term, which corresponds to a new arrival, changes because now the arrival joins an idle queue and therefore runs on $1 \leq i \leq \mathbf{d}$ servers:

$$\begin{aligned} Y_{1,i}(t, t+r) = Y_{1,i}(0, t+r) \\ + i\lambda \int_{u=0}^{t} \binom{\mathbf{d}}{i} Y_1(u,u)^{\mathbf{d}-i} \cdot (1 - Y_1(u,u))^i \bar{G}^{(i)}(t+r-u)du \\ + I_{i=1} \int_{u=0}^{t} (-\partial_r Y_2(u,u))\bar{G}(t+r-u)du, \end{aligned} \tag{6.13}$$

where $\bar{G}^{(i)}(x)$ is the probability that a job's runtime exceeds x given that the job is running on exactly i servers.

Summing over i in (6.12), we find

$$\begin{aligned} Y_n(t, t+r) = Y_n(0, t+r) \\ + \lambda \int_{u=0}^{t} (Y_{n-1}(u,u)^{\mathbf{d}} - Y_n(u,u)^{\mathbf{d}}) \cdot \frac{Y_{n-1}(u, t+r) - Y_n(u, t+r)}{Y_{n-1}(u,u) - Y_n(u,u)} du \\ + \int_{u=0}^{t} (-\partial_r Y_{n+1}(u,u))\bar{G}(t+r-u)du, \end{aligned} \tag{6.14}$$

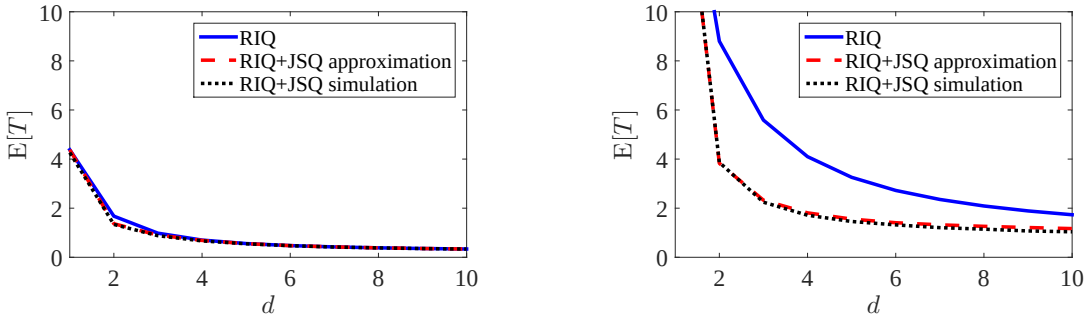
which corresponds to equation (8) in [3] when $\mathbf{d} = 2$. Summing over i in (6.13) yields

$$\begin{aligned} Y_1(t, t+r) = Y_1(0, t+r) \\ + \lambda \int_{u=0}^{t} \sum_{i=1}^{\mathbf{d}} \binom{\mathbf{d}}{i} Y_1(u,u)^{\mathbf{d}-i} \cdot (1 - Y_1(u,u))^i i \bar{G}^{(i)}(t+r-u)du \\ + I_{i=1} \int_{u=0}^{t} (-\partial_r Y_2(u,u))\bar{G}(t+r-u)du. \end{aligned} \tag{6.15}$$

We now use equations (6.14) and (6.15) to numerically estimate π_n, the steady-state probability that a server has at least n jobs (assuming the number of servers k is large). Note that $\pi_1 = \rho$ is the probability that a server is busy. We introduce a mesh of width δ and approximate $Y_n(t, r)$ for $t \in [0, \delta, 2\delta, \ldots, t_0]$ and $r \in [0, \delta, 2\delta, \ldots, r_0]$. Here r_0 is chosen such that $\bar{G}(r_0)$ is negligible, and t_0 is chosen dynamically so that $\sum_{n,r} |Y_n(t+\delta, t+\delta+r) - Y_n(t, t+r)|$ is sufficiently small, e.g., 10^{-10}. We also truncate the queue lengths to at most n_0 jobs.

We start with an empty system at time 0, so $Y_n(0, r) = 0$ for all n and r, and compute $Y_n(t+\delta, t+r+\delta)$ based on $Y_n(t, t+r)$ as follows:

$$\begin{aligned} Y_1(t+\delta, t+r+\delta) = Y_1(t, t+r+\delta) \\ + \lambda \int_{u=t}^{t+\delta} \sum_{i=1}^{\mathbf{d}} \binom{\mathbf{d}}{i} \cdot Y_1(u,u)^{\mathbf{d}-i} \cdot (1 - Y_1(u,u))^i i \bar{G}(t+r+\delta-u)du \\ + \int_{u=t}^{t+\delta} (-\partial_r Y_2(u,u))\bar{G}(t+r+\delta-u)du \end{aligned}$$



(a) $\lambda = 0.3$ (b) $\lambda = 0.7$

Figure 6.11: Comparing baseline RIQ and RIQ+JSQ from our approximation (dashed red line) and simulation (dotted black line; 95% confidence intervals are within the line). Here $S \sim \text{Dolly}(1, 12)$, $X \sim \text{H}_2$ with $\mathbf{E}[X] = \frac{1}{4.7}$ and $C_X^2 = 10$, $Z = 0$, and (a) $\lambda = 0.3$ and (b) $\lambda = 0.7$.

$$\approx Y_1(t, t+r+\delta) - (Y_2(t, t+\delta) - Y_2(t,t))\bar{G}(r+\frac{\delta}{2}) + \lambda\delta \sum_{i=1}^{\mathbf{d}} \binom{\mathbf{d}}{i} Y_1(t,t)^{\mathbf{d}-i}(1-Y_1(t,t))^i i \bar{G}^{(i)}(r+\frac{\delta}{2})$$

$$Y_n(t+\delta, t+r+\delta) \approx Y_n(t, t+r+\delta) - (Y_{n+1}(t, t+\delta) - Y_{n+1}(t,t))\bar{G}(r+\frac{\delta}{2}) + \lambda\delta \frac{Y_{n-1}(t,t)^{\mathbf{d}} - Y_n(t,t)^{\mathbf{d}}}{Y_{n-1}(t,t) - Y_n(t,t)}(Y_{n-1}(t, t+r) - Y_n(t, t+r)).$$

The numerical scheme now uses these two approximations in the following manner.[4] First, we define

$$\bar{Y}_n(t+\delta, t+r+\delta) = \hat{Y}_n(t, t+r+\delta) - (\hat{Y}_{n+1}(t, t+\delta) - \hat{Y}_{n+1}(t,t))\bar{G}(r+\frac{\delta}{2}) + \lambda\delta \frac{\hat{Y}_{n-1}(t,t)^{\mathbf{d}} - \hat{Y}_n(t,t)^{\mathbf{d}}}{\hat{Y}_{n-1}(t,t) - \hat{Y}_n(t,t)}(\hat{Y}_{n-1}(t, t+r) - \hat{Y}_n(t, t+r)),$$

and subsequently we compute $\hat{Y}_n(t+\delta, t+r+\delta)$ as

$$\hat{Y}_n(t+\delta, t+r+\delta) = \hat{Y}_n(t, t+r+\delta) - (\bar{Y}_{n+1}(t+\delta, t+2\delta) - \bar{Y}_{n+1}(t+\delta, t+\delta))\bar{G}(r+\frac{\delta}{2}) + \lambda\delta \frac{\bar{Y}_{n-1}(t+\delta, t+\delta)^{\mathbf{d}} - \bar{Y}_n(t+\delta, t+\delta)^{\mathbf{d}}}{\bar{Y}_{n-1}(t+\delta, t+\delta) - \bar{Y}_n(t+\delta, t+\delta)} \cdot (\bar{Y}_{n-1}(t+\delta, t+r+\delta) - \bar{Y}_n(t+\delta, t+r+\delta)).$$

A similar approach is used to compute $\hat{Y}_1(t+\delta, t+\delta+r)$.

[4]Thanks to Xingjie Li for helpful suggestions to stabilize the numerical scheme.

We now use $\hat{Y}_n(t,t)$ to approximate π_n. The expected number of jobs in the system, $\mathbf{E}[N]$, is

$$\mathbf{E}[N] = \sum_{n=1}^{\infty} \pi_n - \lambda \sum_{i=2}^{\mathbf{d}} \binom{\mathbf{d}}{i} \pi_1^{\mathbf{d}-i}(1-\pi_1)^i (i-1)\mathbf{E}\left[G^{(i)}\right],$$

where $\binom{\mathbf{d}}{i}\pi_1^{\mathbf{d}-i}(1-\pi_1)^i$ is the probability that a job runs on i servers. Note that the second term is included to avoid overcounting jobs that run on $i > 1$ servers. Finally, we find $\mathbf{E}[T]$ by applying Little's Law.

Validation of Approximation

We compare our computed π_n values with simulation; when $k = 1000$, $\mathbf{d} = 10$, $C_X^2 = 10$, $\lambda = 0.7$, and $S \sim \text{Dolly}(1, 12)$, and for $\delta = 0.2$ and $n_0 = 10$, the approximation is within 5% of simulation for π_1, π_2, and π_3 (queue lengths exceed 3 less than 0.5% of the time). The approximation is more accurate when λ, $\mathbf{d}$, C_X^2, and δ are lower and when k is higher.

Performance

We use our new analysis to compare RIQ+JSQ with baseline RIQ, assuming $\mathbf{d}$ is small (for both policies, this is the regime in which our analysis holds). When λ is low, RIQ+JSQ provides very little benefit over RIQ because most jobs find idle servers (see Figure 6.11). When λ is high, RIQ+JSQ beats RIQ because many jobs do not find idle servers and benefit from being dispatched via JSQ rather than randomly.

Note that there is not a single obvious definition of a policy that combines JSQ with Redundancy-$\mathbf{d}$; such a policy would require either polling $n > \mathbf{d}$ servers per job and replicating to the $\mathbf{d}$ shortest queues among the n, or polling $\mathbf{d}$ servers per job and replicating to the $n < \mathbf{d}$ shortest queues among the $\mathbf{d}$. Neither option will solve the stability problems exhibited by Redundancy-$\mathbf{d}$, hence we do not consider a JSQ variant of Redundancy-$\mathbf{d}$.

6.7.2 SMALL: Replicate Only Jobs with Small X

A major concern when replicating jobs is that running multiple copies of the same job adds load to the system. This is particularly true of jobs with a large inherent size X. When a large job runs on many servers, it clogs up these servers for a long time even if it experiences a small slowdown S on some server. This makes it harder for jobs with small X components to find any idle servers on which to run.

One way to prevent the small jobs from being blocked on many servers by large jobs is to allow only jobs with a small X component to create replicas. That is, we define a constant threshold x and only replicate jobs with inherent size $X \leq x$.

Definition 6.4. *Under **RIQ+SMALL**, if an arriving job has inherent size $X \leq x$ it polls $\mathbf{d}$ servers chosen uniformly at random without replacement. If $1 \leq i \leq \mathbf{d}$ of the polled servers are idle, the job replicates itself on all i idle servers. If none are idle, the job joins the queue at one of the $\mathbf{d}$ servers chosen uniformly at random. If the job has inherent size $X > x$, it is dispatched to a single server chosen uniformly at random.*

Definition 6.5. *Under **Redundancy**-d**+SMALL**, if a job has inherent size $X \leq x$ it replicates itself to* $\mathbf{d}$ *servers chosen uniformly at random, whereas if a job has inherent size $X > x$ it is dispatched to a single server chosen uniformly at random.*

Theorem 6.3. *Under RIQ+SMALL, the system is stable if $\lambda \cdot \mathbf{E}[X \cdot S] < 1$.*

Proof. As for the baseline RIQ policy, we will upper bound mean response time under RIQ+SMALL by the mean response time in an M/G/1/vacation system. We first express mean response time under RIQ+SMALL by conditioning on a job's inherent size. We have

$$\begin{aligned}\mathbf{E}[T]^{\text{RIQ+SMALL}} = &\mathbf{P}\{X \leq x\} \cdot \mathbf{E}[T|X \leq x] \\ &+ \mathbf{P}\{X > x\} \cdot \mathbf{E}[T|X > x].\end{aligned}$$

When a small job (with size $X \leq x$) arrives to the system, if it finds i idle servers it enters service on all of these servers. For all i, we have

$$\begin{aligned}\mathbf{E}[T|\text{job finds } i \text{ idle servers } \&\ X \leq x] &= \mathbf{E}[R(i)|X \leq x] \\ &\leq \mathbf{E}[R(1)|X \leq x] \\ &\leq \mathbf{E}[R(1)|X \leq x] + \mathbf{E}\left[T^Q|\text{job finds no idle servers}\right] \\ &= \mathbf{E}[T|\text{job finds no idle servers } \&\ X \leq x]. \end{aligned} \tag{6.16}$$

When a small job arrives to the system and does not find any idle servers, it joins a single queue chosen at random and experiences response time

$$\mathbf{E}[T|\text{job finds no idle servers } \&\ X \leq x]. \tag{6.17}$$

When a large job (with size $X > x$) arrives to the system, the job joins a single queue chosen uniformly at random. The job experiences response time

$$\begin{aligned}&\mathbf{E}[T|X > x] \\ &= \mathbf{E}\left[T \middle| \begin{matrix} X > x \text{ \& job} \\ \text{finds idle servers} \end{matrix}\right] \cdot \mathbf{P}\left\{\begin{matrix} X > x \text{ \& job} \\ \text{finds idle servers} \end{matrix}\right\} \\ &\quad + \mathbf{E}\left[T \middle| \begin{matrix} X > x \text{ \& job finds} \\ \text{no idle servers} \end{matrix}\right] \cdot \mathbf{P}\left\{\begin{matrix} X > x \text{ \& job finds} \\ \text{no idle servers} \end{matrix}\right\} \\ &\leq \mathbf{E}[T|\text{job finds no idle servers } \&\ X > x]. \end{aligned} \tag{6.18}$$

For both small and large jobs that find no idle servers, the job joins the queue at a randomly chosen server with the following properties:

1. When the server is busy, small jobs arrive with rate

$$k\lambda \mathbf{P}\{X \leq x\} \cdot \frac{d}{k} \cdot \mathbf{P}\left\{\begin{matrix} \text{job finds } d-1 \\ \text{other servers busy} \end{matrix}\right\} \cdot \frac{1}{d} = \lambda \mathbf{P}\{X \leq x\} \mathbf{P}\left\{\begin{matrix} \text{job finds } d-1 \\ \text{other servers busy} \end{matrix}\right\}$$

and large jobs arrive with rate

$$k\lambda \mathbf{P}\{X > x\} \cdot \frac{1}{k} = \lambda \mathbf{P}\{X > x\}.$$

2. Small jobs that arrive to the server while it is busy experience runtime $R(1|X \leq x)$, and large jobs that arrive to the server while it is busy experience runtime $R(1|X > x)$.
3. A small job that arrives to the server while it is idle, and finds $i-1$ other idle servers, experiences runtime $R(i|X \leq x) \leq_{st} R(1|X \leq x) + Z$. A large job that arrives to the server while it is idle experiences runtime $R(1|X > x)$.

Since (6.16), (6.17), and (6.18) force all jobs to have a non-zero queueing time in our upper bound, the first job in the busy period (the job that arrives to the server while it is idle) can be viewed as a "dummy" job that does not contribute to response time. Every time a server goes idle, we can imagine that the server is forced to work on a "dummy" job, so the server is always busy when the next real job arrives. This dummy job has size

$$\begin{aligned} R_0 &= \begin{cases} R(1|X > x) & \text{w.p. } \frac{\mathbf{P}\{X>x\}}{\mathbf{P}\{X>x\}+d\cdot\mathbf{P}\{X\leq x\}} \\ R(i)+Z|X \leq x & \text{w.p. } \frac{d\cdot\mathbf{P}\{X\leq x\}}{\mathbf{P}\{X>x\}+d\cdot\mathbf{P}\{X\leq x\}} \\ & \quad \cdot\mathbf{P}\left\{\begin{matrix}\text{job finds } i-1 \\ \text{other servers idle}\end{matrix}\right\} \end{cases} \\ &\leq_{st} \begin{cases} R(1|X > x) & \text{w.p. } \frac{\mathbf{P}\{X>x\}}{\mathbf{P}\{X>x\}+d\cdot\mathbf{P}\{X\leq x\}} \\ R(1)+Z|X \leq x & \text{w.p. } \frac{d\cdot\mathbf{P}\{X\leq x\}}{\mathbf{P}\{X>x\}+d\cdot\mathbf{P}\{X\leq x\}} \end{cases} \\ &\leq_{st} R(1)+Z \end{aligned}$$

The system that runs a "dummy" job of size $R(1)+Z$ every time it becomes idle gives an upper bound on mean response time in our original system, and is exactly an M/G/1/vacation system with vacation time $R(1)+Z$ and arrival rates and runtimes as stated in items 1 and 2 above. We can further upper bound the M/G/1/vacation system by simply increasing the arrival rate of small jobs to $\lambda\mathbf{P}\{X \leq x\}$. We now have an M/G/1/vacation system with arrival rate λ, runtime $R(1)$, and vacation time $R(1)+Z$. This system is stable provided that $\lambda \cdot \mathbf{E}[R(1)] = \lambda \cdot \mathbf{E}[X \cdot S] < 1$. Since mean response time in the M/G/1/vacation system upper bounds mean response time in our original system under RIQ+SMALL, we also have that our system is stable under RIQ+SMALL, for all d, if $\lambda \cdot \mathbf{E}[X \cdot S] < 1$. □

Response Time Analysis: RIQ+SMALL

The analysis of response time under RIQ+SMALL follows the same approach as that under the baseline RIQ policy. Here we show the analysis for the Laplace transform of response time, following the approach in Section 6.4.3 We begin by conditioning on a job's inherent size:

$$\widetilde{T}(s) = \widetilde{T}(s \mid X \leq x) \cdot \mathbf{P}\{X \leq x\} + \widetilde{T}(s \mid X > x) \cdot \mathbf{P}\{X > x\}.$$

To find $\widetilde{T}(s \mid X > x)$, observe that any job with inherent size larger than x is dispatched randomly to a single M*/G/1/efs. As before, the exceptional first service comes from the fact that the first job in the busy period may run on anywhere between 1 and d servers. However, now the job's response time is *not* simply the response time in an M*/G/1/efs because jobs with size $X > x$ are less likely to be the first job in a busy period than jobs with size $X < x$. So we condition on whether the job finds the server idle:

$$\widetilde{T}(s \mid X > x) = \widetilde{T}\left(s \,\middle|\, \begin{matrix} X > x \,\&\, \text{job} \\ \text{finds idle servers}\end{matrix}\right) \cdot \mathbf{P}\left\{\begin{matrix}\text{job finds} \\ \text{idle servers}\end{matrix}\right\}$$

$$+\widetilde{T}\left(s\,\middle|\begin{matrix}X>x\ \&\ \text{job finds}\\ \text{no idle servers}\end{matrix}\right)\cdot\mathbf{P}\left\{\begin{matrix}\text{job finds no}\\ \text{idle servers}\end{matrix}\right\}$$
$$=\widetilde{R(1)}(s\mid X>x)\cdot(1-\rho)+\widetilde{T_Q}(s\mid\text{queueing})^{\text{M}^*/\text{G}/1/\text{efs}}\cdot\widetilde{R(1)}(s\mid X>x)\cdot\rho.$$

The arrival and service rates in this M*/G/1/efs differ from those in the original RIQ analysis, and are discussed below.

To find $\widetilde{T}(s\mid X\le x)$, we follow the original RIQ analysis:

$$\widetilde{T}(s\mid X\le x)=\widetilde{T}\left(s\,\middle|\begin{matrix}X\le x\ \&\ \text{job}\\ \text{finds idle servers}\end{matrix}\right)\cdot\mathbf{P}\left\{\begin{matrix}\text{job finds}\\ \text{idle servers}\end{matrix}\right\}$$
$$+\widetilde{T}\left(s\,\middle|\begin{matrix}X\le x\ \&\ \text{job finds}\\ \text{no idle servers}\end{matrix}\right)\cdot\mathbf{P}\left\{\begin{matrix}\text{job finds no}\\ \text{idle servers}\end{matrix}\right\},$$

where $\mathbf{P}\left\{\text{job finds idle servers}\right\}=1-\rho^{\mathbf{d}}$ and ρ is computed numerically using the same approach as in the original RIQ analysis and the M*/G/1/efs parameterization given below.

Similarly to the original analysis, we have:

$$\widetilde{T}\left(s\,\middle|\begin{matrix}X\le x\ \&\ \text{job}\\ \text{finds idle servers}\end{matrix}\right)=\sum_{i=1}^{\mathbf{d}}\mathbf{P}\left\{\begin{matrix}\text{job finds } i\\ \text{idle servers}\end{matrix}\,\middle|\,\begin{matrix}\text{job finds}\\ \text{idle servers}\end{matrix}\right\}\cdot\widetilde{R(i)}(s\mid X\le x)$$
$$=\sum_{i=1}^{\mathbf{d}}\frac{(1-\rho)^i\rho^{\mathbf{d}-i}\binom{\mathbf{d}}{i}}{1-\rho^{\mathbf{d}}}\cdot\widetilde{R(i)}(s\mid X\le x),$$

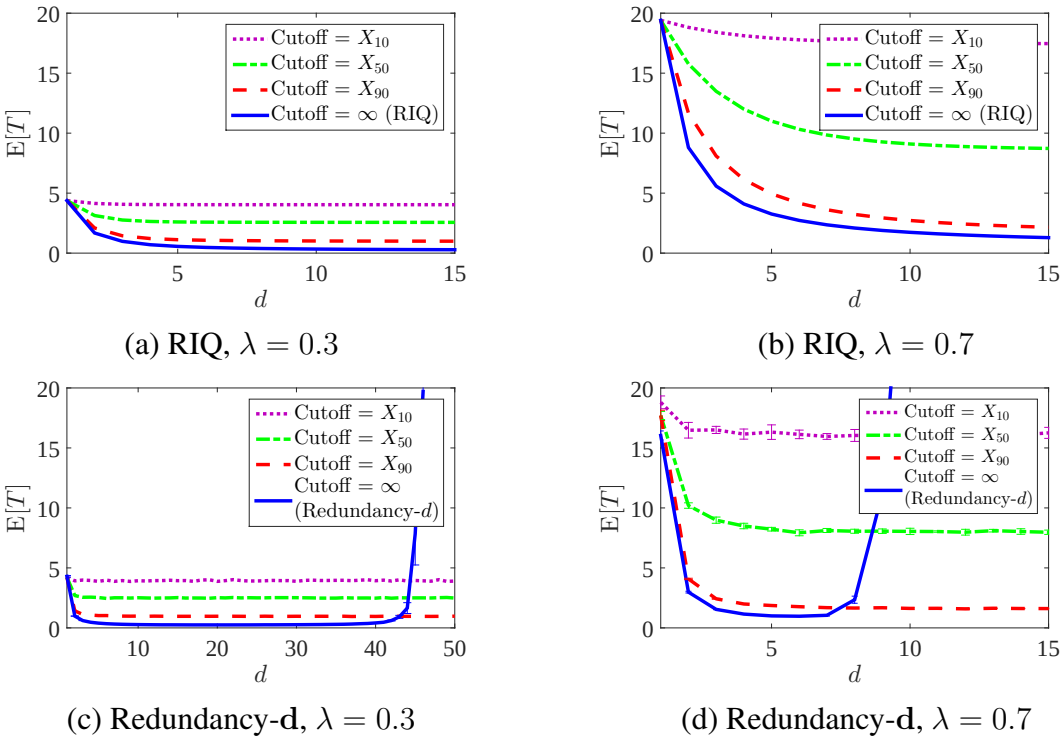
where $\widetilde{R(i)}(s\mid X\le x)=X\cdot\widetilde{\min\{S_1,\ldots,S_i\}}(s\mid X\le x)$.

If a job with $X<x$ finds no idle servers, it simply joins the queue at a single server, which behaves like an M*/G/1/efs:

$$\widetilde{T}\left(s\,\middle|\begin{matrix}X\le x\ \&\ \text{job finds}\\ \text{no idle servers}\end{matrix}\right)=\widetilde{T}(s\mid X\le x)^{\text{M}^*/\text{G}/1/\text{efs}}$$
$$=\widetilde{R(1)}(s\mid X\le x)\cdot\widetilde{T_Q}(s)^{\text{M}^*/\text{G}/1/\text{efs}}.$$

In this system, our M*/G/1/efs has the following parameters:

1. When the server is idle, arrivals form a Poisson process with rate
$$\mathbf{P}\left\{X>x\right\}\cdot\lambda k\cdot\frac{1}{k}+\mathbf{P}\left\{X\le x\right\}\cdot\lambda k\cdot\frac{\mathbf{d}}{k}=\lambda(\mathbf{P}\left\{X>x\right\}+\mathbf{P}\left\{X\le x\right\}\cdot\mathbf{d}).$$
2. When the server is busy, arrivals form a Poisson process with rate
$$\mathbf{P}\left\{X>x\right\}\cdot\lambda k\cdot\frac{1}{k}+\mathbf{P}\left\{X\le x\right\}\cdot\lambda k\cdot\frac{\mathbf{d}}{k}\cdot\rho^{\mathbf{d}-1}\cdot\frac{1}{\mathbf{d}}$$
$$=\lambda(\mathbf{P}\left\{X>x\right\}+\mathbf{P}\left\{X\le x\right\}\cdot\rho^{\mathbf{d}-1}).$$
3. Jobs that find the server idle experience runtime
$$R_0=\begin{cases}R(1)\mid X>x & \text{w.p. }\frac{\mathbf{P}\{X>x\}}{\mathbf{P}\{X>x\}+\mathbf{d}\cdot\mathbf{P}\{X\le x\}}\\ R(i)+Z\mid X\le x & \text{w.p. }\frac{\mathbf{d}\cdot\mathbf{P}\{X\le x\}}{\mathbf{P}\{X>x\}+\mathbf{d}\cdot\mathbf{P}\{X\le x\}}\cdot(1-\rho)^{i-1}\rho^{\mathbf{d}-i}\binom{\mathbf{d}-1}{i-1},\ 1\le i\le\mathbf{d}\end{cases}$$



(a) RIQ, $\lambda = 0.3$

(b) RIQ, $\lambda = 0.7$

(c) Redundancy-$\mathbf{d}$, $\lambda = 0.3$

(d) Redundancy-$\mathbf{d}$, $\lambda = 0.7$

Figure 6.12: Comparing $\mathbf{E}[T]$ under baseline RIQ (top, from analysis) and Redundancy-$\mathbf{d}$ (bottom, simulated; 95% confidence intervals are within the line where not shown) to that under the SMALL variant, which replicates only jobs with inherent size below a certain cutoff X_i, where X_i represents the ith percentile of X. Here $S \sim \text{Dolly}(1, 12)$, $X \sim \text{H}_2$ with $\mathbf{E}[X] = \frac{1}{4.7}$ and $C_X^2 = 10$, $Z = 0$, and $\lambda = 0.3$ (left) and $\lambda = 0.7$ (right).

4. Jobs that find the server busy experience runtime

$$R_b = \begin{cases} R(1) \mid X > x & \text{w.p. } \frac{\mathbf{P}\{X>x\}}{\mathbf{P}\{X>x\}+\mathbf{P}\{X\leq x\}\cdot\rho^{\mathbf{d}-1}} \\ R(1) \mid X \leq x & \text{w.p. } \frac{\mathbf{P}\{X\leq x\}\cdot\rho^{\mathbf{d}-1}}{\mathbf{P}\{X>x\}+\mathbf{P}\{X\leq x\}\cdot\rho^{\mathbf{d}-1}} \end{cases}$$

Here the probabilities are reweighted because the probability that a job of size $> x$ enters the *server* is not equal to the probability that a job of size $> x$ enters the *system*.

Performance

Figure 6.12 compares baseline RIQ and Redundancy-$\mathbf{d}$ to their SMALL variants with inherent size cutoffs at the 10th, 50th, and 90th percentiles of X. The RIQ+SMALL results are from the above analysis; Redundancy-$\mathbf{d}$+SMALL is simulated.

Surprisingly, replicating only the small jobs does not reduce mean response time under RIQ. In fact, as the size cutoff increases (i.e., more jobs are allowed to replicate) $\mathbf{E}[T]$ decreases, and is lowest under baseline RIQ. This is because RIQ is designed to replicate only when the system has spare capacity. Further limiting the number of jobs that replicate leads to a minor increase in the number of idle servers, thereby helping small jobs that can still replicate. But this benefit is

outweighed by the harm experienced by the large jobs that no longer get to see the minimum of multiple server slowdowns.

Similarly, when the system is stable under baseline Redundancy-$\mathbf{d}$, lowering the size cutoff (i.e., replicating fewer jobs) leads to an increase in mean response time. But importantly, the system is stable at higher values of $\mathbf{d}$ under Redundancy-$\mathbf{d}$+SMALL than under baseline Redundancy-$\mathbf{d}$. While Redundancy-$\mathbf{d}$+SMALL eventually becomes unstable (e.g., when $\lambda = 0.3$ and $C_X^2 = 10$, instability occurs around $\mathbf{d} = 680$), at practical (low) values of $\mathbf{d}$ the system is both stable and relatively insensitive to the particular choice of $\mathbf{d}$.

6.7.3 THRESHOLD: Replicate to Short Queues Only

A major advantage of RIQ is that the system cannot become unstable as $\mathbf{d}$ gets large. But RIQ is not perfect: with the right choice of $\mathbf{d}$, Redundancy-$\mathbf{d}$ can achieve lower response time than RIQ. RIQ allows very few jobs to replicate, sacrificing potential response time gains to guarantee stability. On the other hand, Redundancy-$\mathbf{d}$ allows all jobs to replicate, offering the potential for large response time improvements, but risking instability. Ideally, our policy would lie between RIQ and Redundancy-$\mathbf{d}$: we can afford to replicate more than under RIQ, but not as much as under Redundancy-$\mathbf{d}$. To accomplish this, we introduce the THRESHOLD-n policy, which represents a compromise between RIQ and Redundancy-$\mathbf{d}$.

Definition 6.6. *Under **THRESHOLD-**n, each arriving job polls $\mathbf{d}$ servers and joins the queue at those servers with queue length $\leq n$. If all queue lengths are $> n$, the job joins a single queue at random from among the $\mathbf{d}$ polled servers.*

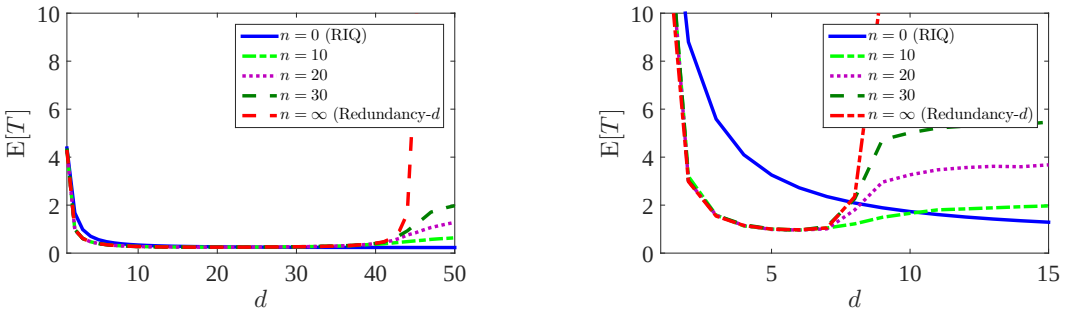
Note that RIQ and Redundancy-$\mathbf{d}$ are the extrema of the THRESHOLD-n policies, where RIQ$\equiv$THRESHOLD-0 and Redundancy-$\mathbf{d}\equiv$THRESHOLD-∞.

Unfortunately, THRESHOLD-n is not analytically tractable. The key feature enabling us to analyze RIQ (and the JSQ and SMALL variants) is that under these policies, all copies of a job start service simultaneously, so we do not need to track the ages of copies in service. This is not true for THRESHOLD-n, hence we study THRESHOLD-n via simulation.

THRESHOLD-n achieves the best features of both RIQ and Redundancy-$\mathbf{d}$ (see Figure 6.13). Like Redundancy-$\mathbf{d}$, THRESHOLD-n allows for enough redundancy to achieve substantial response time improvements. Like for RIQ, we can derive an upper bound on $\mathbf{E}[T]$ that shows that the system does not become unstable as a function of $\mathbf{d}$ (see Theorem 6.4). For example, when $\lambda = 0.7$ (Figure 6.13(b)), THRESHOLD-10 performs nearly identically to Redundancy-$\mathbf{d}$ for $\mathbf{d} \leq 7$; here both policies outperform RIQ. At $\lambda = 8$, Redundancy-$\mathbf{d}$ approaches instability, but under THRESHOLD-10 response time plateaus to a value only slightly higher than that under RIQ. The plateau occurs because all of the queues tend to remain at exactly n, so all jobs have similar queueing times.

While we cannot analyze performance under THRESHOLD-n, we can prove that for all $n < \infty$ and for all $\mathbf{d}$, the system is stable under THRESHOLD-n provided the system is stable when $\mathbf{d} = 1$. This means that we can make n as high as needed to achieve the low response times offered by Redundancy-$\mathbf{d}$ at low $\mathbf{d}$, without worrying that the system will be unstable if we choose $\mathbf{d}$ too high.

Theorem 6.4. *The system is stable under THRESHOLD-n, for all n, if $\lambda \cdot \mathbf{E}[X \cdot S] < 1$.*



(a) $\lambda = 0.3$ (b) $\lambda = 0.7$

Figure 6.13: Mean response time under THRESHOLD-n for different values of n ($n = 0$ is from our RIQ analysis; other values of n are simulated, 95% confidence intervals are within the line). Here $X \sim \mathrm{H}_2$ with $\mathbf{E}[X] = \frac{1}{4.7}$ and $C_X^2 = 10$, $S \sim \mathrm{Dolly}(1, 12)$, $Z = 0$, and (a) $\lambda = 0.3$ and (b) $\lambda = 0.7$.

Proof. We will upper bound mean response time under THRESHOLD-n by the mean response time in an M/G/1/setup system. We begin by expressing mean response time under THRESHOLD-n by conditioning on whether a tagged arrival to the system finds any idle servers. We have

$$\mathbf{E}[T]^{\mathrm{THRESH}-n} = \mathbf{P}\left\{\begin{matrix}\text{job finds servers}\\ \text{with } \leq n \text{ jobs}\end{matrix}\right\} \cdot \mathbf{E}\left[T \middle| \begin{matrix}\text{job finds servers}\\ \text{with } \leq n \text{ jobs}\end{matrix}\right] + \mathbf{P}\left\{\begin{matrix}\text{job finds no servers}\\ \text{with } \leq n \text{ jobs}\end{matrix}\right\} \cdot \mathbf{E}\left[T \middle| \begin{matrix}\text{job finds no servers}\\ \text{with } \leq n \text{ jobs}\end{matrix}\right].$$

When a job arrives to the system, if it finds i servers with fewer than n jobs it joins the queue (or enters service if the queue is empty) at all of these servers. For all i, we have

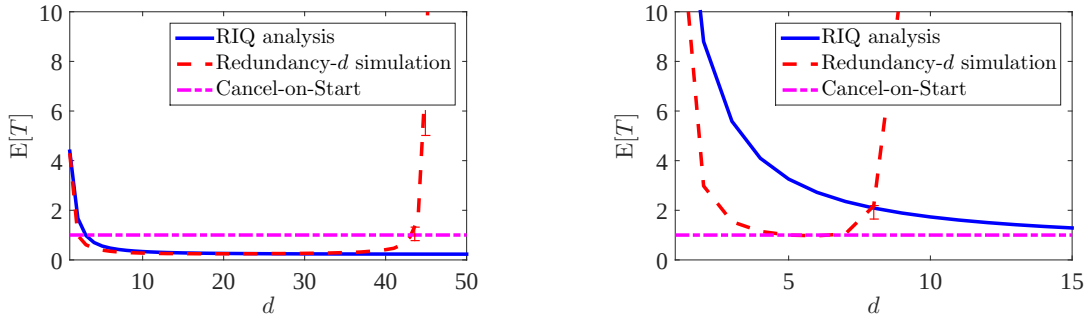
$$\mathbf{E}\left[T \middle| \begin{matrix}\text{job finds } i \text{ servers}\\ \text{with } \leq n \text{ jobs}\end{matrix}\right] \leq \mathbf{E}\left[T \middle| \begin{matrix}\text{job finds no servers}\\ \text{with } \leq n \text{ jobs}\end{matrix}\right].$$

Hence we have the following upper bound:

$$\mathbf{E}[T]^{\mathrm{THRESH}-n} \leq \mathbf{E}\left[T \middle| \begin{matrix}\text{job finds no servers}\\ \text{with } \leq n \text{ jobs}\end{matrix}\right]. \tag{6.19}$$

Given that a tagged arrival found no servers with $\leq n$ jobs, it joins the queue at a randomly chosen server with the following properties:

1. When the server is busy, jobs arrive with rate
$$k\lambda \cdot \frac{d}{k} \cdot \mathbf{P}\{\text{job finds other } d-1 \text{ servers with } > n \text{ jobs}\} \cdot \frac{1}{d} \leq \lambda.$$
2. All jobs that arrive to the server while it is busy experience runtime $R(1)$.
3. A tagged job that arrives to the server while it has $\leq n$ jobs, and finds $i-1$ other servers with $\leq n$ jobs will complete when the first of its i copies completes service.



(a) $\lambda = 0.3$ (b) $\lambda = 0.7$

Figure 6.14: Comparing Redundancy-$\mathbf{d}$ and RIQ under cancel-on-complete to cancel-on-start with $\mathbf{d} = k = 1000$ at (a) $\lambda = 0.3$ and (b) $\lambda = 0.7$. Here $X \sim H2$ with $\mathbf{E}[X] = \frac{1}{4.7}$ and $C_X^2 = 10$, $S \sim \text{Dolly}(1, 12)$, and $Z = 0$.

Since (6.19) forces all jobs to wait behind at least n jobs in our upper bound, we can view any job that arrives while there are $\leq n$ jobs at the server to be a "dummy" job that does not contribute to response time. Every time the queue length drops below n jobs, we can imagine that a "dummy" job with size $R(1) + Z$ is added to the queue, so the server always has greater than n jobs in the queue when the next "real" job arrives. Alternatively (but equivalently), we can imagine that a "real" arrival that finds no other "real" jobs at the server triggers a setup time of length $(R(1) + Z)_e + \sum_{j=1}^{n-1}(R(1) + Z)$ (i.e., the time remaining for the "dummy" job currently in service, plus $n - 1$ additional "dummy" jobs in queue; $(R(1) + Z)_e$ is the excess of $R(1) + Z$). This exactly describes an M/G/1/setup system with arrival rate λ, runtime $R(1)$, and setup time $(R(1)+Z)_e+\sum_{j=1}^{n-1}(R(1)+Z)$. The M/G/1/setup is stable if $\lambda \cdot \mathbf{E}[R(1)] = \lambda \cdot \mathbf{E}[X \cdot S] < 1$. Since the mean response time in the M/G/1/setup upper bounds the mean response time in our original system under THRESHOLD-n, the system is stable under THRESHOLD-n if $\lambda \cdot \mathbf{E}[X \cdot S] < 1$.□

6.8 Cancel-on-Start

Thus far in this thesis we have assumed a *cancel-on-complete* model, in which a job's extra copies are cancelled as soon as the first copy completes service. We now turn to an alternative *cancel-on-start* model, in which a job's extra copies are cancelled as soon as the first copy *enters* service. Unlike under cancel-on-complete, under cancel-on-start no extra load is added to the system because only one copy of each job actually runs. Hence under cancel-on-start there is no disadvantage to making $\mathbf{d}$ as high as possible. While in practice communication overhead may impose an upper limit on $\mathbf{d}$, in theory it is best to set $\mathbf{d} = k$, hence for the remainder of this section we assume $\mathbf{d} = k$.

Cancel-on-start maximizes the queueing benefits of redundancy: each job gets to experience the shortest possible queueing time across all queues. Importantly, this is *not* the same as the Least-Work-Left dispatching policy, under which each arriving job joins the queue with the least work, where "work" is the sum of the inherent sizes of all jobs in the queue, plus the remaining size of the job currently in service if there is one. Least-Work-Left cannot take into account the

server slowdown each job ultimately experiences, because server slowdowns are not known in advance. Cancel-on-start achieves what Least-Work-Left would achieve if both X and S were known in advance for each job (which is equivalent to a central queue); cancel-on-start does this without knowing X or S. In the special case in which $R(1) = X \cdot S$ is a mixture of exponentials, response time can be analyzed numerically using Matrix-Analytic methods. Figure 6.14(b) shows that when λ is high, the queueing benefit allows cancel-on-start to outperform both Redundancy-$\mathbf{d}$ and RIQ at all values of $\mathbf{d}$.

However, cancel-on-start does not allow jobs to experience the minimum slowdown across multiple servers. When λ is low, the system has extra capacity, which is wasted under cancel-on-start but used to reduce runtimes under cancel-on-complete. Hence at low λ Redundancy-$\mathbf{d}$ and RIQ both outperform cancel-on-start (see Figure 6.14(a)).

6.9 Discussion and Conclusion

In this chapter we introduce the $S\&X$ model, a new, more realistic CR model for computer systems with redundancy, where a job's inherent size X is decoupled from the server slowdown S. The model is very general, allowing for any inherent job size distribution X, any server slowdown distribution S, and any cancellation time Z. The $S\&X$ model is designed to be a more realistic alternative to the IR model, in which a job's replicas experience independent runtimes across servers. The IR model is appropriate in settings in which a job's runtime is determined primarily by the server on which it runs, but in many computer systems this is not the case. In contrast, the $S\&X$ model accurately describes systems in which a job's inherent size is non-negligible compared to the server-dependent factors that affect its runtime. This is important because, as we have seen in Chapters 3-5, in the IR model more redundancy is better, whereas empirical results indicate that in computer systems, redundancy may come at a cost. The $S\&X$ model allows us to study the potential costs—as well as the benefits—of redundancy from a theoretical perspective, thereby offering insights into how to design effective redundancy policies for computer systems applications.

In our new $S\&X$ model, dispatching policies such as Redundancy-$\mathbf{d}$ can perform much worse than predicted by our analysis in the IR model (Chapter 5). This motivates us to develop a new dispatching policy designed to perform well in the $S\&X$ model. We introduce the Redundant-to-Idle-Queue policy (RIQ), under which each arriving job creates redundant copies only when the job finds idle servers. We derive a highly accurate approximation for response time under RIQ. We also derive an upper bound on mean response time under RIQ that shows that RIQ does not cause instability even as the redundancy degree $\mathbf{d}$ becomes large. Our results demonstrate that RIQ is extremely *robust* to the system parameters, including the inherent job size distribution, the server slowdown distribution, and $\mathbf{d}$.

The RIQ policy is one example of a redundancy-based policy that performs well in a realistic model. We analyze several variants to the baseline RIQ policy, such as RIQ+JSQ, under which jobs that find no idle servers use Join-the-Shortest-Queue dispatching, and RIQ+SMALL, under which only jobs with a small inherent size are allowed to create redundant copies. RIQ+SMALL is designed to further reduce the amount of extra load introduced to the system, but our results indicate that this does not help. This is because RIQ already significantly limits the number of

jobs that are allowed to replicate, so the SMALL variant ends up being too conservative and therefore sacrifices some of the potential benefit of redundancy.

Instead, we propose the THRESHOLD-n policy, which is a more liberal version of RIQ that allows jobs to replicate at any server with fewer than n jobs in the queue rather than just at idle servers. THRESHOLD-n combines the best features of RIQ (which is equivalent to THRESHOLD-0) and Redundancy-$\mathbf{d}$ (which is equivalent to THRESHOLD-∞): it achieves excellent performance by allowing many jobs to replicate, but avoids instability by ultimately limiting the amount of replication.

The $S\&X$ model bridges the gap between earlier theoretical models of redundancy and the practical characteristics of computer systems that use redundancy. However, our model does not incorporate every aspect of such systems. For example, in practice server slowdowns may be time-dependent or correlated between consecutively processed jobs on the same server. We leave incorporating such features into a theoretical model of redundancy open for future work.

Chapter 7

Conclusion

This thesis has focused on modeling and analyzing systems with redundancy. We developed the first exact analysis of systems with redundancy in the IR model. We use a Markov chain approach to find the limiting distribution on a very detailed state space, and used this result to find the distribution of response time in small systems (Chapter 3) and nested systems with any number of servers (Chapter 4). We also used our limiting distribution, in combination with a new state aggregation approach, to find mean response time under the Redundancy-$\mathbf{d}$ policy in systems of any size (Chapter 5). Also under the Redundancy-$\mathbf{d}$ policy, we found the distribution of response time using a novel differential equations approach (Chapter 5). We studied scheduling in the IR model with the simultaneous goals of improving overall system efficiency and maintaining fairness across different classes of jobs. We proposed the Least Redundant First scheduling policy, which we proved was optimal with respect to minimizing response time, and the Primaries First policy, which we proved was fair (Chapter 4).

We then turned to the $S\&X$ model, which we proposed as a more realistic model for redundancy in computer systems (Chapter 6). In the $S\&X$ model, we observed that policies like Redundancy-$\mathbf{d}$ no longer necessarily perform well, and can in fact lead to instability. We proposed the Redundant-to-Idle-Queue dispatching policy, which we proved maintains system stability. We also provide an approximate analysis of response time under RIQ, and propose several variants that further improve performance (Chapter 6).

The work presented in this thesis provides strong support for the following thesis statement:

> *Redundancy is a powerful technique that has the potential to improve performance significantly in multi-server queueing systems. However, redundancy also is a risky strategy that can degrade performance severely under the wrong design choices. Through mathematical analysis of redundancy systems, we obtain a better understanding of when redundancy helps and hurts, which allows us to design new scheduling and dispatching policies for redundancy systems that are analytically tractable, provably stable, and offer significant response time improvements relative to systems without redundancy.*

The remainder of this chapter is organized as follows. We first discuss the major insights this thesis revealed about how to model systems with redundancy and how to design redundancy-based policies that achieve good performance (Section 7.1). We then discuss open problems about systems with redundancy, which we leave for future work (Section 7.2).

7.1 Lessons Learned about Redundancy

Our study of redundancy began in the IR model in constrained systems, in which each job can replicate itself only at a fixed subset of the servers. This system structure accurately captures systems such as deceased donor organ transplant waitlists, in which patients can only multiple list (i.e., send redundant requests) to a subset of the waitlists based on financial or geographic constraints, and in which the "runtime" experienced by a patient (i.e., the time for an organ to become available to the patient at the head of the queue) is independent across geographic regions. Our results have important implications for how best to manage these waitlists.

First, we find that *more redundancy is better for the overall system*. In Chapter 3, we saw that increasing the fraction of jobs that are redundant reduces overall system mean response time. This suggests that, contrary to the common concerns that multiple listing is harmful, it may be best to allow as many patients as possible to multiple list.

Unfortunately, we have also seen that *the benefits of redundancy are not felt equally across different classes of jobs/patients*. In particular, redundancy benefits the jobs that can become redundant, but can hurt the less redundant or non-redundant jobs. We can overcome this inequity by using smart scheduling policies (Chapter 4). Under our proposed Least Redundant First policy, non-redundant jobs receive full preemptive priority over redundant jobs, so the non-redundant jobs do not suffer due to others' redundancy. Under our proposed Primaries First policy, all classes of jobs—both redundant and non-redundant—perform at least as well as in a system with no redundancy. Our results demonstrate that *redundancy can simultaneously improve overall system efficiency and be fair to all classes of jobs*. Importantly, this means that patients who cannot afford the travel required to multiple list need not suffer longer waiting times due to wealthier patients becoming redundant.

These same messages hold in computer systems applications in which jobs are inherently quite small, while server-side variability is high. Here, the primary factor influencing a job's runtime is unpredictable and time-varying resource contention at the particular server on which the job runs, rather than any inherent properties of the job. This setting arises in applications such as web queries, which may be inherently small but can take a long time to complete if network contention is high, or small computational jobs, which can be greatly slowed down by the presence of other jobs competing for CPU time. Here, we find that *allowing very small jobs to become redundant reduces response time for the system as a whole* (Chapter 3). In many systems, jobs have data locality constraints and run extremely slowly, or cannot run at all, on servers that do not store their required data. As in the organ donation waitlist system, jobs that are non-redundant because their data is only stored at one server can be hurt when jobs with replicated data send redundant requests to multiple servers. But we can *overcome this unfairness using scheduling policies like LRF and PF*, which ensure that the non-redundant jobs are not starved (Chapter 4).

Many computer systems consist of thousands of servers, and again we find that when jobs are inherently small and server-side variability is high, it is best to allow jobs to be as redundant as possible. In constrained class-based systems, "as redundant as possible" may mean that a job can create only a small number of copies. But in flexible systems, in which a job is capable of running on any server, *it is optimal for all jobs to be fully redundant*, sending copies to all servers in the system. In practice, however, it can be costly (e.g., in terms of network bandwidth) to dispatch copies of every single job to every server. Instead, we propose and analyze the Redundancy-d

dispatching policy, under which each job sends copies to some small number $\mathbf{d}$ of the servers. This is similar to the commonly-used JSQ-$\mathbf{d}$ dispatching policy. Our analysis of Redundancy-$\mathbf{d}$ shows that while it is optimal for jobs to be fully redundant, *very little redundancy is needed to achieve substantial response time reductions*. Even creating fewer than two copies per job on average is enough to greatly improve system performance.

In other computer systems, jobs are not necessarily very small, and server-side variability is not necessarily the dominant factor in a job's runtime. Here, the IR model is no longer appropriate, and instead we propose the $S\&X$ model, a new CR model that decouples the job-dependent and server-dependent influences on runtime (Chapter 6). The $S\&X$ model is useful for studying applications such as downloading large files over a network, running computationally intensive jobs in the cloud, or making complex database queries on a shared cluster. In all of these settings, server-dependent factors can slow down a job, but if the job itself is large it will remain large on any server on which it runs. In the $S\&X$ model, we find that Redundancy-$\mathbf{d}$ is no longer a good idea: *Redundancy-$\mathbf{d}$ can lead to unacceptably poor performance and even instability* if too many copies are created per job. This is particularly problematic because Redundancy-$\mathbf{d}$ is difficult to analyze in the $S\&X$ model, meaning that is is impossible to identify the point at which the system abruptly becomes unstable.

We propose the Redundant-to-Idle-Queue (RIQ) dispatching policy, which allows an arriving job to dispatch redundant copies only to those servers that it finds idle among $\mathbf{d}$ randomly polled servers (Chapter 6). Unlike Redundancy-$\mathbf{d}$, RIQ is a "safe" redundancy policy: *RIQ is provable stable*, regardless of the choice of $\mathbf{d}$. Furthermore, *RIQ can lead to an order of magnitude reduction in mean response time* relative to using no redundancy. Yet further improvements are possible by, for example, having jobs that do not find idle servers join the shortest of $\mathbf{d}$ queues instead of a random queue (RIQ+JSQ), or by allowing jobs to replicate at all servers with fewer than n jobs in the queue instead of only at idle servers (THRESHOLD-n). Our analysis of RIQ and its variants in the $S\&X$ model demonstrates that *even in systems where redundancy has costs, it is possible to design policies that are provably stable and that substantially reduce response time*.

The redundancy-based policies that we propose in this thesis to achieve low response time, guarantee stability, and maintain fairness across classes are but a few examples of possible redundancy policies. The common theme behind why policies such as RIQ and PF are able to achieve such good performance provide insight into how to design other redundancy-based policies that are likely to perform well. In both cases, we adopt policies that *limit the amount of redundancy introduced to the system*. In the case of RIQ, this is done explicitly by only creating redundant copies when servers are idle upon a job's arrival to the system. In the case of PF, this is done implicitly by only running redundant copies when a server otherwise would be idle. The key insight is that in order to ensure that desirable system properties such as stability and fairness are maintained, redundancy policies must include some mechanism to ensure that the redundant copies do not "suffocate" the system as a whole, and that the redundant copies do not "suffocate" non-redundant jobs. We hope that this design principle provides a useful guideline for the development of new redundancy-based policies—either extensions of the policies introduced in this thesis, or entirely new policies—that further improve performance.

7.2 Open Problems

7.2.1 Stability in redundancy systems

One of the main concerns about adopting redundancy-based policies in practice is that redundancy can cause the system to become unstable. We have seen this under the Redundancy-$\mathbf{d}$ policy in the $S\&X$ model (Chapter 6): if $\mathbf{d}$ is too high, Redundancy-$\mathbf{d}$ leads to instability. Unfortunately, we do not know how to determine how high is "too high." Without knowing its stability region as a function of both system parameters (e.g., λ, S, X) and policy parameters (e.g., $\mathbf{d}$), Redundancy-$\mathbf{d}$ is a dangerous policy. But it is possible to design redundancy policies that are "safe". For example, our proof that RIQ cannot become unstable as a function of $\mathbf{d}$ (provided that the system is stable when $\mathbf{d} = 1$) demonstrates that RIQ is a "safe" dispatching policy.

The question of stability has important implications for the design of practical redundancy policies, but understanding stability in redundancy systems is difficult. In traditional queueing systems without redundancy, the stability region depends on the arrival rate, service rate, and number of servers in the system. Typically the dispatching and scheduling policies do *not* affect the stability region; we only need to know that the total rate at which work enters the system is less than the maximal rate at which work can leave the system. In contrast, in redundancy systems the stability region depends critically on the specific policy decisions. The total rate at which work enters the system depends on the number of copies created per job. The total rate at which work leaves the system depends on how many of these copies actually enter service, and for how long each copy occupies a server. This clearly depends on the policy choices, and for many policies it is not at all straightforward to compute.

We pose as future work two important questions about stability under redundancy-based policies:

- Define a policy Π to be "safe" if the system is stable under Π for all possible settings of the policy's parameters, provided that the system is stable with no redundancy. Which redundancy-based dispatching and scheduling policies Π are "safe"? Are there general properties of policies that imply "safeness"?
- Suppose that policy Π is unsafe. This means that there is some setting of the policy's parameters such that the stability region under Π is smaller than that in the system with no redundancy. What is the stability region under Π with this problematic parameterization?

Answering the former question allows us to implement only those policies that guarantee stability whenever a non-redundant system guarantees stability. This is appealing in systems in which, for example, the arrival rate might change over time. Answering the latter question allows us to implement a wider range of policies, possibly including policies that achieve better performance when optimal parameter settings are chosen.

7.2.2 Better, analytically tractable redundancy policies

One of the main advantages of RIQ and the variants on RIQ that we propose in Chapter 6 is that they do not cause the system to become unstable. RIQ and the RIQ+SMALL and RIQ+JSQ variants accomplish this by only creating replicas when servers are idle. This allows some jobs to experience a minimum runtime across multiple servers, but it eliminates the possibility for jobs

that do not find idle servers to experience a *queueing time* benefit. This is potentially problematic because a job waiting in a single queue can experience a very long queueing time if it must wait for a job with a large runtime to complete service. Indeed, reducing queueing time by allowing jobs to wait in multiple queues is one important advantage of redundancy. In designing RIQ, which guarantees stability, we have sacrificed some of the potential benefits of redundancy. The THRESHOLD-n policy attempts to fix this by allowing replicas to be created somewhat more liberally, but unfortunately THRESHOLD-n seems difficult to analyze.

One of the main challenges in analyzing redundancy policies in the $S\&X$ model is that it is difficult to track the progress of each replica when the replicas do not enter service at the same time. This makes it hard to understand response time under policies like Redundancy-$\mathbf{d}$ and THRESHOLD-n. But policies that force all of a job's copies to enter service simultaneously lose some of the potential for a queueing time benefit from redundancy. This tradeoff between leveraging the full power of redundancy and analytical tractability gives rise to the following question:

- Are there redundancy-based policies that both offer a greater potential queueing time benefit (as well as a runtime benefit) and are analytically tractable in the $S\&X$ model?

7.2.3 Application-specific models and policies

In Chapter 6 we proposed the $S\&X$ model, which is a realistic model for redundancy that decouples the job-dependent and server-dependent factors that affect a job's runtime. We believe that the $S\&X$ model is an excellent model in which to study redundancy in computer systems. However, there are several characteristics of computer systems that the $S\&X$ model does not yet incorporate. Furthermore, the $S\&X$ model was designed specifically to model computer systems with redundancy. Different modeling decisions are more appropriate in systems such as organ transplant waitlists. We discuss extensions to the $S\&X$ model for computer systems and alternative modeling decisions for organ transplant waitlists below.

Computer systems In the $S\&X$ model, the server draws a new instance of the slowdown random variable every time a new job enters service. This is meant to capture the fact that server slowdown changes over time. We assume that the slowdown stays the same for the duration of the job's service, but this may not be the case in real systems. An alternative—and still more realistic—model of server slowdown would involve a time-dependent slowdown function. This would allow the slowdown to change during a single job's service, and it would also allow slowdowns to be correlated between consecutive jobs.

A second way in which our server slowdown model could be more realistic is by allowing heterogeneous servers. That is, we should be able to allow different servers to have different distributions of S. This would allow us to capture both the fact that servers may have inherently different speeds due to different underlying hardware, and the fact that the servers may have different workloads and so might offer different types of resource contention, and therefore different slowdowns.

We pose the following two questions for future work:

- How can we incorporate time-dependent slowdown functions into the $S\&X$ model? Are policies like RIQ still analytically tractable in this model?
- How can we incorporate heterogeneous slowdown distributions into the $S\&X$ model? Are policies like RIQ still analytically tractable in this model?

We believe that for the latter question, it should be relatively straightforward to include heterogeneous servers in the $S\&X$ model. The analysis of RIQ that we present in Chapter 6 likely can be extended to allow us to first choose an S distribution probabilistically, then instantiate the value of S from the chosen distribution. However the former question is likely to be much more challenging: time-dependent service rates are typically difficult to analyze even in relatively simple queueing systems.

Organ transplant waitlists In organ transplant waitlists, the IR model that we study in Chapters 3-5 is more appropriate than the $S\&X$ model. This is because the time for an organ to become available in one geographic region is independent from the time for an organ to become available in a different region. However, our model in Chapters 3-5 does not capture all of the properties of organ transplant waitlists. In particular, typically these waitlists are in overload, and customers abandon the queue either because they receive an organ from a different matching service or because they become too ill to receive a transplant. The analysis of the limiting distribution of the state space that we present in Chapter 3 applies in this setting: the limiting probabilities will follow the same form in systems with abandonment. However, in systems with abandonment it becomes much more challenging to aggregate states to find the distribution of response time. A second question relates to scheduling. In Chapter 4 we propose Primaries First as a fair scheduling policy that penalizes neither redundant nor non-redundant customers. PF gives low priority to the extra "secondary" copies a customer creates. In a system running in overload, PF does not make sense because the secondary copies will never enter service, so it is as though there is no redundancy at all.

We pose two questions for future work on redundancy in organ transplant waitlist systems:

- Can we aggregate the limiting probabilities from Chapter 3 in systems with abandonment? Alternatively, can we develop approximations for response time in redundancy systems with abandonment in the IR model?
- Are there fair scheduling policies that are appropriate for systems in overload?

Both of the above questions relate to adapting the work presented in this thesis to limiting regimes within the existing models. However, there are other issues that arise in the particular setting of organ transplants. For example, an organ may go to waste if a compatible patient is not identified in the donor's geographic region, so one possible advantage of redundancy is that it can actually increase the overall "service rate" of the system. A detailed model of redundancy in organ transplant systems should take into account this type of interaction between redundancy-based policies and the underlying system parameters.

Bibliography

[1] Joseph Abate and Ward Whitt. Numerical inversion of laplace transforms of probability distributions. *ORSA Journal on computing*, 7(1):36–43, 1995. 6.5

[2] Ivo Adan and Gideon Weiss. A skill based parallel service system under FCFS-ALIS - steady state, overloads, and abandonments. *Stochastic Systems*, 4(1):250–299, 2014. 2.3.3

[3] Reza Aghajani, Xingjie Li, and Kavita Ramanan. Mean-field dynamics of load-balancing networks with general service distributions. *arXiv preprint arXiv:1512.05056*, 2015. 6.7.1, 6.7.1

[4] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI 2013*. 2.1.1, 6.2.1, 1

[5] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. Grass: Trimming stragglers in approximation analytics. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 289–302. USENIX Association, 2014. 2.1.1

[6] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in Map-Reduce clusters using Mantri. In *OSDI 2010*. 2.1.1

[7] Mohammad Sanaei Ardekani and Janis M Orlowski. Multiple listing in kidney transplantation. *American Journal of Kidney Diseases*, 55(4):717–725, 2010. 2.1.2

[8] Barıs Ata, Anton Skaro, and Sridhar Tayur. Organjet: Overcoming geographical disparities in access to deceased donor kidneys in the united states. Technical report, Working paper, Northwestern University, 2012. 2.1.2

[9] François Baccelli and A. Makowski. Simple computable bounds for the fork-join queue. Technical Report RR-0394, Inria, 1985. 2.3.2

[10] François Baccelli, Armand M. Makowski, and Adam Shwartz. The fork-join queue and related systems with synchronization constraints: Stochastic ordering and computable bounds. *Advances in Applied Probability*, 21:629–660, 1989. 2.3.2

[11] Nikhil Bansal and Mor Harchol-Balter. *Analysis of SRPT scheduling: Investigating unfairness*, volume 29. ACM, 2001. 4.1

[12] A. Bassamboo, R. S. Randhawa, and J. A. Van Mieghem. A little flexibility is all you need: On the value of flexible resources in queueing systems. *Operations Research*, 60:1423–1435, 2012. 2.3.3

[13] Andrea Bobbio, Marco Gribaudo, and Miklós Telek. Analysis of large scale interacting systems by mean field method. In *Quantitative Evaluation of Systems, 2008. QEST'08. Fifth International Conference on*, pages 215–224. IEEE, 2008. 5.6

[14] Thomas Bonald and Céline Comte. Networks of multi-server queues with parallel processing. *arXiv preprint arXiv:1604.06763*, April 2016. 3.8

[15] Sem Borst, Onno Boxma, and Miranda Van Uitert. The asymptotic workload behavior of two coupled queues. *Queueing Systems*, 43(1-2):81–102, January 2003. 2.3.1

[16] Robert F. Botta, Carl M. Harris, and William G. Marchal. Characterizations of generalized hyperexponential distribution functions. *Communications in Statistics, Stochastic Models*, 3(1):115–148, 1987. 1

[17] Onno Boxma, Ger Koole, and Zhen Liu. Queueing-theoretic solution methods for models of parallel and distributed systems. In *Performance Evaluation of Parallel and Distributed Systems Solution Methods. CWI Tract 105 & 106*, pages 1–24, 1994. 2.3.2

[18] Maury Bramson, Yi Lu, and Balaji Prabhakar. Asymptotic independence of queues under randomized load balancing. *Queueing Systems*, 71(3):247–292, 2012. 5.4

[19] J. W. Cohen and O. J. Boxma. *Boundary Value Problems in Queueing System Analysis*. North-Holland Publishing Company, 1983. 2.3.1

[20] Lauren Cox and ABC News Medical Unit. Steve jobs' reported liver transplant stirs debate. *ABC News*, June 2009. 2.1.2

[21] Mark E Crovella and Azer Bestavros. Self-similarity in world wide web traffic: evidence and possible causes. *IEEE/ACM Transactions on networking*, 5(6):835–846, 1997. 6.1

[22] Jeffrey Dean and Luis Andre Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, February 2013. 1, 2.1.1, 3.1, 6.2.1, 6.5

[23] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 2.1.1

[24] Guy Fayolle and Roudolf Iasnogorodski. Two coupled processors: The reduction to a Riemann-Hilbert problem. *Zeitschrift fur Wahrscheinlichkeitstheorie und vervandte Gebiete*, 47(3):325–351, 1979. 2.3.1

[25] L. Flatto. Two parallel queues created by arrivals with two demands II. *SIAM Journal on Applied Mathematics*, 45(5):1159–1166, October 1985. 2.3.2

[26] L. Flatto and S. Hahn. Two parallel queues created by arrivals with two demands I. *SIAM Journal on Applied Mathematics*, 44(5):250–255, October 1984. 2.3.2

[27] Eric J Friedman and Shane G Henderson. Fairness and efficiency in web server protocols. In *ACM SIGMETRICS Performance Evaluation Review*, volume 31, pages 229–237. ACM, 2003. 4.1

[28] SW Fuhrmann and Robert B Cooper. Stochastic decompositions in the M/G/1 queue with generalized vacations. *Operations research*, 33(5):1117–1129, 1985.

[29] Kristen Gardner, Mor Harchol-Balter, Esa Hyytiä, and Rhonda Righter. Scheduling for efficiency and fairness in systems with redundancy. *Performance Evaluation*, Under submission.

1.5, 4.1

[30] Kristen Gardner, Mor Harchol-Balter, and Alan Scheller-Wolf. A better model for job redundancy: Decoupling server slowdown and job size. In *MASCOTS*, September 2016. 1.5, 6.1

[31] Kristen Gardner, Mor Harchol-Balter, Alan Scheller-Wolf, and Benny van Houdt. A better model for job redundancy: Decoupling server slowdown and job size. *Transactions on Networking*, Under revision. 1.5, 6.1

[32] Kristen Gardner, Mor Harchol-Balter, Alan Scheller-Wolf, Mark Velednitsky, and Samuel Zbarsky. Redundancy-d: The power of d choices for redundancy. *Operations Research*, 2017. 1.5, 5.1

[33] Kristen Gardner, Samuel Zbarsky, Sherwin Doroudi, Mor Harchol-Balter, Esa Hyytiä, and Alan Scheller-Wolf. Reducing latency via redundant requests: Exact analysis. In *SIGMETRICS*, June 2015. 1.5, 3.1

[34] Kristen Gardner, Samuel Zbarsky, Sherwin Doroudi, Mor Harchol-Balter, Esa Hyytiä, and Alan Scheller-Wolf. Queueing with redundant requests: exact analysis. *Queueing Systems*, 83(3):227–259, 2016. 1.5, 3.1

[35] Kristen Gardner, Samuel Zbarsky, Mor Harchol-Balter, and Alan Scheller-Wolf. The power of d choices for redundancy. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, pages 409–410. ACM, 2016. 1.5, 5.1

[36] Kristen Gardner, Samuel Zbarsky, Mark Velednitsky, Mor Harchol-Balter, and Alan Scheller-Wolf. Understanding response time in the Redundancy-d system. *ACM SIGMETRICS Performance Evaluation Review*, 44(2):33–35, 2016. 1.5, 5.1

[37] P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, s1-10(1):26–30, 1935. 2

[38] Mor Harchol-Balter and Allen B Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems (TOCS)*, 15(3):253–285, 1997. 6.1

[39] Mor Harchol-Balter, Cuihong Li, Takayuki Osogami, Alan Scheller-Wolf, and Mark Squillante. Cycle stealing under immediate dispatch task assignment. In *Annual Symposium on Parallel Algorithms and Architectures*, pages 274–285, June 2003. 2.3.1

[40] Gerard Hooghiemstra, Michael Keane, and Simon Van de Ree. Power series for stationary distributions of coupled processor models. *SIAM Journal on Applied Mathematics*, 48(5):861–878, October 1988. 2.3.1

[41] Longbo Huang, Sameer Pawar, Hao Zhang, and Kannan Ramchandran. Codes can reduce queueing delay in data centers. In *Information Theory Proceedings (ISIT), 2012 IEEE International Symposium on*, pages 2766–2770. IEEE, 2012. 2.2.2

[42] Ali Musa Iftikhar, Fahad Dogar, and Ihsan Ayyub Qazi. Towards a redundancy-aware network stack for data centers. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 57–63. ACM, 2016. 4.6

[43] Gauri Joshi, Yanpei Liu, and Emina Soljanin. Coding for fast content download. In *Allerton Conference'12*, pages 326–333, 2012. 2.2.2, 5.6

[44] Gauri Joshi, Yanpei Liu, and Emina Soljanin. On the delay-storage trade-off in content download from coded distributed storage systems. *IEEE J. Sel. Area. Comm.*, 32(5):989–997, May 2014. 2.2.2, 5.6

[45] J. Keilson and L. Servi. A distributional form of Little's Law. *Operations Research Letters*, 7(5):223–227, 1988. 10

[46] Cheeha Kim and Ashok K. Agrawala. Analysis of the fork-join queue. *IEEE Transactions on Computers*, 38(2):1041–1053, February 1989. 2.3.2

[47] Alan G. Konheim, Isaac Meilijson, and Abraham Melkman. Processor-sharing of two parallel lines. *Journal of Applied Probability*, 18(4):952–956, December 1981. 2.3.1

[48] Ger Koole and Rhonda Righter. Resource allocation in grid computing. *Journal of Scheduling*, 11:163–173, 2009. 2.2.1, 28

[49] Akshay Kumar, Ravi Tandon, and T Charles Clancy. On the latency of heterogeneous MDS queue. In *Global Communications Conference (GLOBECOM), 2014 IEEE*, pages 2375–2380. IEEE, 2014. 2.2.2

[50] Kangwook Lee, Ramtin Pedarsani, and Kannan Ramchandran. On scheduling redundant requests with cancellation overheads. *IEEE/ACM Transactions on Networking*, 2016. 3.8

[51] Bin Li, Aditya Ramamoorthy, and R Srikant. Mean-field-analysis of coding versus replication in cloud storage systems. In *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*, pages 1–9. IEEE, 2016. 5.6

[52] Guanfeng Liang and Ulas C Kozat. Tofec: Achieving optimal throughput-delay trade-off of cloud storage using erasure codes. In *INFOCOM, 2014 Proceedings IEEE*, pages 826–834. IEEE, 2014. 6.2.1

[53] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R Larus, and Albert Greenberg. Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services. *Performance Evaluation*, 68(11):1056–1071, 2011. 2

[54] Robert M Merion, Mary K Guidinger, John M Newmann, Mary D Ellison, Friedrich K Port, and Robert A Wolfe. Prevalence and outcomes of multiple-listing for cadaveric kidney and liver transplantation. *American Journal of Transplantation*, 4(1):94–100, 2004. 2.1.2

[55] Michael Mitzenmacher. The power of two choices in randomized load balancing. *Parallel and Distributed Systems, IEEE Transactions on*, 12(10):1094–1104, 2001. 5.1, 5.1, 5.4, 5.5, 5.6

[56] Leela Nageswaran and Alan Andrew Scheller-Wolf. Queues with redundancy: Is waiting in multiple lines fair? Available at SSRN: https://ssrn.com/abstract=2833549 or http://dx.doi.org/10.2139/ssrn.2833549, 2016. 3.8, 4.1

[57] R. Nelson and A. N. Tantawi. Approximate analysis of fork/join synchronization in parallel queues. *IEEE Transactions on Computers*, 37(6):739–743, 1988. 2.3.2

[58] Takayuki Osogami, Mor Harchol-Balter, and Alan Scheller-Wolf. Analysis of cycle stealing

with switching times and thresholds. In *SIGMETRICS*, pages 184–195, June 2003. 2.3.1

[59] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 69–84, 2013. 1.1, 2.1.1

[60] Xiaoqi Ren, Ganesh Ananthanarayanan, Adam Wierman, and Minlan Yu. Hopper: Decentralized speculation-aware cluster scheduling at scale. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 379–392. ACM, 2015. 2.1.1

[61] Nihar B. Shah, Kangwook Lee, and Kannan Ramchandran. The MDS queue: Analysing latency performance of codes and redundant requests. Technical Report arXiv:1211.5405, November 2012. 2.2.2

[62] Nihar B. Shah, Kangwook Lee, and Kannan Ramchandran. When do redundant requests reduce latency? Technical Report arXiv:1311.2851, June 2013. 2.2.1, 2.2.2

[63] William Stallings and Goutam Kumar Paul. *Operating systems: internals and design principles*, volume 3. prentice hall Upper Saddle River, NJ, 1998. 4.1

[64] Alexander L. Stolyar and Tolga Tezcan. Control of systems with flexible multi-server pools: a shadow routing approach. *Queueing Systems*, 66:1–51, 2010. 2.3.3

[65] Yin Sun, C Emre Koksal, and Ness B Shroff. On delay-optimal scheduling in queueing systems with replications. *arXiv preprint arXiv:1603.07322*, 2016. 2.2.2

[66] Yin Sun, Can Emre Koksal, and Ness B. Shroff. On delay-optimal scheduling in queueing systems with replications. *CoRR*, abs/1603.07322, 2016. 2.2.2

[67] Andrew S Tanenbaum and Albert S Woodhull. *Operating systems: design and implementation*, volume 2. Prentice-Hall Englewood Cliffs, NJ, 1987. 4.1

[68] John Tsitsiklis and Kuang Xu. On the power of (even a little) resource pooling. *Stochastic Systems*, 2:1–66, 2012. 2.3.3

[69] John Tsitsiklis and Kuang Xu. Queueing system topologies with limited flexibility. In *SIGMETRICS*, 2013. 2.3.3

[70] Jeremy Visschers, Ivo Adan, and Gideon Weiss. A product form solution to a system with multi-type jobs and multi-type servers. *Queueing Systems*, 70:269–298, 2012. 2.3.3

[71] Ashish Vulimiri, P. Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Low latency via redundancy. In *CoNEXT*, pages 283–294, December 2013. 2.1.1, 2.2.1, 6.1

[72] N.D. Vvedenskaya, R.L. Dobrushin, and F.I. Karpelevich. Queueing system with selection of the shortest of two queues: An asymptotic approach. *Probl. Peredachi Inf.*, 32(1):20–34, 1996. 5.1, 5.1, 5.4, 5.4, 5.5, 5.6

[73] Da Wang, Gauri Joshi, and Gregory Wornell. Efficient task replication for fast response times in parallel computation. Technical Report arXiv:1404.1328, April 2014. 2.2.2

[74] Peter D Welch. On a generalized M/G/1 queuing process in which the first customer of each busy period receives exceptional service. *Operations Research*, 12(5):736–752, 1964. 6.1,

42, 6.2

[75] Cathy Xia, Zhen Liu, Don Towsley, and Marc Lelarge. Scalability of fork/join queueing networks with blocking. In *SIGMETRICS*, pages 133–144, June 2007. 2.3.2

[76] Yunjing Xu, Michael Bailey, Brian Noble, and Farnam Jahanian. Small is better: Avoiding latency traps in virtualized data centers. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 7. ACM, 2013. 1, 3.7, 5.5.2

[77] Lei Ying, R Srikant, and Xiaohan Kang. The power of slightly more than one sample in randomized load balancing. In *Proc. of IEEE INFOCOM*, 2015. 5.1, 5.4, 5.6

[78] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, volume 8, page 7, 2008. 2.1.1