Speeding up distributed storage and computing systems using codes

Kangwook Lee

Speeding up distributed storage and computing systems using codes

by

Kang Wook Lee

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Kannan Ramchandran, Chair
Professor Jean Walrand
Professor Rhonda Righter

Spring 2016

Speeding up distributed storage and computing systems using codes

Abstract

Speeding up distributed storage and computing systems using codes

by

Kang Wook Lee

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Kannan Ramchandran, Chair

Modern data centers have been providing exponentially increasing computing and storage resources, which have been fueling core applications ranging from search engines in the early 2000's to the real-time, large-scale data analysis of today. All these breakthroughs were made possible only due to the scalability in computing and storage resources offered by modern large-scale clusters, comprising individually small and unreliable low-end devices. Given the individually unpredictable nature of the underlying devices in these systems, we face the constant challenge of securing predictable and high-quality performance of such systems in the face of uncertainty.

In this thesis, distributed storage and computing systems are viewed through a *coding-theoretic lens*. The role of codes in providing resiliency against noise has been studied for decades in many other engineering contexts, especially in communication systems, and codes are parts of our everyday infrastructure such as smartphones, WiFi, cellular systems, etc. Since the performance of distributed systems is significantly affected by anomalous system behavior and bottlenecks, which we call "system noise", there is an exciting opportunity for codes to endow distributed systems with *robustness* against such system noise.

Our key observation – *channel noise in communication systems is equivalent to system noise in distributed systems* – forms the key motivation of this thesis, and raises the fundamental question: "*can we use codes to guarantee robust speedups in distributed storage and computing systems?*". In this thesis, three main layers of distributed computing and storage systems – storage layer, computation layer, and communication layer – are robustified through coding-theoretic tools. For the storage layer, we show that coded distributed storage systems allow *faster* data retrieval in addition to the other known advantages such as higher data durability and lower storage overhead; for the computation layer, we inject computing redundancy into distributed algorithms that are robust to *stragglers* or nodes that are substantially slower than the other nodes; for the communication layer, we propose a novel data caching and communication protocol, based on coding-theoretic principles that can significantly reduce the network overhead of the *data shuffling* operation, which is necessary to achieve higher statistical efficiency when running parallel/distributed machine learning algorithms.

To my dear parents Sangsook and Kisung and my brother Kangmin.

Contents

List of Figures

List of Tables

Acknowledgments

Since I began my long journey of study in Berkeley, I have been always dreaming this moment of writing this page in my thesis. I will try my best to fully express invaluable gratitude to all the people who have helped me complete this thesis.

In my first year when I was taking a digital communication course taught by Prof. Kannan Ramchandran, I casually approached him and asked whether he had any interesting research problem I could start working on, and I never knew that was the beginning of about 6-year long roller-coaster-like apprenticeship. For those 6 years, we worked on a lot of problems – interactive-code design for deletion channels, design of a distributed video-on-demand caching system using codes, queueing analysis of distributed storage systems with coded data, the redundant-requests scheduling problem, compressive phase retrieval, sparse covariance estimation problem, group testing, codes for distributed computation, and coded shuffling for distributed machine learning algorithms. I was very fortunate to learn from him how to find right research problems, how to abstract real problems into solid, theoretic problems, how to solve them, and how to present the results.

What I learned from him was not just about research: it is no exaggeration to say that I have learned literally *everything* from him. First, I truly admire his pure passion as a researcher: just seeing him insatiably working on a various range of problems has been a continuous source of inspiration. Being a GSI for his classes was also priceless: the way he meticulously planned and prepared for lectures, innovated teaching materials, and personally related to all students made me respect him as a teacher. Above all, I truly appreciate his incessant support both as a mentor and a friend. He was the best supporter when I was struggling doing research, when I was homesick, and even when I was suffering from a broken heart. Kannan, I will never forget those great hours spent at your office.

In addition to having a great thesis advisor, I was very fortune to have the best thesis committee members – Prof. Jean Walrand and Prof. Rhonda Righter. I have interacted with Jean a long time ago dating back to the time when I was an undergraduate student in Korea: one of my first research experiences was on the implementation of optimal Carrier Sense Multiple Access (CSMA), which was pioneered by Jean; in my first year in Berkeley, I learned a lot from attending his group meetings and I gave my first conference presentation at the one organized by Jean. Further, I regularly consulted with him and he has always been greatly helpful. I also took a few courses taught by him, and all those lessons helped me a lot complete this thesis. Having Rhonda in my thesis committee was yet another blessing. When I was working on the research topics presented in Chapter 3, I became aware that some of her prior works are very closely related to what I was working on. I asked her for a brief meeting, and she was kind enough to spare her time for a long meeting during which we had an exciting discussion on the research topic. Since then, she kindly agreed to be on my thesis committee and all the meetings I had with her were extremely educative as well as pleasant. Further, I really appreciate her careful, constructive comments on my thesis.

Among all the other professors I have interacted with in Berkeley, I would express my special gratitude to a few more: Prof. Ion Stoica and Prof. Anil Aswani, who were on my Qual Exam committees, provided me invaluable advices before and during the exams; Prof. David Tse was my temporary advisor during my first year, and being advised by him was an exciting learning

experience; Prof. Abhay Parekh, whom I worked with together for my master's thesis as well as his startup, has been super supportive [1]; and all the professors at BLISS – Prof. Venkat Anantharam, Prof. Anant Sahai, Prof. Martin Wainwright, and Prof. Thomas Courtade – were great teachers.

One of the best fortunes I have had in Berkeley is an extensive collaboration with great coworkers. I have been learning so many things from working with a lot of collaborators. In chronological order, I have been fortunate working with Hao Zhang, Sameer Pawar, Salim Rouayheb, Ziyu Shao, Minghua Chen, Lisa Yan, Nick Chang, Nihar Shah, Longbo Huang, Ramtin Pedarsani, Dong Yin, Simon Li, Orhan Ocal, Kihyun Won, Dimitri Papailiopoulos, Maximilian Lam, Jichan Chung, Ryan Kashi, Jiayuan Chen, Alagu Sanjana Haribhaskaran, and Kabir Chandrasekher. I would mention some special remarks to a few of them. I started my master's thesis project which stemmed from Hao Zhang's PhD thesis, and Hao was an extremely helpful collaborator, a great teacher, and a great mentor. My experience of collaboration with Nihar Shah was so invaluable. He has been one of the best teachers in my graduate program – I learned so many invaluable things ranging from how to rigorously prove something to how to draw appropriate figures and how to punctuate. I also had so much fun exchanging childish jokes. Closely working with Ramtin Pedarsani was so much fun. We have been working together on so many problems and successfully nailed many of them. Discussing interesting applications, extracting new research problems from them, brainstorming new approaches to tackle them, and writing amazing papers have been our everyday schedule, and I owe him much intellectual debt. One of the most interesting work I had in Berkeley was done in collaboration with Ramtin Pedarsani, Dimitri Papailiopoulos, Maximilian Lam, and Jichan Chung. Dimitris, who is one the most pleasing collaborators, was the one who helped me learn parallel/distributed machine learning, which eventually became the core ideas of the second half of this thesis. Max and Jichan have been working harder than me, and I was constantly inspired by their energy and passion. Even though I have not done any official collaboration, I owe a huge intellectual debt to Varun Jog and Vijay Kamble, whom I used to consult whenever I had some deep questions. I will always miss the time I had all these great teachers around me.

I also owe a huge personal debt to all my friends. All the hard time taking courses, preparing prelim exams, finding a research advisor, and doing research without straying off will be recollected as precious only with my batchmates – Giulia Fanti, Stephan Adams, TJ Tsai, Daniel Calderone, Cameron Rose, Qie Hu, and Jeannette Chang. Being surrounded by great people in the Berkeley Laboratory of Information and System Sciences (BLISS), also known as Wireless Foundation (WiFo), was truly *blissful*. Thanks to Soheil Mohajer, Seyed Abolfazl Motahari, Eren Sasoglu, Barlas Oguz, Kristen Woyach, Pulkit Grover, Nima Noorshams, Changho Suh, I-Hsiang Wang, Se Yong Park, Naveen Goela, Nebojsa Milosavljevic, Sreeram Kannan, Joseph Bradley, Kate Harrison, Vasuki Narasimha Swamy, Rashmi Vinayak, Po-ling Loh, Sudeep Kamath, Venky Ekambaram, Nan Ma, Ka Kit Lam, Govinda Kamath, Steven Clarkson, Gireeja Ranade, Fanny Yang, Reza Abbasi Asl, Ashwin Pananjady, Vidya Muthukumar, Payam Delgosha, Sang Min Han, Yuting Wei, Leah Dickstein, Raaz Dwivedi, Soham-Rajesh Phade, Ilan Shomorony, and Reinhard

[1]My startup experience with his company as the first employee was so amazing that I almost quit my graduate program.

Heckel. Further, I want to say thank you to the EECS administrative staffs. Among them, Shirley Salanio and Kim Kail have been extremely helpful to resolve many administrative issues.

The long journey has been well flavored with amazing memories I had outside of work. I really had fun playing Starcraft 2 for Lazarus Gaming and for the UC Berkeley collegiate Starcraft team; I am really proud that I was able to make some contribution to the Korean Graduate Student Association at Berkeley as a president; of course, playing soccer for the Berkeley-Korean students' team for 6 years has been my best hobby. I will miss those times and all the friends I had fun with. It was very fortunate to have great Korean friends in the EECS department. Among them, I really acknowledge my gratitude to Jaehwak Kwak, who has been sincerely caring me for the last 6 years as the closest senior of mine. Thanks to Sangyoon Han, Jae Yeon (Claire) Baek, and Dongjin (DJ) Seo for being my best and closest Korean friends. I also want to express my gratitude to all my friends who have been constantly inspired and supported me. I cannot list all of them here but a (noisy) subset of them is as follows: Hyun Oh Song, Jongwoo Lim, Insoon Yang, Kunwoo Lee, Gil Young Cho, Sun Choi, Hwajong Yoo, Jihoon Jang, and Albert No. Also, thanks to my old friends in Korea.

My special thanks go to my past and current housemates: Jaeyoung Choi, Peter Jinwoo Shin, Chan Kim, Eunkwang Joo, and Daniel Oh [2]. They always have been kind enough to live with (and feed) a mere child for many years. If I was not fortunate enough to live with Jaeyoung Choi, I must have suffered from malnutrition. Aside from food, we had so much fun watching TV shows and movies, eating late night food, playing soccer, playing video games, sailing, drinking, fiddling around, hanging out, consulting each other, and laughing over childish jokes. Only with these people, bouncing back and forth between work and home has been extremely delightful in both ways.

I really appreciate the generous support of Korea Foundation for Advanced Studies (KFAS), which had funded my graduate program for 5 years.

My last acknowledgement goes to my beloved family. Thank you for your unlimited support, trust, and love. 감사합니다, 존경합니다, 그리고 사랑합니다.

[2]In the decreasing order of the frequency of feeding me.

Chapter 1

Introduction

Modern data centers have been providing exponentially increasing computing and storage resources, which have been fueling core applications ranging from search engines in the early 2000's to the real-time, large-scale data analysis of today: real time analysis of a huge amount of streaming data from millions of mobile devices or sensors, so called Big data analysis, has transformed how we view data and create value from data; machine learning has quickly evolved to the point where machines can nearly match or even exceed human performance in a diverse range of tasks from classical prediction tasks (e.g. digit recognition [22], image classification [116], and speech recognition [52]) to playing complicated games such as Go [109]. All these breakthroughs were made possible only due to the scalability in computing and storage capacity offered by modern large-scale clusters.

While classical computing and storage clusters were composed of a small number of expensive, custom-designed high-end machines (think IBM mainframes in the 1970's), modern large-scale clusters consist of more than tens of thousands of "commodity" hardware nodes, connected through general-purpose network infrastructure (think Google or Facebook today). Specifically, modern distributed systems like Apache Spark [128] and computational primitives like MapReduce [30] have gained significant traction, as they have enabled the execution of production-scale tasks on data sizes of the order of terabytes in such clusters, comprising individually small and unreliable low-end commodity hardware.

In order to develop and deploy sophisticated solutions and tackle large-scale problems in machine learning, science, engineering, and commerce, it is important to understand and optimize novel and complex trade-offs across the multiple dimensions of computation, communication, storage, and the accuracy of results. Moreover, given the individually unpredictable nature of the underlying nodes in these systems, we face the challenge of securing fast and high-quality algorithmic results in the face of uncertainty. This, coupled with the high level of complexity and heterogeneity of the component hardware, introduces significant *delays* that represent a key bottleneck to attaining the promised speed-ups of these large systems.

In this thesis, distributed storage and computing systems are viewed through a *coding-theoretic lens*. The role of codes in providing resiliency against noise has been studied for decades in many other engineering contexts, especially communication systems, and is part of our everyday infrastructure (smartphones, laptops, WiFi and cellular systems, etc.). Since the performance of distributed systems is also significantly affected by anomalous system behavior and bottlenecks [29], which we call "system noise", there is an exciting opportunity for codes to endow distributed systems with *robustness* against such system noise.

That is, our key observation –

$$\textit{Channel noise : Communication systems} \equiv \textit{System noise : Distributed systems}$$

– forms the key motivation of this thesis, and further raises the fundamental question:

Can we use codes to guarantee robust speedups in distributed storage and computing systems?

Codes are usually perceived as *slow* because in order to achieve a high degree of reliability, they require some redundancy of system resources as well as add encoding/decoding complexity, which are likely to *slow down* systems. This thesis studies how codes can *speed up* large-scale distributed storage and computing systems, hinting at a similar transformational role of codes in the way next-generation distributed storage and computing systems are designed and deployed.

1.1 Main Contribution of the Thesis

The backbone of these large and complex platforms consists of three functional layers: a computational layer, a communication layer, and a storage layer. The role of codes for *each* of these layers is addressed in this thesis, and the main contributions of the thesis are summarized as follows.

- The first main contribution of the thesis is **to propose a queueing model of distributed coded storage systems, to analyze the proposed model, and to study data retrieval performance of coded storage systems.** Codes have already begun to transform the evolution of large-scale distributed storage systems in modern data centers under the umbrella of regenerating and locally repairable codes, which are also having a major impact on industry. However, most of the existing works have been focusing on providing higher data durability while minimizing the storage overhead and the network bandwidth/connectivity/computation overhead of repairing node failures. Regarding whether codes can *speed up* distributed data storage systems, only little has been studied. Our analysis shows that data retrieval performance of coded storage systems is superior to that of uncoded storage systems.

- Many systems possess the possibility of serving the same request in multiple ways, potentially reducing the service time variability. For instance, distributed data storage systems, which inherently have data redundancy in the form of replication or codes, can serve a data retrieval request at more than one storage node. Scheduling of redundant requests can potentially speed up such systems as long as the increased amount of system load is sufficiently low. In this thesis, **optimal scheduling policies are characterized** in various scenarios, and it is shown that one can achieve a significant speed-up by scheduling redundant requests.

- Runtime performance of distributed algorithms is heavily affected by *stragglers*, i.e. "workers" that are substantially *slower* than the others in a distributed computing cluster. By *encoding* distributed algorithms and viewing such stragglers as *erasures*, we propose a novel solution, called 'coded computation', which we show significantly improves the runtime performance of distributed algorithms. This opens up **a new application of codes for the design of efficient distributed algorithms.**

- Parallel/distributed machine learning algorithms can achieve higher statistical efficiency if input data can be shuffled during the runtime of the algorithm. Further, all existing convergence guarantees on parallel/distributed machine learning algorithms assume perfect data-shuffling. Shuffling training data of a large size, however, can completely exhaust the network resource of the computing cluster, which in turn makes data shuffling practically infeasible. We propose a *coded shuffling algorithm*, which can significantly reduce the network burden of the data shuffling operation, **enabling statistically-efficient distributed machine learning algorithms** as well as bridging the chasm between theory and practice.

1.2 Related Works

In this section, we provide a high-level overview of some selected related works. More extensive surveys of the related works are provided in the following chapters.

1.2.1 Coded Storage Systems

Codes have already begun to transform the evolution of large-scale distributed storage systems in modern data centers under the umbrella of regenerating and locally repairable codes for distributed storage [34, 89, 114, 117, 19, 85, 44, 81, 84, 48, 53, 83, 60, 95, 88, 108], which also have major impacts on industry [54, 101, 91, 93]. All of these works have been focusing on providing higher data durability, while minimizing the storage overhead and the network bandwidth/connectivity/computation overhead of repairing node failures. Huang et al. [55] propose the first queueing model of distributed coded storage systems, and show that one can retrieve data *faster* from a coded storage system than an uncoded storage system. However, only a limited class of codes is studied in this work, and data retrieval performance of a more general class of codes was still open. The new framework proposed in this thesis can be used to model and to analyze a larger class of codes, i.e., the class of MDS (maximum distance separable) codes [26]: using this new framework, we rigorously analyze the data retrieval performance of MDS-coded storage systems, and show that it is strictly superior to that of uncoded storage systems.

1.2.2 Straggler Mitigation and Scheduling of Redundant Requests

The straggler problem has been widely observed in distributed computing clusters. The authors of [29, 4] show that running a computational task at a computing node often involves unpredictable latency due to several factors such as network latency, shared resources, maintenance activities, and power limits. Further, they argue that stragglers cannot be completely removed from a distributed computing cluster. One approach to mitigating the adverse effect of stragglers is based on efficient straggler detection algorithms. For instance, the default scheduler of Hadoop constantly detects stragglers while running computational tasks, and relaunches the detected straggler tasks at some other available node. In [129, 4], modifications to the existing straggler detection algorithm are proposed. Another line of approaches is based on appropriate modification of the algorithms: one can design distributed algorithms that are robust to *asynchronous* or delayed updates from the workers. Such robust distributed algorithms can continuously make progress without needing to wait for all the responses from the workers, and hence the overall runtime of these algorithms is less affected by stragglers [1, 99].

Scheduling of redundant requests has also been proposed as a way to tackle the straggler problem: by replicating tasks and scheduling the replicas, the runtime of distributed algorithms can be significantly improved [62, 3, 103, 124, 43, 20, 66]. By collecting outputs of the fast-responding nodes (and potentially canceling all the other slow-responding replicas), such replication-based scheduling algorithms can reduce latency.

1.2.3 Data Shuffling and Communication Overhead

Distributed learning algorithms running on large-scale networked systems have been extensively studied in the literature [11, 80, 18, 8, 35, 21, 31, 74, 63, 111, 67]. Many of the distributed algorithms that are implemented in practice share a similar algorithmic “anatomy": the data set is split among several cores or nodes, each node trains a model locally, then the local models are averaged, and the process is repeated. While training a model with parallel or distributed learning algorithms, it is common to randomly re-shuffle the data a number of times [98, 99, 16, 130, 45, 56, 98, 45, 56]. This essentially means that after each shuffling, the learning algorithm will go over the data in a different order than before. However, the statistical benefits of data shuffling do not come for free: each time a new shuffle is performed, the *entire* dataset is communicated over the network of nodes, resulting in a heavy communication burden.

Our coded shuffling algorithm is built upon the coded caching framework of Maddah-Ali and Niesen [77]. Coded caching is a technique to reduce the communication rate in content delivery networks, mainly motivated by video-sharing applications [78, 86, 61, 57]. The authors of [68] propose coded MapReduce that reduces the communication cost in the process of transferring the results of mappers to reducers.

1.3 Organization

The rest of the thesis is organized as follows:

Chapter 2. In this chapter [1], we study how codes can speed up modern distributed data storage systems. Codes that provide significantly increased data durability and lower overhead of

[1]This chapter is partly done in collaboration with Nihar Shah and Longbo Huang. A part of this chapter was presented in IEEE ISIT 2014 [105], and the whole chapter is under review for publication in IEEE Transactions on Information Theory.

repairing lost data blocks have been extensively studied, and these efficient storage codes are being increasingly adopted in modern data centers. However, it is unclear if such codes can possibly result in compromised 'data retrieval' performance. We introduce a queueing model for distributed data storage systems with codes, called **the MDS queue**, and provide rigorous analysis of how fast one can retrieve data from such systems. Our analysis reveals that data retrieval performance under coded storage systems is superior than uncoded storage systems.

Chapter 3. In this chapter [2], we study how one can improve the data retrieval performance by scheduling **redundant requests**. Distributed data storage systems have inherent data redundancy for maintaining a sufficient level of data durability. Therefore, one can always retrieve data faster by requesting the relevant data blocks from more than the requisite number of storage nodes. This, however, may increase the effective load of the system, and it is not clear whether the overall data retrieval performance improves or degrades. We study when scheduling redundant requests can improve data retrieval performance distributed storage systems (with codes or replication).

Chapter 4. In this chapter [3], we continue our study of scheduling redundant requests, motivated by system-level observations. Scheduling redundant requests in a distributed system is straightforward but *cancelling* straggling requests is not: cancellation of already-issued-requests usually takes non-negligible time, or sometimes is infeasible. This brings us a new question whether one can still improve latency performance even with such a **cancellation overhead**. Focusing on the case of distributed data storage systems with replicated data, we show that one can still achieve improved data-retrieval performance, and that *dynamic* scheduling policies can provide strictly better performance than static policies.

Chapter 5. In this chapter [4], changing gears from storage to computation, we study how one can speed up distributed algorithm with codes. From the previous chapters, we learned that an appropriate scheduling of redundant requests can significantly speed up data-retrieval performance in distributed storage systems with codes. Can we apply a similar idea to speed up distributed computing? We show that by carefully orchestrating a distributed paradigm, a judiciously chosen amount of **computing redundancy** can be injected into the algorithm. Due to this injected redundancy, such 'coded' distributed algorithms can be made very robust to stragglers, the nodes that are significantly slower than the others. We call this new paradigm of distributed computing **coded computation**, and show how one can design efficient coded distributed algorithms.

[2]This chapter is partly done in collaboration with Nihar Shah. A part of this chapter was presented in Allerton 2013 [104], and the whole chapter is published in IEEE Transactions on Communications [103].

[3]This chapter is partly done in collaboration with Ramtin Pedarsani. A part of this chapter was presented in Allerton 2015 [66], and the whole chapter is under review for publication in IEEE/ACM Transactions on Networking.

[4]This chapter is partly done in collaboration with Maximilian Lam, Ramtin Pedarsani, Dimitris Papailiopoulos, and Jichan Chung. A part of this chapter was presented in NIPS 2015 Workshop on Machine Learning Systems [65], and will be presented in IEEE ISIT 2016.

Chapter 6. Finally, in this chapter [5], we show how codes can speed up data-shuffling in a distributed cluster. It has been empirically observed that shuffling training data between distributed workers makes parallel machine learning algorithms – such as parallel stochastic gradient descent – converge faster. Moreover, all of the known convergence guarantees on parallel/distributed machine learning algorithms assume perfect data-shuffling. We show how **coded shuffling** can significantly curtail the network overhead involved with such shuffling procedure.

In **Chapter 7**, we conclude the thesis with the summary of the results and important future research directions.

[5]This chapter is partly done in collaboration with Maximilian Lam, Jichan Chung, Ramtin Pedarsani, Dimitris Papailiopoulos. A part of this chapter was presented in NIPS 2015 Workshop on Machine Learning Systems [65], and will be presented in IEEE ISIT 2016.

Chapter 2

Data Retrieval Performance of Distributed Storage Systems with Coded Data

2.1 Introduction

Two primary objectives of a storage system are to provide reliability and availability of the stored data: the system must ensure that data is not lost even in the presence of individual component failures, and must be easily and quickly accessible to the user whenever required. The classical means of providing reliability is to employ the strategy of replication, wherein identical copies of the (entire) data are stored on multiple servers. However, this scheme is not very efficient in terms of the storage space utilization. The exponential growth in the amount of data being stored today makes storage an increasingly valuable resource, and has motivated data-centers today to increasingly turn to the use of more efficient erasure codes [51, 41, 54, 91].

The most popular, and also most efficient, erasure codes are the class of the Maximum-Distance-Separable (MDS) codes, e.g., Reed-Solomon codes. An MDS code is typically associated to two parameters n and k. Under an (n, k) MDS code, a file is encoded and stored in n servers such that (a) the data stored in *any* k of these n servers suffice to recover the entire file, and (b) the storage space required at each server is $\frac{1}{k}$ of the size of the original file. [1]

While the reliability properties of erasure codes are very well understood, much less is known about their latency performance. In this chapter, we study coded data storage systems based on MDS codes through the lens of queueing theory. We term the queue resulting from the use of codes that allow for recovery of the data from any k of the n servers as "the MDS queue". To understand this queueing-theoretic perspective, consider a simple example with $n = 4$ and $k = 2$. Files $\{F_1, F_2, \ldots\}$ are to be stored in the 4 servers such that no data is lost upon failure of any $(n - k) = 2$ of the $n = 4$ servers. This is achieved via a $(4,\ 2)$ MDS code under which each file is partitioned into two halves $F_i = [f_{i,1}\ f_{i,2}]$, and the 4 servers store $f_{i,1}$, $f_{i,2}$, $(f_{i,1} + f_{i,2})$, and $(f_{i,1} + 2f_{i,2})$ respectively for every i. Requests for reading individual files arrive as a stochastic

[1]While a more precise definition of an MDS code is that it is a code that satisfies the 'Singleton bound' [76], the definition in the main text will suffice for the purposes of this chapter.

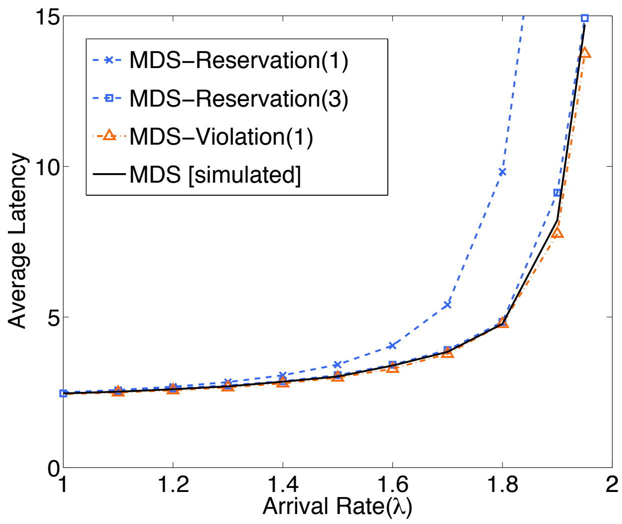


Figure 2.1: **The average latency of a centralized MDS queue using an MDS code with $\mathrm{n} = 10$ and $\mathrm{k} = 5$.** The service time of each job is assumed to be drawn from an exponential distribution with rate $\mu = 1$. The curve titled 'MDS' corresponds to simulations of the exact MDS queue. Also plotted are the analytically computed latencies of the lower bounds (MDS-Reservation(t) queues) and upper bounds (MDS-Violation(t) queues) presented in this chapter.

process, which are buffered and served by the system, and the resulting queue is termed an MDS queue (this will be formalized later in the chapter).

We first study *centralized* MDS queues, which are MDS queues with a central buffer that accepts all incoming requests. While having a centralized buffer allows for efficient load balancing, it comes at the cost of having a centralized architecture, which requires additional system-level overheads. This cost may be prohibitive for large-scale applications, and hence we subsequently study *decentralized* MDS queues as well. A decentralized MDS queue has n servers with their individual buffers, and any request for reading a file is directly scheduled to a randomly chosen set of k servers upon the arrival of the request. This model precisely captures large-scale systems where clients have a limited view of the whole system and submit their jobs to a part of the system at random.

An exact analysis of the MDS queue is hard in general. The Markov chain representation of the centralized MDS queue has a state space that is infinite in at least k dimensions, and the Markov chain representation of the decentralized MDS queue has a state space that is infinite in at least n dimensions. Furthermore, in both settings, the transitions in the respective Markov chains are tightly coupled across dimensions. The highly complex structures of the Markov chains make performance analysis of the MDS queues computationally challenging. In this chapter, we present insightful scheduling policies that provide upper and lower bounds on the performance of the MDS queue, which allow for convenient computation of various metrics of the latency, either analytically or numerically. These bounds are observed to be quite tight for a wide range of system parameters.

Fig. 2.1 illustrates the latency performance of centralized MDS queues. The lower bounds (the 'MDS-Reservation(t)' scheduling policies) and the upper bounds (the 'MDS-Violation(t)' scheduling policies) presented in this chapter are both indexed by a parameter 't'. An increase in the value of t results in tighter bounds, but also increases the complexity of analysis. Furthermore, both classes of scheduling policies converge to the MDS scheduling policy as t$\rightarrow \infty$. However, we observe that the performance of the MDS-Reservation(t) queue is very close to that of the centralized MDS queue for very small values of t (as small as t $= 3$), and the performance of the upper bounds MDS-Violation(t) closely follow that of the MDS queue for values of t as small as t $= 1$. This can be observed in Fig. 2.1. The MDS-Reservation(t) scheduling policies presented here are themselves practical alternatives to the MDS scheduling policy, since they require maintenance of a smaller state, while offering highly comparable performance.

We also consider the problem of degraded reads (i.e., reading of partial data) in distributed storage systems, that has recently attracted considerable interest in the coding-theory community. We employ the framework of the centralized MDS queue to understand and compare, from a queueing theoretic viewpoint, different methods of performing degraded reads.

We summarize our contributions as follows.

(i) We model centralized/decentralized queueing systems that arise in data storage systems based on MDS codes.

(ii) We propose the scheduling policies that form upper and lower bounds on the latency performance of the centralized MDS queues.

(iii) We analyze the latency performance of degraded reads using the centralized MDS queue.

(iv) We find upper and lower bounds on the latency performance of decentralized MDS queues.

(v) We provide extensive simulation results to show how useful the proposed bounds are.

While our analysis of the MDS queues successfully captures the essence of queueing behavior of data storage systems, it is important to note the limitations of the model. We assume that every request asks for an entire file (e.g., as in Facebook's warehouse cluster [91]), but the current model does not capture a system where requests may also ask for partial data blocks. We also make a few simplifying assumptions for analytical tractability: homogeneity of the servers and requests, and the memoryless service time distribution.

The rest of the chapter is organized as follows. Section 2.2 discusses related literature. Section 2.3 presents the centralized MDS queue system model. Section 2.4 gives an overview of the general approach, and also introduces the notation employed in the chapter. Section 2.5 presents the proposed MDS-Reservation(t) queues that lower bound the performance of the centralized MDS queue. Section 2.6 presents the proposed MDS-Violation(t) queues that upper bound the performance. Section 2.7 presents analyses and comparisons of these queues. Section 2.8 presents and analyzes the decentralized MDS queues. Section 2.9 presents conclusions and discusses open problems. The appendix contains proofs of the theorems presented in the chapter.

2.2 Related Works

In this section, we review literature that is most closely related to the setting of this chapter.

A class of queues closely related to the MDS queue is the class of *fork-join queues* [7], and in particular, fork-join queues with variable subtasks [120]. The classical setup of fork-join queues assumes each batch to consist of n jobs, while the setup of fork-join queues with variable subtasks assumes k jobs per batch for some parameter $k \leq n$. However, under a fork-join queue (with variable subtasks), each job must be served by a particular *pre-specified* server, while under an MDS queue, the k jobs of a batch may be processed at any *arbitrary* set of k servers and this choice is governed by the scheduling policy.

Recent works have shown that coded storage systems can achieve improved data serving performance in terms of blocking probability, capacity, and latency. In [37], the authors analyze the coded storage systems that do not have buffers for incoming jobs, and study how codes can reduce the blocking probability compared to that in uncoded storage systems; further, the authors of [38] show that coding can achieve higher capacity in point-to-multipoint storage area networks. The authors of [23] show that coding data across multiple memory units improves read/write capacity of network switches and routers. The authors of [107] provide a general model of distributed storage systems that handle both data retrieval requests and data repair requests, and numerically evaluate a variety of storage codes under the new model. In [127], which appeared subsequent to our conference publication, the authors consider the problem of jointly optimizing latency and cost of coded storage systems. They first find the upper bound on the latency of probabilistic scheduling policies for the coded storage systems with decentralized, heterogenous storage nodes; then, they formulate and solve a joint latency and storage optimization problem. However, a lower bound on the latency is not presented in the paper. Moreover, under the setting of the present chapter, our upper bound (see Theorem 2.14) provides a much tighter bound. For instance, when $\frac{k}{n}$ is fixed and k increases, the proposed bound in [127] scales as $\mathcal{O}(\sqrt{k})$, while our bound of $\mathcal{O}(\log k)$ gives a much sharper result.

A few notable works also study the construction of new codes that are optimized for the data retrieval performance. The authors of [119] study the problem of multilevel diversity coding with regeneration, which allows for different levels of fault tolerance and latency performance for different classes of content; queueing theoretic analysis for such multilevel coding systems is provided in [64]. In [75], the authors propose a coded storage system in which the original data is spread across the storage nodes along with the coded chunks, while maintaining the MDS property of the system. In such a system, the data can be retrieved either by retrieving coded chunks from any k nodes or by collecting the uncoded chunks from all the nodes.

In a system that employs a (n, k) erasure code, the latency of serving the requests can potentially be reduced by sending the requests redundantly to more than k servers. The request is deemed served when it is served in any one of these ways. The other copies of this request may now be removed from the system. In Chapter 3, we will show that scheduling of redundant requests can potentially speed up data retrieval performance.

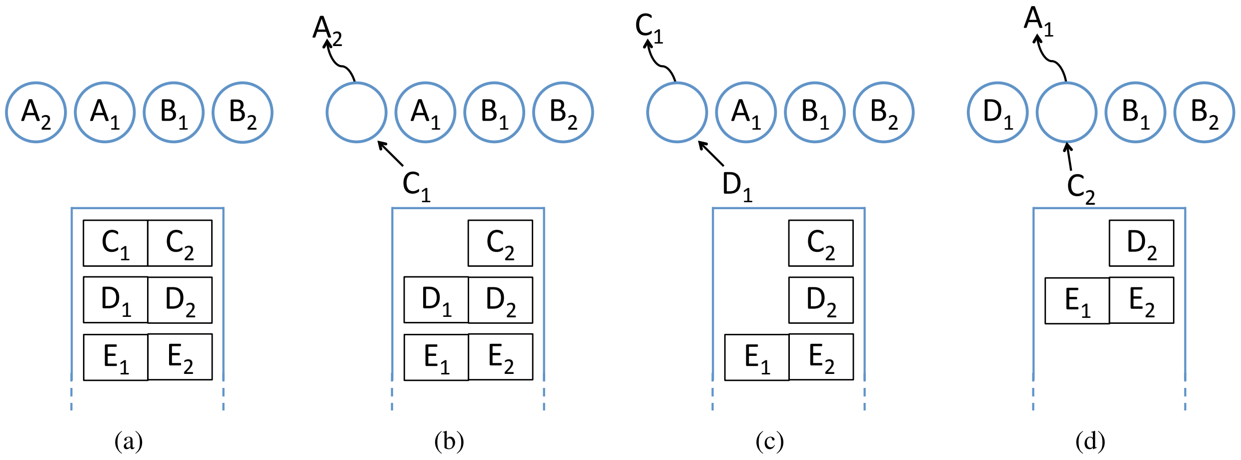


Figure 2.2: **Functioning of the MDS queue.**

2.3 The Centralized MDS Queue System Model

We now describe a queueing theoretic model of a centralized system employing an MDS code. The model of a decentralized MDS queue is provided in Section 2.8. As discussed previously, under an MDS code, a file can be retrieved by downloading data from any k of the servers. We model this by treating each request for reading a file as a *batch* of k *jobs*. The k jobs of a batch represent reading of k encodings of the file from k servers. A batch is considered as served when k of its jobs have been served. For instance, in the example of the $(n = 4,\ k = 2)$ system of Section 2.1, a request for reading file F_i (for some i) is treated as a batch of two jobs. To serve this request, the two jobs may be served by any two of the four servers; for example, if the two jobs are served by servers 2 and 3, then they correspond to reading $f_{i,2}$ and $(f_{i,1} + f_{i,2})$ respectively, which suffice to obtain F_i. We assume homogeneity among files and among servers: the files are of identical size, and the n servers have identical performance characteristics. Such a queueing system is termed as a centralized MDS queue, or simply an MDS queue.

Definition 2.1 ((Centralized) MDS queue)**.** A (centralized) MDS queue is associated with four parameters (n, k) and $[\lambda, \mu]$.

- There are n identical servers
- Requests enter a (common) buffer of infinite capacity
- Requests arrive as a Poisson process with rate λ
- Each request comprises a *batch* of k *jobs*
- Each of the k jobs in a batch can be served by an *arbitrary* set of k *distinct* servers
- The service time for a job at any server is exponentially distributed with rate μ, independent of all else

- The jobs are processed in order, i.e., among all the waiting jobs that an idle server is allowed to serve, it serves the one which had arrived the earliest.

Algorithm 1 formalizes the scheduling policy of the MDS queue.

Algorithm 1 MDS scheduling policy

```
On arrival of a batch
  Assign as many of its jobs as possible to idle servers
  Append remaining jobs (if any) as new batch at end of buffer
On departure from a server (say, server $s$)
  If $\exists$ at least one batch in the buffer such that no job of this batch has been served by $s$
    Among all such batches, find batch that arrived earliest
    Assign a job from this batch to $s$
```

The following example illustrates the functioning of the MDS scheduling policy and the resulting MDS queue.

Example 2.1. *Consider the MDS*$(n = 4,\ k = 2)$ *queue, as depicted in Fig. 2.2. Here, each request comes as a batch of* $k = 2$ *jobs, and hence we denote each batch (e.g., A, B, etc.) as a pair of jobs* ($\{A_1, A_2\}$, $\{B_1, B_2\}$, *etc.). The two jobs in a batch need to be served by (any) two distinct servers. Denote the four servers (from left to right) as servers* $1,\ 2,\ 3$ *and* 4. *Suppose that the system is in the state as shown in Fig. 2.2(a), wherein the jobs* A_2, A_1, B_1 *and* B_2 *are being served by the four servers, and there are three more batches waiting in the buffer. Suppose that server* 1 *completes serving job* A_2 *(Fig. 2.2(b)). This server is now free to serve any of the* 6 *jobs waiting in the buffer. Since jobs are processed only in order, it begins serving job* C_1 *(assignment of* C_2 *would also have been valid). Next, suppose that server* 1 *completes* C_1 *before any other servers complete their tasks (Fig. 2.2(c)). In this case, since server* 1 *has already served a job of batch* C, *it is not allowed to service* C_2. *However, it can service any job from the next batch* $\{D_1, D_2\}$, *and one of these two jobs is (arbitrarily) assigned to it. Finally, when one of the other servers completes its service, that server is assigned job* C_2 *(Fig. 2.2(d)).*

2.3.1 Other Applications

The MDS queue also arises in other applications that require diversity or error correction. For instance, consider a system with n processors, with the arriving jobs comprising computational tasks. It is often the case that the processors are not completely reliable [15], and may give incorrect outputs at random instances. In order to guarantee a correct output, a job may be processed at k different servers, and the results aggregated (perhaps by majority rule) to obtain the final answer. Such a system results in an MDS(n,k) queue (with some arrival and service-time distributions). In general, queues where jobs require diversity, for purposes such as security, error-protection etc., may be modelled as an MDS queue. [2] Finally, even in the setting of distributed storage systems, the MDS queue need not be restricted to analysing Maximum-Distance-Separable codes alone, and can be used for any code that supports recovery of the files from 'any k out of the n' servers.

2.3.2 Exact Analysis

An exact analysis of the MDS queue is hard. The difficulty arises from the special property of the MDS queue that each of the k jobs of a batch must be served by k *distinct* servers. Thus, a Markov-chain representation of this queue is required to have each state encapsulating not only the number of batches or jobs in the queue, but also the configuration of each batch in the queue, i.e., the number of jobs of each batch currently being processed, the number of jobs that have completed processing, and the number of jobs still waiting in the buffer. Thus, when there are b batches in the system, the system can have $\Omega\left(b^k\right)$ possible configurations. Since the number of batches b in the system can take any value in $\{0, 1, 2, \ldots\}$, this leads to a Markov chain which has a state space that has infinite states in at least k dimensions. Furthermore, the transitions along different dimensions are tightly coupled. This makes the Markov chain hard to analyze, and in this chapter, we provide scheduling policies for the MDS queue that lower/upper bound the exact MDS queue.

2.4 Our Approach and Notation for Latency Analysis of the MDS Queue

For each of the scheduling policies presented in this chapter (for lower/upper bounding the MDS queue), we represent the respective resulting queues as continuous time Markov chains. We show that these Markov chains belong to a class of processes known as *Quasi-Birth-Death (QBD)* processes (described below), and obtain their steady-state distribution by exploiting the properties of QBD processes. This is then employed to compute other metrics such as average latency, tail latency, maximum throughput, system occupancy, and waiting probability.

Throughout the chapter, we will refer to the entire setup described in Section 2.3 as the 'queue' or the 'system'. We will say that a batch is waiting (in the buffer) if at-least one of its jobs is still waiting in the buffer (i.e., has not begun service). We will use the term "i^{th} *waiting batch*" to refer to the batch that was the i^{th} earliest to arrive, among all batches currently waiting in the buffer. For example, in the system in the state depicted in Fig. 2.2(a), there are three waiting batches: $\{C_1, C_2\}$, $\{D_1, D_2\}$ and $\{E_1, E_2\}$ are the first, second and third waiting batches respectively.

We will frequently refer to an MDS queue as MDS(n,k) queue, and assume $[\lambda, \mu]$ to be some fixed (known) values. The system will always be assumed to begin in a state where there are no jobs in the system. Since the arrival and service time distributions have valid probability density

[2] An analogy that the academic will relate to is that of reviewing papers. There are n reviewers in total, and each paper must be reviewed by k reviewers. This forms an MDS(n,k) queue. The values of λ and μ considered should be such that $\frac{\lambda}{\mu}$ is close to the maximum throughput, modelling the fact that reviewers are generally busy.

functions, we will assume that no two events occur at exactly the same time. We will use the notation a^+ to denote $\max(a, 0)$.

Review of Quasi-Birth-Death (QBD) processes: Consider a continuous-time Markov process on the states $\{0, 1, 2, \ldots\}$, with transition rate λ_0 from state 0 to 1, λ from state i to $(i+1)$ for all $i \geq 1$, μ_0 from state 1 to 0, and μ from state $(i+1)$ to i for all $i \geq 1$. This is a birth-death process. A QBD process is a generalization of such a birth-death process, wherein, each state i of the birth-death process is replaced by a set of states. The states in the first set (corresponding to $i = 0$ in the birth-death process) is called the set of *boundary* states, whose behaviour is permitted to differ from that of the remaining states. The remaining sets of states are called the *levels*, and the levels are identical to each other (recall that all states $i \geq 1$ in the birth-death process are identical). The Markov chain is allowed to have transitions only within a level or the boundary, between adjacent levels, and between the boundary and the first level. The transition probability matrix of a QBD process is thus of the form

$$\begin{bmatrix} B_1 & B_2 & 0 & 0 & \cdots \\ B_0 & A_1 & A_2 & 0 & \cdots \\ 0 & A_0 & A_1 & A_2 & \cdots \\ 0 & 0 & A_0 & A_1 & \cdots \\ 0 & 0 & 0 & A_0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}. \tag{2.1}$$

Here, the matrices $B_0,\ B_1,\ B_2,\ A_0,\ A_1$ and A_2 represent transitions entering the boundary from the first level, within the boundary, exiting the boundary to the first level, entering a level from the next level, within a level, and exiting a level to the next level respectively. If the number of boundary states is q_b, and if the number of states in each level is q_ℓ, then the matrices B_0, B_1 and B_2 have dimensions $(q_\ell \times q_b)$, $(q_b \times q_b)$ and $(q_b \times q_\ell)$ respectively, and each of A_0, A_1 and A_2 have dimensions $(q_\ell \times q_\ell)$. The birth-death process described above is a special case with $q_b = q_\ell = 1$ and $B_0 = \mu_0$, $B_1 = 0$, $B_2 = \lambda_0$, $A_0 = \mu$, $A_1 = 0$, $A_2 = \lambda$. Figures 2.4, 2.6 and 2.9 in the sequel are also examples of QBD processes.

QBD processes are very well understood [36], and their stationary distribution is fairly easy to compute. In this chapter, we employ the *SMCSolver* software package [13] for this purpose. In the next two sections, we present scheduling policies which lower and upper bound the performance of the MDS queue, and show that the resulting queues can be represented as QBD processes. This representation makes them convenient to analyze, and this is exploited subsequently in the analysis presented in Section 2.7.

2.5 Lower Bounds: MDS-Reservation(t) Queues

This section presents a class of scheduling policies (and resulting queues), which we call the MDS-Reservation(t) scheduling policies (and MDS-Reservation(t) queues), whose performance lower bounds the performance of the MDS queue. This class of scheduling policies is indexed by a parameter ‘t’: a higher value of t leads to a better performance and a tighter lower bound, but on the downside, requires maintenance of a larger state and is more complex to analyze.

The MDS-Reservation(t) scheduling policy, in a nutshell, can be described as follows:

“*Apply the MDS scheduling policy, but with an additional restriction that for any $i \in \{t+1, t+2, \ldots\}$, the i^{th} waiting batch is allowed to move forward in the buffer only when all k of its jobs can move forward together.*”

We first describe in detail the special cases of $t = 0$ and $t = 1$, before moving on to the scheduling policy for a general t.

2.5.1 MDS-Reservation(0)

Scheduling policy

The MDS-Reservation(0) scheduling policy is rather simple: the batch at the head of the buffer may start service only when k or more servers are idle. The policy is described formally in Algorithm 2.

Algorithm 2 MDS-Reservation(0) Scheduling Policy

```
On arrival of a batch
  If number of idle servers $< k$
    append new batch at the end of buffer
  Else
    assign $k$ jobs of the batch to any $k$ idle servers
On departure from server
  If (number of idle servers $\geq k$) and (buffer is non-empty)
    assign $k$ jobs of the first waiting batch to any $k$ idle servers
```

Example 2.2. *Consider the MDS(n=4,k=2) queue in the state depicted in Fig. 2.2(a). Suppose that the server 2 completes processing job A_1 (Fig. 2.3(a)). Upon this event, the MDS scheduling policy would have allowed server 2 to take up execution of either C_1 or C_2. However, this is not permitted under MDS-Reservation(0), and this server remains idle until a total of at least $k = 2$ servers become idle. Now suppose that the third server completes execution of B_1 (Fig. 2.3(b)). At this point, there are sufficiently many idle servers to accommodate all $k = 2$ jobs of the batch $\{C_1, C_2\}$, and hence jobs C_1 and C_2 are assigned to servers 2 and 3.*

We note that the MDS-Reservation(0) queue, when $n = k$, is identical to a “split-merge queue” [50].

Analysis

Observe that under the specific scheduling policy of MDS-Reservation(0), a batch that is waiting in the buffer must necessarily have all its k jobs in the buffer, and furthermore, these k jobs go into the servers at the same time.

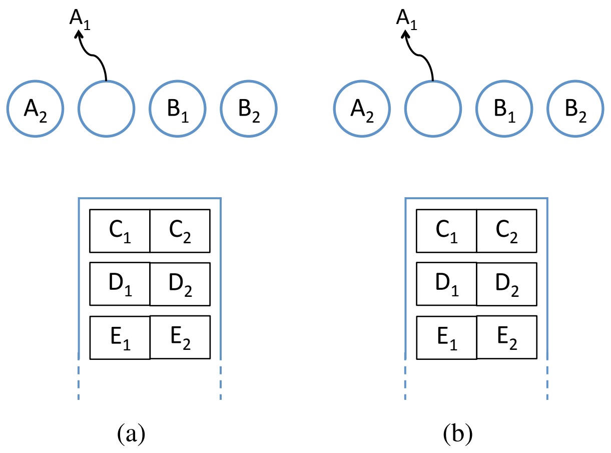


Figure 2.3: **An illustration of the MDS-Reservation(0) scheduling policy for a system with parameters** $(\mathbf{n}=\mathbf{4},\mathbf{k}=\mathbf{2})$. This policy prohibits the servers to process jobs from a batch unless there are k idle servers that can process all k jobs of that batch. As shown in the figure, server 1 is barred from processing $\{C_1, C_2\}$ in (a), but is subsequently allowed to do so when another server also becomes idle in (b).

We now describe the Markovian representation of the MDS-Reservation(0) queue. We show that it suffices to keep track of only the total number of jobs m in the entire system.

Proposition 2.1. *A Markovian representation of the MDS-Reservation(0) queue has a state space* $\{0,1,\ldots,\infty\}$*, and any state* $m \in \{0,1,\ldots,\infty\}$ *has transitions to: (i) state* $(m+k)$ *at rate* λ*, (ii) if* $m \leq n$ *then to state* $(m-1)$ *at rate* $m\mu$*, and (iii) if* $m > n$ *then to state* $(m-1)$ *at rate* $(n-(n-m) \bmod k))\mu$*. The MDS-Reservation(0) queue is thus a QBD process, with boundary states* $\{0,1,\ldots,n-k\}$*, and levels* $m \in \{n-k+1+jk,\ldots,n+jk\}$ *for* $j=\{0,1,\ldots,\infty\}$*.*

The state transition diagram of the MDS-Reservation(0) queue for $(n=4,k=2)$ is depicted in Fig. 2.4. The notation at any state is the number of jobs m in the system in that state. The set of boundary states are $\{0,1,2\}$, and the levels are pairs of states $\{3,4\}$, $\{5,6\}$, $\{7,8\}$, etc. Hence, the transition matrix is of the form (2.1) with

$$B_0=\begin{bmatrix}0&0&3\mu\\0&0&0\end{bmatrix}, B_1=\begin{bmatrix}-\lambda&0&\lambda\mu\\\mu&-(\mu+\lambda)&0\\0&2\mu&-(2\mu+\lambda)\end{bmatrix}, B_2=\begin{bmatrix}0&0\\\lambda&0\\0&\lambda\end{bmatrix}, \tag{2.2}$$

$$A_0=\begin{bmatrix}0&3\mu\\0&0\end{bmatrix}, A_1=\begin{bmatrix}-(3\mu+\lambda)&0\\4\mu&-(4\mu+\lambda)\end{bmatrix}, A_2=\begin{bmatrix}\lambda&0\\0&\lambda\end{bmatrix}. \tag{2.3}$$

Proposition 2.1 shows that the MDS-Reservation(0) queue is a QBD process, allowing us to employ the SMC solver to obtain its steady-state distribution. Alternatively, the MDS-Reservation(0) queue is simple enough to analyze directly as well. To this end, let $y(m)$

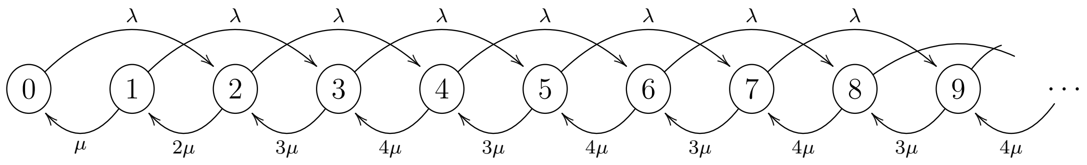


Figure 2.4: **State transition diagram of the MDS-Reservation(0) queue for $n = 4$ and $k = 2$.**

denote the number of jobs being served when the Markov chain is in state m. From the description above, this function can be written as:

$$y(m) = \begin{cases} m, & \text{if } 0 \leq i \leq n \\ n - ((n-m) \bmod k), & \text{if } m > n \ . \end{cases}$$

Let $\pi = [\pi_0 \ \pi_1 \ \pi_2 \ \cdots]$ denote the steady-state distribution of this chain. The global balance equation for the cut between states $(m-1)$ and m gives:

$$\pi_m = \frac{\lambda}{y(m)\mu} \left(\sum_{j=(m-k)^+}^{m-1} \pi_j \right) , \tag{2.4}$$

for every $m > 0$. Using these recurrence equations, for any given (n, k), the distribution π of the number of jobs in steady-state can be computed easily.

2.5.2 MDS-Reservation(1)

Scheduling Policy

The MDS-Reservation(0) scheduling policy discussed above allows the batches in the buffer to move ahead only when all k jobs in the batch can move together. The MDS-Reservation(1) scheduling policy relaxes this restriction for (only) the job at the head of the buffer. This is formalized in Algorithm 3.

Algorithm 3 MDS-Reservation(1) Scheduling Policy

```
On arrival of a batch
  If buffer is empty
    assign one job each from new batch to idle servers
  append remaining jobs of batch to the end of the buffer
On departure from server (say, server $s$):
  If buffer is non-empty and no job from first waiting batch has been served by $s$
    assign a job from first waiting batch to $s$
    If first waiting batch had only one job in buffer & there exists another waiting batch
      to every remaining idle server, assign a job from second waiting batch
```

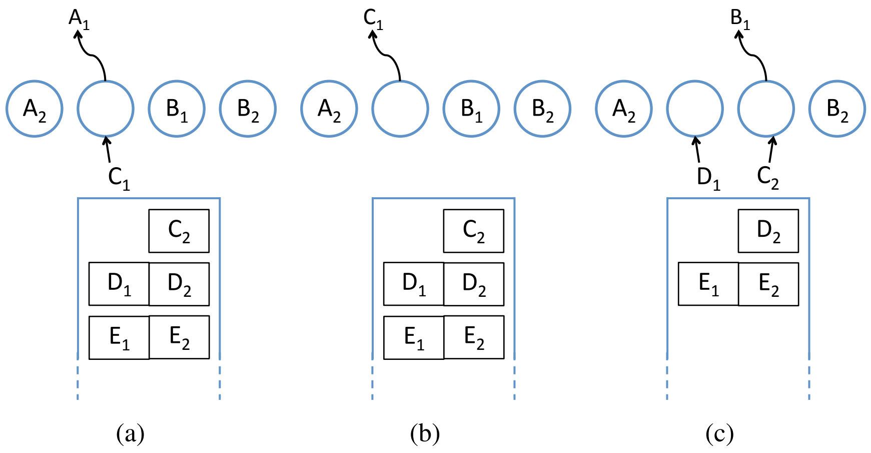


Figure 2.5: **An illustration of the MDS-Reservation(1) scheduling policy, for a system with parameters $(\mathbf{n} = \mathbf{4}, \mathbf{k} = \mathbf{2})$.** As shown in the figure, this policy prohibits the servers from processing jobs of the second or later batches (e.g., $\{D_1, D_2\}$ and E_1, E_2 in (b)), until they move to the top of the buffer (e.g., $\{D_1, D_2\}$ in (c)).

Example 2.3. *Consider the MDS(n=4,k=2) queue in the state depicted in Fig. 2.2(a). Suppose that server 2 completes processing job A_1 (Fig. 2.5(a)). Under MDS-Reservation(1), server 2 now begins service of job C_1 (which is allowed by the MDS scheduling policy, but was prohibited under MDS-Reservation(0)). Now suppose that server 2 finishes this service before any other server (Fig. 2.5(b)). In this situation, since server 2 has already processed one job from batch $\{C_1, C_2\}$, it is not allowed to process C_2. However, there exists another batch $\{D_1, D_2\}$ in the buffer such that none of the jobs in this batch have been processed by the idle server 2. While the MDS scheduling policy would have allowed server 2 to start processing D_1 or D_2, this is not permitted under MDS-Reservation(1), and the second server remains idle. Now, if server 3 completes service (Fig. 2.5(c)), then C_2 is assigned to server 3, allowing batch $\{D_1, D_2\}$ to move up as the first batch. This now permits server 2 to begin service of job D_1.*

Analysis

The following proposition describes the Markovian representation of the MDS-Reservation(1) queue. Each state in this representation is defined by two quantities: (i) the total number of jobs m in the system, and (ii) the number of jobs w_1 of the first waiting batch, that are still in the buffer.

Proposition 2.2. *The Markovian representation of the MDS-Reservation(1) queue has a state space $(w_1, m) \in \{0, 1, \ldots, k\} \times \{0, 1, \ldots, \infty\}$. It is a QBD process with boundary states $\{0, \ldots, k\} \times \{0, \ldots, n\}$, and levels $\{0, \ldots, k\} \times \{n - k + 1 + jk, \ldots, n + jk\}$ for $j = \{1, 2, \ldots, \infty\}$.*

The state transition diagram of the MDS-Reservation(1) queue for $(n = 4, k = 2)$ is depicted in Fig. 2.6.

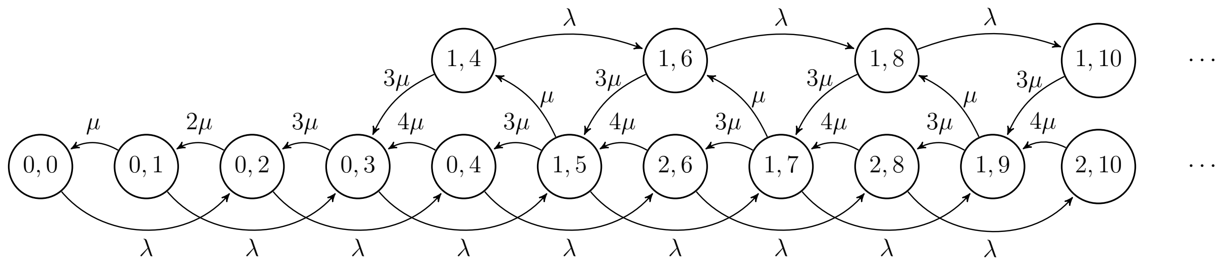


Figure 2.6: **State transition diagram of the MDS-Reservation(1) queue for $\mathbf{n = 4}$ and $\mathbf{k = 2}$.** The notation at any state is (w_1, m). The subset of states that are never visited are not shown. The set of boundary states are $\{0, 1, 2, 3, 4\} \times \{0, 1, 2\}$, and the levels are sets $\{5, 6\} \times \{0, 1, 2\}$, $\{7, 8\} \times \{0, 1, 2\}$, etc.

Note that the state space $\{0, 1, \ldots, k\} \times \{0, 1, \ldots, \infty\}$ has several states that will never be visited during the execution of the Markov chain. For instance, the states $(w_1 > 0, m \leq n - k)$ never occur. This is because $w_1 > 0$ implies existence of some job waiting in the buffer, while $m \leq n - k$ implies that k or more servers are idle. The latter condition implies that there exists at least one idle server that can process a job from the first waiting batch, and hence the value of w_1 must be smaller than that associated to that state, thus proving the impossibility of the system being in that state.

2.5.3 MDS-Reservation(t) for a General t

Scheduling Policy

Algorithm 4 formally describes the MDS-Reservation(t) scheduling policy.

Algorithm 4 MDS-Reservation(t) Scheduling Policy

```
On arrival of a batch
  If buffer has strictly fewer than t batches
    Assign jobs of new batch to idle servers
  Append remaining jobs of batch to end of buffer
On departure of job from a server (say, server $s$)
  Find $\hat{i}$=min{$i$≥1: $s$ has not served job of $i^{\text{th}}$ waiting batch}
  Let $b_{t+1}$ be the $(t+1)^{\text{th}}$ waiting batch (if any)
  If $\hat{i}$ exists & $\hat{i} \leq t$
    Assign a job of $\hat{i}^{\text{th}}$ waiting batch to $s$
    If $\hat{i} = 1$ & the first waiting batch had only one job in the buffer & $b_{t+1}$ exists
      To every remaining idle server, assign a job from batch $b_{t+1}$
```

The following example illustrates the MDS-Reservation(t) scheduling policy when t= 2.

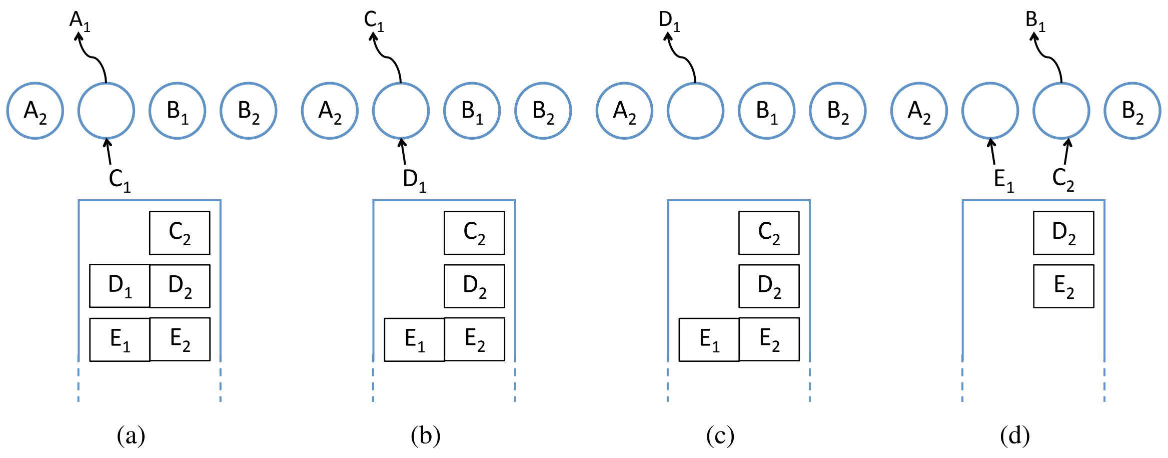


Figure 2.7: **An illustration of the working of the MDS-Reservation(2) scheduling policy, for a system with parameters** $(\mathbf{n}=\mathbf{4},\mathbf{k}=\mathbf{2})$. As shown in the figure, this policy prohibits the servers from processing jobs of the third and later batches (e.g., batch $\{E_1, E_2\}$ in (c)), until they move higher in the buffer (e.g., as in (d)).

Example 2.4. *(t=2). Consider the MDS(n=4,k=2) queue in the state depicted in Fig. 2.2(a). Suppose the second server completes processing job A_1 (Fig. 2.7(a)). Under the MDS-Reservation(2) scheduling policy, server 2 now begins service of job C_1. Now suppose that server 2 finishes this service as well, before any other server completes its respective service (Fig. 2.7(b)). In this situation, while MDS-Reservation(1) would have mandated server 2 to remain idle, MDS-Reservation(2) allows it to start processing a job from the next batch $\{D_1, D_2\}$. However, if the server also completes processing of this job before any other server (Fig. 2.7(c)), then it is not allowed to take up a job of the third batch $\{E_1, E_2\}$. Now suppose server 3 completes service (Fig. 2.7(d)). Server 3 can begin serving job C_2, thus clearing batch $\{C_1, C_2\}$ from the buffer, and moving the two remaining batches up in the buffer. Batch $\{E_1, E_2\}$ is now within the threshold of t= 2, allowing it to be served by the idle server 2.*

Analysis

Theorem 2.3. *The Markovian representation of the MDS-Reservation(t) queue has a state space $\{0, 1, \ldots, k\}^t \times \{0, 1, \ldots, \infty\}$. It is a QBD process with boundary states $\{0, \ldots, k\}^t \times \{0, \ldots, n-k+tk\}$, and levels $\{0, \ldots, k\}^t \times \{n-k+1+jk, \ldots, n+jk\}$ for $j = \{t, t+1, \ldots, \infty\}$.*

The proof of Theorem 2.3 also describes how one can obtain the configuration of the entire system from only the number of jobs in the system, under the MDS-Reservation(t) scheduling policies.

One can see that the sequence of MDS-Reservation(t) queues, as t increases, becomes closer to the MDS queue. This results in tighter bounds, and also increased complexity of the transition diagrams. The limit of this sequence is the MDS queue itself.

Proposition 2.4. *The MDS-Reservation(t) queue, when $t = \infty$, is precisely the MDS queue.*

2.6 Upper Bounds: MDS-Violation(t) Queues

In this section, we present a class of scheduling policies (and resulting queues), which we call the MDS-Violation(t) scheduling policies (and MDS-Violation(t) queues), whose performance upper bounds the performance of the MDS queue. The scheduling policies presented here relax the constraint of requiring the k jobs in a batch to be processed by k distinct servers. While the MDS-Violation(t) scheduling policies and the MDS-Violation(t) queues are not realizable in practice, they are presented here only to obtain upper bounds on the performance of the MDS queue.

The MDS-Violation(t) scheduling policy, in a nutshell, is as follows:

"*apply the MDS scheduling policy whenever there are t or fewer batches in the buffer; when there are more than t batches in the buffer, ignore the restriction requiring the k jobs of a batch to be processed by distinct servers.*"

We first describe the MDS-Violation(0) queue in detail, before moving on to the general MDS-Violation(t) queues.

2.6.1 MDS-Violation(0)

Scheduling Policy

The MDS-Violation(0) scheduling policy operates by completely ignoring the restriction of assigning distinct servers to jobs of the same batch. This is described formally in Algorithm 5.

Algorithm 5 MDS-Violation(0)

```
On arrival of a batch
  assign jobs of this batch to idle servers (if any)
  append remaining jobs at the end of the buffer
On departure from a server
  If buffer is not empty
    assign a job from the first waiting batch to this server
```

Note that the MDS-Violation(0) queue is identical to the MDS-Violation queue, i.e., an $M/M/n$ queue with batch arrivals.

The following example illustrates the working of the MDS-Violation(0) scheduling policy.

Example 2.5. *Consider the MDS(n=4,k=2) queue in the state depicted in Fig. 2.2(a). Suppose the first server completes processing job A_2, as shown in Fig. 2.8(a). Under the MDS-Violation(0) scheduling policy, server 1 now takes up job C_1. Next suppose server 1 also finishes this task before any other server completes service (Fig. 2.8(b)). In this situation, the MDS scheduling policy would prohibit job C_2 to be served by the first server. However, under the scheduling policy of MDS-Violation(0), we relax this restriction, and permit server 1 to begin processing C_2.*

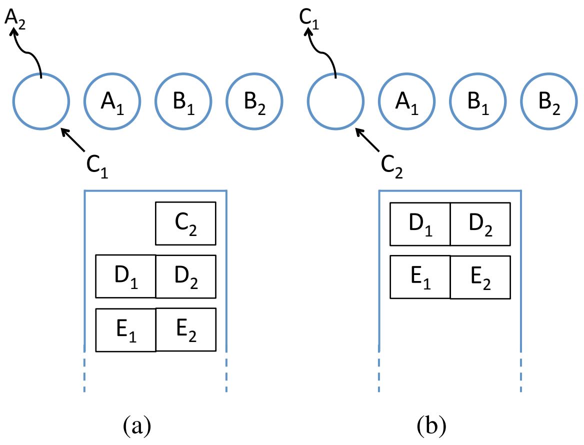


Figure 2.8: **An illustration of the working of the MDS-Violation(0) scheduling policy.** This policy allows a server to process more than one jobs of the same batch. As shown in the figure, server 1 processes both C_1 and C_2.

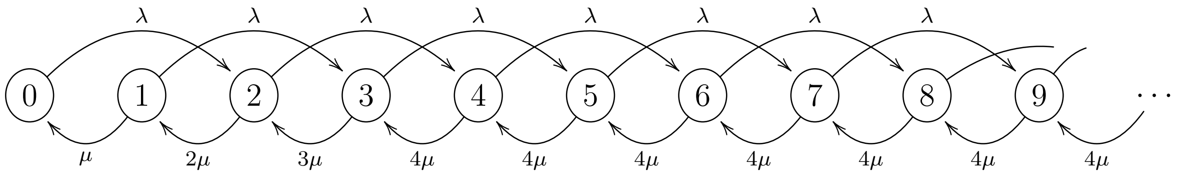


Figure 2.9: **State transition diagram of the MDS-Violation(0) queue for n = 4 and k = 2.** The notation at any state is the number of jobs m in the system in that state. The set of boundary states are $\{0, 1, 2, 3, 4\}$, and the levels are pairs of states $\{5, 6\}$, $\{7, 8\}$, etc. The transition matrix is of the form (2.1) with $B_0 = [0\ 0\ 0\ 0\ 4\mu\,; 0\ 0\ 0\ 0\ 0]$, $B_1 = [-\lambda\ 0\ \lambda\ 0\ 0;\ \mu\ -(\mu{+}\lambda)\ 0\ \lambda\ 0;\ 0\ 2\mu\ -(2\mu{+}\lambda)\ 0\ \lambda;\ 0\ 0\ 3\mu\ -(3\mu{+}\lambda)\ 0;\ 0\ 0\ 0\ 4\mu\ -(4\mu{+}\lambda)]$, $B_2 = [0\ 0\,;\ 0\ 0\,;\ 0\ 0\,;\ \lambda\ 0\,;\ 0\ \lambda]$, $A_0 = [0\ \ 4\mu\ ;\ \ 0\ \ 0]$, $A_1 = [-(4\mu{+}\lambda)\ \ 0\ ;\ \ 4\mu\ \ -(4\mu{+}\lambda)]$, $A_2 = [\lambda\ \ 0\ ;\ \ 0\ \ \lambda]$.

Analysis

We now describe a Markovian representation of the MDS-Violation(**0**) scheduling policy, and show that it suffices to keep track of only the total number of jobs m in the system.

Proposition 2.5. *The Markovian representation of the MDS-Violation(0) queue has a state space $\{0, 1, \ldots, \infty\}$, and any state $m \in \{0, 1, \ldots, \infty\}$, has transitions (i) to state $(m + k)$ at rate λ, and (ii) if $m > 0$, then to state $(m - 1)$ at rate $\min(n, m)\mu$. It is a QBD process with boundary states $\{0, \ldots, k\} \times \{0, \ldots, n\}$, and levels $\{0, \ldots, k\} \times \{n - k + 1 + jk, \ldots, n + jk\}$ for $j = \{1, 2, \ldots, \infty\}$.*

The state transition diagram of the MDS-Violation(0) queue for $n = 4, k = 2$ is shown in Fig. 2.9.

Proposition 2.5 shows that the MDS-Reservation(0) queue is a QBD process, allowing us to employ the SMC solver to obtain its steady-state distribution. Alternatively, the MDS-Violation(0) queue is simple enough to allow for a direct analysis as well. Let π_m denote the stationary probability of any state $m \in \{0, 1, \ldots, \infty\}$. Then, for any $m \in \{1, \ldots, \infty\}$, the global balance equation for the cut between states $(m-1)$ and m gives:

$$\pi_m = \frac{\lambda}{\min(m,n)\mu} \sum_{j=(m-k)^+}^{m-1} \pi_j \ , \tag{2.5}$$

for every $m > 0$. The stationary distribution of the Markov chain can now be computed easily from these equations.

2.6.2 MDS-Violation(t) for a General t

Scheduling policy

Algorithm 6 formally describes the MDS-Violation(t) scheduling policy.

Algorithm 6 MDS-Violation(t) Scheduling Policy

```
On arrival of a batch
  If buffer has strictly fewer than t batches
    Assign jobs of new batch to idle servers
  Else if buffer has t batches
    Assign jobs of first batch to idle servers
    If first batch is cleared
      Assign jobs of new batch to idle servers
  Append remaining jobs of new batch to end of buffer
On departure of job from a server (say, server $s$)
  If number of batches in buffer is strictly greater than t
    Assign job from first batch in buffer to this server
  Else
    Among all batches in buffer that $s$ has not served, find the one that arrived earliest; assign a
  job of this batch to $s$
```

Example 2.6. *(t=1). Consider a system in the state shown in Fig. 2.10(a). Suppose server 1 completes execution of job C_1 (Fig. 2.10(b)). In this situation, the processing of C_2 by server 1 would be allowed under MDS-Violation(0), but prohibited in the MDS queue. The MDS-Violation(1) queue follows the scheduling policy of the MDS queue whenever the total number of batches in the buffer is no more than 1, and hence in this case, server 1 remains idle. Next, suppose there is an arrival of a new batch (Fig. 2.10(c)). At this point there are two batches in the buffer, and the MDS-Violation(1) scheduling policy switches its mode of operation to allowing any server to serve any job. Thus, the first server now begins service of C_2 (Fig. 2.10(d)).*

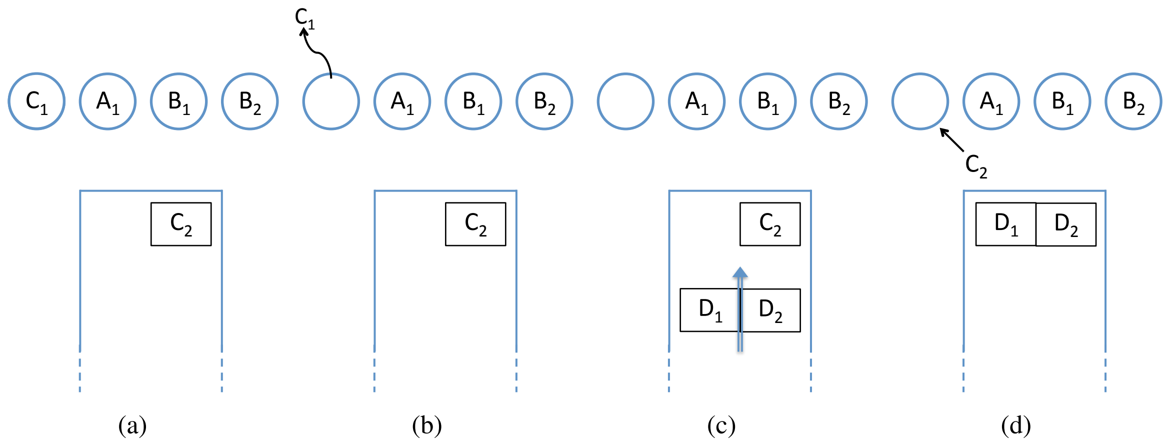


Figure 2.10: **An illustration of the working of the MDS-Violation(1) scheduling policy for $\mathbf{n = 4}$ and $\mathbf{k = 2}$.** This policy allows a server to begin processing a job of a batch that it has already served, unless this batch is the only batch waiting in the buffer. As shown in the figure, server 1 cannot process C_2 in (b) since it has already processed C_1 and C is the only waiting batch; this restriction is removed upon on arrival of another batch in the buffer in (d).

Analysis

Theorem 2.6. *The state transition diagram of the MDS-Violation(t) queue has a state space $\{0,1,\ldots,k\}^t \times \{0,1,2,\ldots\}$. It is a QBD process with boundary states $\{0,\ldots,k\}^t \times \{0,\ldots,n+tk\}$, and levels $\{0,\ldots,k\}^t \times \{n-k+1+jk,\ldots,n+jk\}$ for $j = \{t+1,t+2,\ldots,\infty\}$.*

The proof of Theorem 2.6 also describes how one can obtain the configuration of the entire system from only the number of jobs in the system, under the MDS-Violation(t) scheduling policies.

As in the case of MDS-Reservation(t) queues, one can see that the sequence of MDS-Violation(t) queues, as t increases, becomes closer to the MDS queue. On increase in the value of parameter t, the bounds become tighter, but the complexity of the transition diagrams also increases, and the limit of this sequence is the MDS queue itself.

Proposition 2.7. *The MDS-Violation(t) queue, when $t = \infty$, is precisely the MDS(n,k) queue.*

Remark 2.1. The class of queues presented in this section have another interesting intellectual connection with the MDS queue: the performance of an MDS(n,k) queue is lower bounded by the MDS-Violation(t) queue for all values of t.

2.7 Performance Comparison of Various Scheduling Policies

In this section we analyze the performance of the MDS-Reservation(t) and the MDS-Violation(t) queues. The analysis is performed by first casting these queues as quasi-birth-death processes (as shown in Theorems 2.3 and 2.6), and then building on the properties of quasi-birth-death processes to compute the average latency and throughput of each of these queues. Via simulations, we then validate these analytical results and also look at the performance of these queues with respect to additional metrics. We see that under each of these metrics, for the parameters simulated, the performance of the lower bound MDS-Reservation(t) queues (with $t = 3$) closely follows the performance of the upper bound MDS-Violation(t) queues (with $t = 1$).

2.7.1 Maximum Throughput

The maximum throughput is the maximum possible number of requests that can be served by the system per unit time.

Theorem 2.8. *Let $\rho^*_{\textit{Resv(t)}}$, ρ^*_{MDS}, and $\rho^*_{\textit{Vio(t)}}$ denote the maximum throughputs of the MDS-Reservation(t), MDS, and MDS-Violation(t) queues respectively. Then,*

$$\rho^*_{MDS} = \rho^*_{\textit{Vio}(t)} = \frac{n}{k} \ .$$

When k is treated as a constant,

$$\left(1 - O(n^{-2})\right) \frac{n}{k} \leq \rho^*_{\textit{Resv(t)}} \leq \frac{n}{k} \ .$$

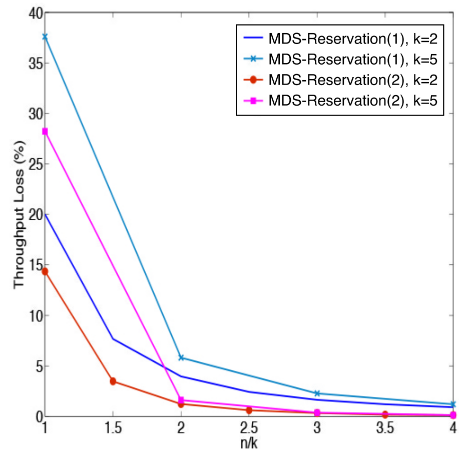
In particular, for any $t \geq 1$, when $k = 2$,

$$\left(1 - \frac{1}{2n^2 - 2n + 1}\right) \frac{n}{k} = \rho^*_{\textit{Resv(1)}} \leq \rho^*_{\textit{Resv(t)}} \ ,$$

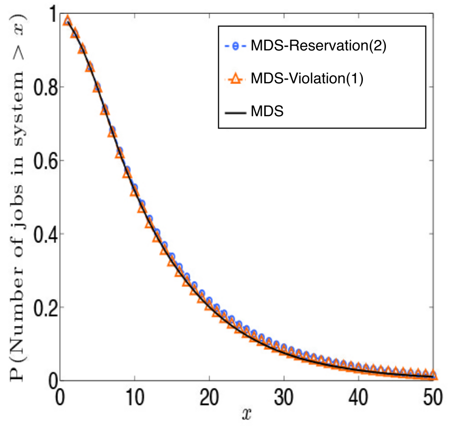
and when $k = 3$,

$$\left(1 - \frac{4n^3 - 8n^2 + 2n + 4}{3n^5 - 12n^4 + 22n^3 - 29n^2 + 26n - 8}\right) \frac{n}{k} = \rho^*_{\textit{Resv(1)}} \leq \rho^*_{\textit{Resv(t)}}$$

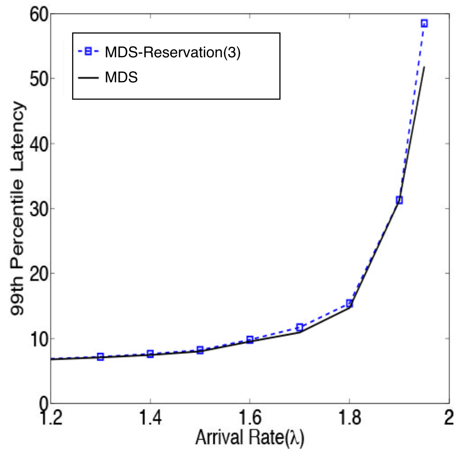
Using the techniques presented in the proof of Theorem 2.8, explicit bounds analogous to those for $k = 2,\ 3$ in Theorem 2.8 can be computed for $k \geq 4$ as well.



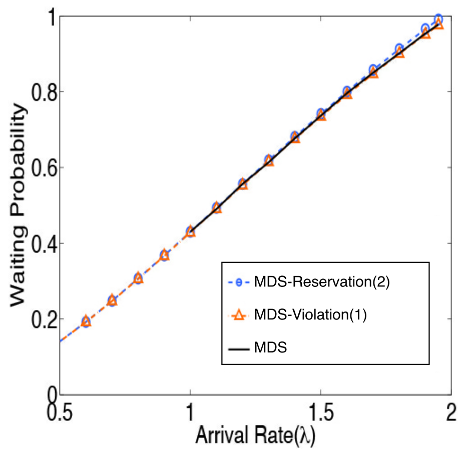
(a) Maximum throughput loss



(b) System occupancy



(c) The 99^{th} percentile of latency



(d) Waiting probability

Figure 2.11: **Performance comparison of various scheduling policies.** In (a), we plot the maximum throughput loss of the MDS-Reservation policies. In (b), we plot the steady state distribution of the system occupancy when $n = 10$, $k = 5$, $\lambda = 1.5$ and $\mu = 1$. In (c), we plot the 99^{th} percentile of latency when $n = 10$, $k = 5$ and $\mu = 1$. In (d), we plot the waiting probability when $n = 10$, $k = 5$ and $\mu = 1$.

Fig. 2.11(a) plots loss in maximum throughput incurred by the MDS-Reservation(1) and the MDS-Reservation(2) queues, as compared to that of the MDS queue. We can observe that for a fixed value of k, the throughput loss decreases as n increases. This is because the MDS-reservation(t) blocks at most $k-1$ servers, and blocking $k-1$ servers among n servers is less wasteful when n is large compared to k.

2.7.2 System Occupancy

The system occupancy at any given time is the number of jobs present (i.e., that have not yet been served completely) in the system at that time. This includes jobs that are waiting in the buffer as well as the jobs being processed in the servers at that time. The distribution of the system occupancy is obtained directly from the stationary distribution of the Markov chains constructed in sections 2.5 and 2.6. Fig. 2.11(b) plots the complementary cdf of the the number of jobs in the system in the steady state. Observe that the analytical upper and lower bounds from the MDS-Reservation(2) and the MDS-Violation(1) queues respectively are very close to each other.

2.7.3 Latency

The latency faced by a batch is the time from its arrival into the system till the completion of service of k of its jobs. Using the obtained steady-state distributions of the MDS-Reservation(t) and the MDS-Violation(t), we can analytically compute the average batch latencies of these systems. Given the steady-state distribution π of a system, for each state i, we can find the average latency d_i faced by a batch entering the system in state i. Since Poisson arrivals see time averages [126], the average latency faced by a batch in the steady state is given by $\sum_i \pi_i d_i$. Fig. 2.1 plots the average latency faced by a batch in the steady state. Observe that the performance of the MDS-Reservation(t) scheduling policy, for t as small as 3, is extremely close to that of the MDS scheduling policy and to the upper bounding MDS-Violation(1) scheduling policy.

We also look at the tails of the latency-distribution via simulations. Fig. 2.11(c) plots the 99th percentile of the distribution of the latency. Also observe how closely the bound MDS-Reservation(3) follows the exact MDS.

While the computation of the stationary distribution of the MDS-Reservation(t) and the MDS-Violation(t) is fairly easy, an exact analysis of these bounds is still difficult. A notable exception is the MDS-Reservation(1) with $k = 2$: one can exactly analyze the MDS-Reservation(1) with $k = 2$ by finding its steady-state distribution. We denote by D the batch latency of a randomly chosen batch in steady state.

Theorem 2.9. *If n is an even number that is larger than 2, and the system is stable, the average request latency of MDS-Reservation(1) with $(n, k = 2)$ is*

$$\mathbb{E}[D] \leq \frac{1}{2\lambda}\left[\sum_{l=1}^{n-1} l\pi_l + \sum_{m=0}^{\infty}(n+2m+1)\pi_{n+2m+1} + \sum_{m=0}^{\infty}(n+2m)(\pi^p_{n+2m} + \pi^g_{n+2m})\right] + \frac{n-1}{n-2}\frac{1}{2\mu} - \frac{1}{(n-2)(n-1)n\mu} - \frac{1}{2(n-1)\mu}, \tag{2.6}$$

where the steady state probabilities $\{\pi_i\}_{i=0}^{\infty}, \{\pi^p_i\}_{i=n}^{\infty}, \{\pi^g_i\}_{i=n}^{\infty}$ can be computed by an iterative process, described in the appendix.

Theorem 2.9 provides an efficient way for analyzing the MDS-Reservation(1) when $k = 2$. We note that these results are not asymptotic and can be applied to systems of any size. Deferring the detailed proof to the appendix, we provide a sketch here. We carefully choose cuts on the Markov chain of the system. We then solve the system of equations, comprising of the flow balance equations derived from each of those chosen cuts, and obtain the iterative process to find the steady-state distribution $\{\pi\}$. By Little's law, the average 'job' latency can be computed with the steady-state distribution. Then, we bound the difference between the average 'job' latency and the average 'request' latency, and hence bound the average request latency.

2.7.4 Waiting Probability

The waiting probability is the probability that, in the steady state, one or more jobs of an arriving batch will have to wait in the buffer and will not be served immediately. As shown previously, under the MDS-Reservation(t) and the MDS-Violation(t) scheduling policies, the system configuration at any time is determined by the state of the Markov chain, which also determines whether a newly arriving batch needs to wait or not. Thus, the waiting probability can be computed directly from the steady-state distribution of the number of jobs in the system. Fig. 2.11(d) plots the waiting probability for the different queues considered in the chapter. Observe how tightly the MDS-Violation(1) and the MDS-Reservation(2) queues bound the waiting probability of the MDS queue.

2.7.5 Degraded Reads

The system considered so far assumed that each incoming request asks for one entire file (from any k servers). In certain applications, incoming requests may sometimes require only a part of the file, say, the part that is stored in one of the servers. In the event that a server is busy or has failed, one may serve requests for that part from the data stored in the remaining servers. This operation is termed a 'degraded read'. Under an MDS code, a degraded read may be performed by obtaining the entire file from the data stored in any k of the $(n-1)$ remaining servers, and then extracting the desired part. Such an operation is generally called 'data-reconstruction'.

Dimakis et al. recently proposed a new model, called the *regenerating codes model*, as a basis to design alternative codes supporting faster degraded reads. Several explicit codes under this model have been constructed subsequently, e.g., [94, 89, 85, 118]. In particular, the *product-matrix (PM) codes* proposed in [89, 92] are practical codes that possess several appealing properties. One feature of the product-matrix codes is that they are associated with an additional parameter $d\ (\leq n-1)$, and can recover the desired data by reading and downloading a fraction $\frac{1}{d-k+1}^{\text{th}}$ of the requisite data from *any* d of the remaining $(n-1)$ servers. This method of performing degraded reads entails a smaller total download but requires connectivity to more servers, and hence the gains achieved by this method in a dynamic setting are unclear. This operation is generally called a 'node-repair' operation in the literature.

We employ the framework of the MDS queue to compare these two methods of performing degraded reads: the data-reconstruction method results in an MDS(n-1,k) queue, while the node-repair operation leads to an MDS(n-1,d) queue with a different service rate. We assume that the average service time is proportional to the number of bits required by a job. Fig. 2.12 depicts

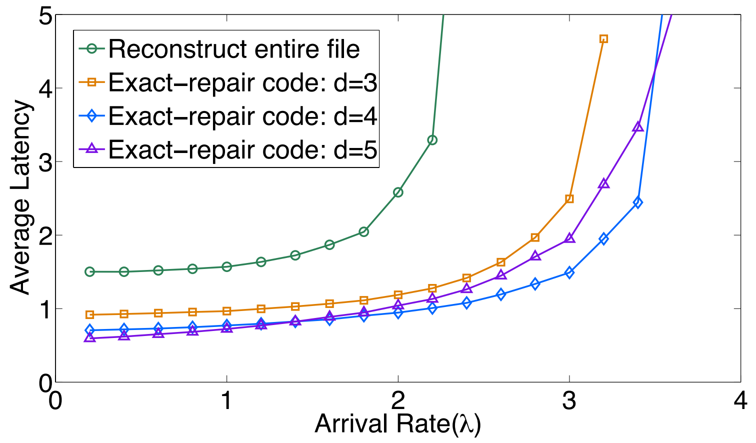


Figure 2.12: **Average latency during degraded reads.** The parameters associated to this system are $n = 6$, $k = 2$. The service time at any server is exponentially distributed with a mean proportional to the amount of data to be read. The performance of the product-matrix (PM) codes are compared for various values of the associated parameter d.

average latency incurred under the two methods when $n = 6$, $k = 2$ and $d = 3$. We see that the product-matrix/regenerating codes perform consistently better in terms of the latency performance. The key insight is that the property of being able to read from *any* d servers provides a great deal of flexibility to the degraded-read operations under a product-matrix code, enabling it to match up (and beat) the performance of the data-reconstruction operation.

2.8 Decentralized MDS Queue

In the previous sections, we considered an MDS queue with a centralized buffer that accepts all arriving requests. However, in some applications, such a centralized scheme is infeasible or inefficient due to practical reasons. For instance, having such a centralized buffer may limit the performance of large-scale storage systems.

In this section, we study a decentralized version of the MDS queue. A *decentralized* MDS queue has n servers with their own buffers, and arriving batches of k jobs are scheduled to k servers, chosen uniformly at random.

Definition 2.2 (Decentralized MDS queue)**.** A decentralized MDS queue is associated to four parameters (n, k) and $[\lambda, \mu]$.

- There are n identical servers with their own buffers
- Requests arrive as a Poisson process with rate λ
- Each request comprises a *batch* of k *jobs*

- Upon arrival of a batch, the k jobs of the batch are scheduled to k (distinct) servers, chosen uniformly at random
- The service time for a job at any server is exponentially distributed with rate μ, independent of all else
- At each server, the jobs are processed in order.

Algorithm 7 formalizes the scheduling policy of the decentralized MDS queue.

Algorithm 7 Decentralized MDS scheduling policy

```
On arrival of a batch
  Randomly choose $k$ servers among the $n$ servers
  Append each job of the arrival to the buffer of each of the $k$ chosen servers
On departure from a server (say, server $s$)
  If $\exists$ at least one job in its own buffer
    Assign the job that arrived earliest
```

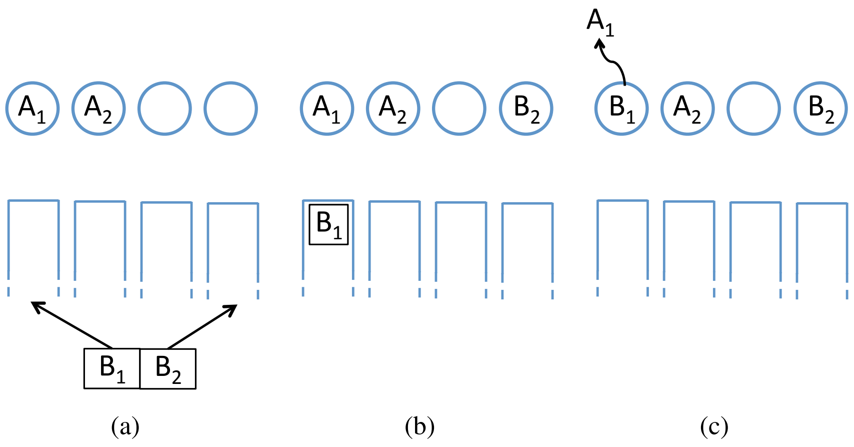


Figure 2.13: **Functioning of the decentralized MDS queue.**

The following example illustrates the functioning of the decentralized MDS scheduling policy and the resultant decentralized MDS queue.

Example 2.7. *Consider a decentralized MDS*$(n = 4,\ k = 2)$ *queue, as depicted in Fig. 2.13. Each request comes as a batch of* $k = 2$ *jobs, and the* 2 *jobs of the batch are scheduled to* 2 *servers, chosen uniformly at random. Suppose that the system is in the state as shown in Fig. 2.13(a), wherein the jobs* A_1, A_2 *are being served by the first two servers, and there are no other jobs in the system. When the new batch* B *arrives, it is randomly dispatched to* 2 *servers among the* 4 *servers. Suppose that the two jobs of the new batch are scheduled to server* 1 *and server* 4*. As shown in Fig. 2.13(b), the job* B_1 *starts waiting in the buffer of server* 1 *as server* 1 *is now busy. On the* *other hand, server 4 immediately starts serving job B_2 since there were no jobs in server 4. Now suppose that server 1 completes service as shown in Fig. 2.13(c). As job B_1 has been waiting in the queue of server 1, server 1 begins serving job B_1.*

We will follow a slightly different proof techniques for deriving results for the decentralized settings, as compared to the techniques employed earlier for the centralized settings. Recall that for the analysis of the centralized MDS queues, we first constructed scheduling policies that form upper and lower bounds to the centralized MDS queues, and then exactly analyzed those bounds to obtain bounds on several metrics including the maximum throughput and the average latency. On the other hand, for the decentralized setting, we find the exact maximum throughput of the decentralized MDS queues. We establish lower bounds on the expected average latency of the decentralized MDS queues also by analyzing these queues directly. The upper bounds on the expected average latency follow a proof technique similar to that in the decentralized case, with a construction of an upper bounding system and an analysis of this system's latency performance.

2.8.1 Throughput

We now analyze decentralized MDS queues focusing on its stability and latency performance. We first define the effective arrival rate of each queue as $\lambda_{\text{eff}} \overset{\text{def}}{=} \frac{k\lambda}{n}$. Since the scheduler chooses a set of k servers at random for any batch, each server is chosen with probability $\frac{k}{n}$. Thus, the arrival process of batches of k jobs is split into n Poisson processes, and the rate of each Poisson process is $\lambda_{\text{eff}} = \frac{k\lambda}{n}$. Therefore, the maximum throughput of a decentralized MDS (n, k) queue can be found by equating the effective arrival rate λ_{eff} and the service rate μ.

Proposition 2.10. *The maximum throughput of a decentralized MDS (n, k) queue is $\frac{n\mu}{k}$.*

Not that the maximum throughput of a decentralized MDS queue is equal to that of a centralized MDS queue.

2.8.2 Lower Bound on the Average Request Latency

We denote by D the batch latency of a randomly chosen batch in steady state. Also, for any $n \geq 1$, we denote the n^{th} harmonic number as $H_n = \sum_{i=1}^{n} \frac{1}{i}$ and the n^{th} generalized harmonic number as $H'_n = \sum_{i=1}^{n} \frac{1}{i^2}$. We now present a lower bound on the average batch latency of a decentralized MDS queue.

Theorem 2.11. *The average batch latency of a decentralized MDS queue $\mathbb{E}[D]$ is lower bounded as*

$$\mathbb{E}[D] \geq \left(\frac{1}{\mu - \lambda_{\mathit{eff}}} - \frac{1}{\mu}\right) + \frac{H_k}{\mu}. \tag{2.7}$$

Note that the batch latency is determined by the maximum of k job latencies. By analyzing the latency of the job whose service time is the maximum among the k jobs, the above lower bound can be easily obtained. We defer the detailed proof of Theorem 2.11 to the appendix.

2.8.3 Upper Bound on the Average Batch Latency

In order to establish an upper bound on the average batch latency of a decentralized MDS queue, we construct a virtual queuing system whose average batch latency is greater than that of the decentralized MDS queue. Consider a virtual queueing system where all the batches arrive in a waiting room before they enter the actual servers. Due to the latency incurred in the waiting room, the latency performance of this new queueing system clearly serves as an upper bound of that of the decentralized MDS queue. We carefully design this system so that the resultant system is amenable to analysis, while not differing much from the actual system.

An exact analysis of decentralized MDS queues is complicated due to dependency across n servers, and this dependency is due to batch arrivals of k jobs. We carefully remove this dependency by generating n independent Poisson processes from the input arrival process of batches of k jobs. We do so by designing a waiting room, which we call the *independent Poisson processes generator (IPPG)*.

Definition 2.3 (Independent Poission Processes Generator (IPPG))**.** An IPPG is associated to four parameters (n, k) and (λ, λ'), where λ' is the virtual service rate of the IPPG.

- Batches of k jobs arrive in an IPPG as a Poisson process with rate λ.
- When a batch of k jobs arrives, the batch enters the central buffer in the IPPG.
- The IPPG has n independent exponential timers each with rate λ'.
- When the i^{th} exponential timer ticks,
  1. if the central buffer is not empty, and the timer has not previously picked any job from the first waiting batch of the central buffer, one job of the first waiting batch is forwarded to the i^{th} server.
  2. if not, a dummy job is created and forwarded to the i^{th} server.
- When all the k jobs of a batch are forwarded to the servers, the batch leaves the IPPG.

It is clear that in this virtual system, the k jobs of a batch will be assigned to k distinct servers, uniformly chosen at random. Therefore, the output processes of the IPPG are independent Poisson processes of rate λ' as the exponential timer independently sends one job, either real or dummy, to the corresponding server whenever its exponential timer ticks. Thus, the latency performance of an IPPG followed by the n servers establishes a strict lower bound of that of the decentralized MDS queues.

The following example illustrates the functioning of the IPPG followed by a decentralized MDS queue.

Example 2.8. *Consider the IPPG followed by the decentralized MDS*$(n = 4,\ k = 2)$ *queue, as depicted in Fig. 2.14. The dotted square with rounded corners represents the IPPG, which consists of the central buffer and the* n *exponential timers (virtual servers). All the arrivals enter the central buffer of the IPPG. Each virtual server is associated with its corresponding real server. Suppose*

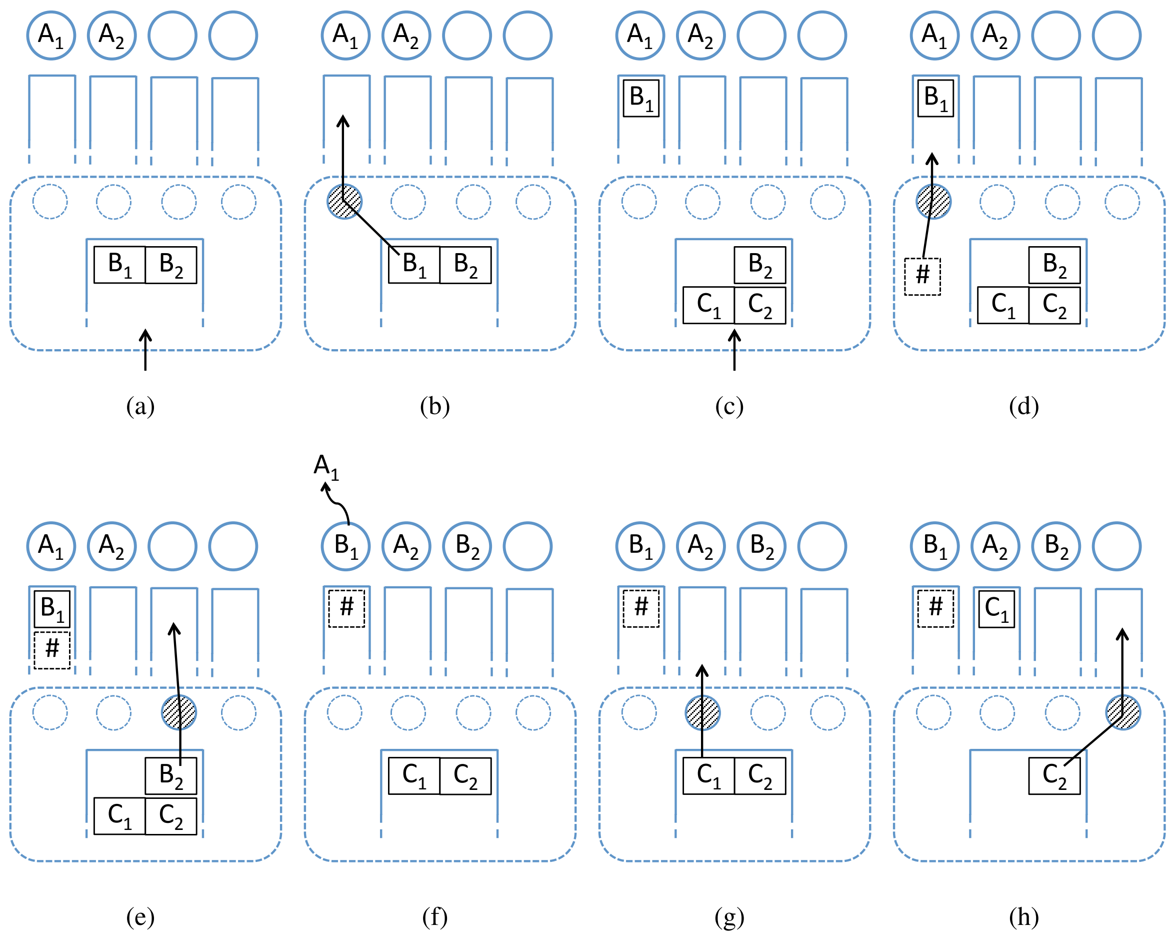


Figure 2.14: **Functioning of the IPPG followed by the decentralized MDS queue.**

that the system is in the state as shown in Fig. 2.14(a), wherein the jobs A_1, A_2 are being served by the first two servers, and there are no other jobs in the system. When the new batch B arrives, the two jobs of the new batch enter the central buffer of the IPPG. In Fig. 2.14(b), suppose that the virtual server 1 expires. Since the virtual server 1 has not served the waiting batch in the IPPG, it forwards one job of the waiting batch to server 1. In Fig. 2.14(c), the new batch C arrives. In Fig. 2.14(d), suppose that the virtual server 1 ticks. As it has already served a job from batch B, it creates a dummy job (tagged with a number sign $\#$) and forwards it to server 1. When the virtual server 3 ticks, the second job of batch B is forwarded to server 3 as shown in Fig. 2.14(e). In Fig. 2.14(f), server 1 finishes its task and starts serving the first waiting job B_1. In Fig. 2.14(g), the virtual server 2 ticks, and it forwards the first job of batch C to server 2. Similarly, the virtual server 4 ticks, and it forward the second job of batch C to server 4.

As the latency of the constructed virtual queueing system upper bounds that of the decentralized MDS queue, we now analyze the batch latency of the constructed virtual queueing system. We denote by D_i job i's latency of a certain batch for $1 \leq i \leq k$. The job latency can be decomposed into latency in the IPPG, $D_{I,i}$, and latency in a server (including its queue), $D_{S,i}$: $D_i = D_{I,i} + D_{S,i}$. Then,

$$D = \max_{1\leq i\leq k} D_i \leq \max_{1\leq i\leq k} D_{I,i} + \max_{1\leq i\leq k} D_{S,i}, \tag{2.8}$$

and

$$\mathbb{E}[D] \leq \mathbb{E}\left[\max_{1\leq i\leq k} D_{I,i}\right] + \mathbb{E}\left[\max_{1\leq i\leq k} D_{S,i}\right], \tag{2.9}$$

where the expectations are over batches.

The following lemma bounds the second term of (2.9), the expected value of the maximum waiting time in the servers.

Lemma 2.12. $\mathbb{E}\left[\max_{1\leq i\leq k} D_{S,i}\right] = \frac{H_k}{\mu-\lambda'}$

This lemma is proved by showing that each of the k jobs of a batch sees an M/M/1 queue that is independent of the other queues, when it exits the IPPG and enters the queue of a server. The detailed proof is given in the appendix.

We now bound the first term in (2.9). First of all, note that this term is the time difference between the arrival time of the batch and the latest job departure time. That is, this term can be viewed as the waiting time of a batch in the IPPG. This term can be analyzed by studying a (centralized) MDS queue with a scheduling policy called the 'redundant-requests scheduler' [103, 58]. In an MDS queue with a redundant-request scheduler, a request starts getting served simultaneously by r $(> k)$ servers. When the first k distinct servers complete their jobs, the request is deemed served and leaves the system. The $r - k$ remaining jobs are immediately cancelled upon the departure. Even though the IPPG is not meant to model such redundant-requests schedulers, the dynamic behavior of the IPPG is equivalent to the MDS queue with such redundant-request schedulers that schedules k jobs of *the first waiting batch* to all the n servers and blocks all the other waiting batches until the first batch is served by k distinct servers. The average batch latency of this system coincides with a bound proposed by the authors of [58]. In order to find the average batch latency of the IPPG, we use their results.

Lemma 2.13 (Joshi et al., 2012)**.** *For every* $\lambda < \frac{\lambda'}{H_n - H_{n-k}}$,

$$\mathbb{E}\left[\max_{1\leq i\leq k} D_{I,i}\right] = \frac{H_n - H_{n-k}}{\lambda'} + \frac{\lambda\left\{(H'_n - H'_{n-k}) + (H_n - H_{n-k})^2\right\}}{2\lambda'^2\left\{1 - \frac{\lambda}{\lambda'}(H_n - H_{n-k})\right\}}. \tag{2.10}$$

When the virtual rate of the IPPG increases, the waiting time in the IPPG decreases, but the waiting time in the actual servers increases due to an increased number of dummy jobs. Thus, for a fixed arrival rate λ, one can optimize over λ' in order to minimize the sum of the waiting time in the IPPG and the latency in the actual servers. By collecting all the above lemmas and observations, we have the following main theorem.

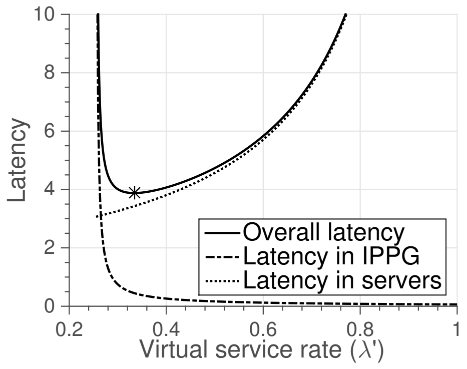


Figure 2.15: **The objective function in the statement of Theorem 2.14 for $\mathbf{n = 100}$, $\mathbf{k = 5}$, and $\boldsymbol{\lambda = 5}$.** The average batch latencies in the IPPG and the servers are also plotted. The best upper bound is obtained by computing the value for λ' that minimizes the overall latency.

Theorem 2.14. *For $\lambda < \frac{\mu}{H_n - H_{n-k}}$, the average batch latency of a decentralized MDS queue $\mathbb{E}[D]$ is upper bounded as*

$$\mathbb{E}[D] \le \min_{\lambda(H_n - H_{n-k}) \le \lambda' \le \mu} \left[\frac{H_n - H_{n-k}}{\lambda'} + \frac{\lambda \left\{ (H'_n - H'_{n-k}) + (H_n - H_{n-k})^2 \right\}}{2\lambda'^2 \left\{ 1 - \frac{\lambda}{\lambda'}(H_n - H_{n-k}) \right\}} + \frac{H_k}{\mu - \lambda'} \right]. \tag{2.11}$$

Note that the objective function can be minimized by numerically solving a polynomial equation of λ'. In Figure 2.15, we plot the objective function in the above theorem when $n = 100$, $k = 5$, and $\lambda = 5$. Also we plot the latencies in the IPPG and the servers. In Figure 2.16, we compare our upper and lower bounds with the simulation results. We observe that our upper bound closely captures the latency performance of decentralized MDS queues when n is large compared to k.

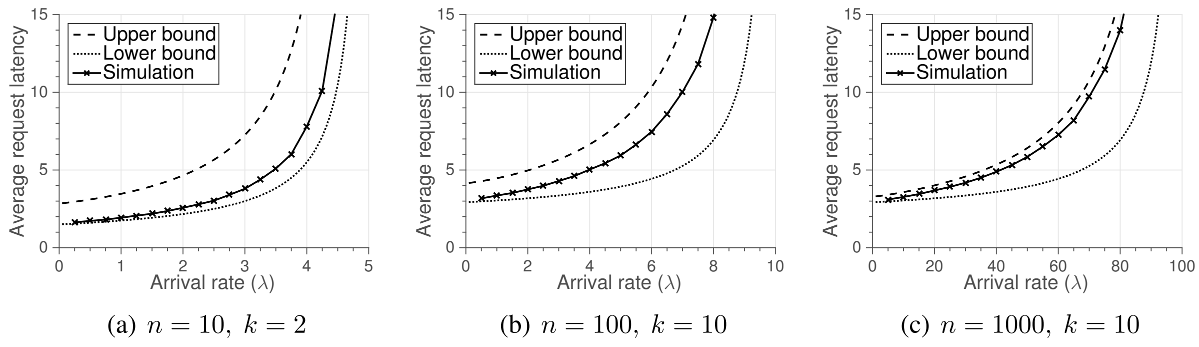


Figure 2.16: **The average batch latency of the decentralized MDS queues with its lower and upper bounds.**

Remark 2.2. Consider a case where $n \gg k$. Then $\lambda(H_n - H_{n-k}) = \frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{n-k+1} \simeq \frac{k}{n}$, and hence one can choose $\lambda' \simeq \frac{k\lambda}{n} = \lambda_{\text{eff}}$. Also, observe that the first two terms of the bound are negligible compared to the last term. Therefore, the upper bound can be simplified as

$$\mathbb{E}[D] \lesssim \frac{H_k}{\mu - \lambda_{\text{eff}}}. \tag{2.12}$$

This approximation can be directly found with the *independence assumption* that the queue lengths of n servers are *independent* of each other. Since the sojourn time of an M/M/1 queue is exponentially distributed with rate $\lambda_{\text{eff}} - \mu$ [49], the maximum of k job latencies can be readily obtained, and that gives us the above approximation. This approximation is used in some related works including [122], and the approximation has been observed to be useful to closely capture the original system's performance. However, it has not been known whether such approximation indeed serves as a bound, either lower or upper, of the original system. The fact that our upper bound asymptotically converges to this simple yet powerful approximation is an interesting theoretical contribution, which may be of independent interest.

2.9 Discussion and Open Problems

In this work, we proposed the MDS queue, which captures the system dynamics of distributed data storage systems that are based on MDS codes. We analyzed both centralized and decentralized MDS queues, and provided extensive simulation results to validate our theoretical analysis.

We aim to use the results of this chapter as a starting point for analysis of more complex systems that mimic the real world more closely. In particular, we intend to build upon the "MDS-Queue" framework presented here to analyze queues that relax one or more of the assumptions made in the chapter, e.g., having general service times, heterogeneous requests or servers, and non-MDS codes.

In this chapter, while we characterized the metrics of average latency and throughput analytically, we also used simulations to examine the performance of the queues in terms of several additional metrics. We observed in these simulations that the MDS-Reservation(t) and the MDS-Violation(t) scheduling policies result in a performance very close to that of the exact MDS queue for values of t as small as 3. Thus, even respect to these metrics, an analysis of the (simpler) scheduling policies of MDS-Reservation(t) and MDS-Violation(t) may provide rather-accurate estimations of the performance of MDS codes.

Chapter 3

Scheduling of Redundant Requests

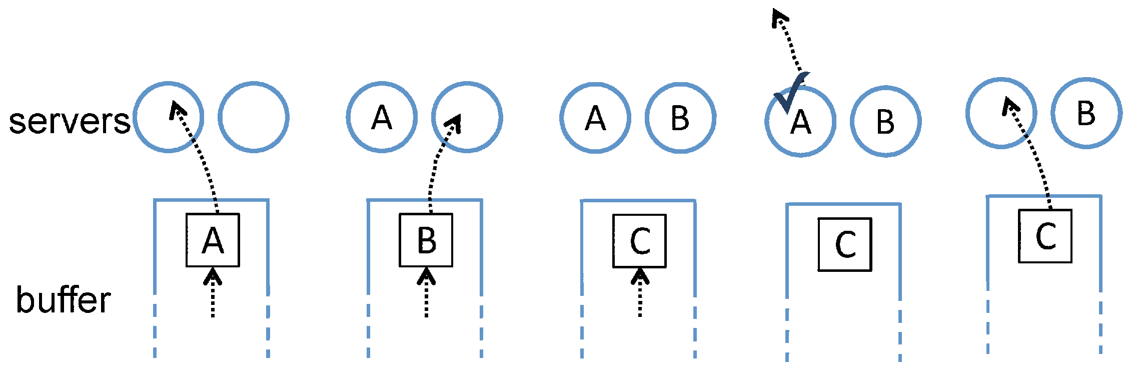
3.1 Introduction

Many systems possess the flexibility to serve requests in more than one way. For instance, in a cluster with n processors, a computation may be performed at any one of the n processors; in a distributed storage system where data is stored using an (n, k) Reed-Solomon code, a read-request may be served by reading data from any k of the n servers; in a network with n available paths from the source to the destination, communication may be performed by transmitting across any one of the n paths. In such settings, the latency of serving the requests can potentially be reduced by sending *redundant requests*. Under a policy of sending redundant requests, each request is attempted to be served in more than one way. The request is deemed served when it is served in any one of these ways. The remaining copies of this request may then be removed from the system.

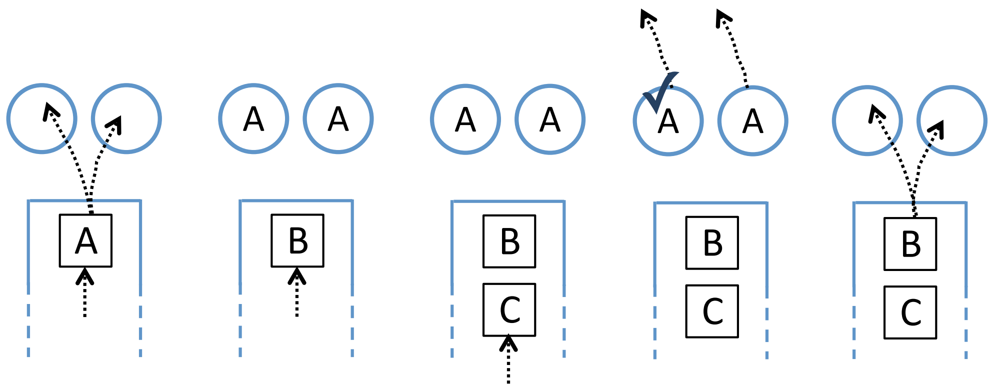
Fig. 3.1 illustrates the concept with a toy model comprising two servers. Requests arrive one-by-one and are stored in a buffer if the servers are busy. Fig. 3.1(a) depicts the traditional policy that does not use redundant requests. Here, each request goes to one server and is deemed served when that server completes service. As shown in the fourth panel of Fig. 3.1(a), when a server completes service, the first request waiting in the buffer is sent to this server. Fig. 3.1(b) illustrates a policy that sends each request redundantly to both servers. Here, each request goes to *both* servers and is deemed served when *any one* of the two servers completes service. As shown in the fourth panel of Fig. 3.1(b), when a server completes service, the copy of this request at the other servers is "cancelled", and a new request moves into both these servers.

It is unclear whether or not such a policy of having redundant requests will actually reduce the latency. On one hand, for any individual request, one would expect the latency to reduce since the time taken to process the request is the *minimum* of the processing times of its multiple copies. On the other hand, introducing redundancy in the requests consumes *additional resources* and increases the number of jobs up for service in the system, thereby adversely affecting the latency.

Many recent works such as [110, 5, 87, 47, 2, 54, 122, 29, 69, 113, 39, 121] perform empirical studies on the latency performance of sending redundant requests, and report reductions in latency in several scenarios (and increases in some others). The goal of this chapter is to obtain an analyt-



(a) Without redundant requests



(b) With redundant requests

Figure 3.1: **An illustration of a system with and one without redundant requests.** The five images (from left to right) respectively depict five consecutive events in the system.

ical understanding of redundant requests. In particular, we consider a model based on the 'MDS queue' model [105], which captures some of the key features of such systems, and can serve as a building block for more complex systems. The model is associated to a set of n servers such that every request can be served by any k of the servers. The requests are served in a first-come-first-served manner. More details of the model, in the context of redundant requests, are described subsequently in Section 3.3 for a centralized setting and in Section 3.6 for a distributed setting.

The contributions of this chapter are summarized as follows. We derive the optimal redundant-requesting policies under a variety of settings: these results are summarized in Table 3.1. Our novel "stochastic coupling" proof techniques allow for arbitrary arrival sequences and are not restricted to any specific arrival patterns. Moreover, our techniques allow for results that are not restricted to steady-state settings. The results are applicable to both centralized (Section 3.4) and distributed (Section 3.6) settings. In addition, we provide extensive simulation results to obtain further insights into the problem. We also apply our results to a standard model of hard-disk service times and use the specifications of a popular line of hard-disks to obtain analytical and numerical results

(Section 3.5).

While we successfully characterize the optimal redundant requests scheduling policies under a variety of settings, the results also have the following limitations. Our techniques allow us to qualitatively compare scheduling policies. However, these techniques do not provide us quantitative analysis, e.g., the value of the latency as a function of system parameters. Secondly, we make the simplifying assumptions of homogeneity of the servers as well as requests. Thirdly, like most literature in this area, we can prove analytical results for the expected latency (although interestingly, our simulations reveal that the tail latencies also follow our predictions that were based on the average latency).

3.2 Related Works

Policies that try to reduce latency by sending redundant requests have been studied previously, largely empirically, in [110, 5, 87, 47, 2, 54, 122, 29, 69, 113, 39, 121]. These works evaluate system performance under redundant requests for several applications, and report a reduction in the latency in many cases. For instance, Ananthanarayanan et al. [2] consider the setting where requests take the form of computations to be performed at processors. In their setting, requests have diffferent workloads, and the authors propose adding redundancy in the requests with lighter workloads. They observe that on the PlanetLab network, the average completion time of the requests with lighter workloads decreases by 47%, at the cost of just 3% extra resources. Huang et al. [54] consider a distributed stoage system where the data is stored using an $(n = 16,\ k = 12)$

Table 3.1: **Summary of analysis of *when* redundant requests reduce latency, and the optimal policies of redundant-requesting for various settings under the models considered in the chapter.** The 'heavy-everywhere' and 'light-everywhere' classes of distributions are defined subsequently in Section 3.4. An example of a heavy-everywhere distribution is a mixture of exponential distributions; two examples of light-everywhere distributions are an exponential distribution shifted by a constant, and the uniform distribution. By 'high load' we mean a 100% utilization of the servers.

n	k	Arrival process	Service distribution	Buffers	Removal cost	Load	Optimal policy (Theorem #)
any	1	any	memoryless	centralized	0	any	send to all (3.1)
any	any	any	memoryless	centralized	0	any	send to all (3.2)
any	1	any	heavy-everywhere	centralized	0	high	send to all (3.3)
any	1	any	light-everywhere	centralized	any	high	no redundancy (3.4)
any	1	any	memoryless	centralized	>0	high	no redundancy (3.5)
any	any	any	memoryless	distributed	0	any	send to all (3.7a)
any	1	any	heavy-everywhere	distributed	0	high	send to all (3.7b)
any	1	any	light-everywhere	distributed	any	high	no redundancy (3.7c)
any	1	any	memoryless	distributed	>0	high	no redundancy (3.7d)

Reed-Solomon code. For $k' \in \{12, 13, 14, 15\}$, they perform the task of decoding the original data by connecting to k' of the nodes and decoding from the k pieces of encoded data that arrive first. They empirically observe that the latency reduces upon increase in k'. In a related setup, codes and algorithms tailored specifically for employing redundant requests in distributed storage are designed in [90] for latency-sensitive settings, allowing for data stored in a busy or a failed node to be obtained by downloading little chunks of data from other nodes. In particular, these codes provide the ability to connect to more nodes than required and use the data received from the first subset to respond, treating the other slower nodes as "erasures". Vulimiri et al. [122] propose sending DNS queries to multiple servers. They observe that on PlanetLab servers, the latency of the DNS queries reduces with an increase in the number of DNS servers queried. Dean and Barroso [29] observe a reduction in latency in Google's system when requests are sent to two servers instead of one. Liang and Kozat [69] perform experiments on the Amazon EC2 cloud. They observe that when the rate of arrival of the requests is low, the latency is lower when the requests are sent to a higher number of servers. However, when the rate of arrival is high, they observe that a high redundancy in the requests increases the latency. Rashmi et al. [93] design erasure codes for distributed storage that given the option to connect to more servers, require a lower download for recovery of partial data. In empirical evaluations on the Facebook warehouse cluster in production, it is shown that under their setting, the combination of higher connectivity and lower download outperforms the lower connectivity and higher download algorithm. Venkataraman et al. [121] consider machine learning applications which operate on subsampled data. Leveraging the fact that any randomly selected choice of input data suffices, they query for additional data and then pick the first to arrive. In empirical evaluations, they observe an 81% reduction in the average job duration when querying for 5% redundant data. A tangential application of redundant requests lies in crowdsourcing tasks [9]. Workers on crowdsourcing platforms are typically noisy, and hence each task is typically given to multiple workers. The work is often required to be completed quickly, and in real-time. A policy of recruiting a few additional workers and using the work of the first requisite number who complete the task may allow for a faster completion of tasks (while, however, trading off with the additional cost of paying the extra workers).

We now discuss related theoretical literature in this area. In [58], Joshi et al. consider the arrival process to be Poisson, and the service to be i.i.d. memoryless, and provide bounds on the average latency faced by a batch in the steady state when the requests are sent (redundantly) to *all* the servers. However, no comparisons are made with other schemes involving redundant requests, including the scheme of having no redundancy in the requests. In fact, our work can be considered as complementary to that of [58], in that we complete this picture by establishing that under the models considered therein, sending (redundant) requests to all servers is indeed the optimal choice. In [69], Liang and Kozat provide an approximate analysis of a system similar to that described in this paper under the assumption that arrivals follow a Poisson process. Using insights from their approximations, they experiment with certain scheduling policies on the Amazon EC2 cloud. However, no analysis or metrics for accuracy of these approximations are provided, nor is there any treatment of whether these approximations lead to any useful upper or lower bounds.

The following two works that appeared subsequent to our conference publication offer further insights into the properties of redundant requests. In [43], the authors study scheduling redundant requests of multiple classes of jobs. Each class of jobs is associated with a certain set of servers, and all jobs of a class are scheduled to the corresponding set of servers. Under this setup, the authors find the steady state distribution of the system, and quantify the gain of scheduling redundant requests. The authors of [115] study the optimal scheduling of redundant requests for data retrieval in storage clouds. The proposed model can be thought as a generalization of our model. They consider various setups, and find the optimal or near-optimal scheduling policies for each of them.

The work perhaps closest to ours is [62], pointed out to us by the second author of [62] subsequent to our conference publication [103]. The paper considers a problem of assigning multiple copies of the same task to different computers to improve latency performance in *grid computing systems*. The authors defined two classes of service time distributions that are independently defined in our conference version: 'new worse than used (NWU)' that corresponds to heavy-everywhere, and 'new better than used (NBU)' that corresponds to light-everywhere. Our contributions over [62] are that we considers the more general setting where each request has k (≥ 1) jobs whereas [62] considers the special case of $k = 1$; certain results derived in [62] are applicable to only $n = 2$ servers. Furthermore, we not only provide optimal scheduling policies but also provide a comparison between the performances of all possible degrees of redundant requests, analytically when $k = 1$ and numerically otherwise.

A related notion is that of 'availability' in erasure coded distributed storage systems, studied in [97, 96], where in the presence of multiple requests, the likelihood of being able to serve any individual 'data-symbol' is to be maximized. These works design erasure codes to maximize the data availability (in the absence of redundant requests).

3.3 System Model: Centralized Buffer

We will first describe the system model followed by an illustrative example. The model is associated to three parameters: two parameters n and k that are associated to the system, and the third parameter r that is associated to the redundant-requesting policy. The system comprises a set of n servers. A request can be served by any *arbitrary* k *distinct* servers out of this collection of n servers. Several applications fall under the special case of $k = 1$: a compute-cluster where computational tasks can be performed at any one of multiple processors, or a data-transmission scenario where requests comprise packets that can be transmitted across any one of multiple routes, or a distributed storage system with data replicated in multiple servers. Examples of settings with $k > 1$ include: a distributed storage system employing an $(n,\ k)$ Reed-Solomon code wherein the request for any data can be served by downloading the data from any k of the n servers; a compute-cluster where each job is executed at multiple processors in order to guard from possible errors during computation; or, a machine-learning compute-cluster where jobs can be computed with any choice of k of the n data blocks.

The policy of redundant requesting is associated to a parameter r $(k \leq r \leq n)$ which we call the *request-degree*. Each request is sent to at most r of the servers, and upon completion of any k of them, it is deemed complete. To capture this feature, we consider each request as a *batch* of r *jobs*. Each of the r jobs of a batch can be served by any arbitrary server, but with the constraint that

Algorithm 8 First-come-first-served scheduling policy with redundant requests

```
on arrival of a request (“batch”)
  divide the batch into $r$ jobs
  assign as many jobs (of the new batch) as possible to idle servers
  append the remaining jobs (if any) as a new batch at the end of the buffer

on completion of processing by a server (say, server $s_0$)
  let set $\mathcal{S} = \{s_0\}$
  if the job that departed from $s_0$ was the $k^{\text{th}}$ job served from its batch then
      for every server that is also serving jobs from this batch do
          remove the job and add this server to set $\mathcal{S}$
      end for
      remove all jobs of this batch from the system
  end if
  for each $s \in \mathcal{S}$ do
      if there exists at least one batch in the buffer such that no job of this batch has been served by $s$ then
          among all such batches, find the batch that had arrived earliest
          assign a job from this batch to $s$
      end if
  end for
```

the r jobs of a batch must be served by r distinct servers. The batch is deemed served when any k of its r jobs are serviced. At this point in time, the remaining $(r-k)$ jobs of this batch are removed from the system. Such a premature removal of a job from a server may lead to certain overheads: the server may need to remain idle for some (random) duration before it is ready to serve another job. We will term this idle time as the *removal cost*.

We assume that the time that a server takes to serve a job is independent of the arrival and service times of other jobs. We further assume that the jobs are processed in a first-come-first-served fashion, i.e., among all the waiting jobs that an idle server can serve, it serves the one which had arrived the earliest. Finally, to be able to perform valid comparisons, we assume that the system is stable in the absence of any redundancy in the requests (i.e., when $r = k$). The arrival process may be *arbitrary*, and the only assumption we make is that the arrival process is independent of the present and past states of the system.

We consider a centralized queueing system in this section, where requests enter into a (common) buffer of infinite capacity. The choice of the server that serves a job may be made at any point in time. (We will consider distributed systems subsequently in Section 3.6, wherein the server must be chosen upon arrival of the request into the system).

The goal is to evaluate whether redundant requests can help in reducing the average latency or not. If yes, then the goal is to establish the redundant-request policy that would minimize the average latency incurred by requests entering the system.

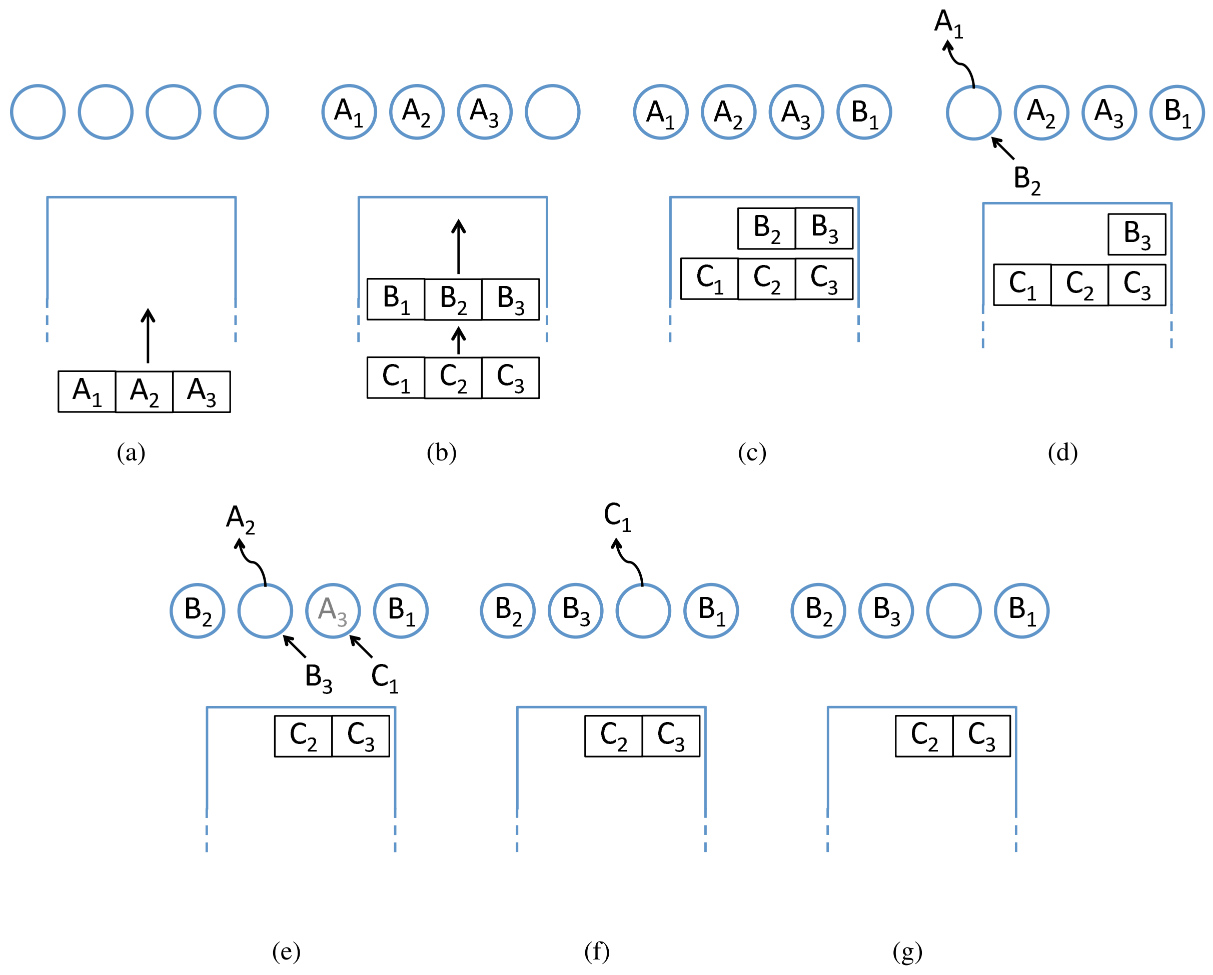


Figure 3.2: **An illustration of the setting for parameters $n = 4$, $k = 2$ and request-degree $r = 3$, as described in Example 3.1.**

The scheduling algorithm for any fixed value of r is formalized in Algorithm 8. Note that the case of $r = k$ corresponds to the case where no redundancy is introduced in the requests, while $r = n$ corresponds to maximum redundancy with each batch being sent to all the servers. The following example illustrates the working of the system.

Example 3.1. *Fig. 3.2 illustrates the system model and the working of Algorithm 8 when $n = 4$, $k = 2$ and $r = 3$. The system has $n = 4$ servers and a common buffer as shown in Fig. 3.2(a). Let us denote the four servers (from left to right) as servers 1, 2, 3 and 4. Each request comes as a batch of $r = 3$ jobs, and hence we denote each batch (e.g., A, B, C, etc.) as a triplet of jobs (e.g., $\{A_1, A_2, A_3\}$, $\{B_1, B_2, B_3\}$, $\{C_1, C_2, C_3\}$, etc.). A batch is considered served if any $k = 2$ of the $r = 3$ jobs of that batch are served.*
Fig. 3.2(a) depicts the arrival of batch A. As shown in Fig. 3.2(b), three of the idle servers begin

serving the three jobs $\{A_1, A_2, A_3\}$. Fig. 3.2(b) depicts the arrival of batch B followed by batch C. Server 4 begins serving job B_1 on arrival of batch B, while the other jobs wait in the buffer. This is depicted in Fig. 3.2(c). Now suppose server 1 completes serving job A_1 (Fig. 3.2(d)). This server now becomes idle to serve any of the jobs remaining in the buffer. We allow jobs to be processed in a first-come-first-served manner, and hence server 1 begins serving job B_2 (assignment of B_3 instead would also have been valid). Next, suppose the second server finishes serving A_2 before any other servers complete their current tasks (Fig. 3.2(e)). This results in the completion of a total of $k = 2$ jobs of batch A, and hence batch A is deemed served and is removed the system. In particular, job A_3 is removed from server 3 (this may cause the server to remain idle for some time, depending on the associated removal cost). Servers 2 and 3 are now free to serve other jobs in the buffer. These are now populated with jobs B_3 and C_1 respectively. Next suppose server 3 completes serving C_1 (Fig. 3.2(f)). In this case, since server 3 has already served a job from batch C, it is not allowed to serve C_2 or C_3 (since the jobs of a batch must be processed by distinct servers). Since there are no other batches waiting in the buffer, server 3 thus remains idle (Fig. 3.2(g)).

3.4 Analytical Results for the Centralized Buffer Setting

In this section, we consider the model presented in Section 3.3 that has a centralized buffer. We find redundant-requesting policies that minimize the average latency under various settings. This minimization is not only over redundant-requesting policies with a fixed value of the request-degree r (as described in Section 3.3) but also over policies that can choose different request-degrees for different batches.

In what follows, we will say that a probability distribution with the cumulative distribution function F *stochastically dominates* another probability distribution with the cumulative distribution function G if

$$F(z) \leq G(z) \quad \forall\, z\ \in \mathbb{R}.$$

with a strict inequality for some interval.

The first two results, Theorems 3.1 and 3.2, consider the service times to follow an exponential (memoryless) distribution.

Theorem 3.1 (memoryless service, no removal cost, $\mathrm{k} = 1$). *Consider a system with n servers such that any one server suffices to serve any request, the service-time is i.i.d. memoryless, and jobs can be removed instantly from the system. For any $r_1 < r_2$, the average latency in a system with request-degree r_1 is larger than the average latency in a system with request-degree r_2. Furthermore, the distribution of the buffer occupancy in the system with request-degree r_1 stochastically dominates that of the system with request-degree r_2. Finally, among all possible redundant requesting policies, the average latency is minimized when each batch is sent to all n servers, i.e., when request-degree $r = n$.*

The proof of this result as well as all other analytical results in this chapter are deferred to the appendix.

Theorem 3.1 provides a total ordering of the performances of redundant requesting policies for all possible values of r, when $k = 1$. Theorem 3.2 below characterizes the optimal policy for any arbitrary value of k.

Theorem 3.2 (**memoryless service, no removal cost, general** k)**.** *Consider a system with n servers such that any k of them can serve a request, the service-time is i.i.d. memoryless, and jobs can be removed instantly from the system. The average latency is minimized when all batches are sent to all the servers, i.e., when $r = n$ for every batch. Furthermore, the distribution of the buffer occupancy in the system with request-degree $r = n$ is stochastically dominated by a system with any other request-degree.*

We conjecture that the same total ordering as in Theorem 3.1 continues to hold for general values of k as well, that is, that the latency reduces with an increase in r when the service times are memoryless. Fig. 3.3(a) depicts simulations that corroborate this conjecture.

We have so far considered the specific case of an exponential service time distribution. We will now move on to some more general classes of service-time distributions. The first class of distributions is what we term *heavy-everywhere*, defined as follows.

Definition 3.1 (**Heavy-everywhere distribution**)**.** A distribution on the non-negative real numbers is termed heavy-everywhere if for every pair of values $a > 0$ and $b \geq 0$ with $P(X > b) > 0$, the distribution satisfies

$$P(X > a + b \,|\, X > b) \;\geq\; P(X > a)\,. \tag{3.1}$$

In words, under a heavy-everywhere distribution, the need to wait for a while makes it more likely that a bad event has occurred, thus increasing the possibility of a greater wait than usual. For example, one can verify that a mixture of independent exponential distributions (namely, a hyperexponential distribution) satisfies (3.1) and hence is heavy-everywhere.

Note that the service times across different servers and across different jobs are independent. However, independence does not hold if the same job is served at a server twice in a row. Thus, cancelling and restarting the same job at the same server always incurs an increase in job latency regardless of the service time distribution.

A second class of distributions is what we call *light-everywhere* distributions, defined as follows.

Definition 3.2 (**Light-everywhere distribution**)**.** A distribution on the non-negative real numbers is termed light-everywhere if for every pair of values $a > 0$ and $b \geq 0$ with $P(X > b) > 0$, the distribution satisfies

$$P(X > a + b \,|\, X > b) \;\leq\; P(X > a)\,. \tag{3.2}$$

In words, under a light-everywhere distribution, waiting for some time brings you closer to completion, resulting in a smaller additional waiting time. For example, one can verify that an exponential distribution that is shifted by a positive constant is light-everywhere, and so is the uniform distribution. Additional properties of heavy-everywhere and light-everywhere distributions are discussed in the appendix.

The following theorems present results for systems with service-times in these classes of distributions.

Theorem 3.3 (**Heavy-everywhere service, no removal cost, k $= 1$, high load**). *Consider a system with n servers such that any one server suffices to serve any request, the service-time is i.i.d. heavy-everywhere, and jobs can be removed instantly from the system. When the system has a 100% server utilization, the average latency is minimized when each batch is sent to all n servers, i.e., when $r = n$ for each batch.*

This is corroborated in Fig. 3.3(b) which depicts simulations with the service time distributed as a mixture of exponentials: $\sim \exp(\text{ rate } = 0.2)$ w.p. 0.1 and $\exp(\text{ rate } = \frac{9}{5})$ w.p. 0.9. Note that Theorem 3.3 addresses only the scenario of high loads and predicts minimization of latency when $r = n$ in this regime; simulations of Fig. 3.3(b) further suggest that the policy of $r = n$ minimizes the average latency for all loads. We have also observed similar phenomena in simulations for $k > 1$.

Theorem 3.4 (**Light-everywhere service, any removal cost, k $= 1$, high load**). *Consider a system with n servers such that any one server suffices to serve any request, and the service-time is i.i.d. light-everywhere. When the system has a 100% server utilization, the average latency is minimized when there is no redundancy in the requests, i.e., when $r = k \; (= 1)$ for all batches.*

This is corroborated in Fig. 3.3(c) which depicts simulations with the service time X distributed as a sum of a constant and a value drawn from an exponential distribution: $P(X > x) = e^{-(x-\frac{1}{2})}$ if $x \geq \frac{1}{2}$ and 1 otherwise. We observe in Fig. 3.3(c) that at high loads, the absence of any redundant requests (i.e., $r = 1$) minimizes the average latency, which is as predicted by the theory. We also observe in the simulations for this setting that redundant requests do help when arrival rates are low, but start hurting beyond a certain threshold on the arrival rate. Similar phenomena are observed in simulations for $k > 1$.

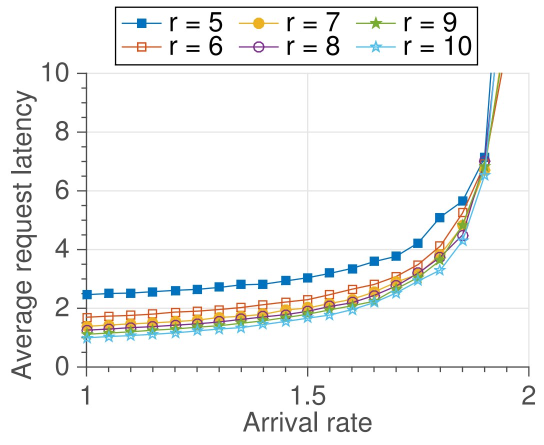
The next theorem revisits memoryless service times, but under non-negligible removal costs.

Theorem 3.5 (**memoryless service, non-zero removal cost, k $= 1$, high load**). *Consider a system with n servers such that any one server suffices to serve any request, and the service-time is i.i.d. memoryless, and removal of a job from a server incurs a non-zero delay. When the system has a 100% server utilization, the average latency is minimized when there is no redundancy in the requests, i.e., when $r = k \; (= 1)$ for all batches.*

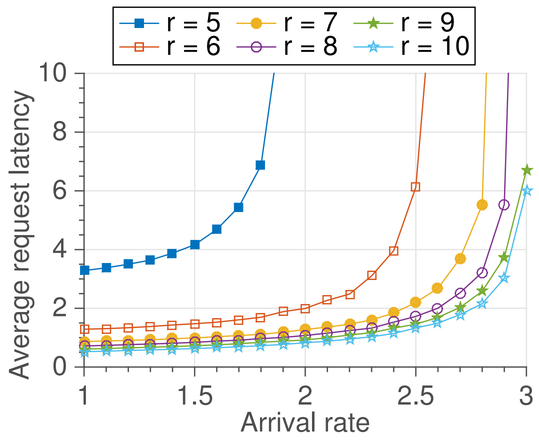
Fig. 3.3(d) presents simulation results for such a setting. The figure shows that under this setting, redundant requests lead to a higher latency at high loads, as predicted by theory.

For each of the settings investigated in this section, we have observed via simulations that interestingly, the tail latencies also follow the theoretical predictions that are provided in this section for the average latency [103].

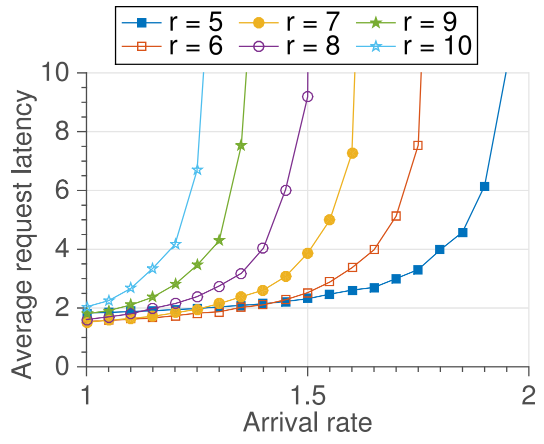
Fig. 3.4 plots the 95th percentile latency for each of the settings simulated in Fig. 3.3. Observe that interestingly, the tail latencies also follow the same predictions as given by our theoretical results in this section that had analytically characterized the average latency.



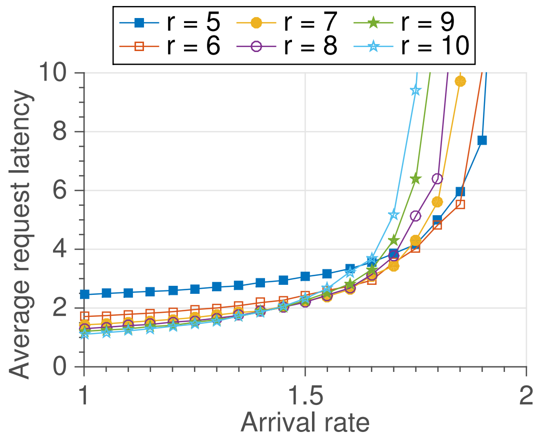
(a)



(b)



(c)

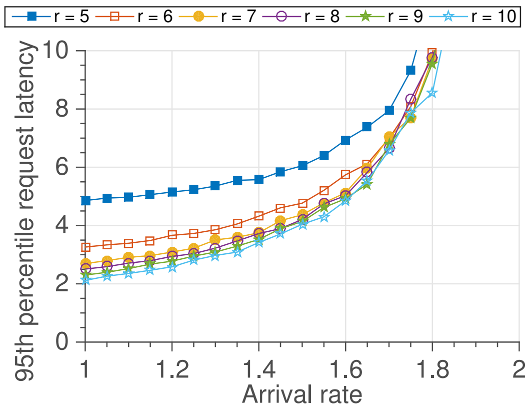


(d)

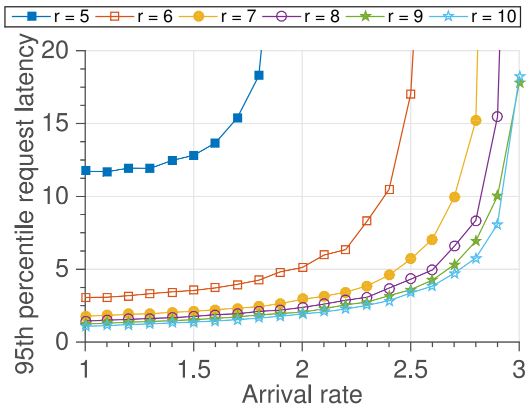
Figure 3.3: **The average latency in an $(n = 10, k = 5)$ system with Poisson arrivals, and various service time distributions.** (a) Exponential with rate 1; (b) Heavy-everywhere: a mixture of exponential distributions; (c) Light-everywhere: a shifted exponential; and (d) Exponential with rate 1 where additionally, removing an unfinished job from a server requires the server to remain idle for a time distributed exponentially with rate 10.

3.5 Analysis Using Disk I/O Service Specifications

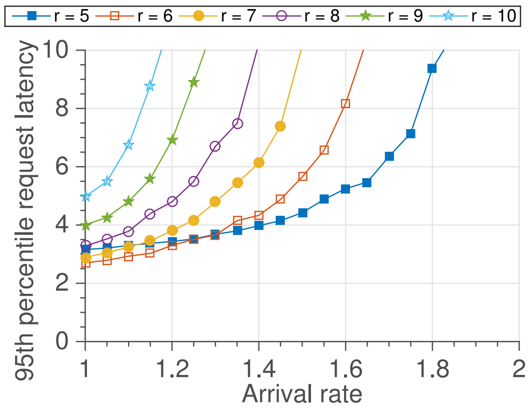
In this section, we investigate the application of the results of this chapter to a popular class of latency-models for hard disks [106, 100, 28]. Under this class of models, the latency T associated to reading data from a disk is a sum of three components: $T = T_{\text{seek}} + T_{\text{rotation}} + T_{\text{transfer}}$. Here, T_{seek} is the "seek latency," that is, the latency for the cylinder to get positioned under the header; T_{rotation} is the "rotation latency," that is, the time taken for the sector to get positioned under the header; and T_{transfer} is the "transfer time," that is, the time taken to transfer a block using an I/O bus [125,



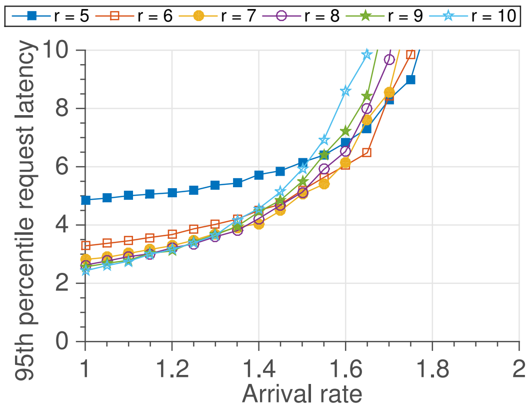
(a)



(b)



(c)



(d)

Figure 3.4: **The 95th percentile latency in a $(n = 10,\ k = 5)$ system with Poisson arrivals, and various service time distributions.** (a) Exponential with rate 1; (b) Heavy-everywhere: a mixture of exponential distributions; (c) Light-everywhere: a shifted exponential; and (d) Exponential with rate 1 where additionally, removing an unfinished job from a server requires the server to remain idle for a time distributed exponentially with rate 10.

Table 3.2: **Performance specifications of WD2500YD.**

$T_{\text{min seek}}$	2.00 ms
$T_{\text{max seek}}$	21.0 ms
Rotation Per Minute (RPM)	7200
T_{rotation}	8.33 ms
Transfer rate	61 MB/s

102].

In order to read/write a requested file block from/to a disk, the drive head needs to move from the current track position to the target track position, the disk has to rotate accordingly, and then the file can be transferred from/to the disk platter. If disk I/O requests are random, the amount of time to seek and rotate can be modeled as an i.i.d. random variable. Modeling the random amount of time for a random disk I/O T as the sum of seek latency (T_{seek}), rotational latency (T_{rotation}), and transfer time (T_{transfer}) has been a common approach in the literature. That is,

$$T = T_{\text{seek}} + T_{\text{rotation}} + T_{\text{transfer}}. \tag{3.3}$$

Each of these terms can be modeled as follows. The seek time is modeled as a linear interpolation between the minimum and maximum seek times: $T_{\text{seek time}} = T_{\text{min seek}} + D(T_{\text{max seek}} - T_{\text{min seek}})$. Here, D is the "normalized seek distance" from the current head position to the target head position, and is modelled as the distribution $P(D \leq d) = 1 - (1-d)^2$ for $0 \leq d \leq 1$. The rotation latency is a random variable distributed uniformly in $[0, T_{\text{rotation period}}]$, and the transfer time is a constant $\frac{F}{R_{\text{transfer}}}$, where F is the size of requested block and R_{transfer} is the transfer rate [102]. The values of $T_{\text{min seek}}$, $T_{\text{max seek}}$, $T_{\text{rotation period}}$ are fixed constants.

We first perform an analytical characterization of this class of service times. We show that this class indeed falls under one of the classes analyzed earlier in the chapter, and hence allows for the use of our results established earlier in Section 3.4.

Proposition 3.6 (Disk I/O time). *For any $T_{\text{max seek}} > T_{\text{min seek}} \geq 0$, $T_{\text{rotation period}}, R_{\text{transfer}} > 0$, the distribution of the disk I/O time for file blocks of a fixed size $F > 0$ is light-everywhere.*

We will now further augment the analytical characterization given by Proposition 3.6 with numerical simulations. In Table 3.2, we list the performance specifications of Western Digital's hard disk model WD2500YD [24]. Using the parameters given by the specifications and the simple model of disk I/O time discussed above, the distribution of disk read time is plotted in Fig. 3.5, for a file block of size $100KB$. We simulate a system with various values of the redundant request degree under this service time distribution. The results from these simulations are depicted in Fig. 3.6(a) and Fig. 3.6(b), showing the average and the 95th percentile latency respectively for a $n = 10$, $k = 5$ disk array. From the figures, we observe that both the average latency and tail latency exhibit tradeoffs identical to those that were established analytically in Section 3.4.

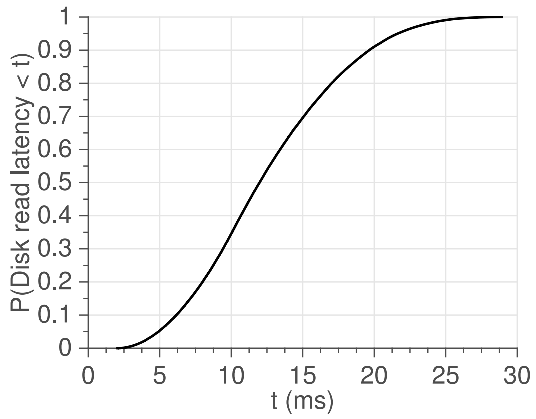
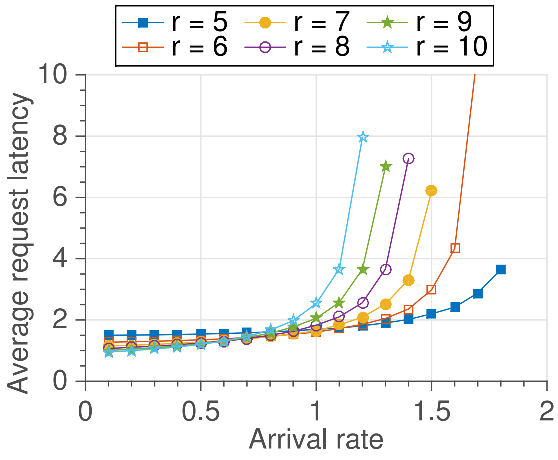
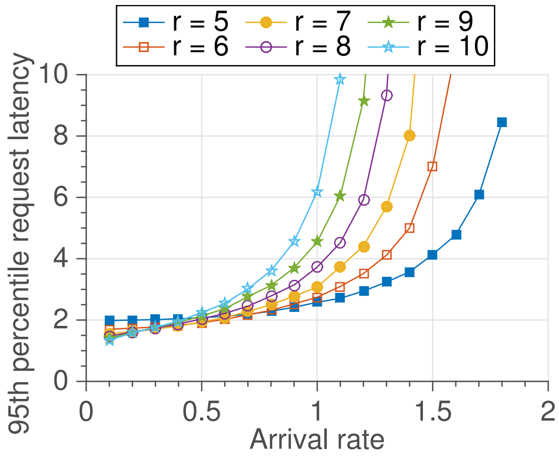


Figure 3.5: **Disk read latency model of the WD2500YD hard disk.**



(a)



(b)

Figure 3.6: **The average request latency of an array of WD2500YD disks with $(\mathbf{n = 10, k = 5})$.** The arrival process is Poisson. Plotted in (a) is the average request latency, and in (b) is the 95th percentile latency.

3.6 Distributed Buffers

The model with distributed buffers closely resembles the case of a centralized buffer. The only difference is that in this distributed setting, each server has a buffer of its own, and the jobs of a batch must be routed to some r of the n buffers *as soon as the batch arrives in the system.* The protocol for choosing these r servers for each batch is allowed to be arbitrary for the purposes of this chapter, but for concreteness, the reader may assume that the r least-loaded buffers are chosen, or the Join the r-Shortest Queue policy is used.

The setting with distributed buffers is illustrated in the following example.

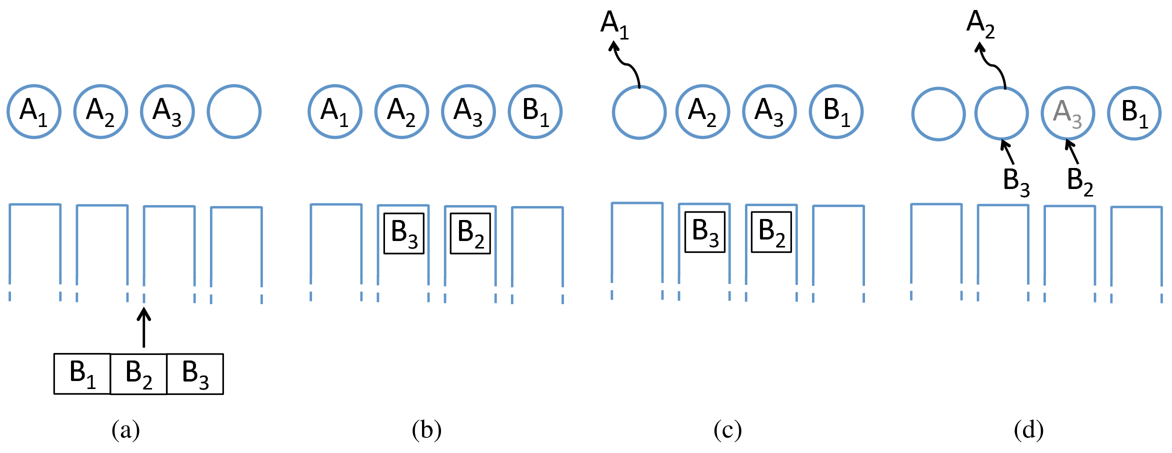


Figure 3.7: **An illustration of the setting with distributed buffers for parameters $\mathrm{n} = 4$, $\mathrm{k} = 2$ and request-degree $\mathrm{r} = 3$, as described in Example 3.2.**

Example 3.2. *Fig. 3.7 illustrates the system model and the working of the system in the distributed setting, for parameters $n = 4$, $k = 2$ and $r = 3$. The system has $n = 4$ servers, and each of these servers has its own buffer, as shown in Fig. 3.7(a). Denote the four servers (from left to right) as servers 1, 2, 3 and 4. Fig. 3.7(a) depicts a scenario wherein batch A is already being served by the first three servers, and batch B just arrives. The three servers (buffers) to which batch B will be sent to must be selected at this time. Suppose the algorithm chooses to send the batch to buffers 2, 3 and 4 (Fig. 3.7(b)). Now suppose server 1 completes service of job A_1 (Fig. 3.7(c)). Since there is no job waiting in the first buffer, server 1 remains idle. Note that in contrast, a centralized setting would have allowed the first server to start processing either job B_2 or B_3. Next, suppose server 2 completes service of job A_2 (Fig. 3.7(d)). With this, $k = 2$ jobs of batch A are served, and the third job A_3 is thus removed. Servers 2 and 3 can now start serving jobs B_3 and B_2 respectively.*

As in the centralized setting of Section 3.4, we continue to assume that the service-time distributions of jobs are i.i.d. and the system operates on a first-come-first-served basis. The following theorems prove results that are distributed counterparts of the results of Section 3.4.

Theorem 3.7. *Consider a system with n distributed servers such that any k of them can serve a request. The service times for different jobs and different servers are i.i.d. Then*
(a) Suppose the service time is memoryless and jobs can be removed instantly from the system. Then average latency is minimized when all batches are sent to all the servers, i.e., when $r = n$ for every batch.
(b) Suppose the service time is heavy-everywhere, $k = 1$, jobs can be removed instantly from the system and the system has a 100% server utilization. Then the average latency is minimized when each batch is sent to all n servers, i.e., when $r = n$ for each batch.
(c) Suppose the service time is light-everywhere and $k = 1$ and the system has a 100% server utilization. Then the average latency is minimized when there is no redundancy in the requests,

i.e., when $r = k\ (= 1)$ for all batches.
(d) Suppose the service time is memoryless, and removal of a job from a server incurs a non-zero delay and the system has a 100% server utilization. Then the average latency is minimized when there is no redundancy in the requests, i.e., when $r = k\ (= 1)$ for all batches.

We see that guidelines identical to those for the centralized setting apply to the distributed case as well.

3.7 General Proof Technique

This section briefly describes the general proof technique employed to prove the aforementioned results. The general proof technique is depicted pictorially in Fig. 3.8.

Consider two identical systems S_1 and S_2 with different redundant-requesting policies. Suppose we wish to prove that the redundant-requesting policy of system S_2 leads to a lower latency as compared to the redundant-requesting policy of system S_1. To this end we first construct two new hypothetical systems T_1 and T_2. The construction is such that the performance of system T_1 is statistically identical or better than S_1, and that of T_2 is statistically identical or worse than S_2. The two systems T_1 and T_2 are also *coupled* in the following manner. The construction establishes a one-to-one correspondence between the n servers of T_1 and the n servers of T_2. Furthermore, it also establishes a one-to-one correspondence between the service events occurring in both systems, i.e., the completion of any job in T_1 is associated to the completion of a unique job in T_2 and vice versa. The same sequence of arrivals is applied to both systems.

Such a coupling facilitates an apples-to-apples comparison between the two systems. We exploit this and show that at any point in time, system T_2 is in a better state than system T_1. Putting it all together, it implies that system S_2 is better than system S_1.

Most interestingly, this technique allows us to handle arbitrary arrival sequences. Furthermore, it does not restrict the results to the (asymptotic) setting when the system is in steady state, but allows the results to be applicable to any interval of time.

3.8 Conclusions and Open Problems

The prospect of reducing latency by means of redundant requests has garnered significant attention among practitioners in the recent past (e.g., [110, 5, 87, 47, 2, 54, 122, 29, 69, 113, 39]). Many recent works empirically evaluate the latency performance of redundant requests under diverse settings. In this chapter, we propose a model that captures key features of such systems, and under this model we analytically characterize several settings wherein redundant requests help and where they do not. For each of these settings, we also derive the optimal redundant-requesting policy.

While we establish the performance of redundant requests for several scenarios in this chapter, the characterization for more general settings remains open. Three questions that immediately arise are: (a) What is the optimal redundant-requesting policy for service-time distributions and removal-costs not considered in this paper? (b) We observed in the simulations (e.g., Fig. 3.3(c))

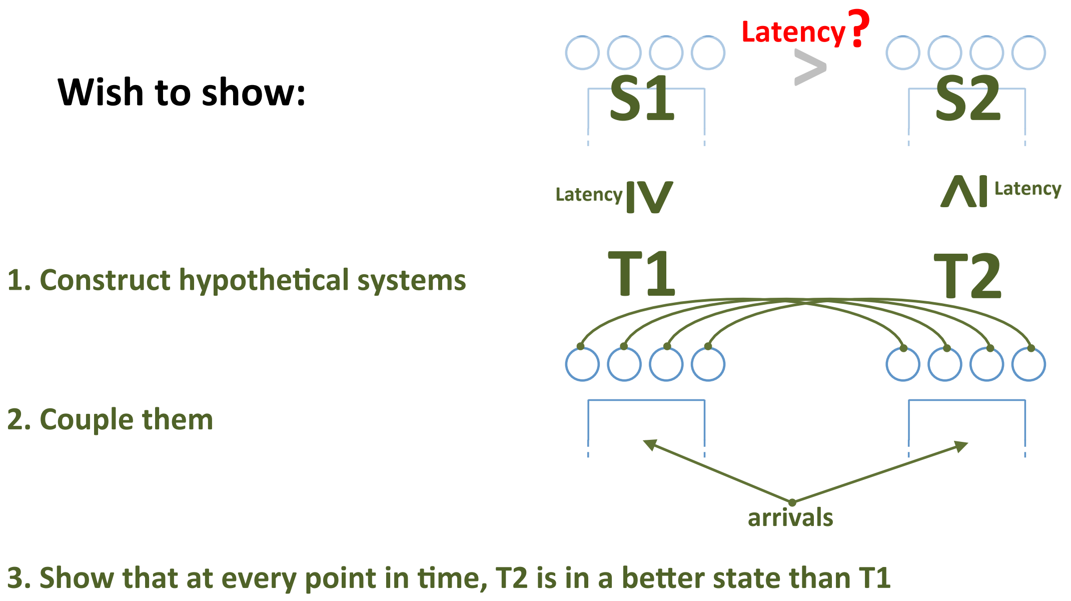

Wish to show:
S1
Latency?
>
S2
Latency ≥
≤ Latency
1. Construct hypothetical systems
T1
T2
2. Couple them
arrivals
3. Show that at every point in time, T2 is in a better state than T1

Figure 3.8: **A pictorial depiction of the general proof technique.**

that for several service-time distributions, redundant requests start hurting when the system is loaded beyond a certain threshold. What are the values of these thresholds? (c) What happens when the requests or the servers are heterogeneous, or if the service-times of different jobs of a batch are not i.i.d.? Addressing these questions constitutes a useful direction of future research. In the following chapter, we attempt to answer the first question.

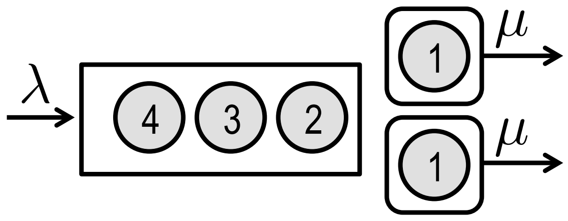
Chapter 4

Exact Analysis of Redundant Requests Scheduling with Cancellation Overhead

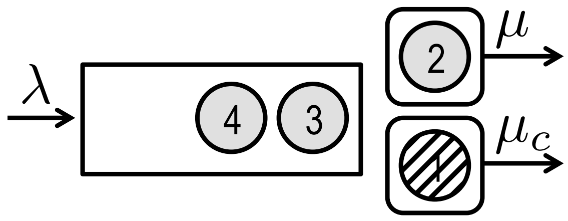
4.1 Introduction

Individual components of a large-scale distributed system exhibit highly variable response times[29]. The high variability is due to many factors: shared resources, queueing delay in multiple layers, and hardware failures. Unfortunately, completely removing such sources of variability in large-scale systems is infeasible. As a result, researchers have proposed several approaches to achieve better latency performance while living with this high variability. One of the most promising approaches is that of scheduling *redundant requests* to multiple components or servers [2, 54, 122, 29, 69]. That is, one can schedule the same job at different servers and obtain the result from the request that responds first in order to reduce latency. Clearly, the technique of scheduling redundant requests can be employed only if requests can be simultaneously served at many different servers. Many systems indeed possess this desired property: in a distributed computing and data storage system, a compute job can be served at any of the servers that stores a copy of the input data; in a network with multiple routes between nodes, the transmitter can choose to establish multiple flows across different paths, and can transmit redundant packets along them.

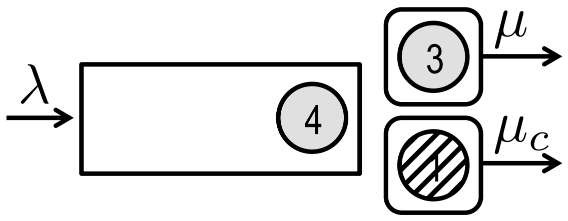
Many recent works [2, 54, 122, 29, 69] empirically observe reductions in job latency by scheduling redundant requests. Following the empirical efficacy of redundant requests, several recent works propose theoretical models of redundant requests scheduling, and find the optimal scheduling policies under the proposed models [103, 58, 69]. However, most existing models commonly assume immediate cancellation of straggling requests; i.e., *all requests except for the finished one are immediately removed from the system.* While this assumption enables an important first step in the analysis of systems with redundant requests, it limits the studied models in several ways. First, cancellation of requests in highly distributed systems typically takes a non-negligible amount of time for exchange of control signals and restoration of server states. Further, job cancellation is not even feasible in some applications where schedulers do not have full control over



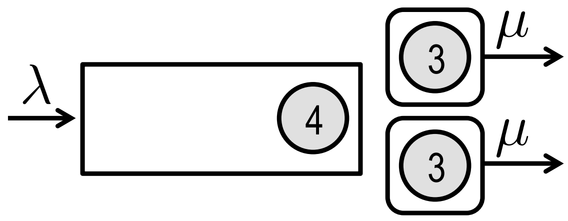
(a) $t = t_1$: 4 jobs are in the system. Job 1 is replicated and both server 1 and 2 are working on it.



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



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



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

Figure 4.1: **An example of scheduling redundant requests with cancellation overheads.** Jobs being canceled are shaded. Job 1 experiences a faster response because two servers work on the same job as depicted in 4.1(a), but the other jobs are delayed because server 2 takes a long time to cancel job 1, and some system resource is not utilized during the cancellation period. The scheduling policy used here is called π_∞, and it is formally defined in Section 4.2.4.

the entire system. That is, the scheduler can issue redundant requests but cannot revoke them. For instance, Vulimiri et al. considers requesting multiple DNS (Domain Name Servers) queries to achieve a lower DNS response delay, and these requests cannot be canceled because the scheduler does not have control over them [122].

Figure 4.1 illustrates how the cancellation overheads negatively affect the overall system performance. In Figure 4.1(a), job 1 is replicated, and it can be served quickly because two servers work on the same job. However, the other jobs are negatively affected if server 2 takes some time to cancel the copy of job 1: during the cancellation period, the system's resource is being wasted. It is therefore no longer clear whether redundant requests will help or not due to this phenomenon. More generally, it is unclear how one can optimally use redundant requests when accounting for this overhead. We attempt to answer this question in this chapter.

Our contributions are as follows. First, bridging the evident gap between the existing models and practice, we propose a new model of redundant requests with cancellation overhead. In order to strike a balance between analytical tractability and capturing key system attributes, we propose a 'memoryless cancellation' model to capture the job cancellation overheads, and study how this cancellation overhead affects the latency performance of several scheduling policies. We show that *maximally scheduling redundant requests is inefficient even with almost negligible cancellation overhead.* This implies the limited applicability of theoretical results that do not consider such cancellation overhead. We then find the latency-optimal scheduling policies and show the necessity of using dynamic policies over static policies: we show that *the optimal dynamic policy can provide up to 16% additional latency reduction over the optimal static policy.* Our analysis uses a unique combination of the Recursive Renewal Reward (RRR) technique [42], a recently developed queueing analysis technique, and dynamic programming. To the best of our knowledge, this is the first work to characterize the optimal dynamic policy of scheduling redundant requests.

The remainder of this chapter is organized as follows. Section 4.2 describes the queueing model that captures redundant requests and cancellation overhead. Section 4.3 and Section 4.4 provide analytical results for static and dynamic schedulers respectively. Finally, Section 4.5 presents conclusions and discusses open problems.

4.1.1 Related Works

While most of the existing works consider static redundant-requests scheduling policies, a few exceptions are as follows. Dynamic scheduling policies are considered in [69] and [123]. Liang et al. [69] propose dynamic scheduling policies based on an approximate analysis. In [123], Wang et al. study scheduling of finite number of jobs in the queue. Sun et al. [115] prove that a certain class of dynamic polices are either delay-optimal or near-optimal with a constant gap. Joshi et al. [59] analyze various scheduling policies including the one that cancels redundant requests when one task starts getting served.

Our work differs from these works in two ways. First, we explicitly include the overhead of canceling redundant requests in the model, and find the optimal scheduling policy. Most of the existing works do not consider this overhead, and propose scheduling policies that may not perform well in practice. Second, to the best of our knowledge, we provide the first dynamic scheduling policy that is provably optimal under the new model.

4.2 Model

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

4.2.1 Arrival and Service Model

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

4.2.2 Redundant Requests

We assume that a job can be scheduled at more than one server: whenever a server becomes available, the scheduler may schedule a new job or a replica of a running job at the available server. When redundant requests are used, the job latency is defined as the difference between the *earliest* departure time of the job and the time of its arrival. When a job is scheduled at multiple servers, we assume that *the redundant jobs experience independent random service times.* The independence assumption is witnessed in previous works. For instance, [70] reports the independence between download times of a file from the cloud server across multiple threads.

4.2.3 Memoryless Cancellation

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

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

1. ($\mu_c = \mu$: Infeasible cancellation) In some systems, it may be impossible or too complicated to cancel the replicas of a job. That is, one cannot cancel a running job even though copies of that job is already served at other servers. This can be simply modeled by setting $\mu_c = \mu$ due to the memoryless property of the exponential distribution. Even though using redundant requests seems very inefficient under this setup, we show that when the arrival rate is smaller than a certain threshold, one can still reduce job latency by using redundant requests.

2. ($\mu_c = \infty$: Immediate cancellation) The immediate cancellation of the redundant requests can be supported by systems that are very centralized and have a negligible communication delay. In other words, one can cancel any job that is submitted to a server if the identical job is served by another server, and the cancellation happens immediately. This can be captured by setting $\mu_c = \infty$.

3. ($\mu_c > \mu$: Slow cancellation) In most systems of interest, cancellation is possible but requires a small yet non-negligible amount of time. For this case, we assume that the cancellation of a job requires a random amount of time that is exponentially distributed with rate μ_c.

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

4.2.4 Scheduling Policies

In a queueing system, the scheduler needs to make a scheduling decision when a resource becomes available. In this work, the system allows the scheduler to schedule a redundant request of a running job or a waiting job in the queue. This flexibility makes the problem of finding the optimal scheduling policy challenging. In this work, we restrict all scheduling policies to be First-come-first-served (FCFS), work-conserving, and non-preemptive. We consider both *static* and *dynamic* scheduling policies: a dynamic policy is one that adapts based on changes in the state of the system (e.g., the number of jobs in the system), whereas a static policy is oblivious to the changes in state [49].

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

Definition 4.1. (Definition of π_γ) A scheduling policy π_γ schedules a redundant request of the least-recently arrived job if and only if the number of distinct jobs in the system at the time of decision $N(t)$ is less than or equal to $\gamma \in \{0, 1, \ldots\}$.

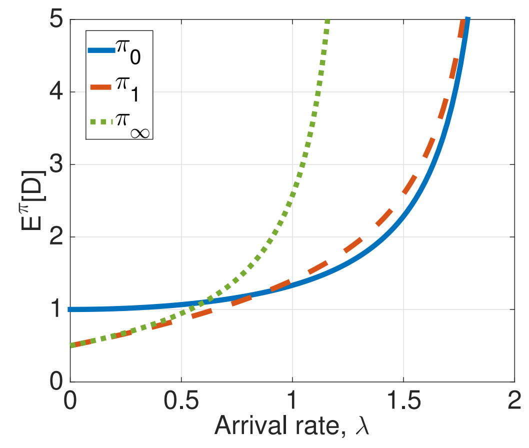
Note that π_0 and π_∞ are two special cases, and they are the only static policies among the class of the schedulers: π_0 never schedules a redundant request, which is the classic FCFS scheduler; π_∞ always schedules redundant requests regardless of the system state. All the other policies are dynamic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

4.2.5 M/M/2 with Redundant Requests

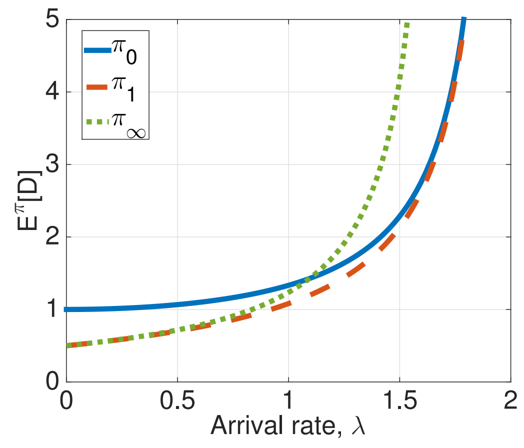
As a first attempt to understand the latency performance of redundant requests, and to find the latency-optimal scheduling policy, we focus on two-servers systems: M/M/2 queueing systems with redundant requests.

4.3 Analysis of Static Schedulers

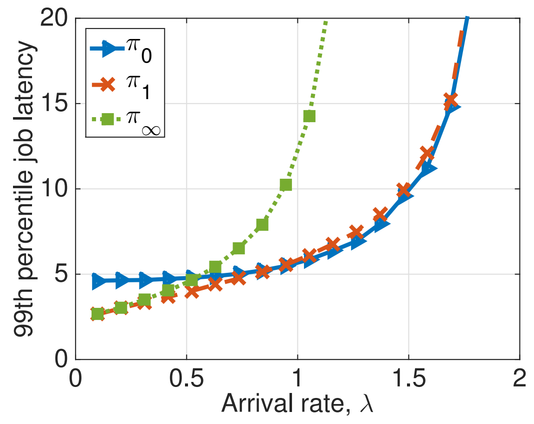
In this section, we analyze and compare the two static schedulers: π_0, which does not use any redundant request, and π_∞, which always schedules redundant requests. The analysis of these static scheduling policies are still useful when one wants to adapt redundant requests but cannot exploit the status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



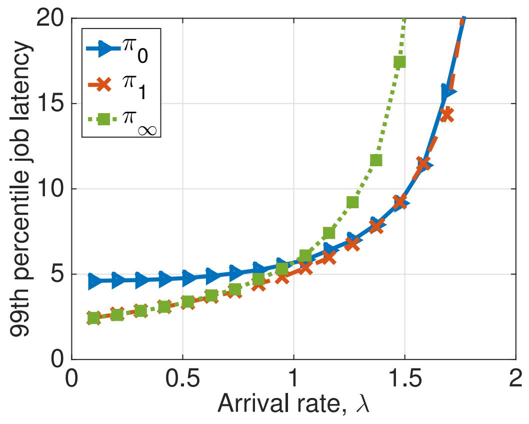
(a) Average latency: $\mu_c = \mu$



(b) Average latency: $\mu_c = 5\mu$



(c) 99th percentile latency: $\mu_c = \mu$



(d) 99th percentile latency: $\mu_c = 5\mu$

Figure 4.2: **The average and tail latency of π_0(solid, blue), π_1(dashed, orange), and π_∞(dotted, green) with $\mu = 1,\ \mu_c = \mu$ and 5μ.** π_0 does not schedule redundant requests, π_1 schedules them adaptively, and π_∞ maximally schedules them.

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

4.3.1 Without Redundant Requests: π_0

Since π_0 does not use any redundant request, an M/M/2/π_0 is identical to an M/M/2/FCFS. Thus, the following lemma immediately follows [49].

Lemma 4.1. *(Stability and job latency of M/M/2/π_0) The maximum arrival rate and the average job latency of an M/M/2/π_0 are as follows:* $\lambda_{max}^{\pi_0} = 2\mu,\ \ \mathbb{E}^{\pi_0}[D] = 4\mu/(4\mu^2 - \lambda^2)$.

Figure 4.2 shows the average job latency of an M/M/2/π_0 system as a function of λ.

4.3.2 With Redundant Requests: π_∞

Now, we consider the other static scheduling policy π_∞. We first illustrate how redundant requests are scheduled using a sample evolution of M/M/2/π_∞ system illustrated in Figure 4.1. In Figure 4.1(a), there are 3 jobs waiting in the queue, and job 1 is replicated, and being served by server 1 and server 2. In Figure 4.1(b), server 1 finishes job 1 earlier than the other, and starts working on job 2 while the other server starts canceling the redundant request of job 1. Note that job 1 is deemed served at this point, and the remaining replica does not affect this job's latency. In Figure 4.1(c), server 1 finishes job 2 and accepts job 3. In Figure 4.1(d), server 2 completes job cancellation, and starts working on job 3's replica.

Since the arrival process, the service process, and the cancellation process are all memoryless and independent, an M/M/2/π_∞ system is Markovian, and the system can be modeled as a Markov chain by defining the state of the system appropriately. We enumerate all the states of an M/M/2 system in Table 4.1. We illustrate the state notation using the example in Figure 4.1. In Figure

Table 4.1: **The states of an M/M/2 system.** $L(S)$ denotes the level of state S. $N(S)$ denotes the number of distinct jobs in the system in state S.

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

4.1(a), 3 jobs are waiting in the queue, and two servers are working on the same job. We denote this state by $(s,3)$, where s means that the same job is being processed by both servers, and 3 indicates the number of jobs in the queue. In Figure 4.1(b), server 1 completes the service of job 1, and starts working on job 2 while server 2 is canceling job 1. We name this state as $(c,2)$, where c stands for cancellation. Similarly, Figure 4.1(c) corresponds to state $(c,1)$, and Figure 4.1(d) corresponds to state $(s,1)$. With this definition of states, the Markov chain of the system can be found, and it is depicted in Figure 4.3. The bottom row shows all the states in which one of the two servers is canceling a job, and the top row shows all the states in which two servers are working on

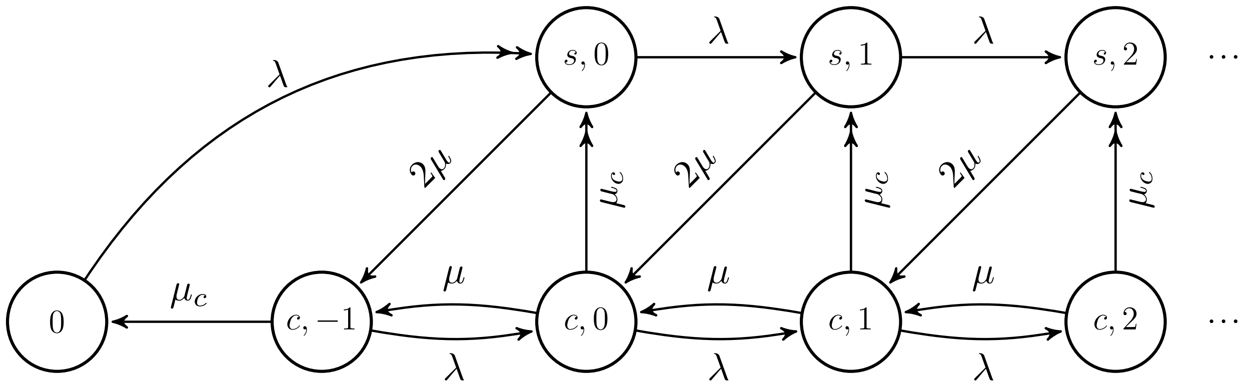


Figure 4.3: **A Markov chain of an M/M/2/π_∞.** States are defined in Table 4.1. States $(s,i),\ i\geq 0$ represent that both servers are working on the same job, and there are i jobs waiting in the queue. States $(d,i),\ i\geq 0$ represent that two servers are working on different jobs, and there are i jobs waiting in the queue. State (0) represent there is no job in the system, and state $(c,-1)$ represents there is only one job that is being canceled by one server.

the same job. We distinguish the transitions corresponding to scheduling redundant requests with double tips.

We are now ready to analyze the latency performance of the system. We first establish the relationship between the average number of 'distinct jobs' in the system and the average job latency of the system using the Little's Law [73].

Lemma 4.2. *(Little's Law with Redundant Requests) The average job latency of a queueing system that schedules redundant requests is $\mathbb{E}[D]=\mathbb{E}[N]/\lambda$, where $\mathbb{E}[N]$ is the average number of distinct jobs (excluding redundant requests and those being canceled).*

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

Lemma 4.2 implies that we can analyze the average job latency if we find the average number of distinct jobs in the system. We do so by using the recursive renewal reward technique, developed recently in [42]. Using this technique, we can exactly analyze the system's performance without relying on the numerical methods. We first present the stability region and the average job latency of an M/M/2/π_∞.

Theorem 4.3. *(Stability and job latency of M/M/2/π_∞) An M/M/2/π_∞ system is stable if and only if the arrival rate λ is strictly less than the maximum arrival rate λ_{max}, where λ_{max} is as follows:*

$$\lambda_{max}^{\pi_\infty}=2\mu(\mu+\mu_c)(2\mu+\mu_c)^{-1}. \tag{4.1}$$

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

$$\mathbb{E}^{\pi_\infty}[D] = \frac{(\mu+\mu_c)\left\{(2\mu_c(\mu+\mu_c)+\lambda(4\mu+\mu_c)\right\}}{\left\{2\mu_c(\mu+\mu_c)+\lambda(2\mu+\mu_c)\right)(2\mu(\mu+\mu_c)-\lambda(2\mu+\mu_c)\}} \tag{4.2}$$

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

$$\mathbb{E}[N] = \frac{\mathbb{E}[\text{accumulated reward over a renewal cycle}]}{\mathbb{E}[\text{length of a renewal cycle}]} \overset{\text{def}}{=} \frac{R}{T}. \tag{4.3}$$

For notational simplicity, we define $L(S)$, the level of state S, as shown in Table 4.1. Note that $L(S) = N(S) + 1$ except for $S = (0)$. Also, we denote 'state s' simply by s unless it makes any confusion.

The average length of a renewal cycle T can be found as follows. Clearly, any renewal path must visit $(c, -1)$ to visit the home state. Thus, if we find the average length of the process from $(s, 0)$ to $(c, -1)$, and that of the process from $(c, -1)$ to the home state, we can find T. By defining the average length of the process from state S to one level left of state S as T^L_S, T can be expressed as follows:

$$T = \lambda^{-1} + T^L_{s,0} + T^L_{c,-1} \tag{4.4}$$

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

$$T^L_{s,0} = (\lambda + 2\mu)^{-1}\left[1 + \lambda(T^L_{s,1} + T^L_{c,0}) + 2\mu \cdot 0\right] \tag{4.5}$$

$$T^L_{c,-1} = (\lambda + \mu_c)^{-1}\left[1 + \lambda(T^L_{c,0} + T^L_{c,-1}) + \mu_c \cdot 0\right] \tag{4.6}$$

$$T^L_{c,0} = (\lambda + \mu + \mu_c)^{-1}\left[1 + \lambda(T^L_{c,1} + T^L_{c,0}) + \mu \cdot 0 + \mu_c T^L_{s,0}\right] \tag{4.7}$$

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

$$R^L_{c,-1} = (\lambda+\mu_c)^{-1}\left[\lambda(R^L_{c,0}+R^L_{c,-1})+\mu_c\cdot 0\right] \tag{4.8}$$
$$R^L_{s,0} = (\lambda+2\mu)^{-1}\left[1+\lambda(R^L_{s,1}+R^L_{s,0})+2\mu\cdot 0\right] \tag{4.9}$$
$$R^L_{c,0} = (\lambda+\mu+\mu_c)^{-1}\left[1+\lambda(R^L_{c,1}+R^L_{c,0})+\mu\cdot 0+\mu_c R^L_{s,0}\right] \tag{4.10}$$

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

$$R^L_{s,1} = R^L_{s,0}+T^L_{s,1} = R^L_{s,0}+T^L_{s,0} \tag{4.11}$$
$$R^L_{c,1} = R^L_{c,0}+T^L_{c,1} = R^L_{c,0}+T^L_{c,0} \tag{4.12}$$

Now, the above equations can be solved, and the solutions of these equations are provided in Appendix A.3. Then, using Lemma 4.2, one can find $E^{\pi_\infty}[D]$, and $\lambda^{\pi_\infty}_{\text{max}}$. □

Note that the maximum arrival rate of an M/M/2/π_∞ system is strictly lower than that of an M/M/2/π_0 system since cancellation overheads induce wastage of system resource. However, the job latency performance under the policy π_∞ is still better if the arrival rate is low enough. Figure 4.2 compares the average job latencies of the two policies with different values for λ and μ_c: the solid (blue) line shows the average job latency of an M/M/2/π_0, and the dotted (green) line shows the average job latency of an M/M/2/π_∞. Observe that there exists a threshold after which π_∞ becomes worse than π_0, and this threshold depends on the cancellation overhead; studying this threshold fully characterizes the optimal static scheduling policies.

4.3.3 The Latency-optimal Static Scheduler

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

Theorem 4.4. *(The latency-optimal static scheduling policy for M/M/2) If the cancellation of jobs is immediate (i.e., $\mu_c = \infty$), the policy π_∞ is always latency-optimal. If the cancellation of jobs is infeasible (i.e., $\mu_c = 0$), the policy π_∞ is latency-optimal if and only if $\lambda < 0.6013\mu$, and the policy π_0 is latency-optimal otherwise. If the job cancellation rate is $\mu < \mu_c < \infty$, the π_∞ policy is latency-optimal if and only if $\lambda < \beta_s(\mu_c)\mu$, where $\beta_s(\mu_c)$ is the unique solution of the following*

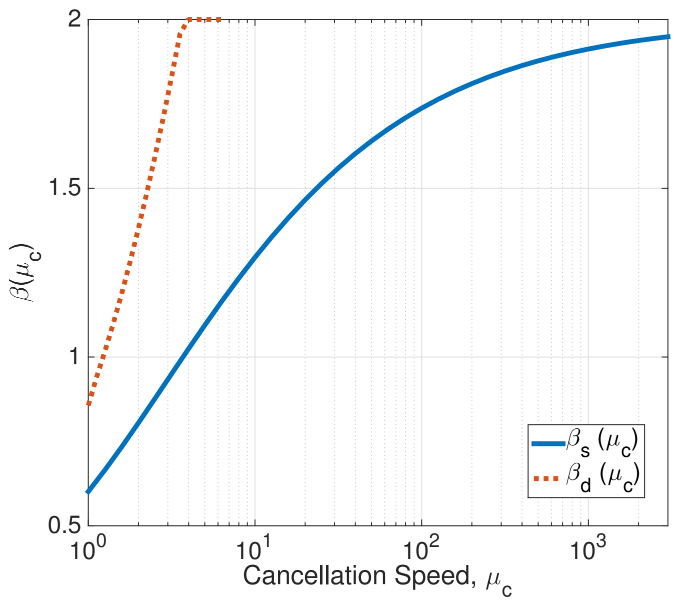


Figure 4.4: **Threshold function $\beta_s(\mu_c)$ and $\beta_d(\mu_c)$ defined in Theorem 4.4 and Theorem 4.6 with $\mu = 1$.** The slow convergence of $\beta_s(\mu_c)$ explains how inefficient the policy π_∞ is even with a negligible cancellation overhead. For instance, $\beta_s(\mu_c) \simeq 1.25\mu = (0.625) \cdot 2\mu$ implies that the policy π_∞ becomes inefficient at 62.5% system if $\mu_c = 10\mu$.

third-order polynomial equation in β.

$$\sum_{i=0}^{3} \alpha_i \beta^i = 0 \tag{4.13}$$

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

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

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

We plot the threshold function $\beta_s(\mu_c)$ in Figure 4.4 when $\mu = 1$. *Note that the threshold function $\beta_s(\mu_c)$ does not approach 2μ as quickly as one would expect.* For instance, $\beta_s(\mu_c) \simeq$

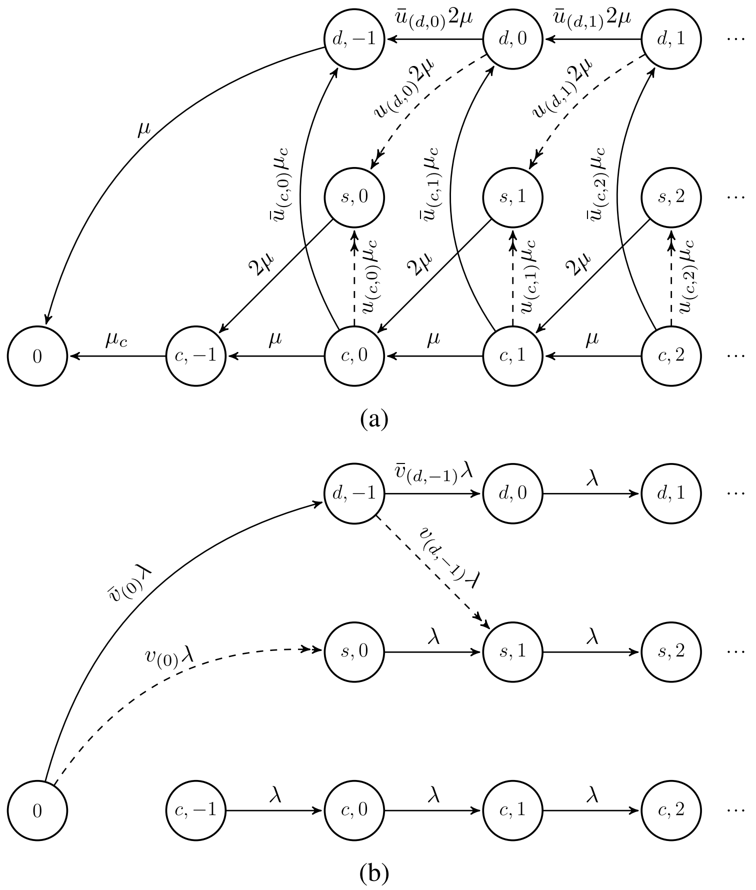


Figure 4.5: **A general Markov chain of an M/M/2 with a dynamic scheduler.** The states are defined in Table 4.1. This Markov chain involves additional states compared to those shown in Figure 4.3. The top row shows states (d, i) for all $i \geq 0$ where each represents a state where servers are working on different job(s), and there are i jobs in the queue. For instance, $(d, -1)$ represents a special case where only one server is working on a job, the other server is idle, and there is no waiting job in the queue. Transitions corresponding to scheduling redundant requests are represented as dashed lines with double arrow tips.

$1.25\mu = (0.625) \cdot 2\mu$, when $\mu_c \simeq 10\mu$, and $\beta_s(\mu_c) \simeq 1.9\mu = (0.95) \cdot 2\mu$, when $\mu_c \simeq 1000\mu$: the policy π_∞ becomes worse than the policy π_0 if the system load is higher than 62.5% even though cancellation overhead is as low as $\mu_c = 10\mu$. *That is, maximally scheduling redundant requests can utterly fail even with a low cancellation overhead.* In Section 4.4, we show that an optimal dynamic scheduling policy can be used to avoid this phenomenon.

4.4 Optimal Dynamic Schedulers

In this section, we consider a class of dynamic schedulers, and find the latency-optimal dynamic scheduling policy.

4.4.1 Problem Formulation

A dynamic scheduling policy decides about scheduling redundant requests, based on the current state of the system. To be more precise, we consider *stationary deterministic* policies that make deterministic decisions based only on the current state of the system. There are three sets of events when the scheduler needs to make a decision: when a job is completed at a server, when a job cancellation is completed, and when a new job arrives.

Figure 4.5 shows all possible transitions using different controls on the Markov chain. In the figure, scheduling of a redundant request is depicted as a dashed line with double arrow tip. Figure 4.5(a) shows all the transitions/controls when a job is served or a redundant job is canceled. The scheduler can decide whether to schedule a redundant request to the idle server or not, according to its decision rules. For instance, consider the state $(d, 1)$, where two servers are working on distinct jobs, and another job is waiting in the queue. Because each of the two servers finishes its service with rate μ, the sum rate of service transitions is 2μ. When this happens, the scheduler has two choices: it can schedule at the idle server either a copy of the running job at the other server or a waiting job in the queue. By defining u_x as an indicator of scheduling redundant requests when a server becomes available at state x, we can draw one transition from state $(d, 1)$ to state $(s, 1)$ with rate $u_{(d,1)}2\mu$, and the other transition to $(d, 0)$ with rate $(1 - u_{(d,1)})2\mu \overset{\text{def}}{=} \bar{u}_{(d,1)}2\mu$. Similarly, Figure 4.5(b) shows all the transitions/controls when a job arrives. We define v_x as an indicator of scheduling redundant requests when a job arrives at state x. When a job arrives, there is an available resource only if the previous state is (0) or $(d, -1)$: only these two states have associated control variables on arrival events. Thus, any dynamic policy can be characterized by specifying the following three sets of controls: $\pi_{\text{a(rrival)}}, \pi_{\text{s(ervice)}}$, or $\pi_{\text{c(ancellation)}}$.

$$\pi_{\text{a}} = (v_{(0)}, v_{(d,-1)}), \quad \pi_{\text{s}} = (u_{(d,0)}, u_{(d,1)}, \ldots), \quad \pi_{\text{c}} = (u_{(c,0)}, u_{(c,1)}, \ldots)$$

Due to Lemma 4.2, the problem of finding the latency-optimal dynamic scheduling policy can be formulated as follows.

$$\pi^* = \underset{\pi=(\pi_{\text{a}},\pi_{\text{s}},\pi_{\text{c}})}{\arg\min}\ \mathbb{E}^{\pi}[D] = \underset{\pi=(\pi_{\text{a}},\pi_{\text{s}},\pi_{\text{c}})}{\arg\min}\ \mathbb{E}^{\pi}[N] \tag{4.14}$$

4.4.2 The Latency-Optimal Dynamic Schedulers

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

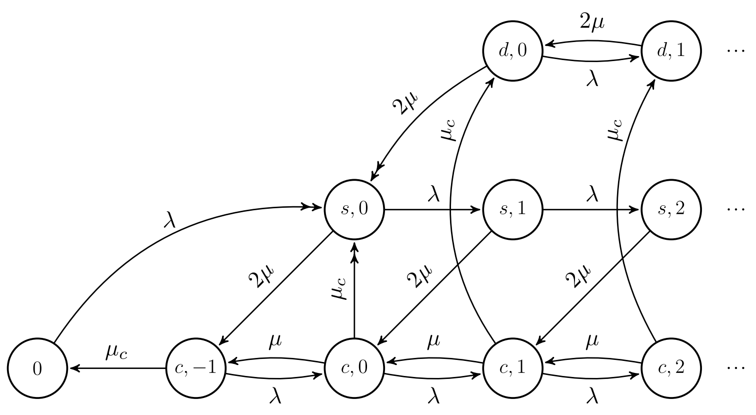


Figure 4.6: **The Markov chain of an M/M/2/π_1 system**

Theorem 4.5. *(Stability and job latency of M/M/2/π_1) An M/M/2/π_1 system is stable if and only if the arrival rate λ is strictly less than the maximum arrival rate $\lambda_{max} = 2\mu$. If the system is stable, the average job latency $\mathbb{E}^{\pi_1}[D]$ can be exactly found using the RRR technique.*

Proof. We defer the detailed proof of the theorem to the Appendix, and provide a brief outline of the proof. Similar to the proof of Theorem 4.3, we first draw the Markov chain of the system under policy π_1. The Markov chain is depicted in Figure 4.6. Note that when a Markov chain starts from a state in the repeating portion, it can enter the one left level through one of the two states, (d, i) or $(c, i+1)$ for some i. Thus, one needs to find with what probabilities the process enters (d, i) and $(c, i+1)$, respectively. This complicates the analysis, but one can still obtain the exact latency performance. □

The following theorem states that depending on the arrival rate, either π_1 or π_0 is the optimal dynamic scheduling policy.

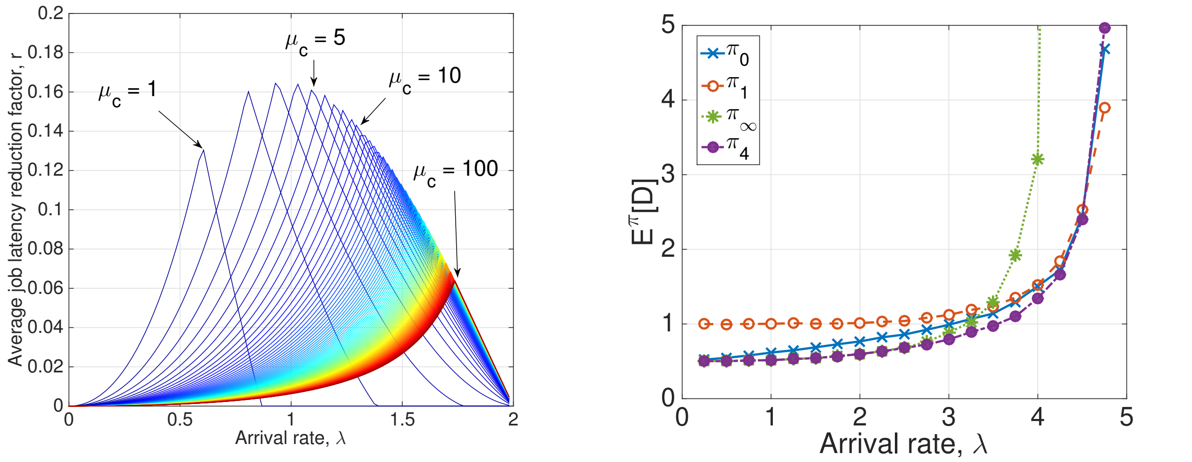
Theorem 4.6. *(The latency-optimal dynamic scheduling policy for M/M/2) The π_1 policy is latency-optimal if and only if $\lambda < \beta_d(\mu_c)\mu$. Otherwise, π_0 is optimal. The threshold function for dynamic scheduling policy $\beta_d(\mu_c)$ is defined as the unique solution of the following equation.*

$$\mathbb{E}^{\pi_1}[D]|_{\lambda=\beta\mu} = \mathbb{E}^{\pi_0}[D]|_{\lambda=\beta\mu} \tag{4.15}$$

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

Proof. By characterizing the structure of the optimal dynamic scheduling policy, one can show that π_0 and π_1 are the only candidates for being the optimal policy. Detailed proof is in the Appendix. □

In Figure 4.4, $\beta_d(\mu_c)$ is plotted. Note that the threshold function $\beta_d(\mu_c)$ approaches 2μ much faster than $\beta_s(\mu_c)$. For instance, $\beta_d(\mu_c) \approx 2\mu$ when $\mu_c \approx 3\mu$: the dynamic policy π_1 becomes optimal at almost all arrival rates if cancellation overhead is $\mu_c \approx 3\mu$. This phenomenon can be



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

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

Figure 4.7: **The value of dynamic scheduling and performance of $\{\pi_\gamma\}$ for M/M/n systems.**

also observed in Figure 4.2: when $\mu_c = 5\mu$, π_1 is strictly better than π_0 at all arrival rates. This observation is in stark contrast to the known fact that the optimal static scheduler performs as good as the optimal dynamic scheduler when cancellation overhead is zero [103], affirming that misleading results may be derived if the cancellation overhead is completely ignored.

4.4.3 Performance Gap between the Optimal Static and the Optimal Dynamic Policies

We found the optimal static schedulers and the optimal dynamic schedulers in Section 4.3 and Section 4.4, respectively. In this section, we compare the optimal static scheduling policy and the optimal dynamic policy. Denote the optimal static policy as π_s^* and the optimal dynamic policy as π_d^*. Theorems 4.4 and 4.6 imply that π_s^* achieves the lower envelope of π_0 and π_∞, and π_d^* achieves the lower envelope of π_0 and π_1. In Figure 4.4, the optimal static policy achieves the lower envelope of the dotted (green) curve and the solid (blue) curve, whereas the dynamic policy achieves the lower envelope of the dashed (orange) curve and the solid (blue) curve.

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

Using Theorem 4.4 and Theorem 4.6, we find r and plot it in Figure 4.7(a).

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

4.5 Conclusion

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

There are many open problems related to this work. First of all, we believe that similar structural results of the optimal dynamic policies can be shown for M/M/n, but the exact analysis of latency performance is still open. It is empirically observed that the class of simple threshold policies $\{\pi_\gamma\}$ perform well even for M/M/n systems. Figure 4.7(b) shows the average job latency under different scheduling policies for M/M/5 systems. We compare the four policies π_0, π_1, π_∞ and π_4. Upto a certain threshold, π_4 provides the best performance, and then π_1 becomes dominant beyond the threshold. We conjecture that appropriate choice of $\{\pi_\gamma\}$ can achieve near-optimal latency performances for general M/M/n systems.

Another important metric to be studied is tail latency. Figure 4.2 plots the 99th percentile job latency obtained via simulations. We observe that the tail latency tradeoffs between scheduling policies are almost identical to the average latency tradeoffs. We conjecture similar thresholding rules can be found for optimizing tail latency, but rigorous analysis is open.

Finally, the optimal scheduling redundant requests under general service, arrival, cancellation models remain open.

Chapter 5

Speeding Up Distributed Computing Using Codes

5.1 Introduction

In recent years, the computational paradigm for large-scale machine learning and data analytics has started to move towards massively large distributed systems, comprising individually small and unreliable computational nodes (low-end, commodity hardware). Specifically, modern distributed systems like Apache Spark [128] and computational primitives like MapReduce [30] have gained significant traction, as they enable the execution of production-scale tasks on data sizes of the order of terabytes. The backbone of these large and complex platforms consists of three functional layers: (i) a computational layer; (ii) a communication layer (to move data around the system as needed); and (iii) a storage layer. In order to develop and deploy sophisticated solutions and tackle large-scale problems in machine learning, science, engineering, and commerce, it is important to understand and optimize novel and complex trade-offs across the multiple dimensions of computation, communication, storage, and the accuracy of results. Moreover, given the individually unpredictable nature of the computational nodes in these systems, we are faced with the challenge of securing fast and high-quality algorithmic results in the face of uncertainty. This, coupled with the high level of complexity and heterogeneity of the component hardware, introduces significant *delays* that represent a key bottleneck to attaining the promised speed-ups of these large systems.

Codes have begun to transform the evolution of large-scale distributed storage systems in modern data centers under the umbrella of regenerating and locally repairable codes for distributed storage [34, 89, 114, 117, 19, 85, 44, 81, 84, 48, 53, 83, 60, 95, 88, 108] , which are also having a major impact on industry [54, 101, 91, 93]. Further, as we saw from the previous chapter, the flexibility of codes even improves data retrieval performance.

We envision codes to play a similar transformational role in the other layers: the computational layer and the communication layer. In this chapter, we use coding theory and focus on improving the runtime performance of distributed computing. In Chapter 6, we focus on reducing the network overhead needed for shuffling input data, which is important for large-scale distributed machine learning algorithms.

In this chapter, we consider how one can design a *coded algorithm* for linear computation such as matrix multiplication, polynomial interpolation, zero-order optimization, and etc. For these linear computation, we show that if the number of distributed workers is n, and the runtime of each subtask has an exponential tail, the optimal coded algorithm is $\Theta(\log n)$ times faster than the uncoded algorithm. Further, we show that the optimal repetition algorithm is still $\Theta(\log n)$ slower than the optimal coded algorithm.

We run Open MPI experiments on Amazon EC2, which highlight significant gains offered by our proposed *coded* algorithms compared to uncoded ones or replication-based solutions. For instance, our preliminary results show that *coded* distributed algorithms can achieve significant speedups of up to 40% compared to *uncoded* distributed algorithms.

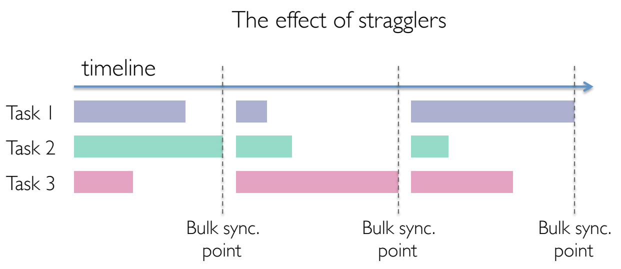


Figure 5.1: **The effects of slow nodes.** In distributed computation, the running time of a single distributed task is governed by that of the slowest node. In this toy figure, we see how slow nodes can significantly impact the running time of distributed computation. Can we use coding to alleviate the straggler's effects?

We would like to remark that a major innovation of our coding solutions in this chapter and Chapter 6 is that they are woven into the fabric of the algorithm design, and encoding/decoding is performed over the representation field of the input data (e.g., floats or doubles). In sharp contrast to most coding applications, we do not need to "re-factor code" and modify the distributed system to accommodate for our solutions; it is all done seamlessly in the algorithmic design layer, an abstraction that we believe is much more impactful as it is located "higher up" in the system layer hierarchy compared to traditional applications of coding that need to interact with the stored and transmitted "bits" (e.g., as is the case for coding solutions for the physical or storage layer).

5.2 Related Works

The straggler problem has been widely observed in distributed computing clusters. The authors of [29] show that running a computational task at a computing node often involves unpredictable latency due to several factors such as network latency, shared resources, maintenance activities, and power limits. Further, they argue that stragglers cannot be completely removed from a distributed computing cluster. The authors of [4] characterize the impacts and causes of stragglers that arise due to resource contention, disk failures, varying network conditions, and imbalanced workload.

One approach to mitigate the adverse effect of stragglers is based on efficient straggler detection algorithms. For instance, the default scheduler of Hadoop constantly detects stragglers while running computational tasks. Whenever it detects a straggler, it relaunches the task that was running on the detected straggler at some other available node. In [129], Zaharia et al. propose a modification to the existing straggler detection algorithm and show that the proposed solution can effectively reduce the completion time of MapReduce tasks. In [4], Ananthanarayanan et al. propose a system that efficiently detects stragglers using real-time progress and cancels those stragglers, and show that the proposed system can further reduce the runtime of MapReduce tasks.

Another line of promising approaches is based on appropriate modification of the algorithms. That is, one can design distributed algorithms that are robust to *asynchronous* or delayed updates from the workers. Such robust distributed algorithms can continuously make progress without needing to wait for all the responses from the workers, and hence the overall runtime of these algorithms is less affected by stragglers. For instance, the authors of [1] study the convergence of asynchronous stochastic gradient descent (SGD) algorithms, and show that their convergence rate is order-optimal. Moreover, in the single-node multi-core setup the authors in [99] propose HOGWILD! an asynchronous multicore implemnentation fo SGD, that runs without any memory locking and synchronization mechanisms between multiple threads. We would like to note that HOGWILD! and other asynchronous approaches do not in general guarantee “correctness" of the result, i.e., the output of the asynchronous algorithm can differ from that of a serial execution with an identical number of iterations; this may not be an issue in statistical problems, as the end solution will be noisy, but can become critically important if one wishes to have a high-precision solution, e.g., as is the case for exact matrix multiplication.

Recently, replication-based approaches have been explored to tackle the straggler problem: by replicating tasks and scheduling the replicas, the runtime of distributed algorithms can be significantly improved [3, 103, 124, 43, 20, 66]. By collecting outputs of the fast-responding nodes (and potentially canceling all the other slow-responding replicas), such replication-based scheduling algorithms can reduce latency. In [66], the authors show that even without replica cancellation, one can still reduce the average task latency by properly scheduling redundant requests. We view these policies as special instances of coded computation: such task replication schemes can be seen as *repetition-coded* computation. In Section 5.3, we describe this connection in detail, and indicate that coded computation can significantly outperform replication (as is usually the case for coding vs. replication in other engineering applications).

5.3 Coded Computation

In this section, we propose a novel paradigm to mitigate the straggler problem. The core idea is simple: *we introduce redundancy into subtasks of a distributed algorithm such that the original task's result can be decoded from a subset of the subtask results, treating uncompleted subtasks as* ***erasures***. For this specific purpose, we use erasure codes to design *coded* subtasks.

5.3.1 Erasure Codes

An erasure code is a method of introducing redundancy to a message that needs to be protected against noise such as erasures in telecommunication channels, packet drops in the routing layer of the Internet, and disk failures in data storage systems [27]. An erasure code *encodes* a message of k symbols into a longer message of n coded symbols such that the original k message symbols can be recovered by decoding a subset of coded symbols [26, 27].

An important class of erasure codes is the class of *repetition codes*. Given k message symbols, an $\frac{n}{k}$-repetition code simply repeats each symbol $\frac{n}{k}$ times. Thus, one can recover the original message as long as at least one of the $\frac{n}{k}$ repeated symbols is not erased for each of the k message symbols. Due to its simplicity, repetition codes have been widely used in many applications including modern distributed storage systems.

Another important class of codes is Maximum-Distance Separable (MDS) codes. A well-known example is the Reed-Solomon code used to protect CDs and DVDs. When a message is encoded using an (n, k) MDS code, any set of k out of the n encoded symbols is sufficient to recover the message of k symbols. As a concrete example, consider a message of two real numbers $\mathbf{m} = (m_A, m_B) \in \mathbb{R}^2$. Consider a code that transforms $\mathbf{m}$ into $\mathbf{c} = (m_A, m_B, m_A + m_B)$. Clearly, the original message $\mathbf{m}$ can be recovered with any $k = 2$ symbols of $\mathbf{c}$, so the code is an $(n = 3, k = 2)$ MDS code.

5.3.2 Coded Computation

We now formally define coded computation.

Definition 5.1 (Coded computation)**.** Consider a computational task $f_{\mathbf{A}}(\cdot)$. A *coded* distributed algorithm for computing $f_{\mathbf{A}}(\cdot)$ is specified by

- local functions $\langle f^i_{\mathbf{A}_i}(\cdot)\rangle_{i=1}^n$ and local data blocks $\langle \mathbf{A}_i \rangle_{i=1}^n$;
- (minimal) decodable sets of indices $\mathcal{I} \in \mathcal{P}([n])$ and a decoding function $\texttt{dec}(\cdot, \cdot)$,

where $[n] \overset{\text{def}}{=} \{1, 2, \dots, n\}$, and $\mathcal{P}(\cdot)$ is the powerset of a set. The decodable sets of indices $\mathcal{I}$ is minimal: no element of $\mathcal{I}$ is a subset of other elements. The decoding function takes a sequence of indices and a sequence of subtask results, and it must correctly output $f_{\mathbf{A}}(\mathbf{x})$ if any decodable set of indices and its corresponding results are given.

A coded distributed algorithm can be run in a distributed computing cluster as follows. Assume that the i^{th} (encoded) data block $\mathbf{A}_i$ is stored at the i^{th} worker for all i. Upon receiving the input argument $\mathbf{x}$, the master node multicasts $\mathbf{x}$ to all the workers, and then waits until it receives the responses from any of the decodable sets. Each worker node starts computing its local function when it receives its local input argument, and sends the task result to the master node. Once the master node receives the results from some decodable set, it decodes the received task results and obtains $f_{\mathbf{A}}(\mathbf{x})$. Algorithms 9 and 10 summarize the described protocols of the master node and the worker nodes.

Algorithm 9 Coded computation: master node's protocol

```
on Receiving an input argument $\mathbf{x}$
  Multicast $\mathbf{x}$ to all the workers.
  $\mathbf{i} = \langle\rangle$
  $\mathbf{y}_{list} = \langle\rangle$
  while $\mathbf{i} \notin \mathcal{I}$ do
    on Receiving a message $\mathbf{y}$ from worker $j$
      $\mathbf{i} \leftarrow \langle \mathbf{i}, j \rangle$
      $\mathbf{y}_{list} \leftarrow \langle \mathbf{y}_{list}, \mathbf{y} \rangle$
  end while
  $\mathbf{y} \leftarrow \texttt{dec}(\mathbf{i}, \mathbf{y}_{list})$
  Return $\mathbf{y}$
```

Algorithm 10 Coded computation: worker node i's protocol

```
on Receiving an input argument $\mathbf{x}$
  Compute $\mathbf{y}_i = f^i_{\mathbf{A}_i}(\mathbf{x})$
  Send $\mathbf{y}_i$ to the master node
```

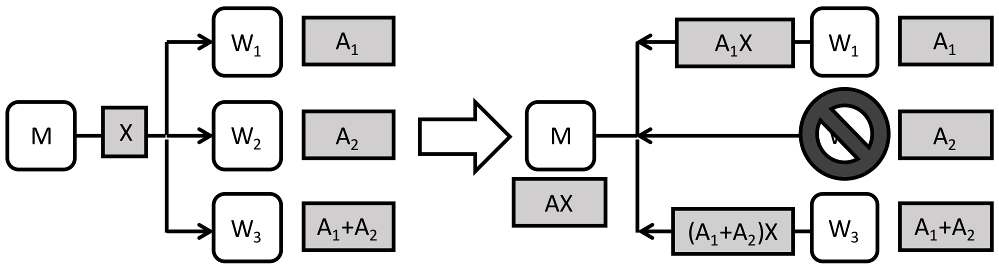


Figure 5.2: **An illustration of *Coded Matrix Multiplication*.** Data matrix $\mathbf{A}$ is partitioned into 2 submatrices: $\mathbf{A}_1$ and $\mathbf{A}_2$. Node W_1 stores $\mathbf{A}_1$, node W_2 stores $\mathbf{A}_2$, and node W_3 stores $\mathbf{A}_1 + \mathbf{A}_2$. Upon receiving $\mathbf{X}$, each node multiplies $\mathbf{X}$ with the stored matrix, and sends the product to the master node. Observe that the master node can always recover $\mathbf{AX}$ upon receiving *any* 2 products, without needing to wait for the slowest response.

The following toy example illustrates the main idea of *Coded Computation*.

Example 5.1. *Consider a system with three worker nodes and one master node, as depicted in Fig. 5.2. The goal is to compute a matrix multiplication $\mathbf{AX}$. The data matrix $\mathbf{A}$ is vertically divided into two (equally tall) submatrices $\mathbf{A}_1$ and $\mathbf{A}_2$, which are stored in node 1 and node 2, respectively. In node 3, we store the matrix sum of the two submatrices $\mathbf{A}_1 + \mathbf{A}_2$. After the master node transmits $\mathbf{X}$ to the worker nodes, each node computes the matrix multiplication of the stored matrix and the received matrix $\mathbf{X}$, and sends the computation result back to the master node. The master node can compute $\mathbf{AX}$ as soon as it receives* any *two computation results. For instance,* *consider a case where it collects* $\mathbf{A}_1\mathbf{X}$ *from node* 1 *and* $(\mathbf{A}_1 + \mathbf{A}_2)\mathbf{X}$ *from node* 3*. By subtracting* $\mathbf{A}_1\mathbf{X}$ *from* $(\mathbf{A}_1 + \mathbf{A}_2)\mathbf{X}$*, it can recover* $\mathbf{A}_2\mathbf{X}$ *and hence* $\mathbf{AX}$*, which is a vertical concatenation of* $\mathbf{A}_1\mathbf{X}$ *and* $\mathbf{A}_2\mathbf{X}$*.*

Coded Computation designs parallel tasks for a linear operation using erasure codes such that its runtime is not affected by up to a certain number of stragglers. Matrix multiplication is one of the most basic linear operations and is the workhorse of a host of machine learning and data analytics algorithms, e.g., gradient descent based algorithm for regression problems, power-iteration like algorithms for spectral analysis and graph ranking applications, etc. Hence, we focus on the example of matrix multiplication in this chapter.

The algorithm described in the previous example is a coded distributed algorithm for matrix multiplication that uses an $(n, n-1)$ MDS code. One can generalize the described algorithm using an (n, k) MDS code as follows. For any $1 \leq k \leq n$, the data matrix $\mathbf{A}$ is first divided into k (equally tall) submatrices [1]. Then, by applying an (n, k) MDS code to each element of the submatrices, n encoded submatrices are obtained. Upon receiving *any* k task results, the master node can use the decoding algorithm to decode k task results. Then, one can find $\mathbf{AX}$ simply by concatenating them.

5.3.3 Runtime of Uncoded/Coded Distributed Algorithms

In this section, we analyze runtime of uncoded and coded distributed algorithms. We first consider the overall runtime of an uncoded distributed algorithm, $T^{\text{uncoded}}_{\text{overall}}$. Assuming that the runtime of each task is identically distributed and independent of others, and denoting the runtime of the i^{th} worker by T^i,

$$T^{\text{uncoded}}_{\text{overall}} = T_{(n)} \overset{\text{def}}{=} \max\{T^1, T^2, \dots, T^n\}, \tag{5.1}$$

where $T_{(i)}$ is the i^{th} smallest one in $\{T^i\}_{i=1}^n$. From (5.1), it is clear that a single straggler can slow down the overall algorithm. A *coded* distributed algorithm is terminated whenever the master node receives results from any decodable set of workers. Thus, the overall runtime of a coded algorithm is *not* determined by the slowest worker, but by the first time to collect results from some decodable set in $\mathcal{I}$, i.e.,

$$T^{\text{coded}}_{\text{overall}} = T_{(\mathcal{I})} \overset{\text{def}}{=} \min_{\mathbf{i} \in \mathcal{I}} \max_{j \in \mathbf{i}} T_j \tag{5.2}$$

We remark that the runtime of uncoded distributed algorithms (5.1) is a special case of (5.2) with $\mathcal{I} = \{[n]\}$. In the following examples, we consider the runtime of the repetition-coded algorithms and the MDS-coded algorithms.

Example 5.2 (Repetition codes)**.** *Consider an* $\frac{n}{k}$*-repetition-code where each local task is replicated* $\frac{n}{k}$ *times. We assume that each group of* $\frac{n}{k}$ *consecutive workers work on the replicas of one local*

[1] If the number of rows of $\mathbf{A}$ is not a multiple of k, one can append zero rows to $\mathbf{A}$ to make the number of rows a multiple of k.

task. Thus, the decodable sets of indices are all the minimal sets that have k distinct task results, i.e., $\mathcal{I} = \prod_{i=1}^{k}\{(i-1)\frac{n}{k}+1, (i-1)\frac{n}{k}+2, \ldots, i\frac{n}{k}\}$. Thus,

$$T_{\textit{overall}}^{\textit{Repetition-coded}} = \max_{i\in[k]} \min_{j\in[\frac{n}{k}]} \{T^{(i-1)\frac{n}{k}+j}\}. \tag{5.3}$$

Example 5.3 (MDS codes)**.** *If one uses an (n,k) MDS code, the decodable sets of indices are the sets of* any *k indices, i.e., $\mathcal{I} = \{\mathbf{i} | \mathbf{i} \in [n],\ |\mathbf{i}| = k\}$. Thus,*

$$T_{\textit{overall}}^{\textit{MDS-coded}} = T_{(k)} \tag{5.4}$$

That is, the algorithm's runtime will be determined by the k^{th} response, not by the n^{th} response.

5.3.4 Probabilistic Model of Runtime

In this section, we analyze the runtime of uncoded/coded distributed algorithms assuming that task runtimes, including times to communicate inputs and outputs, are randomly distributed according to a certain distribution. For analytical purposes, we make a few assumptions as follows. We first assume the existence of the *mother runtime distribution* $F(t)$: we assume that running an algorithm using a *single* machine takes a random amount of time T_0, that is a positive-valued, continuous random variable parallelized according to F, i.e. $\Pr(T_0 \le t) = F(t)$. We also assume that T_0 has a probability density function $f(t)$. Then, when the algorithm is distributed into a certain number of subtasks, say ℓ, the runtime distribution of each of the ℓ subtasks is assumed to be a scaled distribution of the mother distribution, i.e., $\Pr(T^i \le t) = F(\ell t)$ for $1 \le i \le \ell$. Finally, the computing times of the k tasks are assumed to be independent of one another.

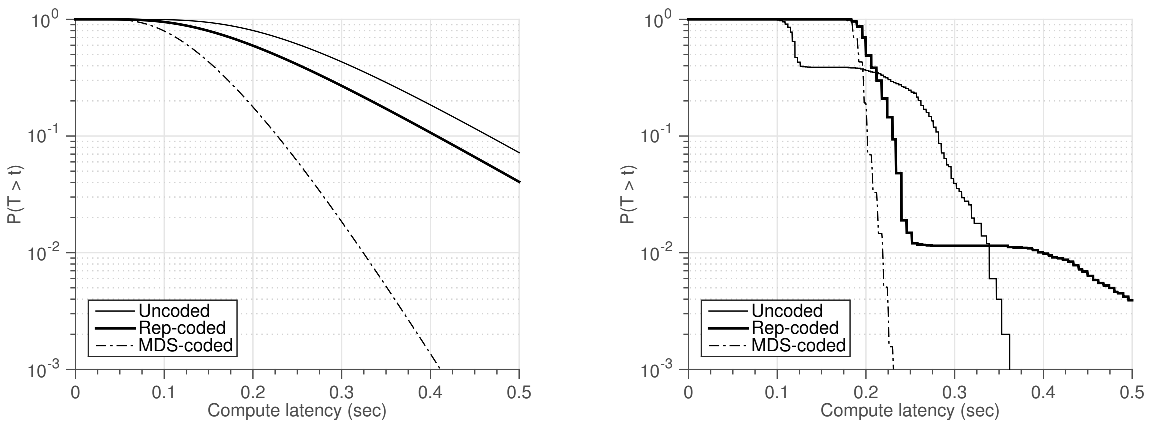
We first consider an uncoded distributed algorithm with n (uncoded) subtasks. Due to the assumptions mentioned above, the runtime of each subtask is $F(nt)$. Thus, the runtime distribution of an uncoded distributed algorithm, denoted by $F_{\text{overall}}^{\text{uncoded}}(t)$, is simply $[F(nt)]^n$.

When repetition codes or MDS codes are used, an algorithm is first divided into k $(< n)$ systematic subtasks, and then $n-k$ coded tasks are designed to provide an appropriate level of redundancy. Thus, the runtime of each task is distributed according to $F(kt)$. Using (5.3) and (5.4), one can easily find the runtime distribution of an $\frac{n}{k}$-repetition-coded distributed algorithm, $F_{\text{overall}}^{\text{Repetition}}$, and the runtime distribution of an (n,k)-MDS-coded distributed algorithm, $F_{\text{overall}}^{\text{MDS-coded}}$. For an $\frac{n}{k}$-repetition-coded distributed algorithm, one can first find the distribution of $\min_{j\in[\frac{n}{k}]}\{T^{(i-1)\frac{n}{k}+j}\}$, and then find the distribution of the maximum of k such terms:

$$F_{\text{overall}}^{\text{Repetition}}(t) = \left[1 - [1-F(kt)]^{\frac{n}{k}}\right]^k. \tag{5.5}$$

The runtime distribution of an (n,k)-MDS-coded distributed algorithm is simply the k^{th} order statistic:

$$F_{\text{overall}}^{\text{MDS-coded}}(t) = \int_{\tau=0}^{t} nkf(k\tau)\binom{n-1}{k-1}F(k\tau)^{k-1}\left[1-F(k\tau)\right]^{n-k} d\tau. \tag{5.6}$$



(a) Shifted-exponential distribution (b) Empirical distribution

Figure 5.3: **Runtime distributions of uncoded/coded distributed algorithms.** We plot the runtime distributions of uncoded/coded distributed algorithms. For the uncoded algorithms, we use $n = 10$, and for the coded algorithms, we use $n = 10$ and $k = 5$. In (a), we plot the runtime distribution when the runtime of tasks are distributed according to the shifted-exponential distribution. In (b), we use the empirical task runtime distribution measured on an Amazon EC2 cluster.

Remark 5.1. For the same value of n and k, the runtime distribution of a repetition-coded distributed algorithm strictly dominates that of an MDS-coded distributed algorithm. This can be shown by observing that the decodable sets of the MDS-coded algorithm is strictly larger than that of the repetition-coded algorithm.

In Fig. 5.3, we compare the runtime distributions of uncoded and coded distributed algorithms. We compare the runtime distributions of uncoded algorithm, repetition-coded algorithm, and MDS-coded algorithm with $n = 10$ and $k = 5$. For (a), we use a shifted-exponential distribution as the mother runtime distribution. That is, $F(t) = 1 - e^{t-1}$ for $t \geq 1$. For (b), we use the empirical task runtime distribution that is measured on an Amazon EC2 cluster [2]. Observe that for both cases, the runtime distribution of the MDS-coded distribution has the lightest tail.

5.3.5 Optimal Code Design for Coded Distributed Algorithms: the Shifted-exponential Case

When a coded distributed algorithm is used, the original task is divided into a fewer number of tasks at first compared to the case of uncoded algorithms. Thus, the runtime of each task of a coded algorithm, which is $F(kt)$, is stochastically larger than that of an uncoded algorithm, which is $F(nt)$. If the value that we choose for k is too small, then the runtime of each task becomes so large that the overall runtime of the distributed coded algorithm will eventually increase. If k is too

[2]The detailed description of the experiments is provided in Section 5.4.

large, the level of redundancy may not be sufficient to prevent the algorithm from being delayed by the stragglers.

Given the mother runtime distribution and the code parameters, one can easily compute the overall runtime distribution of the coded distributed algorithm using (5.5) and (5.6). Then, one can optimize the design based on various target metrics, e.g., the expected overall runtime, the 99th percentile runtime, etc.

In this section, we show how one can design an optimal coded algorithm that minimizes *the expected overall runtime* for a shifted-exponential mother distribution. The shifted-exponential distribution strikes a good balance between accuracy and analytical tractability. This model is motivated by the model proposed in [70]: the authors used this distribution to model latency of file queries from cloud storage systems. The shifted-exponential distribution is the sum of a constant and an exponential random variable, i.e.,

$$\Pr(T_0 \leq t) = 1 - e^{-\mu(t-1)}, \ \ \forall t \geq 1, \tag{5.7}$$

where the exponential rate μ is called the *straggling parameter*.

With this shifted-exponential model, we first find exact, closed-form expressions for the average runtime of uncoded/coded distributed algorithms. We assume that n is large, and k is linear in n. Accordingly, we approximate $H_n \overset{\text{def}}{=} \sum_{i=1}^{n} \frac{1}{i} \simeq \log n$ and $H_{n-k} \simeq \log{(n-k)}$. We first note that the expected value of the maximum of n independent exponential random variables with rate μ is $\frac{H_n}{\mu}$. Thus, the average runtime of an uncoded distributed algorithm is

$$\mathbb{E}[T_{\text{overall}}^{\text{uncoded}}] = \frac{1}{n}\left(1 + \frac{1}{\mu}\log n\right) = \Theta\left(\frac{\log n}{n}\right). \tag{5.8}$$

For the average runtime of an $\frac{n}{k}$-Repetition-coded distributed algorithm, we first note that the minimum of $\frac{n}{k}$ independent exponential random variables with rate μ is distributed as an exponential random variable with rate $\frac{n}{k}\mu$. Thus,

$$\mathbb{E}[T_{\text{overall}}^{\text{Repetition-coded}}] = \frac{1}{k}\left(1 + \frac{k}{n\mu}\log k\right) = \Theta\left(\frac{\log n}{n}\right). \tag{5.9}$$

Finally, we note that the expected value of the k^{th} statistic of n independent exponential random variables of rate μ is $\frac{H_n - H_{n-k}}{\mu}$. Therefore,

$$\mathbb{E}[T_{\text{overall}}^{\text{MDS-coded}}] = \frac{1}{k}\left(1 + \frac{1}{\mu}\log\left(\frac{n}{n-k}\right)\right) = \Theta\left(\frac{1}{n}\right). \tag{5.10}$$

Using these closed-form expressions of the average runtime, one can easily find the optimal value of k that achieves the optimal average runtime. The following lemma characterizes the optimal repetition code for the repetition-coded algorithms and their runtime performances.

Lemma 5.1 (Optimal repetition-coded distributed algorithms)**.** *If $\mu \geq 1$, the average runtime of an $\frac{n}{k}$-Repetition-coded distributed algorithm, in a distributed computing cluster with n workers, is* *minimized by* not *replicating tasks or setting $k = n$. If $\mu = \frac{1}{v}$ for some integer $v > 1$, the average runtime is minimized by setting $k = \mu n$, and the corresponding minimum average runtime is $\frac{1}{n\mu}(1 + \log(n\mu))$.*

Proof. It is easy to see that (5.9) as a function of k has a unique extreme point. By differentiating (5.9) with respect to k and equating it to zero, we have $k = \mu n$. Thus, if $\mu \geq 1$, one should set $k = n$; if $\mu = \frac{1}{v} < 1$ for some integer v, one should set $k = \mu n$. □

The above lemma reveals that the optimal repetition-coded distributed algorithm can achieve a lower average runtime than the uncoded distributed algorithm if $\mu < 1$; however, the optimal repetition-coded distributed algorithm still suffers from the factor of $\Theta(\log n)$, and cannot achieve the order-optimal performance. The following lemma, on the other hand, shows that the optimal MDS-coded distributed algorithm can achieve the order-optimal average runtime performance.

Lemma 5.2 (Optimal MDS-coded distributed algorithms)**.** *The average runtime of an (n, k)-MDS-coded distributed algorithm, in a distributed computing cluster with n workers, can be minimized by setting $k = k^\star$ where*

$$k^\star = \left[1 + \frac{1}{W_{-1}(-e^{-\mu-1})}\right] n \overset{\text{def}}{=} \alpha^\star(\mu) n, \tag{5.11}$$

and $W_{-1}(\cdot)$ is the lower branch of Lambert W function [3] *Thus,*

$$\min_k \mathbb{E}[T^{\text{MDS-coded}}_{\text{overall}}] = \frac{-W_{-1}(-e^{-\mu-1})}{\mu n} \overset{\text{def}}{=} \frac{\gamma^\star(\mu)}{n}. \tag{5.12}$$

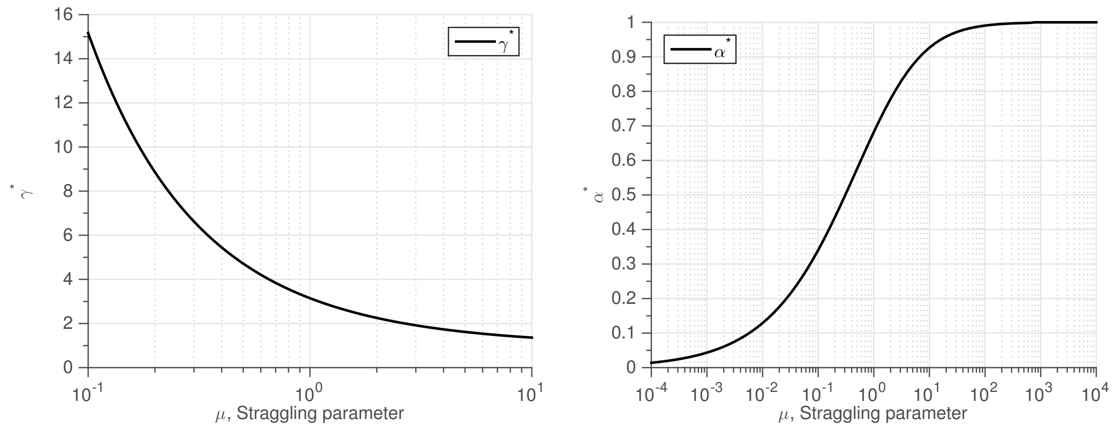
Proof. It is easy to see that (5.10) as a function of k has a unique extreme point. By differentiating (5.10) with respect to k and equating it to zero, we have $\frac{1}{k^\star}\left(1 + \frac{1}{\mu}\log\left(\frac{n}{n-k^\star}\right)\right) = \frac{1}{\mu}\frac{1}{n-k^\star}$. By setting $k = \alpha^\star n$, we have $\frac{1}{\alpha^\star}\left(1 + \frac{1}{\mu}\log\left(\frac{1}{1-\alpha^\star}\right)\right) = \frac{1}{\mu}\frac{1}{1-\alpha^\star}$, which implies $\mu + 1 = \frac{1}{1-\alpha^\star} - \log\left(\frac{1}{1-\alpha^\star}\right)$. By defining $\beta = \frac{1}{1-\alpha^\star}$ and exponentiating both the sides, we have $e^{\mu+1} = \frac{e^\beta}{\beta}$. Note that the solution of $\frac{e^x}{x} = t$, $t \geq e$ and $x \geq 1$ is $x = -W_{-1}(-\frac{1}{t})$. Thus, $\beta = -W_{-1}(-e^{-\mu-1})$. By plugging the above equation into the definition of β, the claim is proved. □

We plot $\gamma^*(\mu)$ and $\alpha^\star(\mu)$ in Fig. 5.4.

In addition to the order-optimality of MDS-coded distributed algorithms, the above lemma precisely characterizes the gap between the achievable runtime and the optimistic lower bound of $\frac{1}{n}$. For instance, when $\mu > 1$, the optimal average runtime is $\frac{\gamma^\star(\mu)}{n} \lesssim \frac{3.15}{n}$, which is only 3.15 away from the lower bound.

Remark 5.2 (Storage overhead). So far, we have considered only the runtime performance of distributed algorithms. Another important metric to be considered is the storage cost. When coded

[3] $W_{-1}(x)$, the lower branch of Lambert W function evaluated at x, is the unique solution of $te^t = x$ and $t \leq -1$.



(a) $\gamma^\star$ in Lemma 5.2 as a function of μ. (b) $\alpha^\star$ in Lemma 5.2 as a function of μ.

Figure 5.4: **$\gamma^\star$ and $\alpha^\star$ in Lemma 5.2.** As a function of the straggling parameter, we plot γ^* and α^*, which help quantify the runtime overhead of the straggler problem and design the optimal MDS-coded computation, respectively.

computation is being used, the storage overhead may increase. For instance, the MDS-coded distributed algorithm for matrix multiplication, described in Section 5.3.2, requires $\frac{1}{k}$ of the whole data to be stored at each worker, while the uncoded distributed algorithm requires $\frac{1}{n}$. Thus, the storage overhead factor is $\frac{\frac{1}{k}-\frac{1}{n}}{\frac{1}{n}} = \frac{n}{k} - 1$. If one uses the runtime-optimal MDS-coded distributed algorithm for matrix multiplication, the storage overhead is $\frac{n}{k^\star} - 1 = \frac{1}{\alpha^\star} - 1$.

5.3.6 Coded Gradient Descent: an MDS-coded Distributed Algorithm for Linear Regression

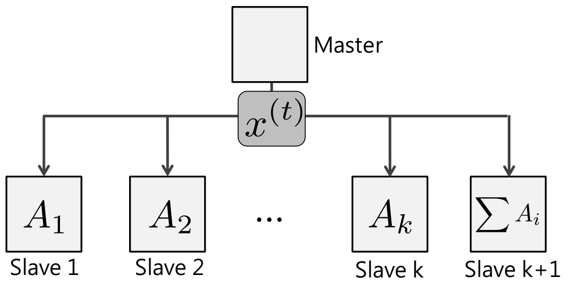
In this section, as a concrete application of coded matrix multiplication, we propose the *coded gradient descent* for solving large-scale linear regression problems.

We first describe the (uncoded) gradient-based distributed algorithm. Consider the following linear regression,

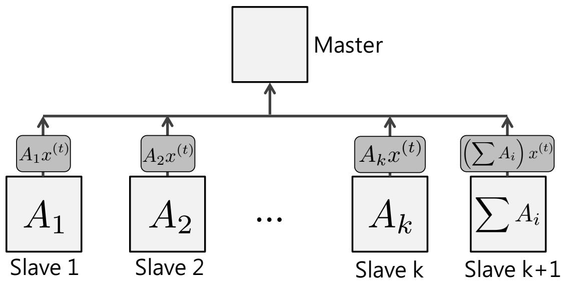
$$\min_{\mathbf{x}} f(\mathbf{x}) \overset{\text{def}}{=} \min_{x} \frac{1}{2}\|\mathbf{A}\mathbf{x} - \mathbf{y}\|_2^2, \tag{5.13}$$

where $\mathbf{y} \in \mathbb{R}^q$ is the label vector, $\mathbf{A} = [\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n]^T \in \mathbb{R}^{q\times r}$ is the data matrix, and $\mathbf{x} \in \mathbb{R}^r$ is the unknown weight vector to be found. We seek a distributed algorithm to solve this regression problem. Since $f(\mathbf{x})$ is convex in $\mathbf{x}$, the gradient-based distributed algorithm works as follows. We first compute the objective function's gradient: $\nabla f(\mathbf{x}) = \mathbf{A}^T(\mathbf{A}\mathbf{x} - \mathbf{y})$. Denoting by $\mathbf{x}^{(t)}$ the estimate of $\mathbf{x}$ after the t^{th} iteration, we iteratively update $\mathbf{x}^{(t)}$ according to the following equation.

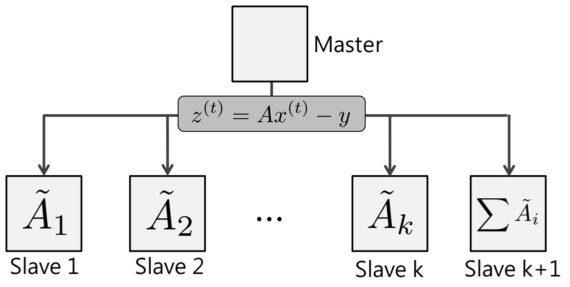
$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \gamma\nabla f(\mathbf{x}^{(t)}) = \mathbf{x}^{(t)} - \gamma\mathbf{A}^T(\mathbf{A}\mathbf{x}^{(t)} - \mathbf{y}) \tag{5.14}$$



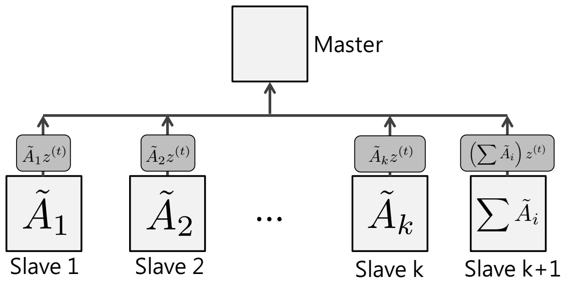
(a) In the beginning of the t^{th} iteration, the master node multicasts $\mathbf{x}^{(t)}$ to the worker nodes.



(b) The master node waits for the earliest responding k worker nodes, and computes $\mathbf{Ax}^{(t)}$.



(c) The master node computes $\mathbf{z}^{(t)}$ and multicasts it to the worker nodes.



(d) The master node waits for the k earliest responding worker nodes, and computes $\mathbf{A}^T\mathbf{z}^{(t)}$ or $\nabla f(\mathbf{x}^{(t)})$.

Figure 5.5: **An illustration of the coded gradient descent for linear regression.** The coded gradient descent computes a gradient of the objective function using *coded matrix multiplication* twice: in each iteration, it first computes $\mathbf{Ax}^{(t)}$ as depicted in (a) and (b), and then computes $\mathbf{A}^T(\mathbf{Ax}^{(t)} - \mathbf{y})$ as depicted in (c) and (d).

The above algorithm is guaranteed to converge to the optimal solution if we use a small enough step size γ [17], and can be easily distributed. We describe one simple way of parallelizing the algorithm, which is implemented in many open-source machine learning libraries including Spark `mllib` [79]. As $\mathbf{A}^T(\mathbf{Ax}^{(t)} - \mathbf{y}) = \sum_{i=1}^{q} \mathbf{a}_i(\mathbf{a}_i^T\mathbf{x}^{(t)} - \mathbf{y}_i)$, gradients can be computed in a distributed way by computing partial sums at different worker nodes and then adding all the partial sums at the master node. This distributed algorithm is an uncoded distributed algorithm: in each round, the master node needs to wait for all the task results in order to compute the gradient. Thus, the runtime of each update iteration is determined by the slowest response among all the worker nodes.

We now propose the *coded gradient descent*, a coded distributed algorithm for linear regression problems. Note that in each iteration, the following two matrix-vector multiplications are computed.

$$\mathbf{Ax}^{(t)}, \quad \mathbf{A}^T(\mathbf{Ax}^{(t)} - \mathbf{y}) \overset{\text{def}}{=} \mathbf{A}^T\mathbf{z}^{(t)} \tag{5.15}$$

In Section 5.3.2, we proposed the MDS-coded distributed algorithm for matrix multiplication.

Here, we apply the algorithm twice to compute these two multiplications in each iteration. More specifically, for the first matrix multiplication, we choose $1 \leq k_1 < n$ and use an (n, k_1)-MDS-coded distributed algorithm for matrix multiplication to encode the data matrix $\mathbf{A}$. Similarly for the second matrix multiplication, we choose $1 \leq k_2 < n$ and use a (n, k_2)-MDS-coded distributed algorithm to encode the *transpose* of the data matrix. Denoting the i^{th} row-split (column-split) of $\mathbf{A}$ as $\mathbf{A}_i$ ($\tilde{\mathbf{A}}_i$), the i^{th} worker stores both $\mathbf{A}_i$ and $\tilde{\mathbf{A}}_i$. In the beginning of each iteration, the master node multicasts $\mathbf{x}^{(t)}$ to the worker nodes, each of which computes the local matrix multiplication for $\mathbf{A}\mathbf{x}^{(t)}$ and sends the result to the master node. Upon receiving *any* k_1 task results, the master node can start decoding the result and obtain $\mathbf{z}^{(t)} = \mathbf{A}\mathbf{x}^{(t)}$. The master node now multicasts $\mathbf{z}^{(t)}$ to the workers, and the workers compute local matrix multiplication for $\mathbf{A}^T\mathbf{z}^{(t)}$. Finally, the master node can decode $\mathbf{A}^T\mathbf{z}^{(t)}$ as soon as it receives any k_2 task results, and can proceed to the next iteration. Fig. 5.5 illustrates the protocol with $k_1 = k_2 = n - 1$.

Remark 5.3 (Storage overhead of the coded gradient descent). The coded gradient descent requires each node to store a $(\frac{1}{k_1} + \frac{1}{k_2} - \frac{1}{k_1 k_2})$-fraction of the data matrix. As the minimum storage overhead per node is a $\frac{1}{n}$-fraction of the data matrix, the relative storage overhead of the coded gradient descent algorithm is at least about factor of 2, if $k_1 \simeq n$ and $k_2 \simeq n$.

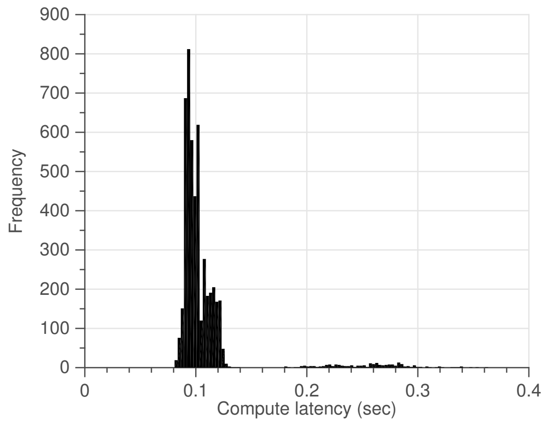
5.4 Coded Computation: Experiment Results

In order to see the efficacy of coded computation, we implement the proposed algorithms and test them on an Amazon EC2 cluster. In this section, we provide the experiment setups and the results.

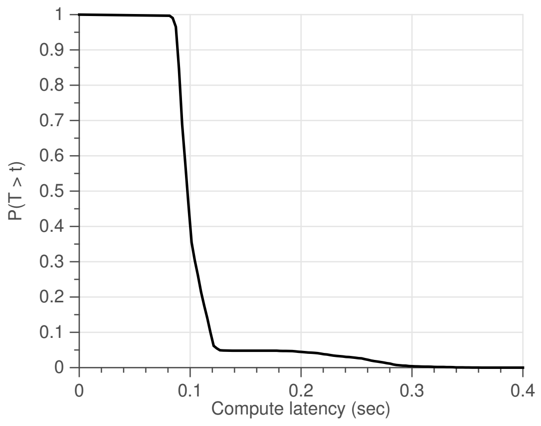
5.4.1 Task Runtime

We first obtain the empirical distribution of task runtime in order to observe how frequently stragglers appear in our testbed by measuring round-trip times between the master node and each of 10 worker instances on an Amazon EC2 cluster. Each worker computes a matrix-vector multiplication and passes the computation result to the master node, and the master node measures round trip times that include both computation time and communication time. Each worker repeats this procedure 500 times, and we obtain the empirical distribution of round trip times across all the worker nodes.

In Fig. 5.6, we plot the histogram and CCDF of measured computing times; the average round trip time is 0.11 second, and the 95th percentile latency is 0.20 second, i.e., roughly five out of hundred tasks are going to be roughly two times slower than the average tasks. Assuming the probability of a worker being a straggler is 5%, if one runs an uncoded distributed algorithm with 10 workers, the probability of not seeing such a straggler is only about 60%, so the algorithm is slowed down by a factor of more than 2 with probability 40%. Thus, this observation strongly emphasizes the necessity of an efficient straggler mitigation algorithm. In Fig. 5.3(a), we plot the runtime distributions of uncoded/coded distributed algorithms using this empirical distribution as the mother runtime distribution. When an uncoded distributed algorithm is used, the overall



(a) Histogram of the measured round trip times

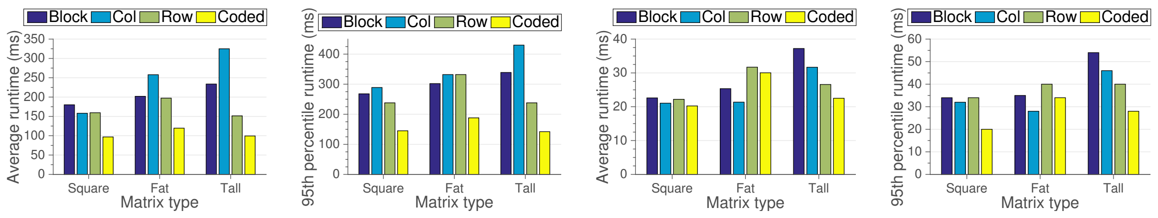


(b) Empirical CCDF of the measured round trip times

Figure 5.6: **Histogram and CCDF of the measured round trip times.** We measure round trip times between the master node and each of 10 worker nodes on an Amazon EC2 cluster. A round trip time consists of transmission time of the input vector from the master to a worker, computation time, and transmission time of the output vector from a worker to the master.

runtime distribution entails a heavy tail, while the runtime distribution of the MDS-coded algorithm has almost no tail.

5.4.2 Coded Matrix Multiplication



(a) m1-small, average runtime (b) m1-small, tail runtime (c) c1-medium, average runtime (d) c1-medium, tail runtime

Figure 5.7: **Comparison of parallel matrix multiplication algorithms.** We compare various parallel matrix multiplication algorithms: block, column-partition, row-partition, and coded (row-partition) matrix multiplication. We implement the four algorithms using OpenMPI and test them on Amazon EC2 cluster of 25 instances. We measure the average and the 95th percentile runtime of the algorithms. Plotted in (a) and (b) are the results with m1-small instances, and in (c) and (d) are the results with c1-medium instances.

The coded matrix multiplication is implemented in C++ using OpenMPI[82] and benchmarked on a cluster of 26 EC2 instances (25 workers and a master). We manage the cluster using the StarCluster toolkit [112]. Input data is generated using a Python script, and the input matrix was row-partitioned for each of the workers (with the required encoding as described in the previous sections) in a preprocessing step. The procedure begins by having all of the worker nodes read in their respective row-partitioned matrices. Then, the master node reads the input vector and distributes it to all worker nodes in the cluster through an asynchronous send (`MPI_Isend`). Upon receiving the input vector, each worker node begins matrix multiplication through a BLAS [14] routine call and once completed sends the result back to the master using `MPI_Send`. The master node waits for a sufficient number of results to be received by continuously polling (`MPI_Test`) to see if any results are obtained. The procedure ends when the master node decodes the overall result after receiving enough partial results. Similarly, three uncoded matrix multiplication algorithms – block, column-partition, and row-partition – are implemented and benchmarked.

We randomly draw a square matrix of size 5750×5750, a fat matrix of size 5750×11500, and a tall matrix of size 11500×5750, and multiply them with a column vector. For the coded matrix multiplication, we choose an $(25, 23)$ MDS code so that the runtime of the algorithm is not affected by any 2 stragglers. Fig. 5.7 shows that the coded matrix multiplication outperforms all the other parallel matrix multiplication algorithms in most cases. On a cluster of `m1-small` instances, compared to the best of the 3 uncoded matrix multiplication algorithms, the coded matrix multiplication achieves about 40% average runtime reduction and about 60% tail reduction. On a cluster of `c1-medium` instances, the coded algorithm achieves the best performance in most of the tested cases: the average runtime is reduced by at most 39.5%, and the 95th percentile runtime is reduced by at most 58.3%.

5.4.3 Coded Linear Regression

We evaluate the performance of the parallel gradient descent algorithms for linear regression. Since matrix multiplication is the core block of the parallel gradient descent for linear regression, the performance of the underlying parallel matrix multiplication algorithm significantly impacts the performance of the overall algorithm's performance.

The coded linear regression procedure is also implemented in C++ using OpenMPI, and benchmarked on a cluster of 11 EC2 machines (10 workers and a master). Like the matrix multiplication procedure, we generate the input data through a Python script. However, since both the transposed and untransposed versions of the input matrix are used in the linear regression algorithm, we duplicate the data and row-partition both versions of the input matrix (with the required encoding). The procedure begins by having all worker nodes load in their respective row-partitioned submatrices (both transposed and untransposed versions). Since each pass of linear regression consists of two matrix-vector multiplications, we split each pass into two iterations – an iteration for each matrix-vector multiplication (transposed multiply $\mathbf{A}\mathbf{x}^{(t)}$, untransposed multiply $\mathbf{A}^T\mathbf{z}^{(t)}$). Then, for every iteration of the procedure, the master node sends the appropriate input vector to each worker node in the cluster through an asynchronous send (MPI_Isend). The message channel through which the input vector is sent determines whether this particular multiplication operation should be a transposed multiplication or not. On the worker side, each worker continually listens for work by probing (Iprobe) for the input vector messages. Upon reception of the input vector, the worker computes the partial matrix multiplication through a call to the BLAS routine. When this completes, it sends the result back to the master node on the same channel from which the message was received. During this process, the master node waits for a sufficient number of partial results by polling for messages (IProbe). When enough partial results have been obtained, the master node decodes the result of the matrix vector multiplication. Then, the necessary computation to move to the next iteration is performed and the next iteration is begun. Similarly, the uncoded linear regression algorithm that is based on the uncoded row-partition matrix multiplication is implemented and benchmarked together with the coded linear regression.

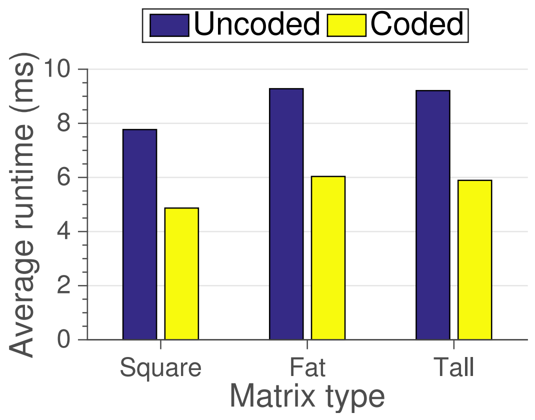
Similar to the previous benchmarks, we randomly draw a square matrix of size 2000×2000, a fat matrix of size 400×10000, and a tall matrix of size 10000×400, and use them as a data matrix. We use a $(10, 8)$-MDS code for the coded linear regression so that each multiplication of the gradient descent algorithm is not slowed down by up to 2 stragglers. Fig. 5.8 shows that the gradient algorithm with the *coded matrix multiplication* significantly outperforms the one with the uncoded matrix multiplication; the average runtime is reduced by 31.3% to 35.7%, and the tail runtime is reduced by 27.9% to 35.6%.

To thoroughly understand how the iterative algorithm with coded matrix multiplication efficiently handles stragglers, we measure the progress of each worker and the master node while running the gradient algorithm with coded matrix multiplication. The Gantt chart depicted in Fig. 5.8(c) visualizes the experiment results. We can observe that since the master node can complete an iteration as soon as it collects any $n-2=8$ workers, the master node can quickly proceed to the next iteration even with the existence of stragglers. This flexibility allows the overall algorithm to constantly progress without getting detained by ubiquitous stragglers, which significantly reduces the overall runtime of the algorithm.

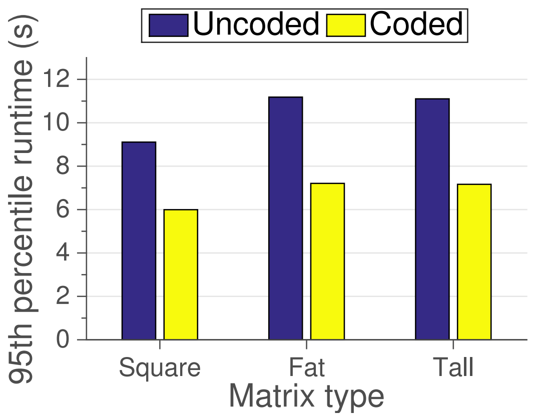
Remark 5.4 (Cancellation of stragglers). In the above experiments on the coded linear regression, we did *not* cancel (or kill) straggler workers. The overhead of cancellation of stragglers in our implementation is too high, so we choose not to cancel those stragglers. In an iterative algorithm such as the coded gradient descent, not canceling stragglers dilutes the gain of codes because only k workers will be available at the beginning of the following iteration. We believe that a delicate implementation of cancellation mechanism can even further improve the runtime performance of iterative coded distributed algorithms.

5.5 Conclusion

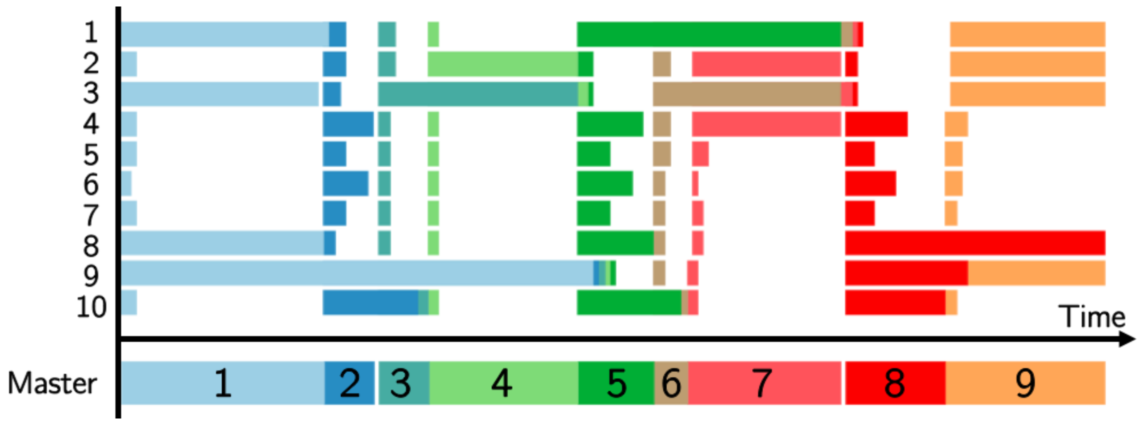
In this chapter, we have explored the power of coding in order to make distributed algorithms robust to stragglers. We propose a novel *Coded Computation* framework that can significantly speed up existing distributed algorithms, by cleverly introducing redundancy through codes into the computation. Our preliminary experiment results validate the power of our proposed scheme in effectively curtailing the negative effects of system bottlenecks, and attaining a significant speedups of up to 40%, compared to the current state-of-the-art methods.



(a) Average



(b) Tail



(c) Gantt chart of coded gradient algorithm with an $(10, 8)$-MDS code

Figure 5.8: **Comparison of parallel gradient algorithms.** We compare parallel gradient algorithms for linear regression problems, with different matrix multiplication algorithms: row-partition and coded (row-partition) matrix multiplication. We implement the two gradient algorithms using Open MPI, and test them on an Amazon EC2 cluster of 10 worker instances. We measure the average and the 95^{th} percentile runtime of the algorithms. Plotted in (a) and (b) are the measured results. We visualize via a Gantt chart how the coded gradient algorithm handles the stragglers in (c). In each row, we visualize which iteration each worker is in, and we visualize the master node's progress in the bottom row. With a $(10, 8)$-MDS coded matrix multiplication, the master node can proceed to the next iteration whenever the first 8 workers complete their local computations, treating the 2 stragglers as erasures.

Chapter 6

Data Shuffling with Codes

6.1 Introduction

We shift our focus from solving the straggler problem to solving the communication bottleneck problem. In this chapter, we focus on the problem of data-shuffling, propose the *Coded Shuffling* algorithm, and analyze its performance.

Consider a master-worker distributed setup, where the master node has access to the entire data-set. Before each *epoch* of the distributed algorithm, the master node shuffles the data points and sends each worker node some *coded functions* of the data, the worker nodes use the received information to decode and extract actual (shuffled) data points and train a local model; at the end of an epoch, the local models are averaged, and the process is repeated. See Fig. 6.1 for a toy illustration. We design a transmission strategy for the worker node, and caching and decoding strategies for the worker nodes that minimize the data communicated across all the shufflings performed. For completeness, we present the high-level description of a distributed machine learning proto-algorithm. Let $\mathbf{A} \in \mathbb{R}^{q \times r}$ be a data matrix. Consider an optimization problem that can be expressed as $\min_{\mathbf{x} \in \mathbb{R}^r} \ f(\mathbf{x}) = \sum_{i=1}^{q} \ell_i(\mathbf{a}_i, \mathbf{x})$, where $\mathbf{a}_i$ is the i^{th} data row of $\mathbf{A}$, and ℓ_i is a local function of the variable $\mathbf{x}$ and data point $\mathbf{a}_i$. The above problem can be solved by an iterative distributed algorithm that operates on rounds, where at each round each worker locally trains a model (variable) $\mathbf{x}_i$ of dimension r that is communicated back to the master. Upon receiving all the local models, the master averages them into a single model, and multicasts it back to the workers. More precisely, at iteration t of the algorithm, the data set is partitioned randomly into n subsets, say $\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_n$. Worker i computes a local update vector $\mathbf{x}_i = h(\mathbf{x}^t, \mathbf{A}_n)$, where $\mathbf{x}^t$ is the model (variable) at iteration t. The master node then aggregates the results by simply averaging the local updates. The algorithm continues by iterating $\mathbf{x}^{t+1} = \frac{1}{n} \sum_{k=1}^{n} h(\mathbf{x}^t, \mathbf{A}_k)$. A prototypical exemplar of the described parallel learning proto-algorithm is the parallel stochastic gradient descent [132].

6.1.1 Coded Shuffling Algorithm

Before we rigorously explain the technical details of the coded shuffling algorithm and our preliminary results, we illustrate the main idea of the coded shuffling algorithm with a toy example,

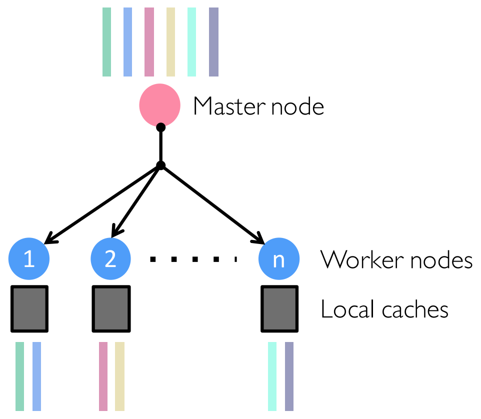


Figure 6.1: **Distributed setup.** We consider a master worker setup, where the master node communicates data points (or coded functions of them), and the worker nodes can store a limited number of data points. The topology of the network (be it tree-like, mesh, or over a shared bus) allows for a degree of multicasting gains. That is, we assume that multicasting the same information to all users is “cheaper" than sending them individual messages.

shown in Fig. 6.2.

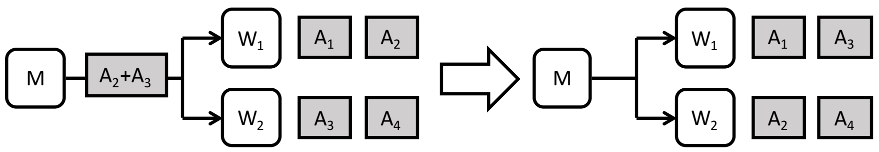


Figure 6.2: **An illustration of *Coded Shuffling*.** The data matrix $\mathbf{A} \in \mathbb{R}^{q\times r}$ is partitioned into 4 submatrices: $\mathbf{A}_1 \in \mathbb{R}^{q/4\times r}$ to $\mathbf{A}_4 \in \mathbb{R}^{q/4\times r}$. Before shuffling, worker 1 has cached $\mathbf{A}_1$ and $\mathbf{A}_2$ (from a previous iteration of the algorithm), and worker 2 has cached $\mathbf{A}_3$ and $\mathbf{A}_4$. Assuming that a new shuffling requires node 1 to receive $\mathbf{A}_3$ and node 2 to receive $\mathbf{A}_2$, the master node can send $\mathbf{A}_2 + \mathbf{A}_3$ (the addition is over the representation field of $\mathbf{A}$, not over bits) in order to shuffle the data stored at the two workers. Observe that in this case, and by using the cached information, the amount of communication (assuming full multicast gain over unicasting) is 50%.

Consider a system with two worker nodes and one master node. Assume that the master node holds the entire data set $\mathbf{A}$. For clarity of exposition of this example, assume that the data is equipartitioned into 4 batches $\mathbf{A}_1, \ldots, \mathbf{A}_4$. Assume that worker 1 already has $\mathbf{A}_1$ and $\mathbf{A}_2$ cached locally, and worker 2 has $\mathbf{A}_3$ and $\mathbf{A}_4$ cached locally. To shuffle the data, the master node's objective is to transmit $\mathbf{A}_3$ to worker 1 and $\mathbf{A}_4$ to worker 2, so that the local functions are now computed on the data points of $(\mathbf{A}_1, \mathbf{A}_3)$ and $(\mathbf{A}_2, \mathbf{A}_4)$ by worker nodes 1 and 2, respectively. For this purpose, the master node can simply multicast a *coded* message $\mathbf{A}_2 + \mathbf{A}_3$ to the worker nodes. Since node 1 has access to $\mathbf{A}_2$, it can subtract $\mathbf{A}_2$ from the received message $\mathbf{A}_2 + \mathbf{A}_3$, and replace $\mathbf{A}_2$ with $\mathbf{A}_3$. Similarly, node 2 can replace $\mathbf{A}_3$ with $\mathbf{A}_2$. Compared to the naïve (or uncoded) shuffling scheme in which the master node transmits $\mathbf{A}_2$ and $\mathbf{A}_3$ separately, this new shuffling scheme can save 50% of the communication cost, speeding up the overall run-time of the distributed machine learning algorithm. This is true assuming that *multicasting* a message to all workers is significantly cheaper than sending individual messages to each worker; that is, assuming that $\gamma(n) = n$, where $\gamma(n)$ is the advantage of using multicasting over unicasting:

$$\gamma(n) = \frac{\text{cost of unicasting } n \text{ separate messages to } n \text{ workers}}{\text{cost of multicasting a common message to } n \text{ workers}}.$$

In this section, our theoretical results assume that $\gamma(n) = n$. For general cases, see Remark 6.1.

General Coded Shuffling Scheme To formally describe the coded shuffling algorithm, we define some notation. Let $\mathbf{A}(\mathcal{J}) \in \mathbb{R}^{|\mathcal{J}|\times r}$, $\mathcal{J} \subset [q]$ be the concatenation of $|\mathcal{J}|$ rows of matrix $\mathbf{A}$ with indices in $\mathcal{J}$. Assume that each worker node has a cache of size s data rows (or $s \times r$ real numbers). In order to be able to fully store the data matrix across the worker nodes, we impose the inequality condition $q/n \leq s$. Further, clearly if $s > q$, the data matrix can be fully stored at each worker node, eliminating the need for any shuffling. Thus, without loss of generality we assume that $s \leq q$. As explained earlier working on the same data points at each worker node in all the iterations of the iterative optimization algorithm leads to slow convergence. Thus, to enhance the statistical efficiency of the algorithm, the data matrix is shuffled after each iteration. More precisely, at each iteration, the set of data rows $[q]$ is partitioned uniformly at random into n subsets S_i, $1 \leq i \leq n$ so that $\cup_{i=1}^{n} S_i = [q]$ and $S_i \cap S_j = \emptyset$ when $i \neq j$; thus, each worker node computes a fresh local function of the data. Clearly, the data set that worker i works on has cardinality q/n, i.e., $|S_i| = q/n$. We refer to each of these subsets as a *mini-batch* of the dataset. Note that the sampling we consider here is *without replacement*, and hence the mini-batches are non-overlapping.

We now provide details of the coded shuffling algorithm after each iteration of the parallel machine learning algorithm. Let C_i^t be the cache content of node i (set of row indices stored in cache i) at the end of iteration t. We design a transmission algorithm (by the master node) and a cache update algorithm to ensure that (i) $S_i^t \subset C_i^t$; and (ii) $C_i^t \setminus S_i^t$ is distributed uniformly at random without replacement in the set $[q] \setminus S_i^t$. The first condition ensures that at each iteration, the workers have access to the data set that they are supposed to work on. The second condition provides the opportunity of effective coded transmissions for shuffling in the next iteration as will be explained later.

After iteration t, the master node aggregates the local functions by averaging them, draws a random permutation on $[q]$, say π^t, to find S_i^{t+1}, and transmits a message $m(t+1)$ (i.e., a number of "coded" data-points) such that S_i^{t+1} can be recovered by worker node i from the current cache content C_i^t and the transmitted message $m(t+1)$. The cache content is then updated according to the following rule: the new cache will contain the subset of the data points used in the current iteration (this is need for the local computation), plus a random subset of the previous cached contents. More specifically, q/n rows of the new cache are precisely the rows in S_i^{t+1}, and $s - q/n$ rows of the cache are sampled points from the set $C_i^t \setminus S_i^{t+1}$, uniformly at random without replacement. Since the permutation π^t is picked uniformly at random, the marginal distribution of the cache contents at iteration $t+1$ given $S_i^{t+1},\ 1 \leq i \leq n$ is described as follows: $S_i^{t+1} \subset C_i^{t+1}$ and $C_i^{t+1} \setminus S_i^{t+1}$ is distributed uniformly at random in $[q] \setminus S_i^{t+1}$ without replacement.

Example 6.1. *To further clarify the algorithm, we revisit our toy example again based on this notation. Let $n = 2$ and $s = q/2$. Each worker node has half of the data stored in its cache. Consider an iteration of the algorithm, where a new permutation of the data rows is drawn by the master node resulting in subsets S_1 and S_2. As q gets large, half of the mini-batch S_1 (that should be processed by worker node 1) is stored in cache 1, i.e. $|S_1 \cap C_1| = q/4$. Similarly, $|S_2 \cap C_2| = q/4$. Thus, without the ability to exploit the power of coding, the master node needs to transmit a total of $q/4 + q/4 = q/2$ data rows to the computing nodes that are rows corresponding to $S_1 \cap \bar{C}_1$ and $S_2 \cap \bar{C}_2$. By contrast, in order to exploit the power of coding, the key observation is that the data rows that are in S_1 but not stored in C_1, are stored in C_2 (since $S_1 \cup S_2 = [q]$), and vice versa. Thus, the master node can code by sending only the* sum *of the rows indexed by $S_1 \cap C_2$ and $S_2 \cap C_1$, denoted by $\mathbf{A}(S_1 \cap C_2) + \mathbf{A}(S_2 \cap C_1)$. The transmission rate is thus $q/4$, which leads to a factor of 2 reduced communication cost. For decoding, since computing node 1 has access to $\mathbf{A}(S_2 \cap C_1)$, it can recover $\mathbf{A}(S_1 \cap C_2)$. Similarly, node 2 can recover $\mathbf{A}(S_2 \cap C_1)$.*

We now formally describe two methods for transmitting the message $m(t)$: (1) uncoded transmission and (2) coded transmission.

Uncoded Transmission Consider the cache content $C_i^t,\ 1 \leq i \leq n$, and the new data rows required by each worker, $S_i^{t+1},\ 1 \leq i \leq n$. In the following description, without loss of generality, we drop the iteration index t (and $t+1$) for the ease of notation. We find how many data rows in S_i are already cached in C_i, i.e. we find $|C_i \cap S_i|$. Since, the new permutation (partitioning) is picked uniformly at random, s/q fraction of the data row indices in S_i are cached in C_i, so as q gets large, we have $|C_i \cap S_i| = \frac{q}{n}(1 - s/q)$. Thus, without coding, the master node needs to transmit $\frac{q}{n}(1 - s/q)$ data points to each of the n worker nodes. The total communication rate (in data points transmitted per iteration) of the uncoded scheme is then

$$R_u = n \times \frac{q}{n}(1 - s/q) = q(1 - s/q). \tag{6.1}$$

Coded Transmission The delivery algorithm of the coded transmission is as follows. Define the set of "exclusive" cache content as $\tilde{C}_{\mathcal{I}} = (\cap_{i \in \mathcal{I}} C_i) \cap \left(\cap_{i' \in [n] \setminus \mathcal{I}} \bar{C}_{i'}\right)$ that denotes the set of rows that are stored at the caches of $\mathcal{I}$, and are *not* stored at the caches of $[n] \setminus \mathcal{I}$. For each subset $\mathcal{I}$ with $|\mathcal{I}| \geq 2$, the master node will multicast $\sum_{i \in \mathcal{I}} \mathbf{A}(S_i \cap \tilde{C}_{\mathcal{I} \setminus \{i\}})$ to the worker nodes. This is the summation (over i) of the data points that should be processed by worker i, and these data points are not stored in the cache of worker i, but are instead stored in the caches of every other worker in set $\mathcal{I}$. Thus, this message enables simultaneous decoding of some missing data points for all the workers in $\mathcal{I}$, by exploiting this multicast coding opportunity. This completes the description of the coded transmission algorithm. [1] A key novelty of our scheme is that the coding is performed over the representation field of the data matrix $\mathbf{A}$, and not over bits. The following example further illustrates the explained procedure.

Example 6.2. *Let $n = 3$. Recall that worker node i needs to obtain $\mathbf{A}(S_i \cap \bar{C}_i)$ for the next iteration of the algorithm. Consider $i = 1$. The data rows in $S_1 \cap \bar{C}_1$ are stored either exclusively in C_2 or C_3 (i.e. $\tilde{C}_2$ or $\tilde{C}_3$), or stored in both C_2 and C_3 (i.e. $\tilde{C}_{2,3}$). The transmitted message consists of 4 parts:*

- *(Part 1) $M_{\{1,2\}} = \mathbf{A}(S_1 \cap \tilde{C}_2) + \mathbf{A}(S_2 \cap \tilde{C}_1)$,*
- *(Part 2) $M_{\{1,3\}} = \mathbf{A}(S_1 \cap \tilde{C}_3) + \mathbf{A}(S_3 \cap \tilde{C}_1)$,*
- *(Part 3) $M_{\{2,3\}} = \mathbf{A}(S_2 \cap \tilde{C}_3) + \mathbf{A}(S_3 \cap \tilde{C}_2)$, and*
- *(Part 4) $M_{\{1,2,3\}} = \mathbf{A}(S_1 \cap \tilde{C}_{2,3}) + \mathbf{A}(S_2 \cap \tilde{C}_{1,3}) + \mathbf{A}(S_3 \cap \tilde{C}_{1,2})$.*

We show that worker node 1 can recover its unstored data rows, $\mathbf{A}(S_1 \cap \bar{C}_1)$. First, observe that node 1 stores $S_2 \cap \tilde{C}_1$. Thus, it can recover $\mathbf{A}(S_1 \cap \tilde{C}_2)$ using part 1 of the message since $\mathbf{A}(S_1 \cap \tilde{C}_2) = M_1 - \mathbf{A}(S_2 \cap \tilde{C}_1)$. Similarly, node 1 recovers $\mathbf{A}(S_1 \cap \tilde{C}_3) = M_2 - \mathbf{A}(S_3 \cap \tilde{C}_1)$. Finally, from part 4 of the message, node 1 recovers $\mathbf{A}(S_1 \cap \tilde{C}_{2,3}) = M_4 - \mathbf{A}(S_2 \cap \tilde{C}_{1,3}) - \mathbf{A}(S_3 \cap \tilde{C}_{1,2})$.

6.2 Related Works

Distributed learning algorithms on large-scale networked systems have been extensively studied in the literature [11, 80, 18, 8, 35, 21, 31, 74, 63, 111, 67]. Many of the distributed algorithms that are implemented in practice share a similar algorithmic “anatomy": the data set is split among several cores or nodes, each node trains a model locally, then the local models are averaged, and the process is repeated. While training a model with parallel or distributed learning algorithms, it is common to randomly re-shuffle the data a number of times [98, 99, 16, 130, 45, 56]. This essentially means that after each shuffling the learning algorithm will go over the data in a different order than before. Although the effects of random shuffling are far from understood theoretically, the large statistical gains have turned it into a common practice. Intuitively, data shuffling before a new pass over the data, implies that nodes get a nearly “fresh" sample from the data set, which experimentally leads to better statistical performance. Moreover, bad orderings of the data—known to lead to slow convergence in the worst case [98, 45, 56]—are “averaged out". However, the statistical benefits of data shuffling do not come for free: each time a new shuffle is performed, the *entire* dataset is communicated over the network of nodes. This inevitably leads to performance bottlenecks due to heavy communication.

[1]The master node also needs to transmit metadata associated with each message. Note that the size of such metadata is negligible compared to the size of coded messages.

In this chapter, we propose to use coding opportunities to significantly reduce the communication cost of some distributed learning algorithms that require data shuffling. Our coded shuffling algorithm is build upon the coded caching scheme by Maddah-Ali and Niesen [77]. Coded caching is a technique to reduce the communication rate in content delivery networks. Mainly motivated by video sharing applications, coded caching exploits the multicasting opportunities between users that request different video files to significantly reduce the communication burden of the server node that has access to the files. Coded caching has been studied in many scenarios such as decentralized coded caching [78], online coded caching [86], hierarchical coded caching for wireless communication [61], and device-to-device coded caching [57]. Recently, [68] proposed coded MapReduce that reduces the communication cost in the process of transferring the results of mappers to reducers. Our proposed approach is significantly different from all related studies on coded caching in two ways: (i) we shuffle the *data points* among the computing nodes to *increase the statistical efficiency* of distributed computation and ML algorithms; and (ii) we *code the data over their actual representation* (i.e., over the doubles or floats) unlike the traditional coding schemes over bits. In the remainder of this chapter, we describe how coded shuffling can remarkably speed up the communication phase of large-scale parallel machine learning algorithms, and provide extensive numerical experiments to validate our results.

We would also like to remark that there has been significant work in communication avoiding algorithms in the context of parallel numerical analysis and linear algebra [12, 72, 33, 32]. In contrast to this line of work, we propose the use of coding opportunities to strike a balance between statistical efficiency due to shuffling and the cost of communicating data points across different data passes.

6.3 Main Results

We now present the main result of this section, which characterizes the communication rate of the coded scheme. Let $p = \frac{s-q/n}{q-q/n}$.

Theorem 6.1 (Coded Shuffling Rate)**.** *Coded shuffling achieves communication rate*

$$R_c = \frac{q}{(np)^2}\left((1-p)^{n+1} + (n-1)p(1-p) - (1-p)^2\right) \tag{6.2}$$

(in number of data rows transmitted per iteration from the master node), which is significantly smaller than R_u in (6.1).

The reduction in communication rate is illustrated in Fig. 6.3 for $n = 50$ and $q = 1000$ as a function of s/q, where $1/n \le s/q \le 1$. For instance, when $s/q = 0.1$, the communication overhead for data-shuffling is reduced by more than 81%. Thus, at a very low storage overhead for caching, the algorithm can be significantly accelerated. We now prove Theorem 6.1.

Proof. To find the transmission rate of the coded scheme we first need to find the cardinality of sets $S_i^{t+1} \cap \tilde{C}_{\mathcal{I}}^t$ for $\mathcal{I} \subset [n]$ and $i \notin \mathcal{I}$. To this end, we first find the probability that a random

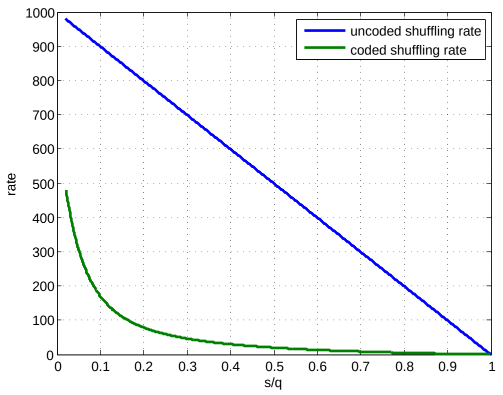


Figure 6.3: **The achievable rates of coded and uncoded shuffling schemes.** This figure shows the achievable rates of coded and uncoded schemes versus the cache size for parallel stochastic gradient descent algorithm.

data row, $\mathbf{r}$, belongs to $\tilde{C}_{\mathcal{I}}^t$. Denote this probability by $\Pr(\mathbf{r} \in \tilde{C}_{\mathcal{I}}^t)$. Recall that the cache content distribution at iteration t: q/n rows of cache j are stored with S_j^t and the other $s - q/n$ rows are stored uniformly at random. Thus, we can compute $\Pr(\mathbf{r} \in \tilde{C}_{\mathcal{I}}^t)$ as follows.

$$\Pr(\mathbf{r} \in \tilde{C}_{\mathcal{I}}^t) = \sum_{i=1}^{n} \Pr(\mathbf{r} \in \tilde{C}_{\mathcal{I}}^t | \mathbf{r} \in S_i^t) \Pr(\mathbf{r} \in S_i^t) \tag{6.3}$$

$$= \sum_{i=1}^{n} \frac{1}{n} \Pr(\mathbf{r} \in \tilde{C}_{\mathcal{I}}^t | \mathbf{r} \in S_i^t) \tag{6.4}$$

$$= \sum_{i \in \mathcal{I}} \frac{1}{n} \Pr(\mathbf{r} \in \tilde{C}_{\mathcal{I}}^t | \mathbf{r} \in S_i^t) \tag{6.5}$$

$$= \sum_{i \in \mathcal{I}} \frac{1}{n} \left(\frac{s - q/n}{q - q/n} \right)^{|\mathcal{I}|-1} \left(1 - \frac{s - q/n}{q - q/n} \right)^{n-|\mathcal{I}|} \tag{6.6}$$

$$= \frac{|\mathcal{I}|}{n} p^{|\mathcal{I}|-1} (1-p)^{n-|\mathcal{I}|}. \tag{6.7}$$

(6.3) is by the law of total probability. (6.4) is by the fact that $\mathbf{r}$ is chosen randomly. To see (6.5), note that $\Pr(\mathbf{r} \in \tilde{C}_{\mathcal{I}}^t | \mathbf{r} \in S_i^t, i \notin \mathcal{I}) = 0$. Thus, the summation can be written only on the indices of $\mathcal{I}$. We now explain (6.6). Given that $\mathbf{r}$ belongs to S_i^t, and $i \in \mathcal{I}$, then $\mathbf{r} \in C_i$ with probability 1. The other $|\mathcal{I}| - 1$ caches with indices in $\mathcal{I} \setminus \{i\}$ contain $\mathbf{r}$ with probability $\frac{s-q/n}{q-q/n}$ independently. Further, the caches with indices in $[n] \setminus \mathcal{I}$ do not contain $\mathbf{r}$ with probability $1 - \frac{s-q/n}{q-q/n}$. By defining $p \overset{\text{def}}{=} \frac{s-q/n}{q-q/n}$, we have (6.7).

We now find the cardinality of $S_i^{t+1} \cap \tilde{C}_{\mathcal{I}}^t$ for $\mathcal{I} \subset [n]$ and $i \notin \mathcal{I}$. Note that $|S_i^{t+1}| = q/n$. Thus, as q gets large (and n remains sub-linear in q), by the law of large numbers,

$$|S_i^{t+1} \cap \tilde{C}_{\mathcal{I}}^t| \simeq \frac{q}{n} \times \frac{|\mathcal{I}|}{n} p^{|\mathcal{I}|-1}(1-p)^{n-|\mathcal{I}|}.$$

Recall that for each subset $\mathcal{I}$ with $|\mathcal{I}| \geq 2$, the master node will send $\sum_{i\in\mathcal{I}} \mathbf{A}(S_i \cap \tilde{C}_{\mathcal{I}\setminus\{i\}})$. Thus, the total rate of coded transmission is

$$R_c = \sum_{i=2}^{n} \binom{n}{i} \frac{q}{n} \frac{i-1}{n} p^{i-2}(1-p)^{n-(i-1)}. \tag{6.8}$$

To complete the proof, we simplify the above expression. Let $x = \frac{p}{1-p}$. Taking derivative with respect to x from both sides of the equality $\sum_{i=1}^{n} \binom{n}{i} x^{i-1} = \frac{1}{x}\left[(1+x)^n - 1\right]$, we have

$$\sum_{i=2}^{n} \binom{n}{i} (i-1)x^{i-2} = \frac{1+(1+x)^{n-1}(nx-x-1)}{x^2}. \tag{6.9}$$

Using (6.9) in (6.8) completes the proof. □

Corollary 6.2. *Consider the case that the cache sizes are just enough to store the data required for processing; that is $s = q/n$. Then, $R_c = \frac{1}{2}R_u$. Thus, one gets a factor 2 reduction gain in communication rate by exploiting coded caching.*

Note that when $s = q/n$, $p = 0$. Finding the limit $\lim_{p\to 0} R_c$ in (6.2), after some manipulations, one calculates

$$R_c = q\left(1 - \frac{s}{q}\right)\frac{1}{1+ns/q} = R_u/2, \tag{6.10}$$

which shows Corollary 6.2.

Corollary 6.3. *Consider the regime of interest where n, s, and q get large, and $s/q \to c > 0$ and $n/q \to 0$. Then,*

$$R_c \to q\left(1 - \frac{s}{q}\right)\frac{1}{ns/q} = \frac{R_u}{ns/q} \tag{6.11}$$

Thus, using coding, the communication rate is reduced by $\Theta(n)$.

Remark 6.1. **(The advantage of using multicasting over unicasting)** It is clearly true that $\gamma(n) \simeq n$ for wireless architecture that is of great interest with the emergence of wireless data centers, e.g. [46, 131], and mobile computing platforms [6]. However, still in many applications, the network topology is based on point-to-point communication, and the multicasting opportunity is not fully available, i.e., $\gamma(n) < n$. For these general cases, we have to renormalize the communication cost of coded shuffling since we have assumed that $\gamma(n) = n$ in our results. For instance, in the regime

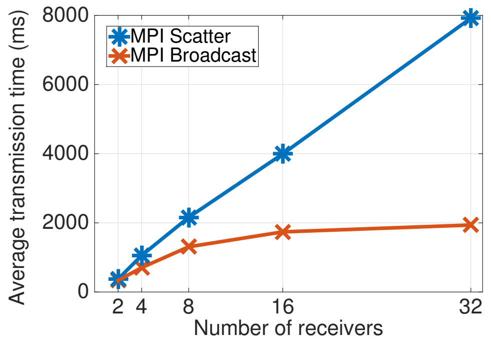


Figure 6.4: **Gains of multicasting over unicasting in distributed systems.** We measure the time taken for a data block of size of 4.15 MB to be transmitted to a targeted number of workers on an Amazon EC2 cluster, and compare the average transmission time taken with Message Passing Interface (MPI) scatter (unicast) and that with MPI broadcast. Observe that the average transmission time increases linearly as the number of receivers increases, but with MPI broadcast, the average transmission time increases logarithmically.

considered in Corollary 6.3, the renormalized communication cost of coded shuffling R_c^γ given $\gamma(n)$ is

$$R_c^\gamma = \frac{n}{\gamma(n)} R_c \to \frac{R_u}{\gamma(n)s/q}. \tag{6.12}$$

Thus, the communication cost of coded shuffling is smaller than uncoded shuffling if $\gamma(n) > q/s$. Note that s/q is the fraction of the data matrix that can be stored at each worker's cache. Thus, in the regime of interest where s/q is a constant independent of n, and $\gamma(n)$ scales with n, the reduction gain of coded shuffling in communication cost is still unbounded and increasing in n.

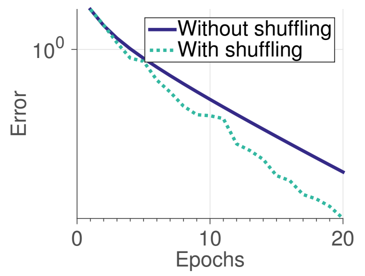
We emphasize that even in point-to-point communication networks, *multicasting the same message to multiple nodes is still significantly faster than unicasting different message (of the same size) to multiple nodes*, i.e., $\gamma(n) \gg 1$, justifying the advantage of using coded shuffling. For instance, the MPI broadcast API (`MPI_Bcast`) utilizes a tree multicast algorithm, which achieves $\gamma(n) = \Theta\left(\frac{n}{\log n}\right)$. Shown in Fig. 6.4 is the time taken for a data block to be transmitted to an increasing number of workers on an Amazon EC2 cluster, which consists of a point-to-point communication network. We compare the average transmission time taken with MPI scatter (unicast) and that with MPI broadcast. Observe that the average transmission time increases linearly as the number of receivers increases, but with MPI broadcast, the average transmission time increases logarithmically.

6.4 Simulation Results

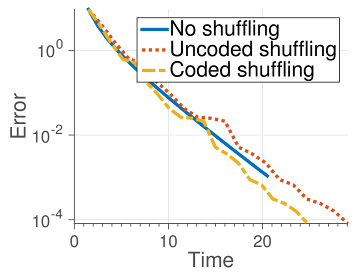
In this section, we simulate the performance of *Coded Shuffling*. More specifically, we compare the performance of parallel stochastic gradient descent (PSGD) algorithms with different shuffling schemes: *Coded Shuffling*, *Uncoded Shuffling*, and *No Shuffling*.

6.4.1 Linear Regression

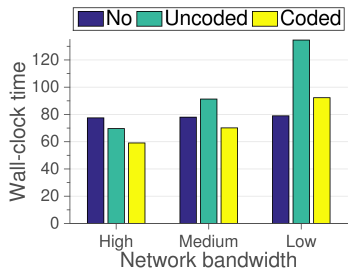
We first simulate the performance of PSGD algorithm for a simple linear regression. That is, given a data matrix $\mathbf{A} \in \mathbb{R}^{q \times r}$ and a vector $\mathbf{y} \in \mathbb{R}^q$, we want to solve the following optimization problem: $\min_{\mathbf{x}} ||\mathbf{A}\mathbf{x} - \mathbf{y}||_2^2$.



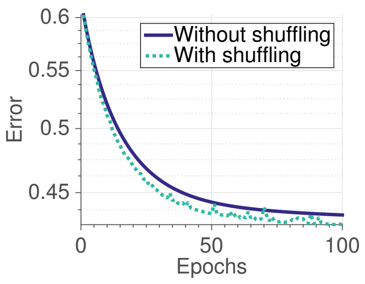
(a) Convergence speed in epochs



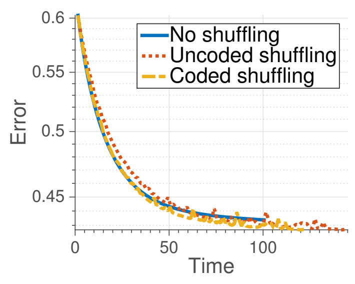
(b) Convergence speed in wall time ($\alpha = 0.5$)



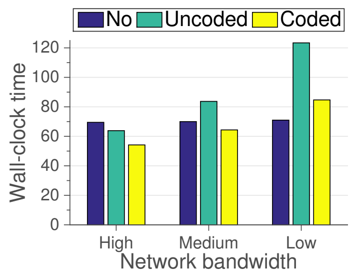
(c) Average convergence speed in wall time



(d) Convergence speed in epochs



(e) Convergence speed in wall time ($\alpha = 0.5$)



(f) Average convergence speed in wall time

Figure 6.5: **Linear regression**: Simulation results with synthetic (upper row) and real data (lower row). Plotted in (a) and (d) are the convergence performance of the algorithm with three different shuffling schemes: coded shuffling, uncoded shuffling, and no shuffling. In these figures, the x-axis is the number of passes, and the y-axis is the error. Plotted in (b) and (e) are the convergence performance of the algorithm as a function of the wall time. Plotted in (c) and (f) are the average convergence performance in wall time: we compute the wall clock time to achieve a target error with different values of bandwidth – high network bandwidth ($\alpha = 0.5$), medium network bandwidth ($\alpha = 1$), and low network bandwidth ($\alpha = 2$) –, and find the average convergence time in wall clock time over 100 runs.

We test the algorithm with both synthetic and real data. For the synthetic data, each element of the data matrix $\mathbf{A} \in \mathbb{R}^{10^4 \times 10^3}$, $\mathbf{x}^\star \in \mathbb{R}^{10^3}$, $\mathbf{w} \in \mathbb{R}^{10^4}$ is chosen uniformly at random from $[0, 1]$. Then, $\mathbf{y} = \mathbf{A}\mathbf{x}^\star + \mathbf{w}$ is given to the optimization problem as an input. With regard to experimental validation using “real" data, we use the Wine Quality Data Set from UCI Machine Learning Repository [25, 71]: Each row of the data matrix includes 11 physicochemical features of a wine such as pH, alcohol, density, acidity, etc., and the task is to predict the quality of a wine based on these features.

We plot in Fig. 6.5 the simulation results. The figures in the upper row are the simulation results with the synthetic data, and those in the lower row are the simulation results with the real data. We first compare the convergence performance of the PSGD with and without shuffling in Fig. 6.5(a). We can see that the convergence rate is significantly improved by shuffling data between epochs. For instance, after 20 epochs, the algorithm with shuffling achieves a 10 times lower error performance compared to the one without shuffling. Let us now consider the cost of communication to be fair in our comparison of the schemes. In order to account for the cost of communication, we consider the *wall time*. The wall time consists of computation time and communication time: computation of each iteration is assumed to take a unit time, and communication of the entire data matrix $\mathbf{A}$ is assumed to take α time units. Assuming n, s, and q are large and hence using Corollary 6.3, we can find the wall time after the i^{th} epoch is completed. We denote by $t_{\text{NS}}(i)$ the wall time after the i^{th} epoch when shuffling is not used, by $t_{\text{NS}}(i)$ the wall time when uncoded shuffling is used, and by $t_{\text{CS}}(i)$ the wall time when coded shuffling is used. Then,

$$t_{\text{NS}}(i) = \alpha + i, \quad t_{\text{US}}(i) = \alpha \left\{ 1 + (i-1)\left(1 - \frac{s}{q}\right) \right\} + i, \tag{6.13}$$

$$t_{\text{CS}}(i) = \alpha \left\{ 1 + (i-1)\left(1 - \frac{s}{q}\right) \frac{1}{1 + ns/q} \right\} + i. \tag{6.14}$$

Note that the common term α is for the communication cost of distributing the input data before the algorithm begins, and that the common term i is for the computation cost of the algorithm. For the uncoded shuffling and coded shuffling schemes, we take account of different shuffling costs.

Using the above equations, the convergence performance as a function of the wall time can be found. In Fig. 6.5, we compare the convergence performance of the three different schemes. Plotted in the upper row are the simulation results with synthetic data, and in the lower row are the ones with real data. In Fig. 6.5(a) and 6.5(d), the convergence of the algorithm is shown without considering the cost of communication; the algorithm that shuffles data after each iteration converges significantly faster than the one that does not shuffle data. Plotted in Fig. 6.5(b) and 6.5(e) are the convergence of the algorithm measured in wall time. Although the algorithm with uncoded shuffling converges faster than the one without shuffling in terms of the number of epochs, when the communication cost is taken into account, the actual convergence time, measured in wall time, can be slower; however, the algorithm with coded shuffling still converges faster than the others because of its reduced communication overhead. We vary the network bandwidth (communication overhead α) and measure the average convergence time (in wall time) over 100 runs, and the results are shown in Fig. 6.5(c) and 6.5(f). The algorithm with coded shuffling converges faster than the

the others in all the cases, except for the cases where the bandwidth is too low (communication cost is too high).

6.4.2 Classification

We also compare the performance of the algorithms for classification. More specifically, we solve a logistic regression problem and classify all the data points into two classes; given a data matrix $\mathbf{A} = (\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_q)^T \in \mathbb{R}^{q \times r}$ and a label vector $\mathbf{y} \in \{0,1\}^q$, we want to solve the following optimization problem: $\min_{\mathbf{x}} \sum_{i=1}^{q} -\log \Pr(y_i|\mathbf{a}_i;\mathbf{x})$, where $\Pr(y_i = 0|\mathbf{a}_i;\mathbf{x}) = \frac{\exp(-\mathbf{a}_i^T\mathbf{x})}{1+\exp(-\mathbf{a}_i^T\mathbf{x})}$ and $\Pr(y_i = 1|\mathbf{a}_i;\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{a}_i^T\mathbf{x})}$. We run the PSGD algorithm to solve the above optimization problem and measure the convergence performance of the algorithm with the different shuffling schemes.

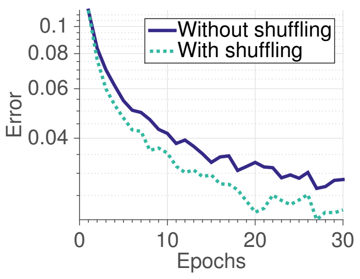
For synthetic data, each element of the data matrix $\mathbf{A} \in \mathbb{R}^{10^4 \times 10^3}$ is drawn from standard normal distribution, and each element of $\mathbf{x}^\star \in \mathbb{R}^{10^3}$ is chosen uniformly at random from $[0,1]$. Then, $\mathbf{y}$ is drawn according to the logistic distribution and is given to the optimization problem as an input. For the real data, we use Spambase Data Set from UCI Machine Learning Repository [71]. Each row of the data matrix includes 48 attributes of emails, and the task is to determine whether an email is spam or not based on these features.

Plotted in Fig. 6.6 are classification simulation results with synthetic and real data. Similar to the simulation results with linear regression, we observe that shuffling significantly improves the convergence performance of the PSGD algorithms, as shown in Fig. 6.6(a) and 6.6(d). When communication overhead is considered, the algorithm with coded shuffling provides the best performance among the three schemes unless the communication cost is too high.

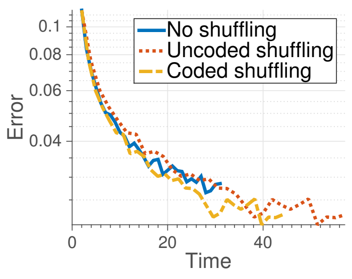
6.5 Conclusion

In this chapter, we propose a new *Coded Shuffling* framework, which can significantly reduce the overhead of data-shuffling, a costly process used in current machine-learning algorithms.

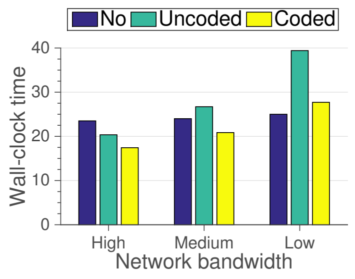
There exists a whole host of theoretical and practical open problems related to the results of this chapter. Convergence analysis of distributed machine learning algorithms under shuffling is not well understood. As we observed in the experiments, shuffling significantly reduces the number of iterations required to achieve a target reliability, but missing is a rigorous analysis that compares the convergence performances of algorithms with shuffling or without shuffling. Further, the trade-offs between bandwidth, storage, and the statistical efficiency of the distributed algorithms are not well understood. Moreover, it is not clear how far our achievable scheme, which achieves a bandwidth reduction gain of $\mathcal{O}(\frac{1}{n})$, is from the fundamental limit of communication rate for coded shuffling. Therefore, finding an information-theoretic lower bound on the rate of coded shuffling is another interesting open problem.



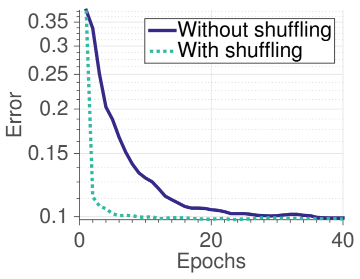
(a) Convergence speed in epochs



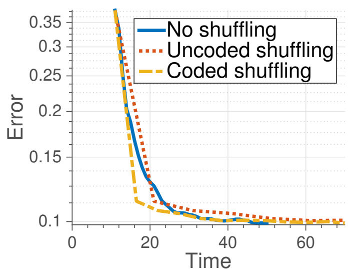
(b) Convergence speed in wall time ($\alpha = 1$)



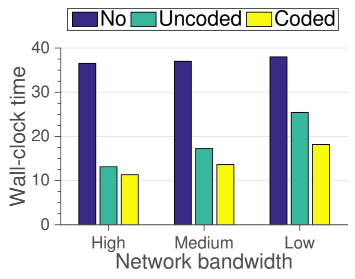
(c) Average convergence speed in wall time



(d) Convergence speed in epochs



(e) Convergence speed in wall time ($\alpha = 10$)



(f) Average convergence speed in wall time

Figure 6.6: **Classification**: Simulation results with synthetic (upper row) and real data (lower row). Plotted in (a) and (d) are the convergence performance of the algorithm with three different shuffling schemes: coded shuffling, uncoded shuffling, and no shuffling. In these figures, the x-axis is the number of passes, and the y-axis is the error. Plotted in (b) and (e) are the convergence performance of the algorithm as a function of the wall time. Plotted in (c) and (f) are the average convergence performance in wall time: we compute the wall clock time to achieve a target error with different values of bandwidth – high network bandwidth ($\alpha = 0.5$), medium network bandwidth ($\alpha = 1$), and low network bandwidth ($\alpha = 2$) –, and find the average convergence time in wall clock time over 100 runs.

Chapter 7

Conclusion and Future Research Directions

7.1 Conclusion

In this thesis, the problem of using codes to speed up distributed storage and computing systems is studied.

In Chapter 2, data retrieval performance of distributed data storage systems with coded data is studied. A novel queueing model for such systems, called the MDS queue, is proposed, and its latency performance is rigorously analyzed via queueing theoretic tools: as a result, it is shown that the data stored in coded systems can be accessed faster than those in uncoded systems. In Chapter 3 and Chapter 4, we explore how we can maximally utilize excess resources or redundancy in distributed systems by scheduling redundant requests. As a concrete example, distributed data storage systems with coded data or replicated data can handle a data access request in multiple ways, and one may achieve an improved level of data-retrieval performance by appropriately scheduling redundant requests. We characterize under which conditions, one can reduce the average latency performance by judiciously scheduling redundant requests.

In Chapter 5, we focus on designing distributed algorithms that are robust to stragglers, and provide a systematic way to design such algorithms, inspired by the principles of coding theory. In Chapter 6, we identify an underlying communication problem, which needs to be solved to achieve a higher statistical efficiency for distributed machine learning algorithms. We propose an efficient data communication and caching protocol, and show that our proposed solution can significantly reduce the communication overhead, enabling statistically efficient distributed machine learning algorithms.

To summarize, we show that 1) coded distributed data storage systems allows faster data access than uncoded ones, 2) appropriately scheduled redundant requests can significantly speed up data access in those systems, 3) coding can provide a systematic way to add redundancy into distributed algorithms so that their runtime is not affected by stragglers, and 4) coding can curtail the network overhead, and hence improve the statistical efficiency of distributed machine learning algorithms.

7.2 Future Research Directions

The new idea of using codes for speeding up distributed storage and computing systems has a huge potential for a wider range of applications but still a lot of open questions need to be addressed. There are several interesting future research directions that are closely related to the topics covered in this thesis, and we conclude the thesis by providing an overview of these new directions.

- **Computationally effective codes for distributed computation.** In Chapter 5, we provide preliminary results on how codes can speed up distributed computing, focusing on algorithms that require synchronization. In order to obtain the results of coded computation, one needs to decode the received results of computation. The decoding complexity can be ignored when the number of workers (or the block length of a code) is small enough. However, once the computing cluster is scaled up to the point where the decoding time is prohibitive, the coding gain might be completely nullified by the overhead of decoding. One promising approach is to design a coded algorithm based on sparse-graph-codes, which are the codes that allow a computationally efficient decoding algorithm based on simple "onion-peeing" operations. Similarly, locally repairable codes can also provide a coded computation algorithm that can be efficiently decoded because partial coded computation results can be locally decoded while the other results are still being computed.

- **Joint optimization of coded computation with parallel/distributed machine learning algorithms.** Parallel/distributed machine learning algorithms that are trained with a large amount of data indeed possess inherent *data redundancy*. That is, one does not need to necessarily run a training algorithm with the entire data: it can still achieve a high enough level of statistical efficiency even with most of the data. This implies that for such applications, one can design a coded computation based on a code with a small error-floor. This opens up a new tradeoff across multiple dimensions – statistical efficiency, runtime performance, and computation/network cost.

Appendix A

Proof of Theorems

A.1 Proof of Theorems in Chapter 2

Table A.1 enumerates notation for various parameters that describe the system at any given time. To illustrate this notation, consider again the system depicted in Fig. 2.2(a). Here, the parameters listed in Table A.1 take values $m = 10$, $z = 0$, $b = 3$, $s_1 = 0$, $s_2 = 0$, $s_3 = 0$, $w_1 = 2$, $w_2 = 2$ and $w_3 = 2$. One can verify that keeping track of these parameters leads to a valid Markov chain (under each of the scheduling policies discussed in this paper). Note that we do not keep track of the jobs of a batch once all k jobs of that batch have begun to be served, nor do we track what servers are serving what jobs. This is to ensure a smaller complexity of representation an2d computation. Further note that in terms of the parameters listed in Table A.1, the number of servers that are busy at any given time is equal to $(n - z)$. For batch i in the buffer ($i \in [b]$), the number of jobs that have completed service is equal to $(k - s_i - w_i)$. For any integer i, $w_i = 0$ will mean that there is no i^{th} waiting batch in the buffer.

Proof of Proposition 2.1. Since the scheduling policy mandates all k jobs of any batch to start service together, the number of jobs in the buffer is necessarily a multiple of k. Furthermore, when the buffer is not empty, the number of servers that are idle must be strictly smaller than k (since otherwise, the first waiting batch can be served). It follows that when $m \leq n$, the buffer is empty ($b = 0$), and all m jobs are being served by m servers (and $z = (n - m)$ servers are idle). When $m > n$, the buffer is not empty. Assuming there are z idle servers, there must be $(n - z)$ jobs

Table A.1: **Notation used to describe state of the system.**

Value	Meaning	Range
m	number of jobs in the entire system	0 to ∞
z	number of idle servers	0 to n
b	number of waiting batches	0 to ∞
$\{s_i\}_{i=1}^{b}$	number of of jobs of i^{th} waiting batch, in the servers	0 to $k-1$
$\{w_i\}_{i=1}^{b}$	number of of jobs of i^{th} waiting batch, in the buffer	0 to k

currently being served, and hence there are $m - (n - z)$ jobs in the buffer. However, since the number of jobs in the buffer must be a multiple of k, and since $z \in \{0, 1, \ldots, k-1\}$, it must be that $z = (n - m) \bmod k$. Thus, when $m > n$, there are $b = \frac{m-n+z}{k}$ batches waiting in the buffer, and $w_i = k,\ s_i = 0\ \forall\ i \in \{1, \ldots, b\}$. We have thus shown that the knowledge of m suffices to completely describe the system.

Once we have determined the configuration of the system as above, it is now easy to obtain the transitions between the states. An arrival of a batch increases the total number of jobs in the system by k, and hence the transition from state m to $(m+k)$ at rate λ. When $m \leq n$, all m jobs are being served, and the buffer is empty. Thus, the total number of jobs in the system reduces to $(m-1)$ at rate $m\mu$. When $m > n$, the number of jobs being served is $n - z = n - ((n-m) \bmod k)$, and thus there is a transition from state m to $(m-1)$ at rate $(n - ((n-m) \bmod k))\mu$. □

Proof of Proposition 2.2. Results as a special case of Theorem 2.3. As a side note, in any given state $(w_1, m) \in \{0, 1, \ldots, k\} \times \{0, 1, \ldots, \infty\}$ of the resulting Markov chain, the number of idle servers is given by $z = n - m$ if $m \leq n - k$, and $z = (n + w_1 - m) \bmod k$ otherwise. The state (w_1, m) has transitions to state:

- $((m+k-n)^+, m+k)$ at rate λ, if $w_1 = 0$.
- $(w_1, m+k)$ at rate λ, if $w_1 \neq 0$
- $(w_1, m-1)$ at rate $m\mu$, if $w_1 = 0$.
- $(w_1, m-1)$ at rate $(k - w_1 - z)\mu$, if $w_1 \neq 0$
- $(w_1 - 1, m-1)$ at rate $(n - k + w_1)\mu$, if $(w_1 > 1$ or $(w_1 = 1\ \&\ m \leq n+1))$
- $((k-z)^+, m-1)$ at rate $(n - k + w_1)\mu$, if $(w_1 = 1\ \&\ m > n+1)$. □

Proof of Theorem 2.3. For $(w_1, w_2, \ldots, w_t, m) \in \{0, 1, \ldots, k\}^t \times \{0, 1, \ldots, \infty\}$, define

$$q = \begin{cases} 0 & \text{if } w_1 = 0 \\ t & \text{else if } w_t \neq 0 \\ \operatorname{argmax}\{t' : w_{t'} \neq 0,\ 1 \leq t' \leq t\} & \text{otherwise.} \end{cases} \tag{A.1}$$

It can be shown that

$$b = \begin{cases} 0 & \text{if } q = 0 \\ q & \text{if } 0 < q < t \\ t + \left\lfloor \frac{m - \sum_{j=1}^{t} w_i - n}{k} \right\rfloor & \text{otherwise,} \end{cases} \tag{A.2}$$

$$z = n - (m - \sum_{j=1}^{t} w_j - (b-t)^+ k)\ , \tag{A.3}$$

$$s_i = \begin{cases} w_{i+1} - w_i & \text{if } i \in \{1, \ldots, q-1\} \\ k - z - w_q & \text{if } i = q \\ 0 & \text{if } i \in \{q+1, \ldots, b\}, \end{cases} \tag{A.4}$$

and

$$w_i = k, \quad \text{for } i \in \{t+1, \ldots, b\}. \tag{A.5}$$

Given the complete description of the state of the system as above, the characterization of the transition diagram is a straightforward task.

It is not difficult to see that the MDS-Reservation(t) queue has the following two key features: (a) any transition event changes the value of m by at most k, and (b) for $m \geq n-k+1+tk$, the transition from any state $(w_1,\ m)$ to any other state $(w_1',\ m' \geq n-k+1+tk)$ depends on $m \bmod k$ and not on the actual value of m. This results in a QBD process with boundary states and levels as specified in the statement of the theorem. Intuitively, this says that when $m \geq n-k+1+tk$, the presence of an additional batch at the end of the buffer has no effect on the functioning of the system. (In contrast, when $m < n-k+1+tk$, the system may behave differently if there was to be an additional batch, due to the possibility of this batch being within the threshold t. For instance, a job of this additional batch may be served upon completion of service at a server, which is not possible if this additional batch was not present). □

Proof of Proposition 2.4. The MDS-Reservation(t) scheduling policy treats the first t waiting batches in the buffer as per the MDS scheduling policy, while imposing an additional restriction on batches $(t+1)$ onwards. When t= ∞, every batch is treated as in MDS, thus making MDS-Reservation(∞) identical to MDS. □

Proof of Proposition 2.5. Under MDS-Violation(0), any job can be processed by any server, and hence a server may be idle only when the buffer is empty. Thus, when $m \leq n$, all m jobs are in the servers, and the buffer is empty. When $m > n$, all the n servers are full and the remaining $(m-n)$ jobs are in the buffer. The transitions follow as a direct consequence of these observations. It also follows that when $m \leq n$, $b = 0$ and $z = n - m$. In addition, in state $m\,(> n)$ it must be that $w_1 = (m-n)\text{mod}k$, $b = \lceil \frac{m-n}{k} \rceil$, and for $i \in \{2, \ldots, b\}$, $w_i = k$. Thus the knowledge of m suffices to describe the configuration of the entire system. □

Proof of Theorem 2.6. For any state $(w_1,w_2,\ldots,w_t,m)$, define q as in (A.1). The values of b, z, w_i, are identical to that in the proof of Theorem 2.3. Given the complete description of the state of the system as above, the characterization of the transition diagram is a straightforward task. It is not difficult to see that the MDS-Violation(t) queues have the following two key features: (a) any transition event changes the value of m by at most k, and (b) for $m \geq n+1+tk$, the transition from any state (w_1,m) to any other state $(w_1',m' \geq n+1+tk)$ depends on $m \bmod k$ and not on the actual value of m. This results in a QBD process with boundary states and levels as specified in the statement of the theorem. Intuitively, this says that when $m \geq n+1+tk$, the total number of waiting batches is strictly greater than t. In this situation, the presence of an additional batch at the end of the buffer has no effect on the functioning of the system. □

Proof of Proposition 2.7. The MDS-Violation(t) scheduling policy follows the MDS scheduling policy when the number of batches in the buffer is less than or equal to t. Thus, MDS-Violation(∞) is always identical to MDS. □

***Proof of Theorem* 2.8**. In the MDS queue, suppose there are a large number of batches waiting in the buffer. Then, whenever a server completes a service, one can always find a waiting batch that has not been served by that server. Thus, no server is ever idle. Since the system has n servers, each serving jobs with times i.i.d. exponential with rates μ, the average number of jobs exiting the system per unit time is $n\mu$. The above argument also implies that under no circumstances (under any scheduling policy), can the average number of jobs exiting the system per unit time exceed $n\mu$. Finally, since each batch consists of k jobs, the rate at which batches exit the system is $\lambda^*_{MDS} = \frac{n\mu}{k}$ per unit time. Since the MDS-Violation(t) queues upper bound the performance of the MDS queue, $\lambda^*_{\text{M}^k\text{/M/n(t)}} = \frac{n\mu}{k}$ for every t.

We shall now evaluate the maximum throughput of MDS-Reservation(1) by exploiting properties of QBD systems. In general, the maximum throughput λ^* of any QBD system is the value of λ such that: $\exists$ **v** satisfying $\mathbf{v}^T(A_0 + A_1 + A_2) = 0$ and $\mathbf{v}^T A_0 \mathbf{1} = \mathbf{v}^T A_2 \mathbf{1}$, where $\mathbf{1} = [1\ 1\ \cdots\ 1]^T$. Note that the matrices A_0, A_1 and A_2 are affine transformations of λ (for fixed values of μ and k). Using the values of A_0, A_1, A_2 in the QBD representation of MDS-Reservation(1), we can show that $\lambda^*_{\text{Resv(1)}} \geq (1 - O(n^{-2}))\frac{n}{k}\mu$. For t$\geq$ 2, each of the MDS-Reservation(t) queues upper bound MDS-Reservation(1), and are themselves upper bounded by the MDS queue. It follows that $\frac{n}{k}\mu \geq \lambda^*_{\text{Resv(t)}} \geq (1 - O(n^{-2}))\frac{n}{k}\mu$ for t$\geq$ 1.

The value of $\lambda^*_{\text{Resv(t)}}$ can be explicitly computed for any value of n, k and t via the method described above. We perform this computation for $k = 2$ and $k = 3$ when t$= 1$ to obtain the result mentioned in the statement of the theorem. We show the computation for $k = 2$ here.

When $k = 2$ and t$= 1$, the j^{th} level of the QBD process consists of states $\{0, 1, 2\} \times \{n - 1 + 2j, n + 2j\}$ for $j \geq 1$. However, as seen in Fig. 2.6, several of these states never occur. In particular, in level j, only the states $(1, n - 1 + 2j)$, $(1, n + 2j)$ and $(2, n + 2j)$ may be visited. Thus, to simplify notation, in the following discussion we consider the QBD process assuming the existence of only these three states (in that order) in every level. Under this representation, we have

$$A_0 = \begin{bmatrix} 0 & \mu & (n-1)\mu \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad A_1 = \begin{bmatrix} -n\mu - \lambda & 0 & 0 \\ (n-1)\mu & -(n-1)\mu - \lambda & 0 \\ n\mu & 0 & -n\mu - \lambda \end{bmatrix}, \quad A_2 = \begin{bmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & \lambda \end{bmatrix}. \tag{A.6}$$

$$\Rightarrow A_0 + A_1 + A_2 = \begin{bmatrix} -n & 1 & n-1 \\ n-1 & -(n-1) & 0 \\ n & 0 & -n \end{bmatrix} \mu \tag{A.7}$$

One can verify that the vector

$$\mathbf{v} = \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix}^T = \begin{bmatrix} n-1 & 1 & \frac{(n-1)^2}{n} \end{bmatrix}^T \tag{A.8}$$

satisfies

$$\mathbf{v}^T(A_0 + A_1 + A_2) = 0. \tag{A.9}$$

Thus,

$$\mathbf{v}^T A_0 \mathbf{1} = n(n-1)\mu, \tag{A.10}$$

and

$$\mathbf{v}^T A_2 \mathbf{1} = \lambda \left(n + \frac{(n-1)^2}{n} \right) . \tag{A.11}$$

According to the properties of QBD processes, the value of $\lambda = \lambda^*_{\text{Resv}(1)}$ must satisfy $\mathbf{v}^T A_0 \mathbf{1} = \mathbf{v}^T A_2 \mathbf{1}$. Thus,

$$\lambda^*_{\text{Resv}(1)} = \frac{n^2(n-1)}{2n^2 - 2n + 1} = \left(1 - \frac{1}{2n^2 - 2n + 1}\right) \frac{n}{2}\mu. \tag{A.12}$$

□

Proof of Theorem 2.11. Denote by D_i the latency experienced by the job enqueued to the i^{th} chosen server. As each server is an M/M/1 queue, $\mathbb{E}[D_i] = \frac{1}{\mu - \lambda_{\text{eff}}}$. The latency of a job consists of the waiting time in the queue and the service time. That is, $D_i = W_i + S_i$, where W_i is the waiting time in the queue of the i^{th} server and S_i is the service time at the i^{th} server. Note that $\{S_i\}$ are independent exponential random variables with rate μ. Thus, for all $1 \le i \le k$, $\mathbb{E}[W_i] = \frac{1}{\mu - \lambda_{\text{eff}}} - \frac{1}{\mu}$. Therefore,

$$D = \max_{1 \le i \le k} D_i = \max_{1 \le i \le k} \{W_i + S_i\} \ge \max_{1 \le i \le k} S_i + W_{\arg\max_i S_i}. \tag{A.13}$$

Note that $\mathbb{E}[W_{\arg\max_i S_i}] = \mathbb{E}[W_1]$ because $\{W_i\}$ are i.i.d., and $\{S_i, W_i\}$ are mutually independent. By taking the expectation of both sides,

$$\mathbb{E}[D] \ge \mathbb{E}\left[\max_{1 \le i \le k} S_i\right] + \mathbb{E}\left[W_1\right] = \frac{H_k}{\mu} + \left(\frac{1}{\mu - \lambda_{\text{eff}}} - \frac{1}{\mu}\right), \tag{A.14}$$

where the equality is due to the fact that the expected value of the maximum of k i.i.d. exponential random variables of rate μ is $\frac{H_k}{\mu}$. □

Proof of Lemma 2.12. Assume that all the n servers are in steady state for $t \ge 0$. Then, the n servers are independent M/M/1 queues in steady state because the arrival process to the n servers are n independent Poisson processes, generated by the IPPG. We now show that the k jobs of a batch experiences mutually independent queue lengths (and hence mutually independent latencies), and each queue length is distributed according to the steady-state queue length distribution of an M/M/1 queue, say $\tilde{X}$. Let T_k denote the time when the k^{th} job of a certain batch enters in any of the n servers. As all the n servers are independent of the arrival process and the state of the IPPG, $\langle T_i \rangle_{i=1}^k$ are independent of the queue lengths $\langle X_i \rangle_{i=1}^n$. We also denote by S_i the randomly chosen server by the i^{th} job of the batch. Then,

$$\begin{aligned}
&\Pr(X_{S_1}(T_1) = x_1, X_{S_2}(T_2) = x_2, \dots, X_{S_k}(T_k) = x_k) \\
&= \Pr(X_1(T_1) = x_1, X_2(T_2) = x_2, \dots, X_k(T_k) = x_k) \\
&= \int_{t_1=0}^{\infty} \cdots \int_{t_k=0}^{\infty} \Pr(X_1(t_1) = x_1, \dots, X_k(t_k) = x_k | T_1 = t_1, \dots, T_k = t_k) f_{T_1,\dots,T_k}(t_1, \dots, t_k)\, \mathrm{d}t_1 \dots \mathrm{d}t_k \\
&= \int_{t_1=0}^{\infty} \cdots \int_{t_k=0}^{\infty} \Pr(X_1(t_1) = x_1, \dots, X_k(t_k) = x_k) f_{T_1,\dots,T_k}(t_1, \dots, t_k)\, \mathrm{d}t_1 \dots \mathrm{d}t_k \\
&= \int_{t_1=0}^{\infty} \cdots \int_{t_k=0}^{\infty} \Pr(\tilde{X} = x_1) \cdots \Pr(\tilde{X} = x_k) f_{T_1,\dots,T_k}(t_1, \dots, t_k)\, \mathrm{d}t_1 \dots \mathrm{d}t_k \\
&= \Pr(\tilde{X} = x_1) \cdots \Pr(\tilde{X} = x_k),
\end{aligned}$$

where the first equality is due to symmetry, the third equality is due to independence of $\langle T_i \rangle_{i=1}^{k}$ and $\langle X_i \rangle_{i=1}^{n}$, and the fourth equality follows from the fact that the n queues are stationary and independent. As the queue lengths faced by the k jobs are independent of the others, the latencies of the k jobs are also independent. The latency of a job that enters a steady-state M/M/1 queue is exponentially distributed with rate $\lambda' - \mu$ [49]. Thus, the expectation of the maximum of k latencies in the k servers is simply $\frac{H_k}{\mu - \lambda'}$. □

Proof of Theorem 2.9. The theorem immediately follows from the application of Lemma A.1 and Lemma A.2. Using Lemma A.2 and Little's law, the average job latency in the system can be computed as:

$$\mathbb{E}[D_{\text{job}}] = \frac{1}{2\lambda} \left[\sum_{l=1}^{n-1} l\pi_{0,l} + \sum_{m=0}^{\infty} (n + 2m + 1)\pi_{1,n+2m+1} + \sum_{m=0}^{\infty} (n + 2m)(\pi_{2,n+2m} + \pi_{1,n+2m}) \right]. \tag{A.15}$$

Plugging this in the inequality of Lemma A.1, the statement of the theorem is obtained. □

Lemma A.1. *Let $\mathbb{E}[D]$ and $\mathbb{E}[D_{job}]$ be the average request delay and average job request delay under the MDS-Reservation(1) with $k = 2$, then:*

$$\mathbb{E}[D] \leq \mathbb{E}[D_{job}] + \frac{n-1}{n-2}\frac{1}{2\mu} - \frac{1}{(n-2)(n-1)n\mu} - \frac{1}{2(n-1)\mu}. \tag{A.16}$$

Proof. Consider any request that enters and departs from the system. Let W_1 and W_2 be the waiting times of its first and second jobs in the queue, and S_1 and S_2 be the service times of them.

We now derive a relationship between W_1 and W_2. Without loss of generality, job 1 always starts getting served before job 2, thus $W_1 \leq W_2$. Now suppose job 1 goes to server j at time t.

Case 1: At time t, only one server is idle. Then, the extra waiting time of job request 2 in the queue is exactly the time it takes for any of the other $n-1$ servers to be free. By the exponential service time nature, this time is exponentially distributed with rate $(n-1)\mu$, i.e.,

$$W_2 = W_1 + \tau, \quad \text{where } \tau \sim \exp((n-1)\mu). \tag{A.17}$$

Now let D_1 and D_2 be the total times job request 1 and 2 stay in the system, and let D be the time the request spends in the system, we have:

$$D_1 = W_1 + S_1, \quad D_2 = W_1 + \tau + S_2, \quad D = W_1 + \max(S_1, S_2 + \tau). \tag{A.18}$$

Denote $Z = \max(S_1, S_2 + \tau)$. It can be verified that:

$$f_Z(z) = \frac{n-1}{n-2}\mu e^{-\mu z} - \frac{(n-1)\mu}{n-2}e^{-(n-1)\mu z} + \mu e^{-\mu z} - \frac{n-1}{n-2}2\mu e^{-2\mu z} + \frac{1}{n-2}\mu e^{-n\mu z}, \tag{A.19}$$

and:

$$\mathbb{E}[Z] = \frac{1}{\mu} + \frac{n-1}{n-2}\frac{1}{2\mu} - \frac{1}{(n-2)(n-1)n\mu}. \tag{A.20}$$

It thus follows that the difference between the average request delay and the average job delay under the MDS-Reservation(1) is given by:

$$\begin{aligned}
\mathbb{E}[D] - \mathbb{E}[D_{\text{job}}] &= \mathbb{E}[D] - \frac{1}{2}\left(\mathbb{E}[D_1] + \mathbb{E}[D_2]\right) && \text{(A.21)}\\
&= \mathbb{E}[W_1] + \mathbb{E}[Z] - \frac{1}{2}(2\mathbb{E}[W_1] + 2\mathbb{E}[S_1] + \mathbb{E}[\tau]) && \text{(A.22)}\\
&= \mathbb{E}[Z] - \mathbb{E}[S_1] - \frac{1}{2}\mathbb{E}[\tau] && \text{(A.23)}\\
&= \frac{n-1}{n-2}\frac{1}{2\mu} - \frac{1}{(n-2)(n-1)n\mu} - \frac{1}{2(n-1)\mu}. && \text{(A.24)}
\end{aligned}$$

Now, consider the other case.

Case 2: At time t, two servers are idle. Then, both job 1 and job 2 start getting served at time t. Thus,

$$D_1 = W_1 + S_1, \quad D_2 = W_1 + S_2, \quad D = W_1 + \max(S_1, S_2). \tag{A.25}$$

Hence,

$$\mathbb{E}[D] - \mathbb{E}[D_{\text{job}}] = \mathbb{E}[\max(S_1, S_2)] - \mathbb{E}[S_1] = \frac{1}{2\mu}. \tag{A.26}$$

By dividing (A.24) by (A.26), one can show that $\mathbb{E}[D] - \mathbb{E}[D_{\text{job}}]$ is strictly larger in the first case because

$$\frac{\frac{n-1}{n-2}\frac{1}{2\mu} - \frac{1}{(n-2)(n-1)n\mu} - \frac{1}{2(n-1)\mu}}{\frac{1}{2\mu}} = 1 + \frac{n-2}{n(n-1)(n-2)} > 1. \tag{A.27}$$

Therefore, in any case,

$$\mathbb{E}[D] - \mathbb{E}[D_{\text{job}}] \le \frac{n-1}{n-2}\frac{1}{2\mu} - \frac{1}{(n-2)(n-1)n\mu} - \frac{1}{2(n-1)\mu}, \tag{A.28}$$

which completes the proof. □

Lemma A.2. *The Markov chain of MDS-Reservation(1) with $(n, k = 2)$ is depicted in Fig. A.1. Denoting by π the stationary distribution of the Markov chain,*

$$\pi_{0,0} = \frac{1-\eta}{(1-\eta)\sum_{l=0}^{n-2} a_l + \frac{\lambda a_{n-2}}{n\mu} + a_{n-1}}, \tag{A.29}$$

$$\pi_{0,l} = a_l \pi_{0,0},\ \forall\, l \in \{1, ..., n-1\}, \tag{A.30}$$

$$\pi_{1,n+2m-1} = \frac{\lambda}{n\mu}(\pi_{2,n+2m-2} + \pi_{1,n+2m-2} + \pi_{1,n+2m-3}), \tag{A.31}$$

$$\pi_{0,n} = \pi_{2,n} = \frac{1}{\gamma_2}\left[\beta_2(\pi_{0,n-1} + \pi_{0,n-2}) + \lambda \pi_{0,n-2}\right], \tag{A.32}$$

$$\pi_{1,n} = \frac{1}{\gamma_1}\left[\beta_1(\pi_{0,n-1} + \pi_{0,n-2}) + \lambda \pi_{0,n-2}\right] \tag{A.33}$$

$$\pi_{2,n+2m} = \frac{1}{\gamma_2}[\beta_2(\pi_{2,n+2m-2} + \pi_{1,n+2m-2} + \pi_{1,n+2m-1}) \tag{A.34}$$

$$+ \lambda\pi_{2,n+2m-2} - (n-1)\lambda\pi_{1,n+2m-2}], \tag{A.35}$$

$$\pi_{1,n+2m} = \frac{1}{\gamma_1}[\beta_1(\pi_{2,n+2m-2} + \pi_{1,n+2m-2} + \pi_{1,n+2m-1}) \tag{A.36}$$

$$+ \lambda\pi_{2,n+2m-2} - (n-1)\lambda\pi_{1,n+2m-2}],\ \forall\ m \geq 1, \tag{A.37}$$

where $\pi_{2,n}$ is introduced for notational simplicity, and

$$\eta = \frac{\lambda}{n\mu} + \frac{\lambda(n-1)\mu}{(n\mu)^2} + \frac{\lambda\mu}{(n-1)\mu n}, \tag{A.38}$$

$$\gamma_1 = \frac{-(n-1)\mu(\lambda + n\mu)}{n\mu} - (n-1)(\lambda + (n-1)\mu), \tag{A.39}$$

$$\gamma_2 = \frac{n\mu(\lambda + (n-1)\mu)}{\mu} + (\lambda + n\mu), \tag{A.40}$$

$$\beta_1 = \frac{-\lambda(\lambda + n\mu)}{n\mu},\quad \beta_2 = \frac{\lambda(\lambda + (n-1)\mu)}{\mu}, \tag{A.41}$$

$$a_0 = 1, a_1 = \frac{\lambda}{\mu}a_0, a_l = \frac{\lambda}{l\mu}(a_{l-1} + a_{l-2}),\ l \in \{0, ..., n-1\}. \tag{A.42}$$

Proof. We find the stationary distribution of the Markov chain by carefully choosing sets of global balance equations, which we call "cuts." Our approach is to first compute $\pi_{0,0}$. Then starting from $\pi_{0,0}$, we iteratively compute all the other probabilities.

First consider the sets that contain the $(2, n + 2m)$ and $(1, n + 2m)$ states, i.e., the Type 3 cuts in Fig. A.1 [1]. Note that we use a shorthand n' for $n - 1$ for an improved visualization. We note that if the system is stable, then the total transition rate going out from any set of states must be equal to the total rate going into them. Thus, we first have the following equation for the states

[1]For notational simplicity, we use the following exchangeable notation: state $(2, n)$ denotes state $(0, n)$.

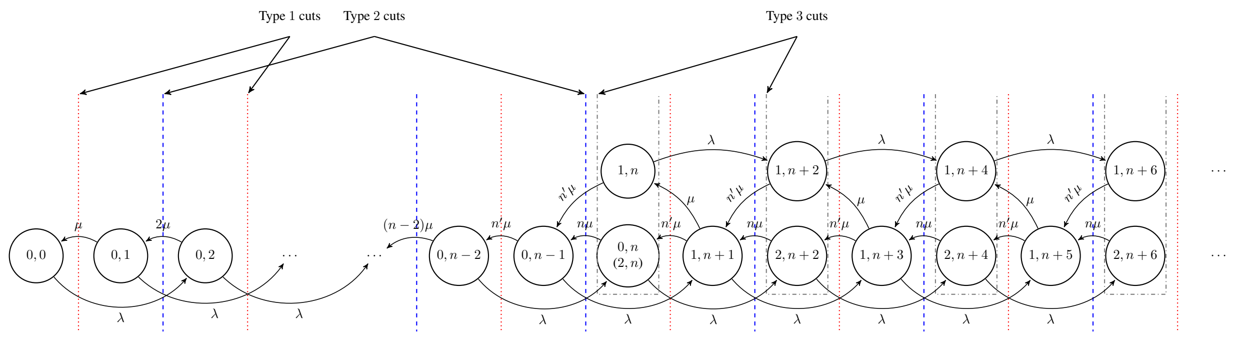


Figure A.1: **State transition diagram of the MDS-Reservation(1) queue for n and $k = 2$, where n is an even integer larger than 2.** We use a shorthand notation n' for $n-1$ for an improved visualization. The notation at any state is (w_1, m).

$(2, n)$ and $(1, n)$:

$$\pi_{2,n}(\lambda + n\mu) + \pi_{1,n}(\lambda + (n-1)\mu) = \pi_{0,n-2}\lambda + \pi_{1,n+1}n\mu. \tag{A.43}$$

Then for states $(2, n+2m)$ and $(1, n+2m)$ with $m \geq 1$, we have:

$$\pi_{2,n+2m}(\lambda + n\mu) + \pi_{1,n+2m}(\lambda + (n-1)\mu) = \lambda(\pi_{2,n+2m-2} + \pi_{1,n+2m-2}) + \pi_{1,n+2m+1}n\mu. \tag{A.44}$$

Summing (A.43) and (A.44) over $m = 1, 2, ...$, we get:

$$n\mu \sum_{m=0}^{\infty} \pi_{2,n+2m} + (n-1)\mu \sum_{m=0}^{\infty} \pi_{1,n+2m} = \lambda\pi_{0,n-2} + n\mu \sum_{m=0}^{\infty} \pi_{1,n+2m+1}. \tag{A.45}$$

Now consider the Type 2 cuts, starting from states $(2, n)$ and $(1, n)$. We have:

$$(\pi_{0,n-2} + \pi_{0,n-1})\lambda = \pi_{2,n}n\mu + \pi_{1,n}(n-1)\mu, \tag{A.46}$$

$$(\pi_{2,n+2m} + \pi_{1,n+2m} + \pi_{1,n+2m+1})\lambda = \pi_{2,n+2m+2}n\mu + \pi_{1,n+2m+2}(n-1)\mu, \ \ m \geq 0. \tag{A.47}$$

Summing (A.46) and (A.47) over $m = 0, 1, ...$, we get:

$$n\mu \sum_{m=0}^{\infty} \pi_{2,n+2m} + (n-1)\mu \sum_{m=0}^{\infty} \pi_{1,n+2m} = \lambda \left[1 - \sum_{l=0}^{n-3} \pi_{0,l}\right]. \tag{A.48}$$

Using (A.45) and (A.48), we thus obtain:

$$n\mu \sum_{m=0}^{\infty} \pi_{1,n+2m+1} = \lambda \left[1 - \sum_{l=0}^{n-2} \pi_{0,l}\right]. \tag{A.49}$$

We now try to first find $\pi_{0,0}$. Consider the states with $w_1 = 2$ and $m \geq n$.

$$pi_{2,n}(\lambda + n\mu) = \pi_{0,n-2}\lambda + \pi_{1,n+1}(n-1)\mu, \tag{A.50}$$
$$\pi_{2,n+2m}(\lambda + n\mu) = \pi_{2,n+2m-2}\lambda + \pi_{1,n+2m+1}(n-1)\mu. \tag{A.51}$$

Summing (A.50) and (A.51) over $m \geq 1$, we obtain:

$$n\mu \sum_{m=0}^{\infty} \pi_{2,n+2m} = \lambda\pi_{0,n-2} + (n-1)\mu \sum_{m=0}^{\infty} \pi_{1,n+2m+1}. \tag{A.52}$$

Similarly, we can look at the states with $w_1 = 1$ and $m \geq n$.

$$\pi_{1,n}(\lambda + (n-1)\mu) = \pi_{1,n+1}\mu, \tag{A.53}$$
$$\pi_{1,n+2m}(\lambda + (n-1)\mu) = \pi_{1,n+2m-2}\lambda + \pi_{1,n+2m+1}\mu. \tag{A.54}$$

Summing these up, we get:

$$(n-1)\mu \sum_{m=0}^{\infty} \pi_{1,n+2m} = \mu \sum_{m=0}^{\infty} \pi_{1,n+2m+1}. \tag{A.55}$$

Using (A.49), (A.52) and (A.55), we obtain:

$$\sum_{m=0}^{\infty} [\pi_{2,n+2m} + \pi_{1,n+2m} + \pi_{1,n+2m+1}] = \left[\frac{\lambda}{n\mu} + \frac{\lambda(n-1)\mu}{(n\mu)^2} + \frac{\lambda}{(n-1)\mu n}\right]\left[1 - \sum_{l=0}^{n-2}\pi_{0,l}\right] + \frac{\lambda}{n\mu}\pi_{0,n-2}. \tag{A.56}$$

Therefore, we get:

$$\left[\frac{\lambda}{n\mu} + \frac{\lambda(n-1)\mu}{(n\mu)^2} + \frac{\lambda}{(n-1)\mu n}\right]\left[1 - \sum_{l=0}^{n-2}\pi_{0,l}\right] + \frac{\lambda}{n\mu}\pi_{0,n-2} = \left[1 - \sum_{l=0}^{n-2}\pi_{0,l}\right] - \pi_{0,n-1}. \tag{A.57}$$

We see that (A.57) provides one equation in terms of only $\pi_{0,0}, ..., \pi_{0,n-1}$. Below we show that all the probabilities $\pi_{0,1}, ..., \pi_{0,n-1}$ can be expressed in terms of $\pi_{0,0}$. In this case (A.57) will allow us to compute $\pi_{0,0}$ exactly. This in turn enables us to compute $\pi_{0,1}, ..., \pi_{0,n-1}$. To do so, we first consider the type 1 and type 2 cuts shown in Fig. A.1 to get:

$$\pi_{0,1} = \frac{\lambda}{\mu}\pi_{0,0}, \tag{A.58}$$
$$\pi_{0,2i} = \frac{\lambda}{2i\mu}(\pi_{0,2i-2} + \pi_{0,2i-1}), \ \ \forall\, i \in \{1, ..., \frac{n}{2} - 1\}, \tag{A.59}$$
$$\pi_{0,2i+1} = \frac{\lambda}{(2i+1)\mu}(\pi_{0,2i-1} + \pi_{0,2i}), \ \forall\, i \in \{1, ..., \frac{n}{2} - 1\}. \tag{A.60}$$

Using (A.58), (A.59) and (A.60), one can obtain:

$$\pi_{0,l} = a_l \pi_{0,0}, \quad \forall\, l \in \{0, ..., n-1\}, \tag{A.61}$$

where $\{a_l, l = 0, ..., n-1\}$ are defined as in (A.42). Plugging (A.61) back into (A.57) and denote $\eta \overset{\text{def}}{=} \frac{\lambda}{n\mu} + \frac{\lambda(n-1)\mu}{(n\mu)^2} + \frac{\lambda}{(n-1)\mu n}$, we have:

$$1 - \sum_{l=0}^{n-2} a_l \pi_{0,0} = \eta - \eta \sum_{l=0}^{n-2} a_l \pi_{0,0} + \frac{\lambda}{n\mu} a_{n-2}\pi_{0,0} + a_{n-1}\pi_{0,0}. \tag{A.62}$$

Therefore:

$$\pi_{0,0} = \frac{1-\eta}{(1-\eta)\sum_{l=0}^{n-2} a_l + \frac{\lambda a_{n-2}}{n\mu} + a_{n-1}}. \tag{A.63}$$

It is not difficult to verify that $\pi_{0,0}$ is a valid probability if $1 - \eta > 0$, i.e.,

$$\frac{\lambda}{n\mu} + \frac{\lambda(n-1)\mu}{(n\mu)^2} + \frac{\lambda}{(n-1)\mu n} < 1. \tag{A.64}$$

This implies that the supportable rate is:

$$\lambda < \frac{n\mu}{2} \frac{1-\frac{1}{n}}{1-\frac{1}{n}+\frac{1}{2n^2}} = \frac{n\mu}{2}\left(1 - \frac{1}{2n^2-2n+1}\right), \tag{A.65}$$

which matches the result of Theorem 2.8.

We can now use (A.61) to compute $\pi_{0,0}, ..., \pi_{0,n-1}$. To compute $\pi_{2,n+2m}, \pi_{1,n+2m}$, for all $m \geq 0$, we start from $m = 0$. We have from (A.46) that:

$$\pi_{2,n} n\mu + \pi_{1,n}(n-1)\mu = \lambda(\pi_{0,n-1} + \pi_{0,n-2}). \tag{A.66}$$

Now if we look at the state $(2, n)$ and $(1, n)$ separately, we get:

$$\pi_{2,n}(\lambda + n\mu) = \pi_{0,n-2}\lambda + \pi_{1,n+1}(n-1)\mu, \tag{A.67}$$

$$\pi_{1,n}(\lambda + (n-1)\mu) = \pi_{1,n+1}\mu. \tag{A.68}$$

Canceling the term $\pi_{1,n+1}$ in (A.67) and (A.68), we get that:

$$\pi_{2,n}(\lambda + n\mu) - \pi_{1,n}(n-1)(\lambda + (n-1)\mu) = \lambda\pi_{0,n-2}. \tag{A.69}$$

With (A.66) and (A.69), we can now compute $\pi_{2,n}, \pi_{1,n}$. To make the expressions more concise, we define:

$$\gamma_2 \overset{\text{def}}{=} \frac{n\mu(\lambda + (n-1)\mu)}{\mu} + (\lambda + n\mu), \tag{A.70}$$

$$\gamma_1 \overset{\text{def}}{=} \frac{-(n-1)\mu(\lambda + n\mu)}{n\mu} - (n-1)(\lambda + (n-1)\mu), \tag{A.71}$$

$$\beta_2 \overset{\text{def}}{=} \frac{\lambda(\lambda + (n-1)\mu)}{\mu}, \quad \beta_1 \overset{\text{def}}{=} \frac{-\lambda(\lambda + n\mu)}{n\mu}. \tag{A.72}$$

Then we get:

$$\pi_{2,n} = \frac{1}{\gamma_2}\left[\beta_2(\pi_{0,n-1} + \pi_{0,n-2}) + \lambda\pi_{0,n-2}\right], \tag{A.73}$$

$$\pi_{1,n} = \frac{1}{\gamma_1}\left[\beta_1(\pi_{0,n-1} + \pi_{0,n-2}) + \lambda\pi_{0,n-2}\right]. \tag{A.74}$$

Now for all the states $(2, n+2m)$ and $(1, n+2m)$ with $m \geq 1$, using (A.47), (A.51) and (A.54), we get:

$$n\mu\pi_{2,n+2m} + (n-1)\mu\pi_{1,n+2m} = \lambda(\pi_{2,n+2m-2} + \pi_{1,n+2m-2} + \pi_{1,n+2m-1}),$$
$$(\lambda + n\mu)\pi_{2,n+2m} - (n-1)(\lambda + (n-1)\mu)\pi_{1,n+2m} = \lambda\pi_{2,n+2m-2} - (n-1)\lambda\pi_{1,n+2m-2}.$$

We can thus obtain the following equations for all states $n+2m, m \geq 1$:

$$\pi_{2,n+2m} = \frac{1}{\gamma_2}\left[\beta_2(\pi_{2,n+2m-2} + \pi_{1,n+2m-2} + \pi_{1,n+2m-1}) + \lambda\pi_{2,n+2m-2} - (n-1)\lambda\pi_{1,n+2m-2}\right],$$
$$\pi_{1,n+2m} = \frac{1}{\gamma_1}\left[\beta_1(\pi_{2,n+2m-2} + \pi_{1,n+2m-2} + \pi_{1,n+2m-1}) + \lambda\pi_{2,n+2m-2} - (n-1)\lambda\pi_{1,n+2m-2}\right],$$

both of which can be verified to be positive.

Finally, the probabilities $\pi_{1,n+2m-1}$ with $m \geq 1$ can be computed using type 1 cuts, i.e.,

$$n\mu\pi_{1,n+2m-1} = \lambda(\pi_{2,n+2m-2} + \pi_{1,n+2m-2} + \pi_{1,n+2m-3}). \tag{A.75}$$

$\square$

A.2 Proof of Theorems in Chapter 3

In this section, we first derive some properties of heavy-everywhere (3.1) and light-everywhere (3.2) classes of distributions. We also provide examples of distributions that fall into these classes. Then, we present the proofs of the results in Section 3.

A.2.1 Heavy-everywhere and Light-everywhere Distributions

In this section we derive some properties of heavy-everywhere (3.1) and light-everywhere (3.2) classes of distributions. We also provide examples of distributions that fall into these classes.

Proposition A.3. *The expected value of the minimum of n random variables, each drawn independently from a distribution that is heavy-everywhere, is no larger than $\frac{1}{n}$ times the expected value of that distribution. The expected value of the minimum of n random variables, each drawn independently from a distribution that is light-everywhere, is no smaller than $\frac{1}{n}$ times the expected value of that distribution.*

Proposition A.4. *Consider a finite set of independent random variables $X_1, \ldots, X_L$, each of whose (marginal) distributions is heavy-everywhere, such that for every i, j and every $a \geq 0,\ b \geq 0$,*

$$P(X_i > a) > P(X_j > a) \Rightarrow P(X_i > b) \geq P(X_j > b)\ .$$

Then, any mixture of $X_1, \ldots, X_L$ is also a heavy-everywhere distribution.

Proposition A.5. *The following distributions are heavy-everywhere:*

1. *A mixture of a finite number of independently drawn exponential distributions.*
2. *A Weibull distribution with scale parameter smaller than 1, i.e., with a pdf*

$$f(x) = \frac{k}{\lambda}\left(\frac{x}{\lambda}\right)^{k-1} e^{-(x/\lambda)^k}$$

for any $k \in (0, 1]$ and any $\lambda > 0$.

Proposition A.6. *The sum of a finite number of independent random variables, each of which has a (marginal) distribution that is light-everywhere, also has a distribution that is light-everywhere.*

Proposition A.7. *The following distributions are light-everywhere:*

1. *For any $c > 0$, the constant distribution with entire mass on c.*
2. *An exponential distribution that is shifted by a positive constant.*
3. *The uniform distribution.*
4. *For any pair of non-negative constants c_1 and c_2 with $2c_1 > c_2 > c_1$, a distribution with its support comprising only the two constants c_1 and c_2.*

We now present the proofs of these claims.

Proof of Proposition A.3. Let X be a random variable with a distribution that is heavy-everywhere. Consider any $x > 0$. Using the property of being heavy-everywhere, we have

$$\begin{aligned} P(X > nx) &= P(X > nx, X > x) = P(X > nx|X > x)P(X > x) \\ &\geq P(X > (n-1)x)P(X > x) \geq P(X > (n-2)x)P(X > x)P(X > x) \cdots \geq P(X > x)^n. \end{aligned}$$

Now consider i.i.d. random variables $X_1, \ldots, X_n$ drawn from this distribution. The expected value of their minimum is given by

$$\begin{aligned} E[\min\{X_1, \ldots, X_n\}] &= \int P(X_1 > x, \ldots, X_n > x) d\mu(x) = \int P(X_1 > x) \cdots P(X_n > x) d\mu(x) \\ &= \int P(X > x)^n d\mu(x) \leq \int P(X > nx) d\mu(x) = \frac{1}{n} E[X]. \end{aligned}$$

If the distribution is light-everywhere, then each of the inequalities in the entire proof above are flipped, leading to the result $E[\min\{X_1, \ldots, X_n\}] \geq \frac{1}{n} E[X]$. □

Proof of Proposition A.4. Suppose X is drawn from a mixture of L independent random variables $X_1, \ldots, X_L$ for some $L \geq 1$ whose (marginal) distributions satisfy the conditions stated in the proposition. In particular, suppose X takes value X_i with probability $p_i \geq 0$ (with $\sum_{i=1}^{L} p_i = 1$). Then

$$\begin{aligned} P(X > a + b) &= \sum_{i=1}^{L} p_i P(X_i > a + b) \geq \sum_{i=1}^{L} p_i P(X_i > a) P(X_i > b) \\ &= \sum_{j=1}^{L} \sum_{i=1}^{L} p_i p_j P(X_i > a) P(X_i > b) \\ &= \left(\sum_{i=1}^{L} p_i P(X_i > a) \right) \left(\sum_{j=1}^{L} p_j P(X_j > b) \right) \\ &+ \frac{1}{2} \sum_{j=1}^{L} \sum_{i=1}^{L} p_i p_j (P(X_i > a) - P(X_j > a))(P(X_i > b) - P(X_j > b)) \end{aligned}$$

Our assumption of $P(X_i > a) \geq P(X_j > a) \Rightarrow P(X_i > b) \geq P(X_j > b)$ then gives

$$P(X > a + b) \geq \left(\sum_{i=1}^{L} p_i P(X_i > a) \right) \left(\sum_{j=1}^{L} p_j P(X_j > b) \right) = P(X > a) P(x > b).$$

□

Proof of Proposition A.5. Let X be a random variable drawn from the distribution under consideration.

1. A mixture of a finite number of independently drawn exponential distributions.
   The exponential distribution trivially satisfies (3.1) and hence is heavy-everywhere. Furthermore, if X_i and X_j are exponentially distributed with rates μ_i and μ_j,
   $$P(X_i > a) > P(X_j > a) \Rightarrow e^{-\mu_i a} > e^{-\mu_j a} \Rightarrow -\mu_i > -\mu_j \Rightarrow P(X_i > b) \geq P(X_j > b) \ .$$
   This allows us to apply Prop. A.4, giving the desired result.

2. A Weibull distribution with scale parameter smaller than 1.
   The Weibull distribution has a complementary c.d.f.
   $$P(X > x) = e^{-(x/\lambda)^k} \ .$$
   For $k \in (0,1]$, and for any $a, b > 0$, we know that
   $$\begin{aligned} &(a+b)^k \leq a^k + b^k && \Rightarrow -\left(\frac{a+b}{\lambda}\right)^k \geq -\left(\frac{a}{\lambda}\right)^k - \left(\frac{b}{\lambda}\right)^k \\ &\Rightarrow e^{-\left(\frac{a+b}{\lambda}\right)^k} \geq e^{-\left(\frac{a}{\lambda}\right)^k} e^{-\left(\frac{b}{\lambda}\right)^k} && \Rightarrow P(X > a+b) \geq P(X > a)P(X > b) \ . \end{aligned}$$

□

Proof of Proposition A.6. Let X_1 and X_2 be independent random variables whose (marginal) distributions are light-everywhere. Let $X = X_1 + X_2$, Then,

$$\begin{aligned} P(X > a+b|X > b) &= P(X_1 + X_2 > a+b, X_2 > b|X_1 + X_2 > b) \\ &+ P(X_1 + X_2 > a+b, X_2 \leq b|X_1 + X_2 > b) \\ &= P(X_1 + X_2 > a+b|X_2 > b, X_1 + X_2 > b)P(X_2 > b|X_1 + X_2 > b) \\ &+ P(X_1 + X_2 > a+b|X_2 \leq b, X_1 + X_2 > b)P(X_2 \leq b|X_1 + X_2 > b). \end{aligned} \tag{A.76}$$

Now,

$$\begin{aligned} P(X_1 + X_2 > a+b|X_2 > b, X_1 + X_2 > b) &= P(X_1 + X_2 > a+b|X_2 > b) \qquad \text{(A.77)} \\ &= P(X_2 > a+b-X_1|X_2 > b) \\ &\leq P(X_2 > a - X_1) = P(X_1 + X_2 > a) \ , \end{aligned}$$

where the final inequality utilizes the light-everywhere property of the distribution of X_2. Also,

$$\begin{aligned} P(X_1 + X_2 > a+b|X_2 \leq b, X_1 + X_2 > b) &= P(X_1 > a+b-X_2|X_2 \leq b, X_1 > b - X_2) \\ &\leq P(X_1 > a) \leq P(X_1 + X_2 > a) \ , \end{aligned}$$

where the final inequality utilizes the light-everywhere property of the distribution of X_1. Putting it back together in (A.76) we get

$$\begin{aligned} P(X>a+b|X>b) &\leq P(X_1+X_2 > a)P(X_2>b|X_1+X_2>b)+P(X_1+X_2 > a)P(X_2\leq b|X_1+X_2>b) \\ &= P(X > a). \end{aligned}$$

□

Proof of Proposition A.7. Let X be a random variable drawn from the distribution under consideration.

1. For any $c > 0$, the constant distribution with entire mass on c.
   If $a \leq c$ then $P(X > a) = 1$. If $a > c$ then $P(X > a + b) = 0$. Thus the constant distribution satisfies (3.2).

2. An exponential distribution that is shifted by a positive constant.
   The exponential distribution trivially satisfies (3.2) and is light-everywhere. A constant is also light-everywhere as shown above. Applying Proposition A.6, we get the desired result.

3. The uniform distribution.
   We first show that for every $M > 0$. the uniform distribution on the interval $[0, M]$ is light-everywhere. If $a + b \geq M$ then $P(X > a + b) = 0$, thus trivially satisfying (3.2). If $a + b < M$ then $P(X > a) = \frac{M-a}{M}$ and $P(X > a + b | X > b) = \frac{M-a-b}{M-b}$. Using the fact that $a \geq 0, b \geq 0$, some simple algebraic manipulations of these expressions lead to (3.2). Since a constant is light-everywhere, Proposition A.6 completes the result.

4. For any pair of non-negative constants c_1 and c_2 with $2c_1 > c_2 > c_1$, a distribution with its support comprising only the two constants c_1 and c_2.
   If $a + b \geq c_2$ then $P(X > a + b) = 0$. If $a < c_1$ then $P(X > a) = 1$. Finally, if $a \geq c_1$ and $a + b < c_2$ then the constraint of $2c_1 > c_2$ implies $b < c_1$. Thus in this setting, $P(X > a + b | X > b) = P(X = c_2) = P(X > a)$.

□

A.2.2 Proof of Theorems in Chapter 3

Proof of Theorem 3.1 *(centralized, memoryless service, no removal cost, $k = 1$).* Consider two systems, system S_1 with request-degree r_1 and system S_2 with request-degree r_2 ($> r_1$), both having system parameters $(n,\ k = 1)$, the same arrival process, and the same rate of service. In the proof, we shall construct two new hypothetical systems T_1 and T_2 such that the statistics of T_1 are identical to S_1, and the statistics of T_2 are identical to S_2. We shall then show that system T_2 outperforms system T_1, and conclude that S_2 outperforms S_1.

The new system T_1 is defined as follows. The system T_1 is also associated to parameters $(n,\ k = 1)$, has the same arrival and service processes as S_1, and follows the scheduling protocol described in Algorithm 8 with request-degree r_1. However, after every service-event, we perform a specific permutation of the n servers. Since the n servers have independent and memoryless service time distributions with identical rates, the system T_1 remains statistically identical to S_1. In particular, the two systems T_1 and S_1 have identical distributions of the latency and buffer occupancy. The specific permutation applied is as follows. At any point in time, consider denoting the n servers by indices '1',. . .,'n'. Upon completion of any job at any server, the servers are permuted such that the busy servers have the lowest indices and the idle servers have the higher indices. In a similar manner, we construct T_2 to be a system identical to S_2, but again permuting the servers in T_2 after every job completion such that the busy servers have the lowest indices. Thus T_2 is statistically identical to S_2.

In the system under consideration, at any point in time, there are $(n + 1)$ processes simultaneously going on: the arrival process and the processes at the n servers. The assumption of memoryless service times allows us to assume that a (fictitious) service process continues to execute even in an idle server, although no job is counted as served upon completion of the process. Let us call the completion of any of these processes as an *event*. In this proof, we assume the occurrence of any arbitrary sequence of events, and evaluate the performance of systems T_1 and T_2 under this sequence of events. Since the arrivals into the system and the memoryless processes at the servers are all independent of the state of the system, we can assume *the same* sequence of events to occur in the two systems.

We begin by showing that under an identical sequence of events (the arrivals and server completions) in systems T_1 and T_2, the number of batches remaining to be completely served in T_2 at any point of time is no more than number of batches remaining in T_1 at that time. Without loss of generality, we will prove this statement only at times immediately following an event, since the systems do not change state between any two consecutive events. With some abuse of notation, for $z \in \{0, 1, 2, \ldots\}$, we will use the term "time z" to denote the time immediately after the z^{th} event.

Assume that the two systems begin in identical states at time 0. For system T_i $(i \in \{1, 2\})$, let $b_i(z)$ denote the number of batches remaining in system T_i at time z. The proof proceeds via induction on z. The induction hypothesis is that at any time z, we have $b_1(z) \geq b_2(z)$. Since the two systems begin in identical states, $b_1(0) = b_2(0)$. Now suppose the induction hypothesis is satisfied at time $(z - 1)$. We shall now show that it is satisfied at time z as well.

Suppose the z^{th} event is the arrival of a new batch. Then

$$b_1(z) = b_1(z-1) + 1 \geq b_2(z-1) + 1 = b_2(z),$$

where the inequality results from the induction hypothesis. The hypothesis is thus satisfied at time z.

Now suppose the z^{th} event is the completion of the exponential timer of one of the n servers (in both the systems). We first consider the case $b_1(z-1) \geq b_2(z-1) + 1$. Since the completion of the timer at a server can lead to the completion of the service of at most one batch, it follows that $b_1(z) \geq b_2(z)$ in this case. Now consider the case $b_1(z-1) = b_2(z-1)$. Since $k = 1$, the number of servers occupied in system T_i $(i \in \{1, 2\})$ at time $(z-1)$ is equal to $\min\{r_i b_i(z-1),\ n\}$. Furthermore, from the construction of systems T_1 and T_2 described above (recall the permutation of servers), it must be that the first $\min\{r_i b_i(z-1),\ n\}$ servers are occupied at time $(z-1)$ in system T_i. Thus, since $r_1 < r_2$ and $b_1(z-1) = b_2(z-1)$, the set of servers occupied at time $(z-1)$ in T_1 is a subset of the servers occupied in T_2. Now, since $k = 1$, an event at a server triggers the completion of service of a batch if and only if that server was not idle. Thus, if this event leads to the completion of service of a batch in T_1, it also leads to the completion of service of a batch in system T_2. It follows that $b_1(z) \geq b_2(z)$. We have thus shown that at any point in time, the number of batches remaining in system T_2 is no more than that under system T_1.

The arguments above show that the distribution of the number of batches remaining in T_1 dominates that in T_2: with B_1 and B_2 denoting the number of batches in the system T_1 and T_2

respectively under steady state, $P(B_1 > x) \geq P(B_2 > x)$ for all $x \geq 0$. Since the average latency is proportional to the average system occupancy, it follows that the latency faced by a batch on an average in system T_2 is no more than that in T_1. These properties carry over to S_1 and S_2 since the statistics of S_1 and S_2 are identical to those of T_1 and T_2 respectively.

From arguments identical to the above, it follows that having a request degree of n for each batch minimizes the average latency as compared to any other redundant requesting policy, including ones where a different request degree may be chosen (adaptively) for different batches.

Finally, we show that if T_1 employs a fixed request-degree $r < n$ for *all* batches, and T_2 employs $r = n$ for all batches, then the average latency under T_2 is strictly smaller. At any given time, there is a non-zero probability of the occurrence of a sequence of service-events that empty system T_1 (which also results in T_2 getting emptied). Now, upon arrival of a batch, this new batch is served in $r < n$ servers of T_1 and in all n servers of T_2, and hence there is a strictly positive probability that the batch completes service in T_2 before it completes service in T_1 and also before a new batch arrives. This event results in $b_2(\cdot) < b_1(\cdot)$, and since this event occurs with a non-zero probability, we can draw the desired conclusion. □

***Proof of Theorem* 3.2** *(centralized, memoryless service, no removal cost, general k).* Consider two systems, system S_1 with an arbitrary redundant-requesting policy and system S_2 with request-degree n, both having system parameters $(n,\ k)$, the same arrival process, and the same rate of service. In the proof, we shall construct two new systems T_1 and T_2 such that the statistics of T_1 are identical to S_1, and the statistics of T_2 are identical to S_2. We shall then show that system T_2 outperforms system T_1, and conclude that S_2 outperforms S_1.

In either system, at any point in time, there are $(n+1)$ processes simultaneously going on: the arrival process and the processes at the n servers. The assumption of memoryless service times allows us to assume that a (fictitious) service process continues to execute even in an idle server, although no job is counted as served upon completion of the process. Let us term the completion of any of these $(n+1)$ timers as the an *event*. In this proof, we assume the occurrence of any arbitrary sequence of events, and evaluate the performance of systems T_1 and T_2 under this sequence of events. Since the arrivals into the system and the memoryless processes at the servers are all independent of the state of the system, we can assume *the same* sequence of events to occur in the two systems.

We shall now show that under an identical sequence of events (arrivals and server completions) in T_1 and T_2, the number of batches remaining in system T_1 is at least as much as that in T_2 at any given time. Without loss of generality, we shall prove this statement only at times immediately following an event, since the states of the systems do not change in between any two events. Abusing some notation, for $z \in \{0, 1, 2, \ldots\}$, we shall use the term "time z" to denote the time immediately following the z^{th} event.

Assume that the two systems begin in the same state at time $z = 0$. For system T_i $(i \in \{1, 2\})$, let $b_i(z)$ denote the number of batches remaining in system T_i at time z. The proof proceeds via induction on the time z. The induction hypothesis is that at any time z:

(a) $b_1(z) \geq b_2(z)$, and

(b) for any $z' > z$, if there are no arrivals between time z and z' (including at time z'), then $b_1(z') \geq b_2(z')$.

The hypotheses are clearly true at $z = 0$, when the two systems are in the same state. Now, let us consider them to be true for time z (≥ 0). Suppose the next event occurs at time $(z+1)$. We need to show that the hypotheses are true even after this event at time $(z+1)$.

First suppose the event was the completion of an exponential-timer at one of the n servers. Then there has been no arrival between times z and $(z+1)$. This allows us to apply hypothesis (b) at time z with $z' = z+1$, which implies the satisfaction of both the hypotheses at time $(z+1)$.

Now suppose the event at time $(z+1)$ is the arrival of a new batch. Then, hypothesis (a) is satisfied at time $(z+1)$ since $b_1(z+1) = b_1(z)+1 \geq b_2(z)+1 = b_2(z+1)$. We now show that hypothesis (b) is also satisfied. Consider any sequence of server-events, and any time $z' > z+1$ such that there were no further arrivals between times $(z+1)$ and z'.

Let $a_1(z')$ and $a_2(z')$ be the number of batches remaining in the two systems at time z' *if the new batch had not arrived* but the sequence of server-events was the same as before. From hypothesis (b) at time z, we know that $a_1(z') \geq a_2(z')$. Also note that the scheduling protocol described in Algorithm 8 gives priority to the batch that had arrived earliest, and as a consequence, a server serves a job from the new batch only when it cannot serve any other batch. It follows that under any sequence of server-events, for $i \in \{1,2\}$, $b_i(z') = a_i(z')+1$ if k jobs of the new batch have not completed service in T_i, else $b_i(z') = a_i(z')$. When $b_1(z') = a_1(z')+1$, it follows that $b_1(z') = a_1(z')+1 \geq a_2(z')+1 \geq b_2(z')$. It thus remains to show that $b_1(z') = a_1(z') \Rightarrow b_2(z') \leq b_1(z')$. The condition $b_1(z') = a_1(z')$ implies that k jobs of the new batch have completed service in system T_1 at or before time z'. Let $z_1, \ldots, z_k$ $(z_1 < \ldots < z_k \leq z')$ be the events when the k jobs of the new batch are served in system T_1. Then, at these times, the corresponding servers must have been idle in system T_1 if the new batch had not arrived.

Consider another sequence of events that is identical to that discussed above, but excludes the server-events that happened at times $z_1, \ldots, z_k$, and also excludes the arrival at time $(z+1)$. Let $c_i(z')$ denote the number of batches remaining in this situation at time z'. From the arguments above, we get $c_1(z') = a_1(z')$. From the second hypothesis, we also have $c_2(z') \leq c_1(z')$. Thus we already have $c_2(z') \leq c_1(z') = a_1(z') = b_1(z')$, and hence for our goal of showing $b_2(z') \leq b_1(z')$, it now suffices to show that $b_2(z') \leq c_2(z')$.

If $b_2(z') = 0$ then we automatically have $b_2(z') \leq b_1(z')$ and there is nothing left to show. Thus, we consider the case $b_2(z') > 0$, i.e., system T_2 is non-empty at time z'. We shall now see how to count the number of batches in any system at time z', under the condition that there were *no arrivals* between time $(z+1)$ and z'. Consider $(n+1)$ counters: one counter each for the n servers and one 'global' counter. At time $(z+1)$, let the value of the counter of any server be equal to the number of jobs that this server has finished serving from the batches that are still remaining in the system. Let the value of the global counter be 0 at this time. Now, whenever a server-event occurs, add 1 to the counter associated to that server, irrespective of whether the server had a job or not. Whenever the counters of any k servers become greater than zero, add 1 to the global counter, and subtract 1 from the counters of these k servers. One can see that in this process, the value of the global counter at any time gives the number of batches that have finished service since the time we started counting. With this in mind, we shall compare the sequence of events that includes the events at $z_1, \ldots, z_k$ to that which excludes these events. Since the events $z_1, \ldots, z_k$ must correspond to events at k *distinct* servers, the service-events at $z_1, \ldots, z_k$ cause the global counter of system T_2 to increase by one. Since T_2 also had one additional arrival as compared to the system of $c_2(\cdot)$, it must be that $b_2(z') = c_2(z')$. Putting the pieces together, we get that the number of batches served in T_2 at any time is at least as much as that served in T_1 at any time.

Since the average latency is proportional to the average system occupancy, it follows that the latency faced by a batch on an average in system T_2 is smaller than that in T_1. These properties carry over to S_1 and S_2 since the statistics of S_1 and S_2 are identical to those of T_1 and T_2 respectively.

Finally, we show that if T_1 employs a fixed request-degree $r < n$ for *all* batches, and T_2 employs $r = n$ for all batches, then the average latency under T_2 is strictly smaller. At any given time, there is a non-zero probability of the occurrence of a sequence of service-events that empty system T_1 (which also results in T_2 getting emptied). Now, upon arrival of a batch, this new batch is served in $r < n$ servers of T_1 and in all n servers of T_2, and hence there is a strictly positive probability that the batch completes service in T_2 before it completes service in T_1 and also before a new batch arrives. This event results in $b_2(\cdot) < b_1(\cdot)$, and since this event occurs with a non-zero probability, we can draw the desired conclusion. Thus, the distribution of the system occupancy in T_2 is strictly dominated by that of T_1. $\square$

Proof of Theorem 3.3 *(centralized, heavy-everywhere service, no removal cost, $k = 1$, high load).* Consider two systems, system S_1 with some arbitrary redundant-requesting policy, and system S_2 with request-degree n for all batches. We shall now construct two new hypothetical systems T_1 and T_2 such that T_1 is statistically identical to S_1 and T_2 is worse than S_2, and show that the performance of T_1 is worse than that of T_2.

The two new systems T_1 and T_2 are constructed as follows. Both systems have the same parameters n and $k = 1$, and retain the redundant-requesting policies of S_1 and S_2 respectively. The service-time distribution in T_1 is identical to that in S_1. On the other hand, we shall make the service time distribution of T_2 *worse* than that of S_2 in the manner described below.

Let us fix some arbitrary one-to-one correspondence between the n servers of system T_1 and the n servers of system T_2. Consider any point in time when a server in system T_2 is just beginning the service of a job. Let X denote the random variable corresponding to this service time. Let P_H denote the law associated to the heavy-everywhere service-time distribution under consideration. Since the systems operate at 100% server utilization, the corresponding server in T_1 is not idle at this point in time and is serving some job. When this server in system T_2 begins service, suppose the job in the corresponding server of system T_1 began to be serviced $t > 0$ units of time ago. Then we modify the distribution of X and let it follow the law

$$P(X > x) = P_H(X > x + t | X > t) \quad \forall x \geq 0 \, .$$

Since the distribution P_H is heavy-everywhere (3.1), the service under system T_2 is no better than that under S_2. As a result of the construction above, whenever a job begins to be processed in system T_2, it has a service-time distribution that is identical to the distribution of the service-time of the job in the corresponding server in T_1. We couple the servers even further by assuming whenever a server in T_2 begins a new job, the time taken for this job to be completed is *identical* to that taken for the job in the corresponding server in T_1 (unless, of course, some other job of the batch completes service first and this job is removed). We also feed an identical sequence of arrivals to the two systems T_1 and T_2. This completes the construction of the two systems T_1 and T_2.

Note that the aforementioned coupling of service-times between corresponding servers of systems T_1 and T_2 only takes place when the server in T_2 begins a job. The case when a server of T_1 begins serving a new job when the corresponding server of T_2 is already serving a job is not accounted for. The induction hypothesis below handles such situations.

We start at any point in time when the two systems are in an identical state, and show that the average latency faced by the batches in T_2 from then on is no larger than that faced by batches in T_1. We shall now show the following two properties via an induction on time:

a) At any point in time, the number of batches in system T_2 is no more than the number of batches in system T_1.

b) At any point in time, if a server in system T_1 begins service of a job, the corresponding server in T_2 also begins service of some job.

Part (b) of the hypothesis ensures that the service times of the jobs in corresponding servers of systems T_1 and T_2 are always identical (via the construction above).

As mentioned previously, let us start at any point in time when the two systems are in an identical state. Since the systems are in an identical state, both hypotheses hold true at this time. Without loss of generality, we shall now consider only the times immediately following an event in either system, where an event is defined as an arrival of a batch or the completion of processing by a server. First consider any time that immediately follows an arrival. By our induction hypothesis, just before the arrival, the number of batches in T_2 was no more than that in T_1. The arrival only increases the number of batches in both systems by 1, and hence induction hypothesis (a) still stands. Under a 100% server utilization, an arrival does not trigger the beginning of a service in either system. Thus, hypothesis (b) continues to hold. Let us now consider an event where a server completes processing a job. Due to hypothesis (b), the service times at corresponding servers in the two systems were coupled. As a result, the next service completes at the same time in corresponding servers of both systems. This reduces the number of batches in both systems by one, thus continuing to satisfy hypothesis (a). Furthermore, since we have assumed a 100% utilization of the servers, there is at least one batch waiting in the buffer in both the systems. In system T_2, since we had $k = 1$, $r = n$ and no removal cost, at any given time each of the n servers in system T_2 will be serving jobs of the same batch. Thus jobs in all the servers of T_2 are removed from the system, and are replaced by (new) jobs of the next batch. As a result, upon any service-event, each of the servers in T_2 begin serving new jobs, thus satisfying hypothesis (b). Due to the specific construction of the two systems, the service times of these new jobs in the servers of T_2 are identical to those of jobs in corresponding servers of T_1.

This completes the proof of the induction hypothesis, and in particular that the number of batches in T_2 at any time is no more than the number of batches in system T_1. The fact that the average latency is proportional to the average number of batches in the system implies that the average latency in system T_1 is no smaller than in T_2. Finally, the constructions of the two systems T_1 and T_2 ensured that system T_2 is worse than S_2, and system T_1 is statistically identical to S_1, thus leading to the desired result.

Finally, suppose system T_1 employs a fixed request-degree $r < n$ for all batches. Further suppose that the heavy-everywhere distribution is such that (3.1) holds with a strict inequality for a set of events that have a probability bounded away from zero. Under this setting, the aforementioned construction is such that system T_2 is worse than system S_2 by a non-trivial amount, and as a result, the average latency in system S_2 is strictly smaller than that of S_1. $\square$

Proof of Theorem 3.4 *(centralized, light-everywhere service, any removal cost, $k = 1$, high load).* Consider two systems, system S_1 with some arbitrary redundant-requesting policy, and system S_2 with request-degree $r = k = 1$ for all batches. We shall now construct two new hypothetical systems T_1 and T_2 such that T_1 is statistically identical to S_1 and T_2 is worse than S_2, and show that the performance of T_1 is worse than that of T_2.

The two new systems T_1 and T_2 are constructed as follows. Both systems have the same parameters n and $k = 1$, and retain the redundant-requesting policies of S_1 and S_2 respectively. The service-time distribution in T_2 is identical to that in S_2. On the other hand, we shall make the service time distribution of T_1 *better* than that of S_1 in the manner described below.

Let us fix some arbitrary one-to-one correspondence between the n servers of system T_1 and the n servers of system T_2. Consider any point in time when a server in system T_1 is just beginning the service of a job. Let X denote the random variable corresponding to this service time. Let P_L denote the law associated to the heavy-everywhere service-time distribution under consideration. Since the systems operate at 100% server utilization, the corresponding server in T_2 is not idle at this point in time and is serving some job. When this server in system T_2 begins service, suppose the job in the corresponding server of system T_1 began to be served $t > 0$ units of time ago. Then we modify the distribution of X and let it follow the law

$$P(X > x) = P_L(X > x + t | X > t) \quad \forall x \geq 0\ .$$

Since the distribution P_L is light-everywhere (3.2), the service under system T_1 is no better than that under S_1. As a result of the construction above, whenever a job begins to be processed in system T_1, it has a service-time distribution that is identical to the distribution of the service-time of the job in the corresponding server in T_2. We couple the servers even further by assuming whenever a server in T_1 begins a new job, the time taken for this job to be completed is *identical* to that taken for the job in the corresponding server in T_2 (unless, of course, some other job of the batch completes service first and this job is removed). We also feed an identical sequence of arrivals to the two systems T_1 and T_2. This completes the construction of the two systems T_1 and T_2.

Note that the aforementioned coupling of service-times between corresponding servers of systems T_1 and T_2 only takes place when the server in T_1 begins a job. The case when a server of T_2 begins serving a new job when the corresponding server of T_1 is already serving a job is not accounted for. The induction hypothesis below handles such situations.

We start at any point in time when the two systems are in an identical state, and show that the average latency faced by the batches in T_2 from then on is no more than that faced by batches in T_1. We shall now show the following two properties via an induction on time:

a) At any point in time, the number of batches in system T_2 is no more than the number of batches in system T_1.

b) At any point in time, if a server in system T_2 begins service of a job, the corresponding server in T_1 also begins service of some job.

Part (b) of the hypothesis ensures that the service times of the jobs in corresponding servers of systems T_1 and T_2 are always identical (via the construction above).

As mentioned previously, let us start at any point in time when the two systems are in an identical state. Since the systems are in an identical state, both hypotheses hold true at this time. Without loss of generality, we shall now consider only the times immediately following an event in either system, where an event is defined as an arrival of a batch or the completion of processing at server. First consider any time that immediately follows an arrival. By our induction hypothesis, just before the arrival, the number of batches in T_2 was no more than that in T_1. The arrival only increases the number of batches in both systems by 1, and hence induction hypothesis (a) still stands. Under a 100% server utilization, an arrival does not trigger the beginning of a service in either system. Thus, hypothesis (b) continues to hold. Let us now consider an event where a server completes processing a job. Due to hypothesis (b), the service times at corresponding servers in the two systems were coupled. As a result, the next service completes at the same time in corresponding servers of both systems. This reduces the number of batches in both systems by one, thus continuing to satisfy hypothesis (a). Furthermore, since we have assumed a 100% utilization of the servers, there is at least one batch waiting in the buffer in both the systems. In system T_2, since we had $k = 1$ and $r = k = 1$, only this server begins serving a new job, while all remaining $(n - 1)$ servers continue processing the jobs they already have. Now, the corresponding server in T_1 also had a server-event and begins serving a new job at this moment. Thus this satisfies hypothesis (b). Due to the specific construction of the two systems, the service times of these new jobs in the servers of T_2 are identical to those of jobs in corresponding servers of T_1.

This completes the proof of the induction hypothesis, and in particular that the number of batches in T_2 at any time is no more than the number of batches in system T_1. The fact that the average latency is proportional to the average number of batches in the system implies that the average latency in system T_1 is no smaller than in T_2. Finally, the constructions of the two systems T_1 and T_2 ensured that system T_2 is worse than S_2, and system T_1 is statistically identical to S_1, thus leading to the desired result.

Finally, suppose system T_1 employs a fixed request-degree $r < n$ for all batches. Further suppose that the light-everywhere distribution is such that (3.2) holds with a strict inequality for a set of events that have a probability bounded away from zero. Under this setting, the aforementioned construction is such that system T_1 is better than system S_1 by a non-trivial amount, and as a result, the average latency in system S_2 is strictly smaller than that of S_1. $\square$

Proof of Theorem 3.5 *(centralized, memoryless service, non-zero removal cost, $k = 1$, high load).* The proof is identical to that of Theorem 3.4. The system T_2 is constructed to be 'better' than system S_2 by assuming zero removal costs in T_2. □

Proof of Proposition 3.6. (disk I/O time is light-everywhere) From Proposition A.6, we know that the sum of a finite number of independent light-everywhere random variables is light-everywhere. Thus, in order to show that T is light-everywhere, we need to show that T_{seek}, T_{rotation}, and T_{transfer} are all light-everywhere. From Proposition A.7, the transfer time T_{transfer} and the rotational latency T_{rotation} are light-everywhere. We now show that T_{seek} is light-everywhere. We first note that seek distance D is a light-everywhere distribution as for every pair of values $0 \leq a \leq 1 - b$ and $0 \leq b < 1$, $\frac{(a+b-1)^2}{(b-1)^2} \leq (a-1)^2$. Thus, T_{seek} is also light-everywhere as it is a sum of the scaled D and a constant $T_{\text{min seek}}$. Therefore, T is light-everywhere. □

Proof of Theorem 3.7 *(distributed setting).* (a) In order to get to the desired results, we shall first compare a system with distributed buffers to an analogous system that has a central buffer. The scheduling policy in either setting needs to make two kinds of decisions: the number of redundant requests for each batch, and the precise set of servers to which these requests are assigned. Firstly, observe that under the redundant requesting policy of $r = n$ for all batches, the choice of the r $(= n)$ servers to which the jobs are assigned does not require any decision to be made. The centralized and the distributed systems thus are identical in this case and hence have the same average latency. Secondly, we know from the results of Section 3.4 that under all the arrival and service time distributions considered here, the choice of $r = n$ for all batches is optimal under a centralized scheme. Thirdly, for any fixed redundant requesting policy, the average latency under the centralized scheme will be no more than the average latency under the distributed scheme. This is because the first-come first-served policy as described in Algorithm 8 minimizes average latency, and moreover, the policy of the system with a centralized buffer has more information (about the system) as compared to the one operating under distributed buffers. It thus follows that even in the case of distributed buffers, the average latency is minimized when the request-degree for each batch is n.
Parts (b), (c), (d) are identical to the proofs of Theorems 3.3, 3.4 and 3.5 respectively. □

A.3 Proof of Theorems in Chapter 4

Proof of Theorem 4.3. In order to complete the proof, we solve the equations (4.4)-(4.12) in Theorem 4.5. We can exactly solve them since they are 8 linear equations with 8 variables after replacing several terms using the recursive structure. We provide the solutions of the equations here.

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

Using Lemma 4.2, one can find $E^{\pi_\infty}[D]$, and $\lambda^{\pi_\infty}_{\text{max}}$. □

Proof of Theorem 4.5. Recall that π_1 is the policy that redundant request is used only when the queue is empty. To analyze this policy we use the RRR technique in [42].

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

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

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

$$q = (\lambda + 2\mu)^{-1} \left[2\mu \cdot 0 + \lambda(q + (1-q)(1-r))\right] \tag{A.78}$$

$$r = (\lambda + \mu + \mu_c)^{-1} \left[\mu \cdot 1 + \mu_c \cdot 0 + \lambda \cdot r^2\right] \tag{A.79}$$

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

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

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

Note that in the last three equations, we used the following relationships exploiting the structure of the Markov chain: $R^L_{d,1} = T^L_{d,0} + R^L_{d,0}$, $R^L_{s,2} = R^L_{s,1} + T^L_{s,1}$, and $R^L_{c,2} = R^L_{c,1} + T^L_{c,1}$. Thus, one can solve the equations exactly to find R in closed form. By applying Lemma 4.2, we obtain the average job latency in closed form; that is, $\mathbb{E}^{\pi_1}[D] = \frac{1}{\lambda}\frac{R}{T}$. □

Proof of Theorem 4.6. We first formulate a β-discounted problem whose cost function under policy π is defined as $J^\pi = \lim_{t\to\infty} \mathbb{E}^\pi \left\{\int_0^t e^{-\beta t} N(s(t))\, dt\right\}$, where $s(t)$ is the random process of states under policy π, and the expectation is over $s(t)$. We first find the structure of the optimal policy of the β-discounted problem, and later we will relate the found structure to the structure of the optimal policy of the average problem. The optimal policy for a β-discounted problem can be found by uniformizing the continuous Markov chain and by applying Bellman's equation. Define the transition rate from state x to y as r_{xy}, the sum transition rates from state x as r_x, and the maximum sum transition rate as $\nu = \max_x r_x$. Recall that r_{xy} is a function of u_x as depicted in Figure 4.5. Then, we have the following lemma.

Lemma A.8. *(Bellman equation and value iteration for a continuous time Markov chain) Consider a β-discounted problem with the objective $\lim_{t\to\infty} \mathbb{E}^\pi \left\{\int_0^t e^{-\beta t} C(s(t))\, dt\right\}$, where $C(\cdot)$ is the cost function. Define $J(\cdot)$ as the optimal value function that satisfies the following Bellman equations,*

$$J(x) = \eta \left\{ C(x) + (\nu - r_x)J(x) + \min_{u_x} \left(\sum_y r_{xy}(u_x) J(y) \right) \right\}, \tag{A.80}$$

where $\eta = 1/(\beta+\nu)^{-1}$. Then, the optimal policy of the β-discounted problem can be constructed from $u_x^* = \arg\min_{u_x} \left(\sum_y r_{xy}(u_x) J(y)\right)$. Also, the optimal value function can be found by using value iteration. That is, one can initiate $J_0(\cdot)$ with arbitrary values, and recursively update value functions using the following update rules.

$$J_{k+1}(x) = \eta \left\{ C(x) + (\nu - r_x)J_k(x) + \min_{u_x} \left(\sum_y r_{xy}(u_x) J_k(y) \right) \right\} \tag{A.81}$$

Then, the sequence of value functions converges to $J(\cdot)$; that is, $J_k(\cdot) \to J(\cdot)$ as $k \to \infty$.

The proof can be found in Chapter 5 of [10]. Applying Lemma A.8 to the formulated β-discounted problem, we have the following equations: [2]

$$J(0) = \eta(N(0) + (\mu + \mu_c)J(0) + \lambda \min_{v_{(0)}}(v_{(0)} J(s,0) + \bar{v}_{(0)} J(d,-1))) \tag{A.82}$$

$$J(c,-1) = \eta(N(c,-1) + \mu J(c,-1) + \lambda J(c,0) + \mu_c J(0)) \tag{A.83}$$

$$J(d,-1) = \eta(N(d,-1) + \mu_c J(d,-1) + \mu J(0) + \lambda \min_{v_{(d,-1)}} (v_{(d,-1)} J(s,1) + \bar{v}_{(d,-1)} J(d,0))), \tag{A.84}$$

[2]For simplicity, we abuse notations with $J(\cdot)$ and $N(\cdot)$ in order to avoid double parenthesis; e.g., $J(\cdot, i) \overset{\text{def}}{=} J((\cdot, i))$

and for all $i \geq 0$,

$$\begin{aligned} J(d,i) &= \eta(N(d,i) + (\mu_c - \mu)J(d,i) + \lambda J(d,i+1) \\ &\quad + 2\mu \min_{u_{(d,i)}}(u_{(d,i)}J(s,i) + \bar{u}_{(d,i)}J(d,i-1))), \end{aligned} \tag{A.85}$$

$$\begin{aligned} J(s,i) &= \eta(N(s,i) + (\mu_c - \mu)J(s,i) \\ &\quad + \lambda J(s,i+1) + 2\mu J(c,i-1), \end{aligned} \tag{A.86}$$

$$\begin{aligned} J(c,i) &= \eta(N(c,i) + \lambda J(c,i+1) + \mu J(c,i-1) \\ &\quad + \mu_c \min_{u_{(c,i)}}(u_{(c,i)}J(s,i) + \bar{u}_{(c,i)}J(d,i-1))). \end{aligned} \tag{A.87}$$

We now show that the optimal value function $J(\cdot)$ has a certain structure by identifying a certain structure in the sequence of value functions $\{J_k(\cdot)\}$.

Lemma A.9. *(Structure of the β-optimal policy (i)) The optimal scheduling policy $\pi^* = (\pi_a^*, \pi_s^*, \pi_c^*)$ for a β-discounted program satisfies $\pi_c^* = \pi_s^*$. Also, $\pi_a^* = (u^*_{(c,0)}, u^*_{(c,1)})$. That is, the optimal policy can be fully characterized by π_c^*.*

Proof. From Figure 4.5(a), one can observe that states (d,i) and (c,i) share a common decision problem. This can be formally shown by comparing inner terms in the Bellman equations (A.85) and (A.87). Similarly, one can show that the optimal controls on arrival events are the same at states $(c,0)$ and $(c,1)$. □

Theorem A.10. *(Structure of the β-optimal policy (ii)) The optimal scheduling policy π^* for a β-discounted program satisfies $\pi_c^*(i) = 0$ for all $i \geq 2$. In other words, $u^*_{(c,i)} = 0$ for all $i \geq 1$.*

Proof. We first assume that $\pi_c^*(1) = u^*_{(c,0)} = 0$. We use the value iteration method; that is, we consider a sequence of functions J_k with the fixed control $u^*_{(c,0)} = 0$. Then, we initiate J_0 with all zeros, and iterate $J_{k+1} = TJ_k$. Then, the sequence of value iterations J_k has the following structure for all finite k: $J_k(d,i) \leq J_k(s,i+1)$, $\forall i \geq 0$. This can be inductively proved as follows by showing a stronger statement: the following sets of inequalities hold when $u^*_{(c,0)} = 0$.

$$\begin{aligned} J_k(0) &\leq J_k(c,-1), \\ J_k(d,i) &\leq J_k(c,i+1), \ \forall i \geq -1 \\ J_k(d,i) &\leq J_k(s,i+1), \ \forall i \geq 0 \\ J_k(s,i) &\leq J_k(s,i+1), \ \forall i \geq 0 \\ J_k(d,i) &\leq J_k(d,i+1), \ \forall i \geq -1 \\ J_k(c,i) &\leq J_k(c,i+1), \ \forall i \geq -1 \end{aligned}$$

For $k = 0$, the above relationships hold by the intialization. Assuming that all inequalities hold for k, we prove the same relationship holds for $k+1$. First, we prove the first inequality as follows.

$$\begin{aligned} J_{k+1}(0) - \eta N(0) &= \eta\{(\mu + \mu_c)J_k(0) + \lambda J_k(d,-1)\} \\ &\leq \eta\{\mu J_k(c,-1) + \mu_c J_k(0) + \lambda J_k(c,0)\} \\ &= J_{k+1}(c,-1) - \eta N(c,-1) = J_{k+1}(c,-1) - \eta N(0) \end{aligned}$$

Then, we prove the second set of inequalities as follows. For the boundary case $i = 0$,

$$\begin{aligned}
&J_{k+1}(d,-1) - \eta N(d,-1)\\
&= \eta\{\mu J_k(0) + \mu_c J_k(d,-1) + \lambda J_k(d,0)\}\\
&\leq \eta\{\mu J_k(c,-1) + \mu_c J_k(d,-1) + \lambda J_k(c,1)\}\\
&= J_{k+1}(c,0) - \eta N(c,0) = J_{k+1}(c,0) - \eta N(d,-1).
\end{aligned}$$

Then, for general cases $i \geq 1$,

$$\begin{aligned}
&J_{k+1}(d,i) - \eta N(d,i)\\
&= \eta\{2\mu J_k(d,i-1) + (\mu_c - \mu) J_k(d,i) + \lambda J_k(d,i+1)\}\\
&\leq \eta\{\mu J_k(c,i) + \mu_c J_k(d,i) + \lambda J_k(c,i+2)\}\\
&= J_{k+1}(c,i+1) - \eta N(c,i+1) = J_{k+1}(c,i+1) - \eta N(d,i)
\end{aligned}$$

Lastly, we prove the third set of inequalities as follows.

$$\begin{aligned}
&J_{k+1}(d,i) - \eta N(d,i)\\
&= \eta\{2\mu J_k(d,i-1) + (\mu_c - \mu) J_k(d,i) + \lambda J_k(d,i+1)\}\\
&\leq \eta\{2\mu J_k(c,i) + (\mu_c - \mu) J_k(s,i) + \lambda J_k(s,i+2)\}\\
&= J_{k+1}(s,i+1) - \eta N(s,i+1) = J_{k+1}(s,i+1) - \eta N(d,i)
\end{aligned}$$

Thus, we have found the optimal structure of the policy when $u^*_{(c,0)} = 0$. Similarly, we can assume $u^*_{(c,0)} = 1$, find a similar structure, and hence prove the structure.

Now, observe that the above structure directly implies that $u_{(c,i)} = 0$ for $i \geq 1$. Then, we use the convergence of the value iteration policy [10]: $J_k \to J$ as $k \to \infty$. Because J_k satisfies the property for all k, J also satisfies the structural property. Thus, $u^*_{(c,i)} = 0$ for $i \geq 1$. □

Theorem A.10 implies that either π_0 or π_1 is the optimal dynamic scheduler for the β-discounted problem. Now, the following Lemma relates the optimal policy's structure of the average cost problem, and that of the β-discounted problem.

Lemma A.11. *(Structure of the average optimal policy) The optimal scheduling policy π^* for the average program satisfies $\pi^*_c(i) = 0$ for all $i \geq 2$. In other words, $u^*_{(c,i)} = 0$ for all $i \geq 1$.*

Proof. The structure of the optimal policy for a β-discounted problem does not depend on the specific value of β. Thus, the optimal policy does not change as $\beta \to 1$ because of its monotonicity. Therefore, the policy is Blackwell optimal, and we apply Theorem 4 in [40] to show the Blackwell optimality implies the average optimality. □

With that, we conclude the proof of Theorem 4.6. □

Bibliography

[1] A. Agarwal and J. C. Duchi. “Distributed Delayed Stochastic Optimization”. In: *Proc. of the 25th Annual Conference on Neural Information Processing Systems (NIPS)*. 2011, pp. 873–881.

[2] G. Ananthanarayanan et al. “Why Let Resources Idle? Aggressive Cloning of Jobs with Dolly”. In: *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*. HotCloud’12. Boston, MA: USENIX Association, 2012, pp. 17–17.

[3] G. Ananthanarayanan et al. “Effective Straggler Mitigation: Attack of the Clones”. In: *Proc. of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2013, pp. 185–198.

[4] G. Ananthanarayanan et al. “Reining in the Outliers in Map-Reduce Clusters using Mantri”. In: *Proc. of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2010, pp. 265–278.

[5] D. G. Andersen et al. “Improving web availability for clients with MONET”. In: *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association. 2005, pp. 115–128.

[6] M. Y. Arslan et al. “Computing while charging: building a distributed computing infrastructure using smartphones”. In: *Proc. of the 8th international conference on Emerging networking experiments and technologies*. ACM. 2012, pp. 193–204.

[7] F. Baccelli, A. M. Makowski, and A. Shwartz. “The fork-join queue and related systems with synchronization constraints: Stochastic ordering and computable bounds”. In: *Advances in Applied Probability* (1989), pp. 629–660.

[8] R. Bekkerman, M. Bilenko, and J. Langford. *Scaling up machine learning: Parallel and distributed approaches*. Cambridge University Press, 2011.

[9] M. S. Bernstein et al. “Crowds in two seconds: Enabling realtime crowd-powered interfaces”. In: *ACM UIST Symposium*. 2011.

[10] D. P. Bertsekas. *Dynamic programming and optimal control*. Vol. 1. 2. Athena Scientific, 1995.

[11] D. P. Bertsekas. *Nonlinear programming*. Athena scientific, 1999.

[12] J. A. Bilmes et al. “Author retrospective for optimizing matrix multiply using PHiPAC: a portable high-performance ANSI C coding methodology”. In: *Proc. of ACM International Conference on Supercomputing 25th Anniversary*. 2014, pp. 42–44.

[13] D. Bini et al. “Structured Markov chains solver: software tools”. In: *Proceedings of the workshop on Tools for solving structured Markov chains*. ACM. 2006, p. 14.

[14] *BLAS (Basic Linear Algebra Subprograms)*. http://www.netlib.org/blas/. Accessed: 2016-05-01.

[15] S. Borkar. “Designing reliable systems from unreliable components: the challenges of transistor variability and degradation”. In: *Micro, IEEE* 25.6 (2005), pp. 10–16. ISSN: 0272-1732.

[16] L. Bottou. “Stochastic Gradient Descent Tricks”. In: *Neural Networks: Tricks of the Trade - Second Edition*. 2012, pp. 421–436.

[17] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[18] S. P. Boyd et al. “Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers”. In: *Foundations and Trends in Machine Learning* 3.1 (2011), pp. 1–122.

[19] V. R. Cadambe et al. “Optimal repair of MDS codes in distributed storage via subspace interference alignment”. In: *arXiv preprint arXiv:1106.1250* (2011).

[20] M. Chaubey and E. Saule. “Replicated Data Placement for Uncertain Scheduling”. In: *Proc. of IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPS)*. 2015, pp. 464–472.

[21] J. Chen and A. H. Sayed. “Diffusion Adaptation Strategies for Distributed Optimization and Learning Over Networks”. In: *IEEE Transactions on Signal Processing* 60.8 (2012), pp. 4289–4305.

[22] D. Ciresan, U. Meier, and J. Schmidhuber. “Multi-column deep neural networks for image classification”. In: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE. 2012, pp. 3642–3649.

[23] R. Cohen and Y. Cassuto. “Algorithms and Throughput Analysis for MDS-Coded Switches”. In: *CoRR* abs/1504.04803 (2015).

[24] W. D. Company. *Specifications for the WD Caviar RE 250 GB Next Generation Serial ATA drive (model WD2500YD)*. 2015.

[25] P. Cortez et al. “Modeling wine preferences by data mining from physicochemical properties”. In: *Decision Support Systems* 47.4 (2009), pp. 547–553.

[26] D Costello and S. Lin. *Error control coding*. New Jersey, 2004.

[27] T. M. Cover and J. A. Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[28] H. G. Cragon. *Computer Architecture and Implementation*. New York, NY, USA: Cambridge University Press, 2000. ISBN: 0-521-65168-9.

[29] J. Dean and L. A. Barroso. “The tail at scale”. In: *Communications of the ACM* 56.2 (2013), pp. 74–80.

[30] J. Dean and S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Proc. of the 6th Symposium on Operating System Design and Implementation (OSDI)*. 2004, pp. 137–150.

[31] J. Dean et al. “Large Scale Distributed Deep Networks”. In: *Proc. of the 26th Annual Conference on Neural Information Processing Systems (NIPS)*. 2012, pp. 1232–1240.

[32] J. Demmel et al. “Communication-optimal Parallel and Sequential QR and LU Factorizations”. In: *SIAM J. Scientific Computing* 34.1 (2012).

[33] J. Demmel. *Communication–Avoiding Algorithms for Linear Algebra and Beyond*. Presented at IEEE 27th International Symposium on Parallel Distributed Processing. 2013.

[34] A. G. Dimakis et al. “Network coding for distributed storage systems”. In: *IEEE Transactions on Information Theory* 56.9 (2010), pp. 4539–4551.

[35] J. C. Duchi, A. Agarwal, and M. J. Wainwright. “Dual Averaging for Distributed Optimization: Convergence Analysis and Network Scaling”. In: *IEEE Transactions on Automatic Control* 57.3 (2012), pp. 592–606.

[36] R. M. Feldman and C. Valdez-Flores. *Applied probability and stochastic processes*. Springer Science & Business Media, 2009. Chap. 13.

[37] U. J. Ferner, M. Medard, and E. Soljanin. “Toward sustainable networking: Storage area networks with network coding”. In: *Proceedings of the 50th Annual Allerton Conference on Communication, Control, and Computing*. 2012, pp. 517–524.

[38] U. J. Ferner et al. “Scheduling advantages of network coded storage in point-to-multipoint networks”. In: *Proceedings of the International Symposium on Network Coding*. IEEE. 2014, pp. 1–6.

[39] T. Flach et al. “Reducing Web Latency: the Virtue of Gentle Aggression”. In: *ACM SIGCOMM*. 2013.

[40] J. Flynn. “Conditions for the equivalence of optimality criteria in dynamic programming”. In: *The Annals of Statistics* (1976), pp. 936–953.

[41] D. Ford et al. “Availability in Globally Distributed Storage Systems”. In: *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*. 2010.

[42] A. Gandhi et al. “Exact Analysis of the M/M/K/Setup Class of Markov Chains via Recursive Renewal Reward”. In: *SIGMETRICS Perform. Eval. Rev.* 41.1 (June 2013), pp. 153–166. ISSN: 0163-5999.

[43] K. Gardner et al. “Reducing Latency via Redundant Requests: Exact Analysis”. In: *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS ’15. Portland, Oregon, USA: ACM, 2015, pp. 347–360. ISBN: 978-1-4503-3486-0.

[44] P. Gopalan et al. “On the locality of codeword symbols”. In: *IEEE Transactions on Information Theory* 58.11 (2011), pp. 6925–6934.

[45] M. Gürbüzbalaban, A. Ozdaglar, and P. Parrilo. “Why Random Reshuffling Beats Stochastic Gradient Descent”. In: *arXiv preprint arXiv:1510.08560* (2015).

[46] D. Halperin et al. “Augmenting data center networks with multi-gigabit wireless links”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 41. 4. ACM. 2011, pp. 38–49.

[47] D. Han et al. “RPT: Re-architecting loss protection for content-aware networks”. In: *USENIX NSDI*. 2011.

[48] J. Han and L. A. Lastras-Montano. “Reliable memories with subline accesses”. In: *Proc. of IEEE International Symposium on Information Theory (ISIT)*. IEEE. 2007, pp. 2531–2535.

[49] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. 1st. New York, NY, USA: Cambridge University Press, 2013. ISBN: 1107027500, 9781107027503.

[50] P. Harrison and S. Zertal. “Queueing models with maxima of service times”. In: *Computer Performance Evaluation. Modelling Techniques and Tools*. Springer, 2003, pp. 152–168.

[51] *HDFS and Erasure Codes (HDFS-RAID)*. http://hadoopblog.blogspot.com/2009/08/hdfs-and-erasure-codes-hdfs-raid.html. Accessed: 2016-05-01. 2009.

[52] G. Hinton et al. “Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups”. In: *Signal Processing Magazine, IEEE* 29.6 (2012), pp. 82–97.

[53] C. Huang, M. Chen, and J. Li. “Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems”. In: *Proc. of the Sixth IEEE International Symposium on Network Computing and Applications (NCA)*. IEEE. 2007, pp. 79–86.

[54] C. Huang et al. “Erasure Coding in Windows Azure Storage”. In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC’12. Boston, MA: USENIX Association, 2012, pp. 2–2.

[55] L. Huang et al. “Codes can reduce queueing delay in data centers”. In: *Information Theory Proceedings (ISIT), 2012 IEEE International Symposium on*. 2012, pp. 2766–2770.

[56] S. Ioffe and C. Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *arXiv preprint arXiv:1502.03167* (2015).

[57] M. Ji, G. Caire, and A. F. Molisch. “Fundamental limits of distributed caching in D2D wireless networks”. In: *IEEE Information Theory Workshop (ITW)*. 2013, pp. 1–5.

[58] G. Joshi, Y. Liu, and E. Soljanin. “Coding for fast content download”. In: *Proceedings of the 50th Annual Allerton Conference on Communication, Control, and Computing*. 2012, pp. 326–333.

[59] G. Joshi, E. Soljanin, and G. Wornell. “Efficient Replication of Queued Tasks to Reduce Latency in Cloud Systems”. In: *53rd Annual Allerton Conference on Communication, Control, and Computing*. 2015.

[60] G. M. Kamath et al. "Codes with Local Regeneration". In: *arXiv preprint arXiv:1211.1932* (2012).

[61] N. Karamchandani et al. "Hierarchical coded caching". In: *Proc. of IEEE International Symposium on Information Theory (ISIT)*. 2014, pp. 2142–2146.

[62] G. Koole and R. Righter. "Resource allocation in grid computing". In: *Journal of Scheduling* 11.3 (2008), pp. 163–173.

[63] T. Kraska et al. "MLbase: A Distributed Machine-learning System". In: *Proc. of the Sixth Biennial Conference on Innovative Data Systems Research (CIDR)*. 2013.

[64] A. Kumar, R. Tandon, and T. Clancy. "On the Latency and Energy Efficiency of Distributed Storage Systems". In: *IEEE Transactions on Cloud Computing* PP.99 (2015), pp. 1–1. ISSN: 2168-7161.

[65] K. Lee et al. *Speeding Up Distributed Machine Learning Using Codes*. Presented at the Workshop on Machine Learning Systems at Neural Information Processing Systems (NIPS). 2015.

[66] K. Lee, R. Pedarsani, and K. Ramchandran. "On Scheduling Redundant Requests with Cancellation Overheads". In: *Proc. of the 53rd Annual Allerton conference on Communication, Control, and Computing*. Oct. 2015.

[67] M. Li et al. "Scaling Distributed Machine Learning with the Parameter Server". In: *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014, pp. 583–598.

[68] S. Li, M. A. Maddah-ali, and S. Avestimehr. *Coded MapReduce*. Presented at the 53rd Annual Allerton conference on Communication, Control, and Computing, Monticello, IL. 2015.

[69] G. Liang and U. Kozat. "FAST CLOUD: Pushing the Envelope on Delay Performance of Cloud Storage With Coding". In: *IEEE/ACM Transactions on Networking* 22.6 (2014), pp. 2012–2025. ISSN: 1063-6692.

[70] G. Liang and U. C. Kozat. "TOFEC: Achieving optimal throughput-delay trade-off of cloud storage using erasure codes". In: *Proc. of IEEE Conference on Computer Communications (INFOCOM)*. 2014, pp. 826–834.

[71] M. Lichman. *UCI Machine Learning Repository*. Accessed: 2015-08-01. 2013.

[72] B. Lipshitz et al. "Communication-avoiding parallel strassen: implementation and performance". In: *Proc. of High Performance Computing Networking, Storage and Analysis (SC)*. 2012, p. 101.

[73] J. D. C. Little. "A Proof for the Queuing Formula: $L = \lambda W$". In: *Operations Research* 9.3 (1961), pp. 383–387.

[74] Y. Low et al. "Distributed GraphLab: A Framework for Machine Learning in the Cloud". In: *PVLDB* 5.8 (2012), pp. 716–727.

[75] Y. Lu et al. “Network coding for data-retrieving in cloud storage systems”. In: *Proceedings of the International Symposium on Network Coding*. 2015, pp. 51–55.

[76] F. J. MacWilliams and N. J. A. Sloane. *The theory of error correcting codes*. Elsevier, 1977.

[77] M. A. Maddah-Ali and U. Niesen. “Fundamental limits of caching”. In: *IEEE Transactions on Information Theory* 60.5 (2014), pp. 2856–2867.

[78] M. A. Maddah-Ali and U. Niesen. “Decentralized Coded Caching Attains Order-Optimal Memory-Rate Tradeoff”. In: *IEEE/ACM Transactions on Networking* 23.4 (2015), pp. 1029–1040.

[79] X. Meng et al. “MLlib: Machine Learning in Apache Spark”. In: *CoRR* abs/1505.06807 (2015).

[80] A. Nedic and A. E. Ozdaglar. “Distributed Subgradient Methods for Multi-Agent Optimization”. In: *IEEE Transactions on Automatic Control* 54.1 (2009), pp. 48–61.

[81] F. Oggier and A. Datta. “Self-repairing homomorphic codes for distributed storage systems”. In: *Proc. of IEEE INFOCOM*. IEEE. 2011, pp. 1215–1223.

[82] *Open MPI: Open Source High Performance Computing*. http://www.open-mpi.org. Accessed: 2016-05-01.

[83] D. S. Papailiopoulos and A. G. Dimakis. “Locally repairable codes”. In: *Proc. of IEEE International Symposium on Information Theory (ISIT)*. IEEE. 2012, pp. 2771–2775.

[84] D. S. Papailiopoulos et al. “Simple regenerating codes: Network coding for cloud storage”. In: *Proc. of IEEE INFOCOM*. IEEE. 2012, pp. 2801–2805.

[85] D. S. Papailiopoulos, A. G. Dimakis, and V. R. Cadambe. “Repair optimal erasure codes through Hadamard designs”. In: *IEEE Transactions on Information Theory* 59.5 (2013), pp. 3021–3037.

[86] R. Pedarsani, M. A. Maddah-Ali, and U. Niesen. “Online coded caching”. In: *Proc. of IEEE International Conference on Communications (ICC)*. 2014, pp. 1878–1883.

[87] M. J. Pitkänen and J. Ott. “Redundancy and distributed caching in mobile dtns”. In: *ACM/IEEE MobiArch*. 2007.

[88] N. Prakash et al. “Optimal linear codes with a local-error-correction property”. In: *Proc. of IEEE International Symposium on Information Theory (ISIT)*. IEEE. 2012, pp. 2776–2780.

[89] K. V. Rashmi, N. B. Shah, and P. V. Kumar. “Optimal exact-regenerating codes for distributed storage at the MSR and MBR points via a product-matrix construction”. In: *IEEE Transactions on Information Theory* 57.8 (2011), pp. 5227–5239.

[90] K. V. Rashmi et al. “Regenerating Codes for Errors and Erasures in Distributed Storage”. In: *IEEE ISIT*. 2012.

[91] K. V. Rashmi et al. “A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster”. In: *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems*. HotStorage’13. San Jose, CA: USENIX Association, 2013, pp. 8–8.

[92] K. Rashmi et al. “Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-bandwidth”. In: *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. Santa Clara, CA: USENIX Association, Feb. 2015, pp. 81–94. ISBN: 978-1-931971-201.

[93] K. Rashmi et al. “A hitchhiker’s guide to fast and efficient data reconstruction in erasure-coded data centers”. In: *ACM SIGCOMM*. 2014.

[94] K. Rashmi et al. “Explicit construction of optimal exact regenerating codes for distributed storage”. In: *Proceedings of the 47th Annual Allerton Conference on Communication, Control, and Computing*. IEEE. 2009, pp. 1243–1249.

[95] A. S. Rawat et al. “Optimal locally repairable and secure codes for distributed storage systems”. In: *IEEE Transactions on Information Theory* 60.1 (2014), pp. 212–236.

[96] A. S. Rawat et al. “Locality and availability in distributed storage”. In: *IEEE ISIT*. 2014.

[97] A. S. Rawat et al. “On codes with availability for distributed storage”. In: *IEEE ISCCSP*. 2014.

[98] B. Recht and C. Ré. “Parallel stochastic gradient algorithms for large-scale matrix completion”. In: *Mathematical Programming Computation* 5.2 (2013), pp. 201–226.

[99] B. Recht et al. “Hogwild: A lock-free approach to parallelizing stochastic gradient descent”. In: *Proc. of the 25th Annual Conference on Neural Information Processing (NIPS)*. 2011, pp. 693–701.

[100] C. Ruemmler and J. Wilkes. “An introduction to disk drive modeling”. In: *Computer* 27.3 (1994), pp. 17–28. ISSN: 0018-9162.

[101] M. Sathiamoorthy et al. “XORing elephants: Novel erasure codes for big data”. In: *Proc. of the VLDB Endowment*. Vol. 6. 5. VLDB Endowment. 2013, pp. 325–336.

[102] P. Scheuermann, G. Weikum, and P. Zabback. “Data Partitioning and Load Balancing in Parallel Disk Systems”. In: *The VLDB Journal* 7.1 (Feb. 1998), pp. 48–66. ISSN: 1066-8888.

[103] N. B. Shah, K. Lee, and K. Ramchandran. “When Do Redundant Requests Reduce Latency?” In: *IEEE Transactions on Communications* 64.2 (2016), pp. 715–722. ISSN: 0090-6778.

[104] N. Shah, K. Lee, and K. Ramchandran. “When do redundant requests reduce latency?” In: *Proceedings of the 51st Annual Allerton Conference on Communication, Control, and Computing*. 2013, pp. 731–738.

[105] N. B. Shah, K. Lee, and K. Ramchandran. “The MDS Queue: Analysing the Latency Performance of Erasure Codes”. In: *IEEE ISIT*. 2014.

[106] E. Shriver. “Performance Modeling for Realistic Storage Devices”. UMI Order No. GAX97-31437. PhD thesis. New York, NY, USA, 1997.

[107] Q Shuai, V. Li, and Y Zhu. “Performance models of access latency in cloud storage systems”. In: *Proceedings of the 4th workshop on Architectures and Systems for Big Data (ASBD 2014)*. 2014.

[108] N. Silberstein, A. Singh Rawat, and S. Vishwanath. “Error resilience in distributed storage via rank-metric codes”. In: *CoRR* abs/1202.0800 (2012).

[109] D. Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529.7587 (2016), pp. 484–489.

[110] A. C. Snoeren, K. Conley, and D. K. Gifford. “Mesh-based content routing using XML”. In: *ACM SIGOPS Operating Systems Review*. Vol. 35. 5. 2001, pp. 160–173.

[111] E. R. Sparks et al. “MLI: An API for Distributed Machine Learning”. In: *Proc. of the IEEE 13th International Conference on Data Mining (ICDM)*. 2013, pp. 1187–1192.

[112] *StarCluster*. http://star.mit.edu/cluster/. Accessed: 2016-05-01.

[113] C. Stewart, A. Chakrabarti, and R. Griffith. “Zoolander: Efficiently meeting very strict, low-latency SLOs”. In: *USENIX ICAC*. 2013.

[114] C. Suh and K. Ramchandran. “Exact-repair MDS code construction using interference alignment”. In: *IEEE Transactions on Information Theory* 57.3 (2011), pp. 1425–1442.

[115] Y. Sun et al. “Provably delay efficient data retrieving in storage clouds”. In: *Computer Communications (INFOCOM), 2015 IEEE Conference on*. 2015, pp. 585–593.

[116] C. Szegedy et al. “Going deeper with convolutions”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 1–9.

[117] I. Tamo, Z. Wang, and J. Bruck. “MDS array codes with optimal rebuilding”. In: *Proc. of IEEE International Symposium on Information Theory (ISIT)*. 2011, pp. 1240–1244.

[118] I. Tamo, Z. Wang, and J. Bruck. “Access vs. bandwidth in codes for storage”. In: *Proceedings of IEEE International Symposium on Information Theory*. IEEE. 2012, pp. 1187–1191.

[119] C. Tian and T. Liu. “Multilevel diversity coding with regeneration”. In: *arXiv preprint arXiv:1503.00013* (2015).

[120] E. Varki, A. Merchant, and H. Chen. *The M/M/1 fork-join queue with variable sub-tasks*. 2008.

[121] S. Venkataraman et al. “The Power of Choice in Data-Aware Cluster Scheduling”. In: *USENIX OSDI*. 2014.

[122] A. Vulimiri et al. “More is less: reducing latency via redundancy”. In: *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. ACM. 2012, pp. 13–18.

[123] D. Wang. “Computing with unreliable resources: design, analysis and algorithms”. PhD thesis. Massachusetts Institute of Technology, 2014.

[124] D. Wang, G. Joshi, and G. Wornell. “Efficient Task Replication for Fast Response Times in Parallel Computation”. In: *SIGMETRICS Perform. Eval. Rev.* 42.1 (June 2014), pp. 599–600. ISSN: 0163-5999.

[125] G. Weikum, P. Zabback, and P. Scheuermann. “Dynamic File Allocation in Disk Arrays”. In: *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’91. Denver, Colorado, USA: ACM, 1991, pp. 406–415. ISBN: 0-89791-425-2.

[126] R. W. Wolff. “Poisson arrivals see time averages”. In: *Operations Research* 30.2 (1982), pp. 223–231.

[127] Y. Xiang et al. “Joint latency and cost optimization for erasurecoded data center storage”. In: *ACM SIGMETRICS Performance Evaluation Review* 42.2 (2014), pp. 3–14.

[128] M. Zaharia et al. “Spark: Cluster Computing with Working Sets”. In: *Proc. 2nd USENIX Workshop Hot Topics in Cloud Computing (HotCloud)*. 2010.

[129] M. Zaharia et al. “Improving MapReduce Performance in Heterogeneous Environments”. In: *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2008, pp. 29–42.

[130] C. Zhang and C. Re. “DimmWitted: A Study of Main-Memory Statistical Analytics”. In: *PVLDB* 7.12 (2014), pp. 1283–1294.

[131] Y. Zhu et al. “Cutting the cord: a robust wireless facilities network for data centers”. In: *Proc. 20th annual international conference on Mobile computing and networking*. ACM. 2014, pp. 581–592.

[132] M. Zinkevich et al. “Parallelized Stochastic Gradient Descent”. In: *Proc. of the 24th Annual Conference on Neural Information Processing (NIPS)*. 2010, pp. 2595–2603.