The MDS Queue: Analysing Latency Performance of Codes and Redundant Requests

Nihar B. Shah, Kangwook Lee, Kannan Ramchandran
Dept. of Electrical Engineering and Computer Sciences
University of California, Berkeley
{nihar, kw1jjang, kannanr}@eecs.berkeley.edu

Abstract

In order to scale economically, data centers are increasingly evolving their data storage methods from the use of simple data replication to the use of more powerful erasure codes, which provide the same level of reliability as replication-based methods at a significantly lower storage cost. In particular, it is well known that Maximum-Distance-Separable (MDS) codes, such as Reed-Solomon codes, provide the maximum storage efficiency. While the use of codes for providing improved reliability in archival storage systems, where the data is less frequently accessed (or so-called "cold data"), is well understood, the role of codes in the storage of more frequently accessed and active "hot data", where latency is the key metric, is less clear.

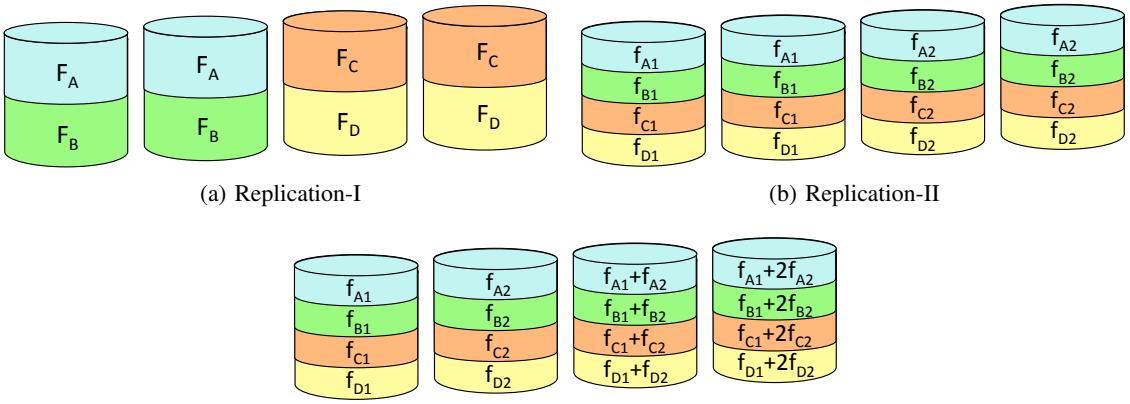
In this paper, we study data storage systems based on MDS codes through the lens of queueing theory, and term this the "MDS queue." We analytically characterize the latency performance of MDS queues, for which we present insightful scheduling policies that form upper and lower bounds to performance, and show that they are quite tight. Extensive simulations using Monte Carlo methods are also provided and used to validate our theoretical analysis. As a side note, our lower-bound analytical method based on the so-called MDS-Reservation(t) queue, represents an elegant practical scheme that requires the maintenance of considerably smaller state, depending on the parameter t, than that of the full-fledged MDS queue (which corresponds to $t = \infty$), and may be of independent interest in practical systems. Comparisons with replication-based systems reveal that codes provide a superior latency-performance (by up to 70%) than replication.

The second part of the paper considers an alternative method of (potentially) reducing latency in data centers, that of sending *redundant requests*. Here, a request is sent to more servers than required, and is deemed served when any requisite number of servers complete service. Several recent works provide empirical evidence of the benefits of redundant requests in various settings, and in this paper, we aim to analytically characterize the situations when can redundant requests actually help. We show that under the MDS queue model (with exponential service times and negligible costs of cancelling jobs), in a replication-based system, the average latency strictly reduces with more redundancy in the requests, and that under a general MDS code, the average latency is minimized when requests are sent to all servers. To the best of our knowledge, these are the first analytical results that prove the benefits of sending redundant requests.

I. Introduction

Two of the 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 storage codes [1]–[4]. While the reliability properties of erasure codes are well understood, much less is known about their latency performance.

In systems that possess the flexibility to serve requests in more than one way (e.g., by reading data from any one of multiple servers where it is stored), latency can (potentially) be reduced by sending redundant requests. Here, a request is sent to an excess number of servers, and is deemed served when the requisite number of servers



(a) Replication-I (b) Replication-II

(c) MDS Code

Fig. 1: An example comparing two storage schemes (a) and (b) based on replication and a storage scheme (c) based on MDS codes. The data to be stored consists of four files F_A, F_B, F_C and F_D. In schemes (b) and (c), each of these files are split into two halves as $F_x = \{f_{x1}, f_{x2}\}$, for $x \in \{A,B,C,D\}$, and 'half' of each file is stored in each server. Observe that in the coded system, any file an be recovered by reading data from *any* two of the four servers.

complete service. Such a mechanism trades off the increase in speed of service of any individual request with an increase in queueing delay due to use of additional resources. Several recent works report empirical observations on latency-benefits of redundant requests under various settings. However, to the best of our knowledge, no analytical characterization of the situations in which redundant requests will help is known.

In this paper, we first provide an analysis of the latency performance of erasure-coded systems. The queueing theoretic model resulting out of the use of erasure codes is termed the 'MDS Queue'. We build on previous work by Huang et al. [5] in this setting, and provide an analysis of a much more general setting and bounds on the latency-performance that are significantly tighter than those from [5].

We then employ this model to study the latency-performance of redundant requests, with an aim of characterizing situations when sending redundant requests helps (and it does not help). In this paper, we show that when service times are memoryless and independent and when cancelling jobs incurs negligible costs, redundant requests necessarily reduce the latency. These results are applicable to replication-based systems (where any one of a set of servers can serve the request) and also to data-centers that employ erasure codes.

We shall now present a brief introduction to MDS codes, along with a comparison with replication. Fig. 1 provides an illustrative example comparing the replication and coded schemes. Here, four files F_A, F_B, F_C and F_D, are to be stored across four servers. One replication strategy (called replication I), as depicted in Fig. 1a, stores $\{F_A, F_B\}$ in the first two servers and $\{F_C, F_D\}$ in the two remaining servers. An alternative strategy of replication (called replication II) is depicted in Fig. 1b wherein each file is first partitioned into two halves as $F_x = [f_{x1} \ f_{x2}]$ for $x \in \{A,B,C,D\}$, and each server stores one 'half' of each file. The strategy of coding as shown in Fig. 1c, also partitions each file into two halves, and stores the four sets $\{f_{x1}\}$, $\{f_{x2}\}$, $\{f_{x1}+f_{x2}\}$, and $\{f_{x1}+2f_{x2}\}$ in four different servers. Observe that the total storage space required under each of these schemes is the same. However, the coded scheme is provides a higher reliability than the replication-based systems. To see this, observe that either replication scheme would lose some data on the failure of the first or the last two servers (e.g., replication-I loses data F_A and F_B if the first two servers fail). On the other hand, the coding scheme loses no data even upon failure of *any* two of the four servers. While this was only a toy example, the amount of gains offered by codes are significantly higher under more general settings. Another disadvantage of replication-based schemes is that any redundancy scheme based on replication must store at least one additional copy of the entire data, which necessitates a storage overhead of at least 100%. This may itself be prohibitive in many cases. On the other hand, codes do not face such a barrier and can support smaller overheads.

The most popular, and also most efficient storage codes are a class of codes known as Maximum-Distance-

Separable (MDS) codes [6]. 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] The Reed-Solomon code [7] is an example of an MDS code, and so is the code depicted in Fig. 1c.

Note that the replication-I scheme can be considered as a special case of an MDS code, corresponding to the parameter $k=1$. For instance, restricting our attention to only the first two servers in Fig. 1a gives a $(n=2,\ k=1)$ MDS code

A. Definition of the MDS queue

The use of an MDS code for storing the data leads to a simple queueing-theoretic model, which we term the 'MDS Queue'. To arrive at this model, some simplifying assumptions are made in the interests of conceptual simplicity. It is assumed that there is homogeneity among files, among requests, and among servers: the files are of identical size, incoming requests are distributed as a Poisson process independent of the state of the system, and every incoming request asks for reading out one particular file with the reading process at every server following an independent and identical exponential distribution. [2]

We note that the model described in this section does *not* consider sending redundant requests, and this extension is provided subsequently in Section VII.

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 the reads of k parts of the file from k servers. A batch is considered as served when all k of its jobs have completed service. For instance, a request for reading file F_A in the system depicted in Fig. 1c is treated as a batch of two jobs. To service 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_{A2} and $(f_{A1}+f_{A2})$ respectively, which suffices to obtain the desired file F_A. This paper thus formalizes the setting considered in [5], and generalizes the work from $k=2$ to a general k.

Definition 1 (MDS queue): An MDS queue is associated to four parameters $(n,\ k)$ and $[\lambda,\ \mu]$.

- There are n identical servers
- Requests enter into a (common) buffer of infinite capacity
- Requests arrive in *batches* of k *jobs* each
- These batches arrive as a Poisson process with a rate of λ
- 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 a rate of μ, and is independent of the arrival and service times of all other jobs
- 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.

The scheduling policy that governs this queue is formalized in Algorithm 1.

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

Example 1: Consider the MDS(n=4, k=2) queue, as depicted in Fig. 2. Here, each request comes as a batch of $k=2$ jobs, and hence we denote each batch (e.g., A, B, C, etc.) as a pair of jobs (e.g., $\{A_1,A_2\}$, $\{B_1,B_2\}$, $\{C_1,C_2\}$, 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 the system is in the state as shown in Fig. 2a, wherein the

[1] A more generic definition of an MDS code is that it is a code that satisfies the 'Singleton bound' [6].

[2] While the service times in practice are unlikely to be distributed exponentially, such an assumption is meant to serve as a starting point for more rigorous theoretical analyses.

Algorithm 1 MDS scheduling policy

on arrival of batch
  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 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 **then**
    among all such batches, find the batch that had arrived earliest
    assign a job from this batch to s
  end if

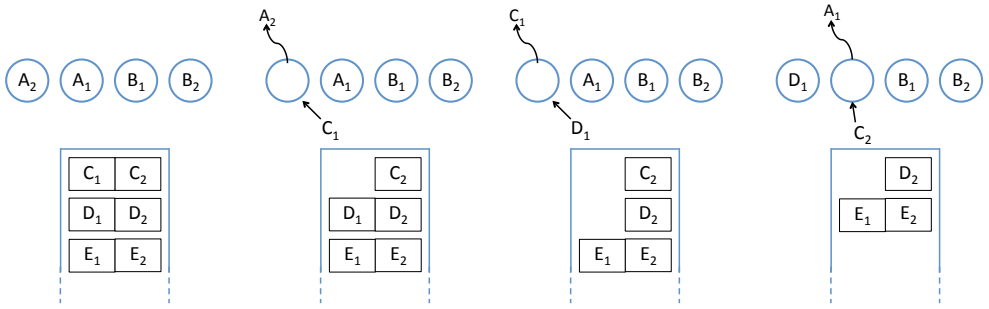


Fig. 2: Illustration of the functioning of the MDS queue. Requests take the form of batches, A, B, C etc., each comprising of $k=2$ jobs $\{A_1,A_2\}$, $\{B_1,B_2\}$, $\{C_1,C_2\}$, etc. The two jobs of any batch can be served by any two distinct servers.

jobs A_2, A_1, B_1 and B_2 are being processed by the four servers, and there are three more batches waiting in the buffer. Suppose server 1 completes servicing job A_2 (Fig. 2b). Now, this server is free to serve any of the 6 jobs waiting in the buffer. However, since we allow jobs to be processed only in order, server 1 begins servicing job C_1 (assignment of C_2 instead would also have been valid). Next, suppose the first server completes service of C_1 before any other servers complete their current tasks (Fig. 2c). In this case, since server 1 has already served a job from the $\{C_1,C_2\}$ batch, it is not allowed to service C_2 (due to the restriction in the 5^{th} bullet of the definition of the MDS queue). 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, then that server is assigned to service C_2 (Fig. 2d).

As noted previously, the replication-I scheme forms a special case corresponding to $k=1$, i.e., the request can be served by any one of the n servers. This special case is simply the M/M/n queue. However, the MDS queue for a general value of k differs from the M/M/n queue in two ways. In the MDS(n,k) queue,

- jobs arrive in batches of k, and
- each job in a batch must be served by a different server.

The extension provided in Section VII incorporating redundant requests has an additional parameter ν $(k \leq \nu \leq n)$ which we term the request-degree. In this setting, a batch is now viewed as comprising ν jobs, that can be served by upto ν arbitrary servers. The batch is considered as served whenever any k of the ν jobs of the batch complete service. We shall not consider the request-degree ν in the paper until Section VII.

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 completed processing, and the number 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

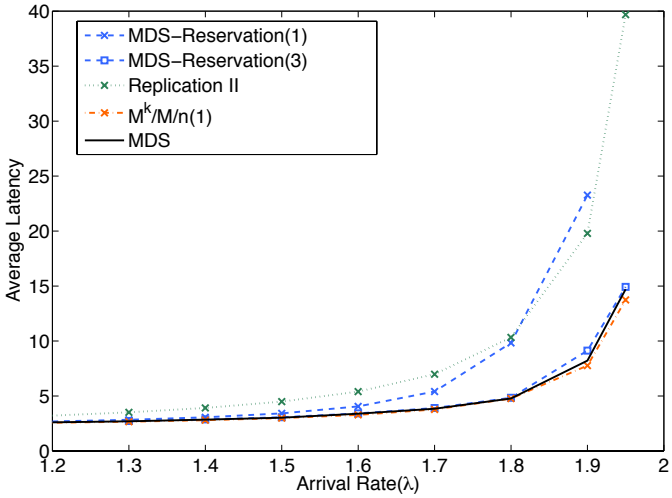


Fig. 3: Average latency under of a storage system employing an MDS code with parameters $n=10$, $k=5$, and with service at each server exponentially distributed with a rate $\mu=1$. The curve titled 'MDS' is the latency under the exact system, obtained via a Monte Carlo simulations. Alongside, analytically-derived performance measures of the lower bounds (MDS-Reservation(t) queues) and upper bounds (M^k/M/n(t) queues) presented in this paper are also plotted. A simulation of the latency performance under replication-II is also provided for comparison. We note that the special case of MDS-Reservation(1) was previously proposed in [5], and was analysed therein for $k=2$.

k dimensions. Furthermore, the transitions along different dimensions are tightly coupled. This makes the Markov chain hard to analyse, and in this paper, we provide scheduling policies for the MDS queue that lower/upper bound its exact performance.

B. Analysis of latency of codes

In this paper, we provide scheduling policies to bound the performance of the MDS queue. Analysing these scheduling policies in terms of several performance metrics, we provide tight bounds on the performance of the MDS queue. In addition, we also provide Monte Carlo (MC) simulations for the MDS queue, which are used to validate our analytical results, and also target other metrics (e.g., tails of the distribution of the latency) that are hard to evaluate otherwise.

The lower bounds (the *MDS-Reservation(t)* scheduling policies) and the upper bounds (the *M^k/M/n(t)* scheduling policies) are both indexed by a parameter 't'. While these policies will be discussed in detail later in the paper, Fig. 3 here gives a brief preview, depicting the average latency faced by a batch under these policies. An increase in the value of t results in tighter bounds (as seen in the figure), but also increases the complexity of analysis. Furthermore, both classes of scheduling policies converge to the (exact) MDS scheduling policy as $t\to\infty$. We note that the MDS-Reservation(t) scheduling policies presented here are themselves a practical alternative to the MDS scheduling policy, since they require maintenance of a smaller state, while offering highly comparable performance. While the performance as well as the complexity of the MDS-Reservation(t) queue increases with an increase in the value of the parameter t, we observe that its performance is very close to that of the MDS queue for very small values of t (as small as $t=3$). Likewise, the performance of the upper bounds closely follow that of the MDS queue for values of t as small as $t=1$.

Our analysis and simulations also reveal that codes can achieve significant gains over replication based systems, under the models considered in this paper. For example, the average latency incurred by a read-request in a coded system may enjoy upto a 70% reduction in latency on an average as compared to replication-II, as shown in Fig. 3; the 99^{th} percentile tails of the latency may reduce by upto 50%, as shown later. The key insight is that the property of being able to recover the data from *any* k of the servers that lends MDS codes a high storage efficiency, also allows for greater parallel access, and is thus instrumental in providing it a low latency.

In order to provide an apples-to-apples comparison, we compare the performance of codes with replication-II alone (and not with replication-I) since the storage space allocation for the files in each server are the same in the coded and replication-II schemes (but are different from that under replication-I).

We also employ the analytical and the simulation framework presented in this paper to model and compare various methods of performing *degraded reads* in distributed storage systems, wherein any requests meant for a busy or failed node must be served by downloading data from other nodes.

To allow the reader to evaluate performance of these queues under any parameters of choice, the source code of the MC simulations as well as those for computing steady-state distributions and performance metrics of the proposed queues have been made freely downloadable from the websites of the first two authors.

C. Latency gains through redundant requests

A mechanism of (potentially) reducing the latency, that is gaining increasing popularity in practice [8], [9], is that of introducing redundancy in the requests. Specifically, under such a mechanism, a request is sent to more servers than the minimum required, and is deemed served when any targeted number of servers complete service. The services at the remaining servers are cancelled at this point. This idea is applicable to any system that possesses redundancy (e.g., redundancy via replication or codes in a distributed storage setting).

Under such a mechanism, processing the jobs at multiple servers in parallel contributes to the reduction in the service time, while on the other hand, the consumption of a greater amount of resources by each request contributes to an increase in the queueing delay. The overall latency under such a system is thus governed by the tradeoff between these two factors. The question that thus arises is to determine under any given system conditions, the optimal number of servers that a request must be (redundantly) sent to.

This paper answers this question in the realm of the MDS queue framework, when the service times are memoryless and jobs can be cancelled with a negligible cost. It is shown that in a replication-based system, the average latency strictly reduces with increase in the redundancy in the requests. We further show that under a general MDS code, the average latency is minimized when requests are sent to all servers. To the best of our knowledge, these are the first analytical results to prove the benefit of sending redundant requests. Finally, simulations evaluating redundant request policies under more general settings are also presented.

D. Organization of the paper

The rest of this paper is organized as follows. Section II provides a background on this problem and presents related literature. Sections III to VI contain the first part of the paper, wherein the latency of MDS coded systems is analysed. Section III describes the general approach and the notation followed in the paper for analysing our scheduling policies. Section IV presents the MDS-Reservation(t) queues, that lower bound the performance of the MDS queue. Section V presents the M^k/M/n(t) queues, that are upper bounds to the performance of the MDS queue. Section VI presents an analysis and comparison of each of these queues, and queues resulting from replication-based storage schemes. The second part of the paper is presented in Section VII. In this section, analytical and simulation-based results on the effects of redundant requests are provided. The paper concludes with a discussion in Section VIII. The appendix contains proofs of the theorems presented in the paper.

The reader interested only in the analysis of redundant requests may jump directly to Section VII without any loss in continuity.

II. Related Literature

A. Latency and storage efficiency in data-centers

1) Latency analysis: The study of latency comparisons between replication and coded systems was initiated by Huang et al. in [5], which we build upon in this paper. They presented a scheduling policy termed the 'block-one-scheduling' policy, that provides a lower bound on the performance of the MDS queue. They analysed this policy

for $k=2$ and showed that this scheme improves the average latency faced by a job by upto 17% as compared to the replication scheme. The block-one-scheduling policy is a special case of the MDS-Reservation(t) queues presented in this paper, and corresponds to the case when t$=1$. While the block-one-scheduling policy was analysed for $k=2$ in [5], the analysis in this paper presents applies to any value of $(n,\ k)$, and recovers the corresponding results of [5] as a special case. In addition, the MDS-Reservation(t) queues of this paper, when t≥ 2, provide significantly tighter bounds to the performance of the MDS queue, as seen in Fig. 3.

2) Blocking probability: The blocking probability of the MDS queue in the absence of a buffer was previously studied in [10]. A theoretical analysis of the blocking probability was presented in [10] for arbitrary service time distributions. In this paper, as a special case of the MDS queue analysis, we present simulations of the blocking probabilities for the case when there is no buffer as well as for the case when there is a buffer with a finite capacity, when the service times are exponentially distributed.

3) Degraded Reads: The coded system presented so far assumes that each incoming request desires one complete file F_x, for some $x\in\{A,B,C,\ldots\}$. However, in certain applications, some incoming requests may require only a part of the file, say, $f_{x\ell}$ for some $\ell\in\{1,\ldots,k\}$. Let us assume that the code employed is systematic, and servers $1,2,\ldots,k$ respectively store the data $f_{x1},f_{x2},\ldots,f_{xk}$. In this case, a request for $f_{x\ell}$ can be served by reading $f_{x\ell}$ directly from the ℓ^{th} server. In the event that server ℓ is busy or unavailable, $f_{x\ell}$ will have to be recovered from the data stored in the remaining $(n-1)$ servers. Such an operation is called a 'degraded read'.

Under any MDS code, a degraded read may be performed by downloading the data stored in any k of the remaining $(n-1)$ servers to obtain F_x, and then extracting the desired chunk $f_{x\ell}$ from F_x. 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., [11]–[14]. In particular, the practical *product-matrix codes* proposed in [12] are associated to an additional parameter d, such that any part $f_{x\ell}$ can be recovered by reading and downloading small fractions of the data stored in *any* d of the remaining $(n-1)$ servers. In this paper, we employ our queueing theoretic framework to analyse and compare the latency-performance of these two methods of performing degraded reads.

B. Diversity and error-correction

The MDS queue also arises in other applications that require diversity or error correction. For instance, consider a system with n processors, with jobs arriving as a Poisson process. It is often the case that the processors are not completely reliable [15], and may give incorrect outputs at random times. In order to guarantee a correct output, a job may be processed at k different servers, and the results aggregated (perhaps by a majority rule) to obtain the final answer. Such a system results precisely in an MDS(n,k) queue. In general, queues where jobs require diversity, for purposes such as security, error-protection etc., may be modelled as an MDS queue. [3]

C. Fork-join queues for parallel processing

A class of queues that are closely related to the MDS queue are the *fork-join queues* [16], [17]. A fork-join queue is used to model systems wherein jobs require multiple resources that can be provided in parallel. These queues are similar to the MDS queue in the sense that in both queues, jobs arrive in batches of k, and each job needs to be served by one distinct server out of the n servers. However, the distinction between these two classes of queues is that under a fork-join queue, each job must be served by a particular *pre-specified* server, while under an MDS queue, the k servers serving the jobs can be chosen by the scheduling policy. It thus follows that the performance of an MDS(n,k) queue is lower bounded by that of an (n,k) fork-join queue, with the gap quantifying the gains due to the flexibility in scheduling. Furthermore, as a special case, the MDS(n,n) queue is identical to the the (n,n) fork-join queue.

[3] 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.

D. Redundant Requests

Policies that try to reduce latency by sending redundant requests have been previously implemented in [8], [9], [18]. These works show that for several diverse applications, sending redundant requests indeed helps. However, to the best of our knoledge, there has been no rigorous theoretical characterization of the settings where sending redundant requests helps (and where it does not help).

The two theoretical results that we know of in this context are [19] and [18]. In [19], Joshi et al. provide bounds on the average latency faced by a batch in the steady state when the requests are sent (redundantly) to *all* n servers. However, no comparisons are made with other schemes of redundant requests, including the scheme of having no redundancy in the requests, thus making it unclear whether redundant requesting actually helps. Our work can be considered as complementary to that of [19], since we complete this picture by proving that under the models considered, sending (redundant) requests to all n servers is indeed an optimal choice. In [18], Liang and Kozat run practical experiments on Amazon EC2 servers, and present empirical evidence of redundant requests helping reduce the latency. They also provide an approximate analysis of the system described in Section I-A. However, no measures of the tightness of these approximations are provided, and neither is it known whether these approximations lead to any bounds on the actual performance, due to which it remained unclear as to whether redundant requests are provably beneficial in the system models originally considered.

III. Our Approach and Notation for Latency Analysis of The MDS Queue

For each of the scheduling policies presented in this paper (that lower/upper bound 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 the average latency, system occupancy, etc.

Throughout the paper, we shall refer to the entire setup described in Section I-A as the 'queue' or the 'system'. We shall 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 shall 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. 2a, 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 shall 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 functions, we shall assume that no two events occur at exactly the same time. We shall 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 may 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

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

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 5, 7 and 10 in the sequel also present examples of QBD processes.

QBD processes are very well understood [20], and their stationary distribution is fairly easy to compute. In this paper, we employ the *SMCSolver* software package [21] 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 them easy to analyse, and this is exploited subsequently in the analysis presented in Section VI.

IV. The MDS-Reservation(t) Queues: Lower Bounds

We now present a class of scheduling policies (and the 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. We shall see subsequently that the MDS-Reservation(t) scheduling policies are practical policies that require maintenance of a much smaller state as compared to the MDS queue, and hence may be of independent interest. This class of scheduling policies are indexed by a parameter 't': a higher value of t leads to a better performance and a tighter lower bound to the MDS queue, but on the downside, requires maintenance of a larger state and is also more complex to analyse.

The MDS-Reservation(t) scheduling policy, in a nutshell, is 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.

A. MDS-Reservation(0)

1) 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.

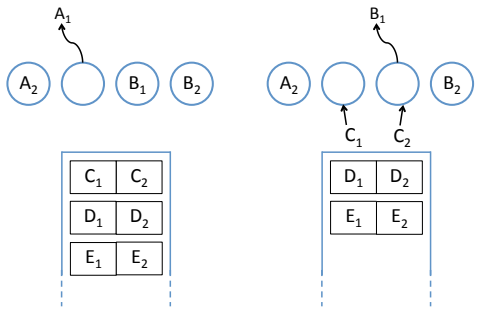


Fig. 4: An illustration of the MDS-Reservation(0) scheduling policy for a system with parameters $(n=4,k=2)$. This policy prohibits the servers to process jobs from a batch unless there are k idle servers that can process all k jobs of that batch. As shown in the figure, server 1 is barred from processing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

Algorithm 2 MDS-Reservation(0) Scheduling Policy

on arrival of a batch
  if number of idle servers $<k$ **then**
    append new batch at end of buffer
  else
    assign k jobs of the batch to any k idle servers
  end if
on departure from server
  if (number of idle servers $\geq k$) and (buffer is non-empty) **then**
    assign k jobs of the first waiting batch to any k idle servers
  end if

Example 2: Consider the MDS(n=4,k=2) queue in the state depicted in Fig. 2a. Suppose the server 2 completes processing job A_1 (Fig. 4a). 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 the third server completes execution of B_1 (Fig. 4b). 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 [22].

2) 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.

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.

Theorem 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. 5.

Theorem 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, this queue is simple enough to analyse analytically as well. Let $y(m)$ denote the number of jobs being served when the Markov chain is in state m. From the description above,

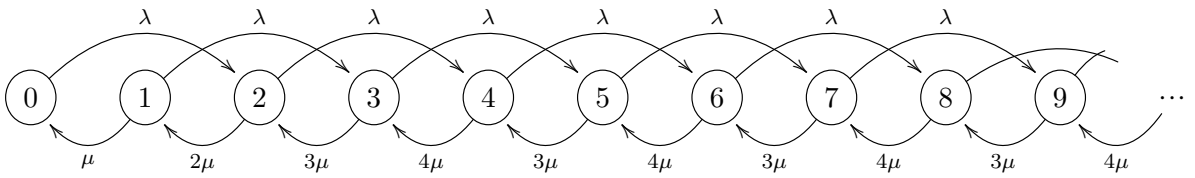


Fig. 5: State transition diagram of the MDS-Reservation(0) queue for $n=4$ and $k=2$. The notation at any state is the number of jobs m in the system in that state. The set of boundary states are $\{0,1,2\}$, and the levels are pairs of states $\{3,4\}$, $\{5,6\}$, $\{7,8\}$, etc. The transition matrix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

this function can be written as:

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

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:

$$\pi_m=\frac{\lambda}{y(m)\mu}\left(\sum_{j=(m-k)^+}^{m-1}\pi_j\right) \quad \forall \; m>0. \tag{1}$$

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

B. MDS-Reservation(1)

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 **then**
    assign one job each from new batch to (at most k) idle servers
  end if
  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 **then**
    assign a job from first waiting batch to s
    if first waiting batch had only one job in buffer & there exists another waiting batch **then**
      to every remaining idle server, assign a job from second waiting batch
    end if
  end if

Example 3: Consider the MDS(n=4,k=2) queue in the state depicted in Fig. 2a. Suppose server 2 completes processing job A_1 (Fig. 6a). Under MDS-Reservation(1), server 2 now begins service of job C_1 (which is allowed by MDS, but was prohibited under MDS-Reservation(0)). Now suppose that server 2 finishes this service before any other server (Fig. 6b). 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

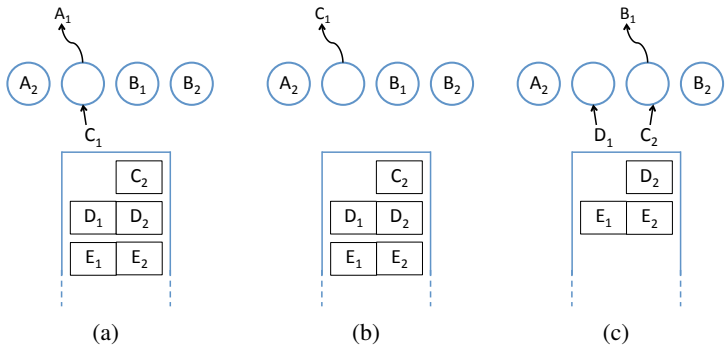


Fig. 6: An illustration of the MDS-Reservation(1) scheduling policy, for a system with parameters $(n=4,k=2)$. As shown in the figure, this policy prohibits the servers from processing jobs of the second or later batches (e.g., $\{D_1,D_2\}$ and E_1,E_2 in (b)), until they move to the top of the buffer (e.g., $\{D_1,D_2\}$ in (c)).

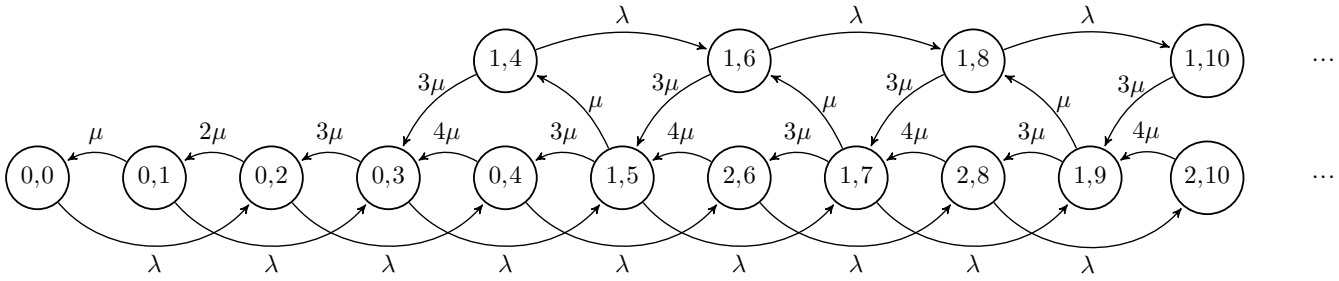


Fig. 7: State transition diagram of the MDS-Reservation(1) queue for $n=4$ and $k=2$. The notation at any state is (w_1,m). The subset of states that are never visited are not shown. The set of boundary states are $\{0,1,2,3,4\}\times\{0,1,2\}$, and the levels are sets $\{5,6\}\times\{0,1,2\}$, $\{7,8\}\times\{0,1,2\}$,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

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. 6c), 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.

The MDS-Reservation(1) scheduling policy is identical to the block-one-Scheduling policy proposed in [5]. While this scheme was analysed in [5] only for the case of $k=2$, in this paper, we present a more general analysis that holds for all values of the parameter k.

2) Analysis: The following theorem 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.

Theorem 2: The Markovian representation of the MDS-Reservation(1) queue has a state space $\{0,1,\ldots,k\}\times\{0,1,\ldots,\infty\}$. 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. 7.

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.

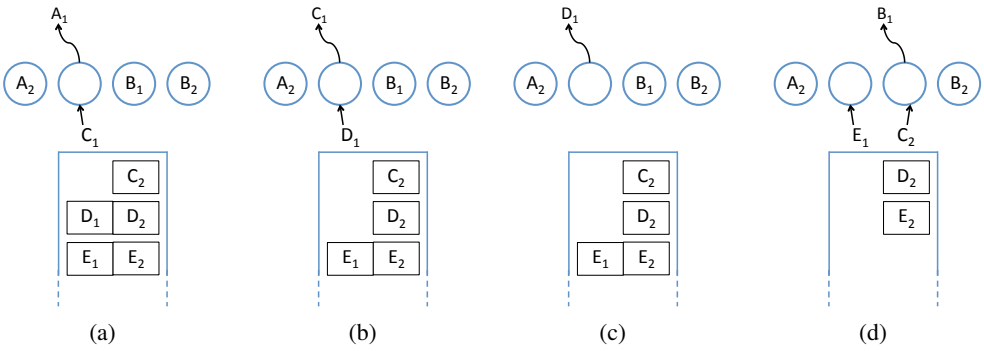


Fig. 8: An illustration of the working of the MDS-Reservation(2) scheduling policy, for a system with parameters $(n=4,k=2)$. As shown in the figure, this policy prohibits the servers from processing jobs of the third and later batches (e.g., batch $\{E_1,E_2\}$ in (c)), until they move higher in the buffer (e.g., as in (d)).

C. MDS-Reservation(t) for a general t

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

Algorithm 4 MDS-Reservation(t)

on arrival of a batch:
  if $\exists$ $<t$ batches in the buffer **then**
    assign 1 job each from new batch to every idle server
  end if
  append remaining jobs of batch to the end of the buffer
on departure from server (say, server s):
  find $\hat{i}=\min\{i\geq 1$:no job of the i^{th} waiting batch has been served by $s\}$
  if $\hat{i}$ exists & $\hat{i}\leq t$ **then**
    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 buffer & there exists another waiting batch **then**
      to every remaining idle server, assign a job from $(t+1)^{\text{th}}$ waiting batch
    end if
  end if

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

Example 4: (t=2). Consider the MDS(n=4,k=2) queue in the state depicted in Fig. 2a. Suppose the second server completes processing job A_1 (Fig. 8a). 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. 8b). 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. 8c), 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. 8d). 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.

2) Analysis:

Theorem 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 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.

Theorem 4: The MDS-Reservation(t) queue, when $t = \infty$, is precisely the MDS queue.

V. The $M^k/M/n(t)$ Queues: upper bounds

In this section, we present a class of scheduling policies (and resulting queues), which we call the M^k/M/n(t) scheduling policies (and M^k/M/n(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 M^k/M/n(t) scheduling policies and the M^k/M/n(t) queues are not realizable in practice, they are presented here only to obtain upper bounds to the performance of the MDS queue.

The M^k/M/n(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 theM^k/M/n(0) queue in detail, before moving on to the general M^k/M/n(t) queues.

A. M^k/M/n(0)

1) Scheduling policy: The M^k/M/n(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 M^k/M/n(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 then
      assign a job from the first waiting batch to this server
  end if
```

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

The following example illustrates the working of the M^k/M/n(0) scheduling policy.

Example 5: Consider the MDS(n=4,k=2) queue in the state depicted in Fig. 2a. Suppose the first server completes processing job A_2, as shown in Fig. 9a. Under the M^k/M/n(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. 9b). 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 M^k/M/n(0), we relax this restriction, and permit server 1 to begin processing C_2.

2) Analysis: We now describe a Markovian representation of the M^k/M/n(0) scheduling policy, and show that it suffices to keep track of only the total number of jobs m in the system.

Theorem 5: The Markovian representation of the M^k/M/n(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 $j = \{1,2,\ldots,\infty\}$.

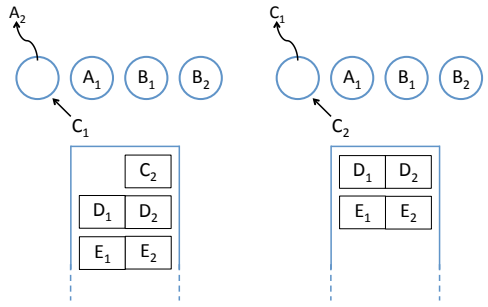


Fig. 9: Illustration of the working of the M^k/M/n(0) scheduling policy. This policy allows a server to process more than one jobs of the same batch. As shown in the figure, server 1 processes both C_1 and C_2.

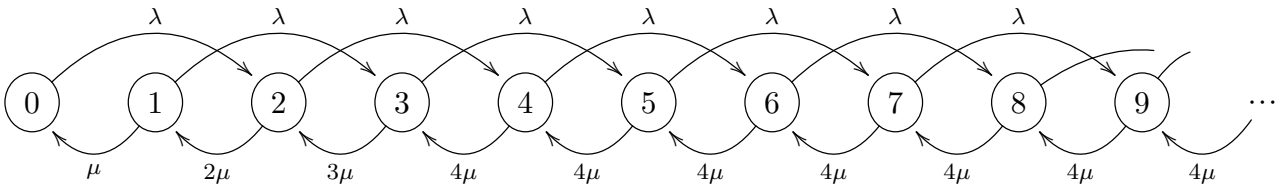


Fig. 10: State transition diagram of the M^k/M/n(0) queue for $n=4$ and $k=2$. The notation at any state is the number of jobs m in the system in that state. The set of boundary states are $\{0,1,2,3,4\}$, and the levels are pairs of states $\{5,6\}$, $\{7,8\}$, etc. The transition matrix is of the form (III) with $B_0=[0\ \ 0\ \ 0\ \ 0\ \ 4\mu\ ;0\ \ 0\ \ 0\ \ 0\ \ 0]$, $B_1=[-\lambda\ \ 0\ \ \lambda\ \ 0\ \ 0;\ \ \mu\ \ -(\mu+\lambda)\ \ 0\ \ \lambda\ \ 0;\ \ 0\ \ 2\mu\ \ -(2\mu+\lambda)\ \ 0\ \ \lambda;\ \ 0\ \ 0\ \ 3\mu\ \ -(3\mu+\lambda)\ \ 0;\ \ 0\ \ 0\ \ 0\ \ 4\mu\ \ -(4\mu+\lambda)]$, $B_2=[0\ \ 0\ ;\ \ 0\ \ 0\ ;\ \ 0\ \ 0\ ;\ \ \lambda\ \ 0\ ;\ \ 0\ \ \lambda]$, $A_0=[0\ \ 4\mu\ ;\ \ 0\ \ 0]$, $A_1=[-(4\mu+\lambda)\ \ 0\ ;\ \ 4\mu\ \ -(4\mu+\lambda)]$, $A_2=[\lambda\ \ 0\ ;\ \ 0\ \ \lambda]$.

The state transition diagram of the M^k/M/n(0) queue for $n=4,k=2$ is shown in Fig. 10.

Theorem 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, this queue is simple enough to analyse analytically 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:

$$\pi_m=\frac{\lambda}{\min(m,n)\mu}\sum_{j=(m-k)^+}^{m-1}\pi_j\ . \tag{2}$$

The stationary distribution of the Markov chain can now be computed easily from these equations.

B. M^k/M/n(t) for a general t

1) Scheduling policy: Algorithm 6 formally describes the M^k/M/n(t) scheduling policy.

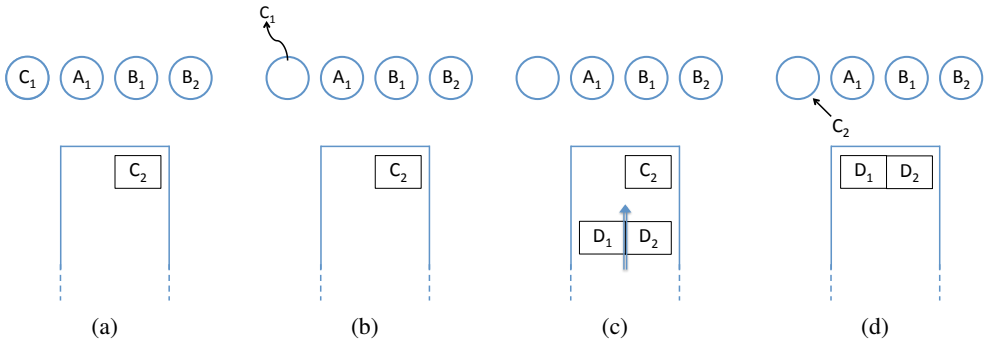


Fig. 11: Illustration of the working of the M^k/M/n(1) scheduling policy. This policy allows a server to begin processing a job of a batch that it has already served, unless this batch is the only batch waiting in the buffer. As shown in the figure, server 1 cannot process C_2 in (b) since it has already 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).

Algorithm 6 M^k/M/n(t)

on arrival of a batch
  if buffer has strictly fewer than t batches **then**
    assign jobs of new batch to idle servers
  else if buffer has t batches **then**
    assign jobs of first batch to idle servers
    if first batch is cleared **then**
      assign jobs of new batch to idle servers
    end if
  end if
  append remaining jobs of the new batch to the end of the buffer
on departure of job from a server (say, server s)
  if number of batches in buffer is strictly greater than t **then**
    assign job from first batch in buffer to this server
  else
    among all batches in the buffer that have not been served by s, find the one that arrived earliest
    assign a job of this batch to s
  end if

Example 6: (t=1). Consider a system in the state shown in Fig. 11a. Suppose server 1 completes execution of job C_1 (Fig. 11b). In this situation, the processing of C_2 by server 1 would be allowed under M^k/M/n(0), but prohibited in the MDS queue. The M^k/M/n(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. 11c). At this point there are two batches in the buffer, and the M^k/M/n(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. 11d).

2) Analysis:

Theorem 6: The state transition diagram of the M^k/M/n(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 6 also describes how one can obtain the configuration of the entire system from only the number of jobs in the system, under the M^k/M/n(t) scheduling policies.

As in the case of MDS-Reservation(t) queues, one can see that the sequence of M^k/M/n(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.

Theorem 7: The M^k/M/n(t) queue, when $t=\infty$, is precisely the MDS(n,k) queue.

Remark 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 M^k/M/(n-k+1)(t) queue for all values of t.

VI. Performance Analysis and Comparison of Various Queues

In this section we analyse the performance of the MDS-Reservation(t) queues and the M^k/M/n(t) queues using the properties of QBD processes. These form lower and upper bounds respectively to the performance of the MDS queue. We also provide a performance analysis of the MDS queue via MC simulations. We compare the performance of these queues with replication-based schemes.

A replication-based schaeme stores multiple copies of the entire data, and hence must have n as a multiple of k. As in systems employing MDS codes, we assume that under a replication-based scheme, each server has a capacity of storing $\frac{1}{k}$ of the total data. As seen previously in Fig. 1, replication can be of two types:

- Replication-I (depicted in Fig. 1a): The n servers are partitioned into a k sets of $\frac{n}{k}$ servers each, and the set of files are partitioned into k sets of equal size. Each set of files is associated to one set of servers, and is replicated in the $\frac{n}{k}$ servers in this set. Thus, a request for reading a file can be completed by any of the $\frac{n}{k}$ servers in the set of servers to which the file belongs.
- Replication-II (depicted in Fig. 1b): The n servers are partitioned into a k sets of $\frac{n}{k}$ servers each, and each individual file is split into k chunks of equal size. For each file, the i^{th} chunk $(1 \leq i \leq k)$ is replicated in the $\frac{n}{k}$ servers of the i^{th} set of servers. Thus, a request for reading a file is split into k identical jobs, and the i^{th} job can be served by any of the $\frac{n}{k}$ servers in the i^{th} set of servers.

Observe that the property of an MDS code wherein each server stores $\frac{1}{k}$ of each file is identical to that of replication-II. Thus, in order to provide an apples-to-apples comparison, in this section, we compare the performance of MDS codes with that of replication-II (and not replication-I).

In the analysis of the latency incurred by coded systems, we assume that the additional delay incurred due to computations required for decoding the data is negligible. This assumption is justified in most systems of interest, since the codes employed typically have small block lengths, allowing for fast decoding.

The following subsections present the analysis and comparisons of the queues with respect to various metrics. Unless mentioned otherwise, the values of the system parameters for the graphs plotted are

$$\mathbf{n=10},\ \mathbf{k=5},\ \text{and}\ \mu=\mathbf{1}\ .$$

The system is assumed to be in steady-state.

A. Maximum throughput

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

Theorem 8: For any given (n,k) and $t \geq 1$, let $\lambda^*_{\text{Resv(t)}}$, λ^*_{MDS}, and $\lambda^*_{M^k/M/n(t)}$ denote the maximum throughputs of the MDS-Reservation(t), MDS, and M^k/M/n(t) queues respectively. Then,

$$\lambda^*_{MDS} \ = \ \lambda^*_{M^k/M/n(t)} \ = \ \frac{n}{k}\mu\ . \tag{3}$$

When k is treated as a constant,

$$\left(1-O(n^{-2})\right)\frac{n}{k}\mu \ \leq \ \lambda^*_{\text{Resv(t)}} \ \leq \ \frac{n}{k}\mu\ . \tag{4}$$

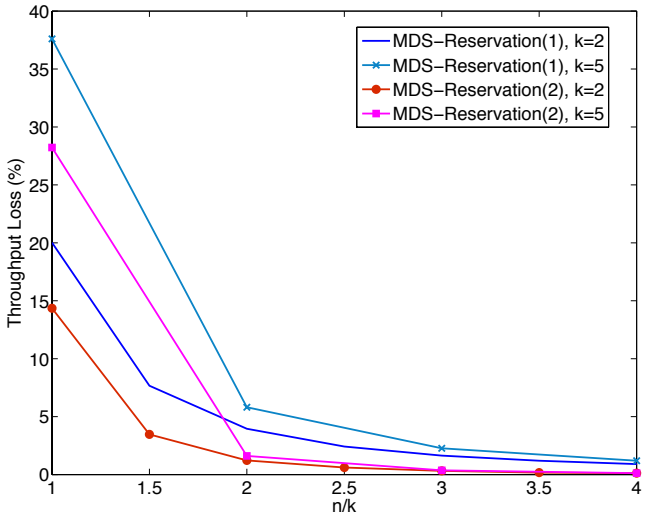


Fig. 12: Loss in maximum throughput incurred by the MDS-Reservation(1) and the MDS-Reservation(2) queues as compared to that of the MDS queue (and replication).

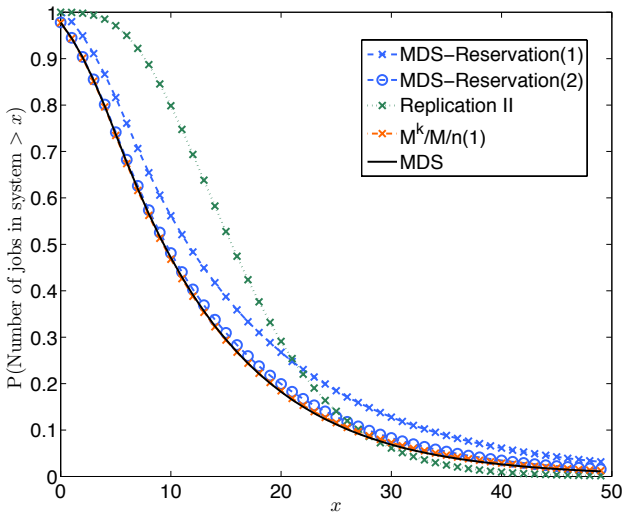


Fig. 13: Steady state distribution (complementary cdf) of the system occupancy when $\lambda = 1.5$.

In particular, when $k = 2$,

$$\left(1-\frac{1}{2n^2-2n+1}\right)\frac{n}{k}\mu \;=\; \lambda^*_{\text{Resv(1)}} \;\leq\; \lambda^*_{\text{Resv(t)}}\;, \tag{5}$$

and when $k = 3$,

$$\left(1-\frac{4n^3-8n^2+2n+4)}{3n^5-12n^4+22n^3-29n^2+26n-8}\right)\frac{n}{k}\mu \;=\; \lambda^*_{\text{Resv(1)}} \;\leq\; \lambda^*_{\text{Resv(t)}}\;. \tag{6}$$

The maximum throughput of replication-I and replication-II is also $\frac{n}{k}\mu$.

Note that the special case of Theorem 8, for the MDS-Reservation(1) queue with $k=2$, also recovers the throughput results of [5]. Moreover, as compared to the proof techniques presented in [5], the proofs in this paper are simpler and do not require computation of the stationary distribution. Using the techniques presented in the proof of Theorem 8, bounds analogous to (5) and (6) can be computed for $k \geq 4$ as well.

Fig. 12 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 (and replication).

B. 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 IV and V. Fig. 13 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 M^k/M/n(1) queues respectively are very close to each other.

C. Average job latency

The latency faced by a job is is the time from its arrival into the system till the time it completes being serviced at some server. Fig. 14 plots the average latency faced by a job in the steady state. Here, the average job latencies of the MDS-Reservation(t) and the M^k/M/n(t) queues have been obtained analytically by applying Little's law to the stationary distribution of the system occupancy. A MC simulation of the average job latency of the (exact) MDS queue is also plotted alongside.

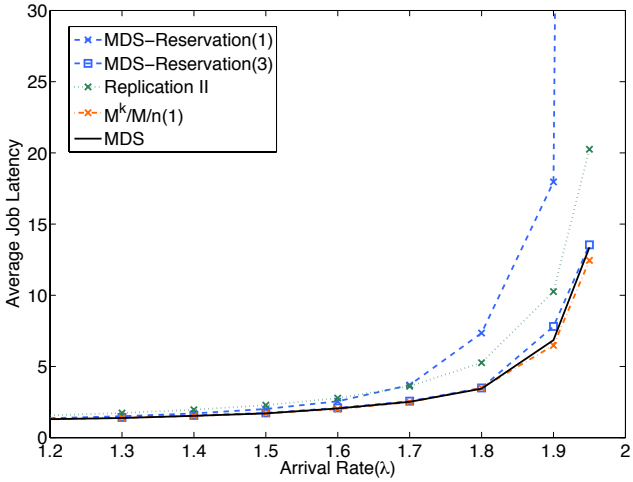


Fig. 14: Average latency faced by a job.

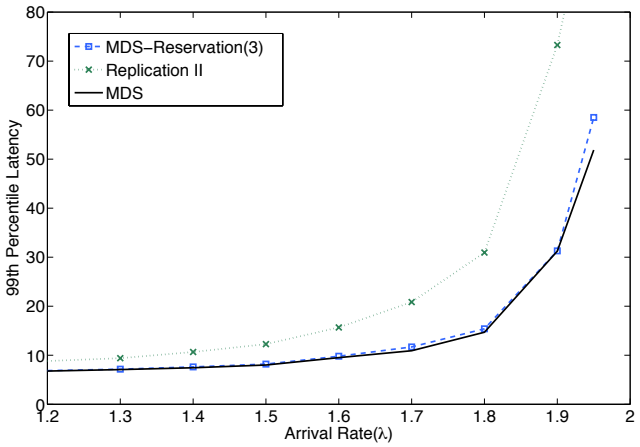


Fig. 15: 99^{th} percentile of the distribution of the latency faced by a batch.

D. Batch latency

The latency faced by a batch is the time from its arrival into the system till the last of its k jobs completes service.

1) Average: We have analytically computed the average batch latencies of the MDS-Reservation(t) and the M^{k}/M/n(t) queues in the following manner. We first compute the steady state distribution π of the number of jobs in the system. As discussed in the previous sections, each state i in the corresponding Markov chain is associated to a unique configuration of the jobs in the system. Now, the average latency d_i faced by a batch entering when the system is in state i can be computed easily as a dynamic program, by employing the Markovian representations of the queues presented previously. Since Poisson arrivals see time averages [23], the average latency faced by a batch in the steady state is given by $\sum_i \pi_i d_i$.

Fig. 3 plots the average latency faced by a batch in the steady state. Observe that in this figure, the coded systems achieve upto 70% reduction in average latency as compared to replication. Also 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 M^{k}/M/n(1) scheduling policy.

2) Tails: It is of significant practical interest to analyse the tails distribution of the latency. We cannot get an analytical handle on this quantity through the methods presented here, and hence provide MC simulations instead. Fig. 15 plots the 99^{th} percentile of the distribution of the latency faced by a batch: for a given value of the arrival rate λ, a curve takes value y_λ if exactly 1% of the batches incur a delay greater than y_λ. Again, we see that coded systems achieve significant gains over replication.

E. Waiting and blocking probabilities

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 M^{k}/M/n(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. 16 plots the waiting probability for the different queues considered in the paper. Observe how tightly the M^{k}/M/n(1) and the MDS-Reservation(2) queues bound the waiting probability of the MDS queue. Also observe that replication fares much worse than codes, even for low arrival rates.

We have assumed throughout the paper that the buffer has an infinite capacity. For a moment, let us suppose that the buffer has a capacity of accommodating at most a certain (finite) number batches. Then, the blocking probability

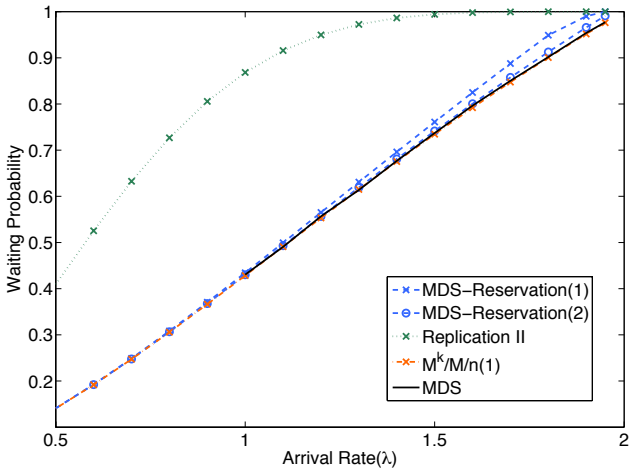


Fig. 16: Waiting probability.

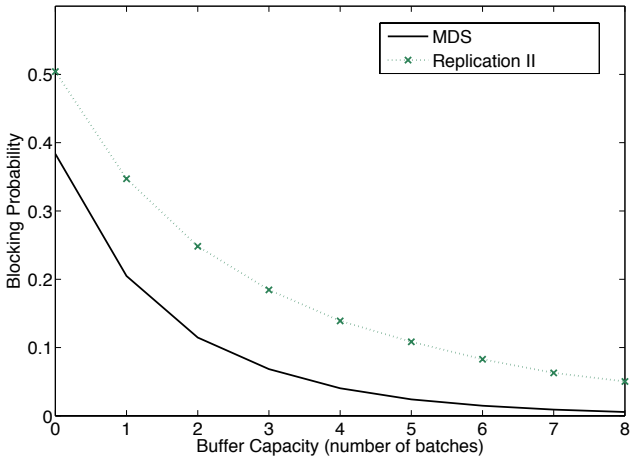


Fig. 17: Blocking probability when the buffer has a finite capacity. The arrival rate is $\lambda = 1.5$.

is the probability that (in the steady state) an arriving batch will find the buffer full and hence will have to be rejected. Fig. 17 plots the blocking probability of a system for various buffer sizes, obtained via MC simulations. Observe that the blocking probability of the coded system is consistently smaller than that of replication.

F. Degraded Reads

The coded system presented so far assumes that each incoming request desires one complete file F_x, for some $x \in \{A,B,C,\ldots\}$. As discussed previously in Section II-A3, in certain applications, incoming requests may sometimes require only a part of the file $f_{x\ell}$, for some $\ell \in \{1,\ldots,k\}$. In this case, assuming the code is systematic, $f_{x\ell}$ can be retrieved by reading it directly from the ℓ^{th} server. However, in the situation that server ℓ is busy, one may need to perform a 'degraded read', i.e., recover $f_{x\ell}$ from the data stored in the remaining $(n-1)$ servers.

Under any MDS code, a degraded read may be performed by reconstructing the entire file F_x from the data stored in any k of the $(n-1)$ remaining servers, and subsequently extracting the desired chunk $f_{x\ell}$ from it. 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 subsequentlys (e.g., [11]–[14]). In particular, the practical *product-matrix (PM) codes* proposed in [12] are associated to an additional parameter d, such that any part $f_{x\ell}$ can be recovered by reading and downloading a fraction $\frac{1}{d-k+1}^{\text{th}}$ of the data stored in *any* d of the remaining $(n-1)$ servers. This method of performing a degraded read is termed as a 'repair' operation, since it was first proposed in the context of repairing failed servers.

In this paper, we employ the framework of the MDS queue developed in this paper, to model and compare the two aforementioned methods of performing degraded reads, and also to compare the performance under various choices of parameter d. We assume that each request desires to read chunk $f_{x\ell}$ for some (uniformly) random $x \in \{A,B,\ldots\}$ and $\ell \in \{1,\ldots,k\}$. We further assume the time required to read data from any server to be exponentially distributed with a mean equal to the amount of data to be read, with each chunk $f_{x\ell}$ assumed to be of unit size. Under this setting, the method of reconstructing F_x from any k servers forms an MDS(n-1,k) queue (with service rate $\mu = 1$), and the method of performing 'repair' forms an MDS(n-1,d) queue (with service rate $\mu = (d-k+1)$). We can now apply the machinery developed in the previous sections to analyse the performance.

For example, consider a system with parameters $n = 6$ and $k = 2$, that stores files encoded by a PM code with some parameter $d \in \{3,4,5\}$. Then, a complete file can be recovered, as in any other MDS code, by downloading the corresponding chunks from any $k = 2$ servers. A part $f_{x\ell}$ of a file may also be recovered by reading and downloading data that is $\frac{1}{d-1}$ the size of $f_{x\ell}$ from each of any d servers. Fig. 18 plots the average latency incurred under these two methods of performing degraded reads. One can see that the method of 'repair' performs consistently better

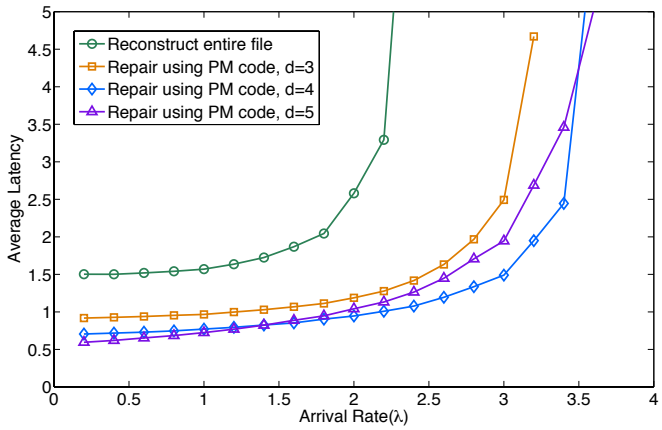


Fig. 18: Average latency during degraded reads. The parameters associated to this system are $n=6$, $k=2$. The service time at any server is exponentially distributed with a mean proportional to the amount of data to be read. The performance of the product-matrix (PM) codes are compared for various values of the associated parameter d.

as compared to reconstructing the entire file F_x, thus corroborating the efficacy of these codes in terms of latency performance for degraded reads.

VII. Redundant Requests

In systems that possess the flexibility to serve requests in more than one way (e.g., by reading data from any one of multiple servers where it is stored), latency can (potentially) be reduced by sending redundant requests. Here, a request is sent to an excess number of servers, and is deemed served when the requisite number of servers complete service. Such a mechanism trades off the increase in the speed of service of any individual request with an increase in queueing delay due to use of additional resources.

Under an (n,k) MDS code, such schemes send each request (redundantly) to some ν $(k \leq \nu \leq n)$ servers. Upon completion of service by any k of these ν servers, the batch is considered to be served, and the services of the remaining $(\nu-k)$ jobs of that batch are cancelled. The case of $\nu=k$ corresponds to the absence of any redundancy in the requests, and is precisely the case considered previously in this paper. The case $k=1$ corresponds to a replication-based scheme. We term ν as the (redundant) *request-degree*. This scheduling policy and the resultant queues are formalized in Algorithm 7 and Definition 2 respectively.

Definition 2 (MDS queue with redundant requests): An MDS queue with redundant requests is associated to five parameters (n,k), $[\lambda,\mu]$ and the request-degree ν.

- There are n identical servers
- Requests enter into a (common) buffer of infinite capacity
- Requests arrive in *batches* of ν *jobs* each
- The batches arrive as a Poisson process with a rate of λ
- Each of the ν jobs in a batch can be served by an **arbitrary** set of ν **distinct** servers
- The service time for a job at any server is exponentially distributed with a rate of μ, and is independent of the arrival and service times of all other jobs
- A batch is served when any k jobs of that batch are served
- 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.

We assume that the overheads associated to cancelling a job are negligible, i.e., when a job is cancelled, the server immediately becomes available to serve any other job.

Algorithm 7 MDS scheduling policy with redundant requests

on arrival of a batch
  divide the batch into ν 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 departure from a server (say, server s_0)
  let set $\mathcal{S} = \{s_0\}$
  if the job that departed from s_0 was the k^{th} job served of the corresponding batch **then**
    for every server that is also serving jobs from this batch **do**
      cancel this job and add this server to set $\mathcal{S}$
    end for
  end if
  for each $s \in \mathcal{S}$ **do**
    if $\exists$ 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

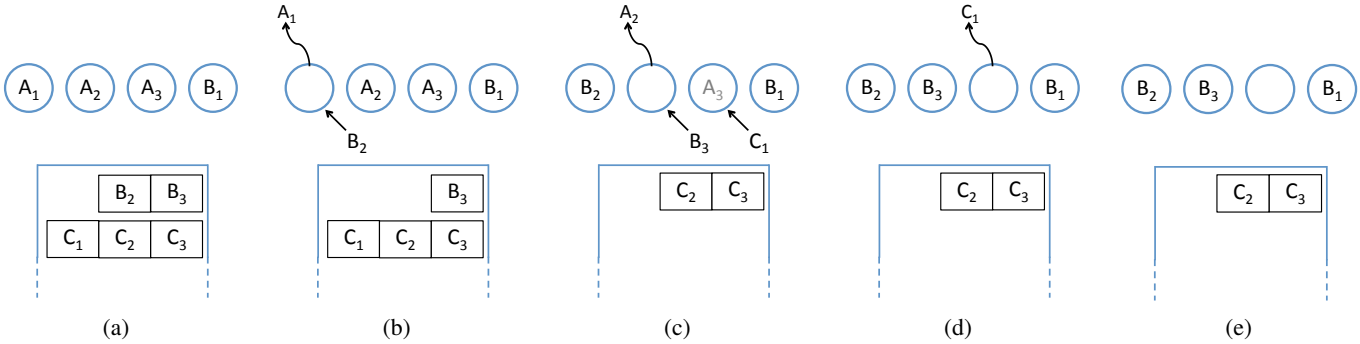


Fig. 19: Illustration of the functioning of the MDS queue with redundant requests for a system with parameters $n=4$, $k=2$ and request-degree $\nu=3$. Requests take the form of batches, A, B, C etc., each comprising of $\nu=3$ jobs $\{A_1,A_2,A_3\}$, $\{B_1,B_2,B_3\}$, $\{C_1,C_2,C_3\}$, etc. out of which any $k=2$ must be served. Jobs of any batch can be served by any distinct servers.

Example 7: Fig. 19 illustrates the model of the MDS queue with redundant requests, and the working of Algorithm 7, when $n=4$, $k=2$ and $\nu=3$. Here, each request comes as a batch of $\nu=3$ 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 three jobs of that batch are served. Let us denote the four servers (from left to right) as servers 1, 2, 3 and 4. Suppose the system is in the state as shown in Fig. 19a, wherein the jobs A_1, A_2, A_3 and B_1 are being processed by the four servers. Suppose server 1 completes servicing job A_1 (Fig. 19b). It now becomes idle to serve any of the jobs remaining in the buffer. We allow jobs to be processed only in order, and hence server 1 begins servicing job B_2 (assignment of B_3 instead would also have been valid). Next, suppose the second server completes service of A_2 before any other servers complete their current tasks (Fig. 19c). Now, $k=2$ jobs of batch A are complete, and hence batch A exits the system. In particular, job A_3 is immediately cancelled at server 3. Thus, servers 2 and 3 are now free to serve other jobs in the buffer. These are now populated with the jobs B_3 and C_1 respectively. Next suppose server 3 completes serving C_1 (Fig. 19d). In this case, since server 3 has already served a job from batch C, it is not allowed to service 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. 19e).

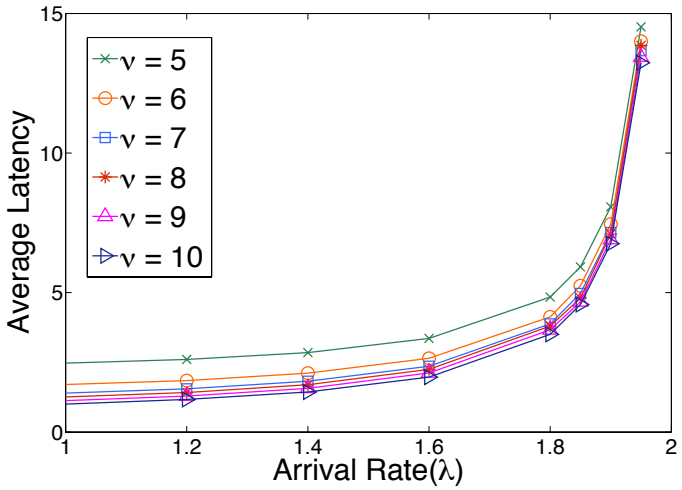


Fig. 20: Simulation results showing the reduction of average latency with an increase in the request-degree ν, in a $(n=10,\ k=5)$ MDS queue with $\mu=1$.

While policies that send redundant requests have been empirically shown [9], [18], [24] to reduce the latency under various settings, to the best of our knowledge, there is no theoretical proof of when this can actually provide any benefits. In this section, we initiate such an analytical investigation, under the MDS queue model presented in Section I-A. The results below prove that under this model, request flooding can indeed reduce the latency in replication as well as in coded systems. Subsequently, a brief discussion on the effect of redundant requests under certain extensions of this model is also presented.

A. Redundant Requests Help Reduce Latency in MDS Queue

Theorem 9 (Replication): Consider a replication-based queue (i.e., an MDS(n,k) queue with $k=1$) with (λ,μ) such that the system is stable in the absence of redundancy in requests. For any $\nu_1<\nu_2$, the average latency in a system with request-degree ν_1 is strictly larger as compared to the average latency in a system with request-degree ν_2. Furthermore, the distribution of the buffer occupancy in the system with request-degree ν_1 strictly dominates (is larger than) that in the system with request-degree ν_2.

Theorem 10 (MDS codes): Consider the MDS$(n,\ k)$ queue with $(\lambda,\ \mu)$ such that the system is stable in the absence of redundancy in requests. A scheduling policy that has $\nu=n$ results in a strictly smaller average latency than any other redundant-request-policy (including that of not having redundancy in requests, and that of adaptively changing the request-degree ν). Furthermore, the distribution of the buffer occupancy in the system with request-degree $\mu=n$ is strictly dominated by (i.e., is smaller than) a system with any other request-degree.

The proofs of both theorems are provided in the Appendix. Fig. 20 depicts simulations that corroborate this result.

B. Simulations of More General Settings

In this paper we showed that redundancy in requests strictly improves the average latency, under a particular model. We wish to employ the results of this paper as building blocks to analyse the effects of having redundancy in requests under more general system models, as discussed below. It also remains to quantify the amount of gains that redundant requests provide.

1) General service times: We saw that when the servers take i.i.d. exponential times to complete service, the average latency strictly reduces with an increase in request-degree. We do not know at present whether such a phenomenon is a result of the memoryless assumption of service times, or whether it carries over to a much larger class of distributions.

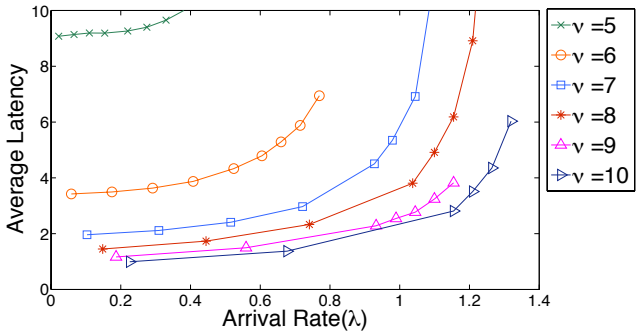


Fig. 21: Average latency in an MDS coded system with the service time distributed as a mixture of exponential distributions. Here, $n=10$ and $k=5$.

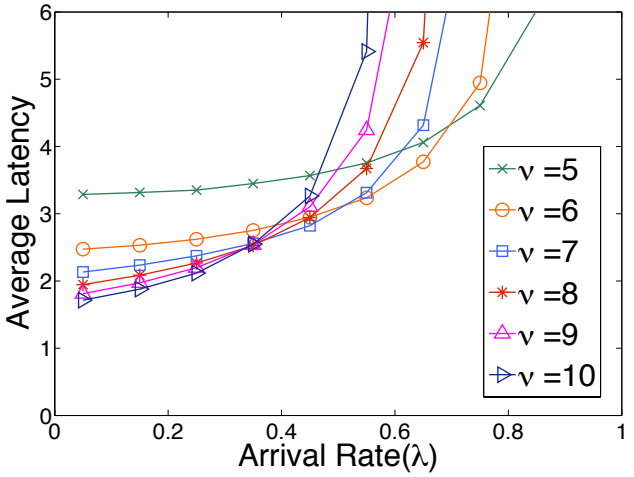


Fig. 22: Average latency in an MDS coded system with the service time following an exponential distribution shifted by a constant. Here, $n=10$ and $k=5$.

We believe that for distributions that are "heavier" than the exponential (in the sense defined below), introducing redundancy in the requests necessarily helps. We say that a distribution is heavier than the exponential if for a random variable X drawn from the given distribution and for any two positive values a and b, the distribution satisfies $P(X>a+b|X>b)\geq P(X>a)$. This is corroborated in Fig. 21 which depicts simulations with the service time X distributed as a mixture of exponentials:

$$X\sim\begin{cases}\exp(0.1) & \text{w.p. } 0.2\\ \exp(1) & \text{w.p. } 0.8\end{cases}.$$

We see in this figure that for all values of the arrival rates, an increase in the request-degree monotonically reduces the the average latency.

On the other hand, we have observed (via simulations) that for several distributions of service times that are not 'heavier' than the exponential, a high request-degree aids when arrival rates are low, but an increase in the arrival rate causes a crossover, with small request-degrees performing better at high enough arrival rates. This is seen, for instance, in Fig. 22 which depicts simulations with the service time X distributed as:

$$X\sim 1+exp(1)\ .$$

The intuition is as follows. If a service time distribution is "heavier" than the exponential distribution, a server's state becomes 'worse' as time goes on. Thus, the cancellation of a job at a server necessarily moves the server to a 'better' state. On the other hand, under a service-time distribution that is not as heavy as the exponential distribution, cancellation moves the server to a 'worse' state. Now, a higher request-degree would typically lead to more cancellations, which would thus benefit systems with heavier service-time distributions but harm those with lighter distributions. Thus the lighter distributions only gain from redundant requests whenever the load is low enough to allow for consumption of extra resources. It is for this reason that we believe that heavier distributions will generally be benefited by a higher request degree, while in the case of lighter distributions it would help only upto a certain threshold on the load on the system.

Thus an interesting open problem is to characterize the distributions of service times, and corresponding ranges of arrival rates, under which introducing redundancy in the requests helps.

2) Cancellation penalties: In the model above, we assumed that once some k jobs of a batch complete service, any jobs of that batch being processed by the servers are immediately cancelled, and these servers become available immediately to serve other jobs. However in practice, such a cancellation may often result in non-negligible costs. For instance, it might take some time for a central schedule to be notified that enough number of jobs are serviced

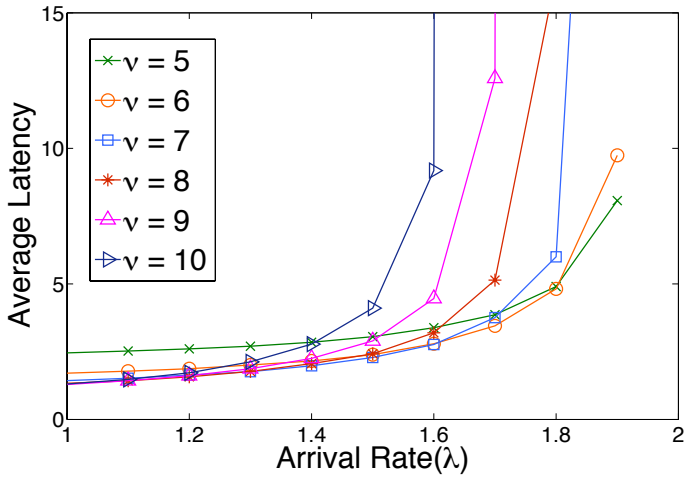


Fig. 23: Simulation results for the average latency for various values of the request-degree ν, in a $(n=10,\ k=5)$ MDS queue with $\mu=1$ and various values of λ.

and to send control signals to cancel servers. Here, we present simulations that are preliminary attempts to capture these effects.

We consider a system model identical to that described previously in Definition 2 and Algorithm 7, with the following additional property. Whenever a job being processed by a server is cancelled, the server must stay idle for a time that is distributed exponentially with a rate 10μ. This, in some sense, corresponds to a cancellation cost of 10%. Fig. 23 depicts the average latency faced by a batch in the steady state under such a system. Observe that for low arrival rates, the latency is higher for higher values of the request-degree ν. However, an increase in the arrival rate reduces these gains, and beyond a certain arrival rate, the smallest latency is obtained under the policy of not having any redundancy in the requests.

VIII. Discussion and Open Problems

In this paper, we study the latency gains achieved through (a) coding the data, and (b) having redundant requests.

We study storage systems that employ Maximum-Distance-Separable(MDS) codes through the lens of queueing theory. The queues under such systems are termed the MDS queues, and an analysis of their latency performance is provided through analytical means as well as via MC-based simulations. Our analysis reveals the superior performance of codes compared to that of currently used replication-based schemes (upto 70%) under the models considered here. The key insight is that, the property of being able to recover the data from *any* k of the servers that lends MDS codes a high storage efficiency and reliability, is also instrumental in providing them a low latency. This paper thus makes a case for the use of codes to store frequently accessed "hot" data.

We present two classes of scheduling policies, the MDS-Reservation(t) and the M^k/M/n(t) scheduling policies, that respectively lower and upper bound the MDS queue. We show that the queues resulting from these scheduling policies belong to the class of Quasi-Birth-Death (QBD) processes, and exploit the properties of the QBD processes in our analysis. While both these classes converge to the MDS queue when the parameter $t=\infty$, we show that even for small values of t (as small as $t=3$), these bounds are quite tight and suffice to characterize the precise performance of the MDS queue. The MDS-Reservation(t) scheduling policy represents a practical scheme that requires the maintenance of a much smaller state, and may be of independent interest in practical systems. We also use the framework of the MDS queue developed in this paper to compare different methods of performing degraded reads. We observe that the average latency incurred for a degraded read can be reduced significantly by employing codes with efficient 'repair' properties, thereby providing a queueing-theoretic evidence for the efficacy of these codes.

In the paper, we also study the efficacy of sending redundant requests. We show that in a replication-based

system, when the service times are exponentially distributed, this policy necessarily reduces the average latency as well as the occupancy in the system. In addition, we show that in a system with an MDS code, sending requests (redundantly) to all servers is strictly better in these respects. To the best of our knowledge, these are the first rigorous analytical proofs showing the benefits of having redundancy in the requests. We also present simulations of certain more general settings, such as alternative service-time distributions and non-zero cancellation penalties.

In the future, we intend to build upon the framework presented in this paper, and analyse the latency performance of codes and redundant requests in settings that relax one or more of the assumptions made in the paper:

- General service times: more accurate modelling based on the underlying properties of the storage devices (e.g., [25]).
- Heterogeneous requests: files of different sizes.
- Heterogeneous servers: the n servers may not have identical service-time distributions.
- Absence of MDS property: a storage system may alternatively employ codes that are not MDS (e.g., [4]), or store different files in different sets of n servers. In such situations, instead of being able to recover the data from *any* set of k servers, there will exist pre-specified subsets of the servers from which a file can be recovered.
- Decentralized MDS queue: each server may have its own buffer, in which case, the k jobs of a batch must be sent to the buffers of k distinct servers, and the choice of these k servers may have to be made much before the jobs are actually serviced.

The characterization of the service-time distributions under which redundant requesting necessarily helps is an interesting open problem. Moreover, even in the case of exponentially distributed service times, the precise amount of gains achieved due to redundant requests is unknown.

We are also presently working on connecting the framework and results presented in this paper to traces from real-world data centers.

REFERENCES

[1] D. Borthakur, "HDFS and Erasure Codes (HDFS-RAID)," 2009. [Online]. Available: http://hadoopblog.blogspot.com/2009/08/hdfs-and-erasure-codes-hdfs-raid.html

[2] "Erasure codes: the foundation of cloud storage," Sep. 2010. [Online]. Available: http://blog.cleversafe.com/?p=508

[3] D. Ford, F. Labelle, F. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems," in *9th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2010.

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

[5] L. Huang, S. Pawar, H. Zhang, and K. Ramchandran, "Codes can reduce queueing delay in data centers," in *Proc. IEEE International Symposium on Information Theory (ISIT)*, Cambridge, Jul. 2012.

[6] F. MacWilliams and N. Sloane, *The Theory of Error-Correcting Codes, Part I*. North-Holland Publishing Company, 1977.

[7] I. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

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

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

[10] U. Ferner, M. Medard, and E. Soljanin, "Toward sustainable networking: Storage area networks with network coding," in *50th Annual Allerton Conference on Communication, Control, and Computing*, Oct. 2012.

[11] K. V. Rashmi, N. B. Shah, P. V. Kumar, and K. Ramchandran, "Explicit construction of optimal exact regenerating codes for distributed storage," in *Proc. 47th Annual Allerton Conference on Communication, Control, and Computing*, Urbana-Champaign, Sep. 2009, pp. 1243–1249.

[12] K. V. Rashmi, N. B. Shah, and P. V. Kumar, "Optimal exact-regenerating codes for the MSR and MBR points via a product-matrix construction," *IEEE Transactions on Information Theory*, vol. 57, no. 8, pp. 5227–5239, Aug. 2011.

[13] D. Papailiopoulos, A. Dimakis, and V. Cadambe, "Repair optimal erasure codes through hadamard designs," in *Proc. 47th Annual Allerton Conference on Communication, Control, and Computing*, Sep. 2011, pp. 1382–1389.

[14] I. Tamo, Z. Wang, and J. Bruck, "Access vs. bandwidth in codes for storage," in *IEEE International Symposium on Information Theory (ISIT)*, Cambridge, Jul. 2012, pp. 1187–1191.

[15] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *Micro, IEEE*, vol. 25, no. 6, pp. 10–16, 2005.

[16] F. Baccelli, A. Makowski, and A. Shwartz, "The fork-join queue and related systems with synchronization constraints: stochastic ordering and computable bounds," *Advances in Applied Probability*, pp. 629–660, 1989.
[17] E. Varki, "Response time analysis of parallel computer and storage systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 11, pp. 1146–1161, 2001.
[18] G. Liang and U. Kozat, "Fast cloud: Pushing the envelope on delay performance of cloud storage with coding," *arXiv:1301.1294*, Jan. 2013.
[19] G. Joshi, Y. Liu, and E. Soljanin, "Coding for fast content download," in *50th Annual Allerton Conference on Communication, Control, and Computing*, Oct. 2012.
[20] R. Feldman and C. Valdez-Flores, *Applied Probability and Stochastic Processes (Chapter 13)*. Springer, 2010.
[21] D. Bini, B. Meini, S. Steffé, and B. Van Houdt, "Structured markov chains solver: software tools," in *Workshop on Tools for solving structured Markov chains*, 2006.
[22] P. Harrison and S. Zertal, "Queueing models with maxima of service times," *Computer Performance Evaluation. Modelling Techniques and Tools*, pp. 152–168, 2003.
[23] R. Wolff, "Poisson arrivals see time averages," *Operations Research*, vol. 30, no. 2, pp. 223–231, 1982.
[24] A. Vulimiri, O. Michel, P. Godfrey, and S. Shenker, "More is less: Reducing latency via redundancy," in *ACM Hotnets*, Oct. 2012.
[25] A. Merchant and P. Yu, "Analytic modeling and comparisons of striping strategies for replicated disk arrays," *IEEE Transactions on Computers*, vol. 44, no. 3, pp. 419–433, 1995.

APPENDIX
PROOFS OF THEOREMS

Additional Notation used in the proofs: Table I enumerates notation for various parameters that describe the system at any given time. To illustrate this notation, consider again the system depicted in Fig. 2a. Here, the parameters listed in Table I 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 I, the number of servers that are busy at any given time is equal to $(n-z)$. For batch i in the buffer ($i \in \{1,\ldots,b\}$), 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 Theorem 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 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,

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

TABLE I: Notation used to describe state of the system.

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 Theorem 2: Results as a special case of Theorem 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 3: For any state of the system $(w_1,w_2,\ldots,w_t,m) \in \{0,1,\ldots,k\}^t \times \{0,1,\ldots,\infty\}$, define

$$q=\begin{cases} 0 & \text{if } \ w_1=0 \\ t & \text{else if } \ w_t \neq 0 \\ \arg\max\{t' : w_{t'} \neq 0, 1 \leq t' \leq t\} & \text{otherwise.} \end{cases} \tag{7}$$

It can be shown that

$$b=\begin{cases} 0 & \text{if } \ q=0 \\ q & \text{if } \ 0<q<t \\ t+\left\lfloor \frac{m-\sum_{j=1}^{t} w_i-n}{k} \right\rfloor & \text{otherwise,} \end{cases} \tag{8}$$

$$z=n-(m-\sum_{j=1}^{t} w_j-(b-t)^+k) \ , \tag{9}$$

$$\text{for } i \in \{1,\ldots,b\}, \ \ s_i=\begin{cases} w_{i+1}-w_i & \text{if } \ i \in \{1,\ldots,q-1\} \\ k-z-w_q & \text{if } \ i=q \\ 0 & \text{if } \ i \in \{q+1,\ldots,b\}, \end{cases} \tag{10}$$

and

$$\text{for } i \in \{t+1,\ldots,b\}, \ \ w_i=k \ . \tag{11}$$

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 Theorem 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$=\infty$, every batch is treated as in MDS, thus making MDS-Reservation(∞) identical to MDS. ■

Proof of Theorem 5: Under M^k/M/n(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 6: For any state $(w_1,w_2,...,w_t,m)$, define q as in (7). The values of b, z, w_i, are identical to that in the proof of Theorem 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 M^k/M/n(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 Theorem 7: The M^k/M/n(t) scheduling policy follows the MDS scheduling policy when the number of batches in the buffer is less than or equal to t. Thus, M^k/M/n(∞) is always identical to MDS. ■

Proof of Theorem 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 M^k/M/n(t) queues upper bound the performance of the MDS queue, $\lambda^*_{M^k/M/n(t)} = \frac{n\mu}{k}$ 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: $\exists$ $\mathbf{v}$ 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≥ 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. 7, 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

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

$$\Rightarrow A_0 + A_1 + A_2 = \begin{bmatrix} -n & 1 & n-1 \\ n-1 & -(n-1) & 0 \\ n & 0 & -n \end{bmatrix} \mu \tag{12}$$

One can verify that the vector

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

satisfies

$$\mathbf{v}^T(A_0 + A_1 + A_2) = 0. \tag{13}$$

Thus,

$$\mathbf{v}^T A_0 \mathbf{1} = n(n-1)\mu, \tag{14}$$

and

$$\mathbf{v}^T A_2 \mathbf{1} = \lambda \left(n + \frac{(n-1)^2}{n} \right) . \tag{15}$$

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,

$$\lambda^*_{\text{Resv(1)}} = \frac{n^2(n-1)}{2n^2 - 2n + 1} = \left(1 - \frac{1}{2n^2 - 2n + 1} \right) \frac{n}{2} \mu. \tag{16}$$

■

Proof of Theorem 9: Consider two systems, system S_1^* with request-degree ν_1 and system S_2^* with request-degree ν_2 $(> \nu_1)$, both having system parameters $(\lambda,\ \mu)$ and $(n,\ k = 1)$. In the proof, we shall construct two new systems S_1 and S_2 such that the statistics of S_1 are identical to S_1^*, and the statistics of S_2 are identical to S_2^*. We shall then show that system S_2^* outperforms system S_1^*, and conclude that S_2 outperforms S_1.

The new system S_1 is defined as follows. The system S_1 is also associated to parameters $(\lambda,\ \mu)$ and $(n,\ k = 1)$ and follows the usual MDS-scheduling protocol with request-degree ν_1. However, after every departure-event, we perform a particular permutation of the n servers. Since the n servers have independent and identical service time distributions, the system S_1 remains statistically identical to S_1^*, and in particular, the two systems have identical distributions of the latency and buffer occupancy. The specific permutation applied is as follows. 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 S_2 to be a system identical to S_2^*, but again permuting the servers in S_2 after every job completion such that the busy servers have the lowest indices. Thus S_2 is stochastically identical to S_2^*. The rest of the proof shows that S_2 has a smaller average latency as compared to S_1, and the distribution of system-occupancy in S_1 dominates that in S_2.

Under an MDS queue, at any point in time, there are $(n+1)$ competing exponential timers: the arrival process and the n servers. Let us term the completion of any of the timers as the *events*. In this proof, we assume the occurrence of any arbitrary sequence of events, and evaluate the performance of systems S_1 and S_2 under this sequence of events. We first show that at any point in time, the number of batches remaining in system S_1 is at least as much as that in S_2. We then show that for a fraction of time that is strictly bounded away from zero, the number of batches in S_1 is strictly larger than that in S_2. This shows that the distribution of the system occupancy in S_1 strictly dominates that in S_2. Furthermore, since the average latency faced by a batch is proportional to the average number of batches in the system, this proves that the average latency is strictly smaller in S_2 as compared to S_1.

We begin by showing that under the same sequence of events (i.e., the arrivals and server completions), the number of batches remaining to be completely served in S_2 at any point of time is no more than number of batches remaining in S_1 at that time. Without loss of generality, we shall prove this statement only at times immediately following an event, since the systems do not change state between any two events. With some abuse of 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 identical states at time 0. For system S_i $(i \in \{1,2\})$, let $b_i(z)$ denote the number of batches remaining in system S_i at time z. The proof proceeds via induction on z. The induction hypothesis is that at any time z, $b_1(z) \geq b_2(z)$. 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

$$\begin{aligned} b_1(z) &= b_1(z-1)+1 && (17)\\ &\geq b_2(z-1)+1 && (18)\\ &= b_2(z) && (19)\end{aligned}$$

where (18) follows from the induction hypothesis. Thus, the hypothesis is satisfied at time z.

Now suppose the z^{th} event is the completion of the exponential timer of one of the n servers. 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 departure 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 at time $(z-1)$ is equal to $\min\{\nu_i b_i(z-1),\ n\}$. Furthermore, from the construction of systems S_1 and S_2 describe above, we have that the first $\min\{\nu_i b_i(z-1),\ n\}$ are occupied at time $(z-1)$ in system $i \in \{1,\ 2\}$. Thus, since $\nu_1 < \nu_2$ and $b_1(z-1) = b_2(z-1)$, the set of servers occupied at time $(z-1)$ in S_1 is a subset of the servers occupied in S_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 a departure of a batch in S_1, it also leads to the departure of a batch in system S_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 S_2 is no more than that under system S_1.

We shall now show that for a fraction of time that is strictly positive, the number of batches remaining in S_2 is strictly smaller than that in S_1. To this end consider some time $(z-1)$ such that $b_2(z-1) = 1$. From the discussion above, it follows that $b_1(z-1) \geq 1$. If the next event is the triggering of any k of the ν_2 servers that are servicing the batch in S_2 then $b_2(z) = 0$. However, the triggering of one server out of only ν_1 $(< \nu_2)$ servers can lead to $b_1(z) = 0$. Thus the system moves to a state with $\{b_1(z) \geq 1,\ b_2(z) = 0\}$ with a probability that is strictly bounded away from zero. From the first part of the proof above, it also follows that the stability region of a system with a larger request-degree contains the stability region of a system with a smaller request-degree. In particular, the stability regions of S_1 and S_2 are subsets of the stability region of a system without any redundancy in the requests (i.e., with $\nu = k$). Since λ and μ are in the stability region of the system with request-degree $\nu = k$, it follows that system S_2 is also stable. Thus, the state $b_2(z) = 1$ has a probability p of occurrence in the steady state that is strictly bounded away from zero. Thus the fraction of time this state occurs is $p > 0$. Thus, the fraction of time f in which $b_1(\cdot) > b_2(\cdot)$ is p times that value which is also thus strictly bounded away from zero.

The arguments above show that the distribution of the number of batches remaining in S_1 dominates that in S_2: with B_1 and B_2 denoting the number of batches in the system S_1 and S_2 respectively under steady state, $P(B_1 > x) \geq P(B_2 > x)$ for all x and $P(B_1 > 0) > P(B_2 > 0)$. 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 S_2 is strictly smaller than that in S_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 S_1 and S_2 respectively. ■

Proof of Theorem 10: Consider two systems, system S_1 with request-degree $\nu < n$ and system S_2 with request-degree n. Under an MDS queue, at any point in time, there are $(n+1)$ competing exponential timers: the arrival process and the n servers. Let us term the completion of any of these $(n+1)$ timers as the *events*. In this proof, we assume the occurrence of any arbitrary sequence of events, and evaluate the performance of both systems under this sequence of events. We show that at any point in time, the number of batches remaining to be completely served in S_1 is at least as much as that in S_2. We then show that the number of batches in S_1 is strictly larger than that in S_2 for a fraction of time that is strictly bounded away from zero. Since the average latency faced by a batch is proportional to the average number of batches in the system, this also proves that the average latency is smaller in S_2 than in S_1.

We begin by showing that under the same sequence of events (i.e., the arrivals and server completions), the number of batches remaining in system S_1 is at least as much as that in S_2 at any given time. Without loss of generality, we shall prove this statement only at times immediately following an event, since the systems do not change state 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 are empty at time $z=0$. For system S_i $(i\in\{1,2\})$, let $b_i(z)$ denote the number of batches remaining in system S_i at time z. The proof proceeds via induction on the time z. The induction hypothesis is that at any time z:

(a) $b_1(z)\geq b_2(z)$
(b) if there are no further arrivals from time z onwards, then at any time $z'>z$, $b_1(z')\geq b_2(z')$

The hypotheses are clearly true at $z=0$, when both systems are empty. Now, let us consider them to hold for some time $z\geq 0$. Suppose the next event occurs at time $(z+1)$. We need to show that the hypotheses are true even after that event, at time $(z+1)$.

First suppose the event was the completion of an exponential-timer at one of the n servers. Then the hypothesis to time z implies the satisfaction of all the hypotheses at time $(z+1)$. This is because there has been no arrival between times z and $(z+1)$, which allows us to apply the second hypothesis at time z with $z'=z+1$.

Now suppose the event at $(z+1)$ is the arrival of a new batch. Then, the first hypothesis 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 the second hypothesis is also satisfied. Consider any sequence of departures, and any time $z'>z+1$. If S_2 is empty at time z' then $b_2(z')=0\leq b_1(z')$ and there is nothing left to prove. Let us consider the case when $b_2(z')>0$.

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*. From the hypothesis at time z, we know that $a_1(z')\geq a_2(z')$. Also note that the MDS scheduling policy 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 departures, for $i\in\{1,2\}$, $b_i(z')=a_i(z')$ if the new batch has not been served in S_i, else $b_i(z')=a_i(z')+1$. 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 suffices to show that $b_1(z')=a_1(z')\Rightarrow b_2(z')=a_2(z')$. The condition $b_1(z')=a_1(z')$ implies that the new batch has been served in S_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 have been served in S_1. Then, at these times, the corresponding servers must have been idle in S_1 if the new batch had not arrived.

Suppose $b_1(z')=a_1(z')$. Consider another sequence of departures that is identical to that discussed above, but excludes the events that happened at times $z_1,\ldots,z_k$. Let $c_i(z')$ denote the number of batches remaining in this situation at time z'. From the arguments above, we get $a_1(z')=c_1(z')$. From the second hypothesis, we also have $c_2(z')\leq c_1(z')$. We are required to show $b_2(z')=a_2(z')+1$, for which it now suffices to show that $b_2(z')=c_2(z')$. We shall consider only system S_2 for the remaining part of this paragraph. Observe that if at time z, all $b_2(z)$ batches present in the system had all its k jobs remaining to be served, and there were no further arrivals, then the total number of batches served between times z and $z'>z$ can be counted as follows. Start a 'global' counter at 0. Also associate one counter to each server, starting at 0. Whenever the exponential timer at a server is completed, add 1 to the counter at that server. Whenever some k servers' counters become larger than zero, add 1 to the global counter, and subtract 1 from the counters of these k servers. The value of the global counter at z' gives the number of batches served upto time z' (assuming that the system is not empty at time z'). 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 the completion of exponential timers at k *distinct* servers, it must be that $b_2(z')=c_2(z')$. Putting the pieces together, we get that the number of batches served in S_2 at any time is at least as much as that served in S_1 at any time.

The fact that we did not use any specific properties of ν in the proof above renders this result applicable to all values of request-degree ν, including that of dynamically adapting it.

We shall now show that when when S_1 employs request-degree $\nu<n$, for a fraction of time that is strictly bounded away from 0, the number of batches remaining in S_2 is strictly smaller than that in S_1. To this end consider some time when both systems are empty. The result that we just proved above also implies that the stability region of a system with $\nu=n$ contains the stability region of a system with a smaller request-degree. In particular, the stability regions of S_1 and S_2 are subsets of the stability region of a system without any redundancy in the requests (i.e., with $\nu=k$). Since λ and μ are in the stability region of the system with $\nu=k$, it follows that

system S_2 is also stable. Thus, the state where both systems are empty has a probability of occurrence that is strictly bounded away from zero in the steady state. Consider the next arrival, whose ν_i jobs are assigned (without loss of generality) to the first ν_i servers under system S_i. Suppose this arrival occurs at time z. Then $b_2(z)=b_1(z)=1$. Suppose the next few events are completion of the exponential timers at different servers. Then since $n>\nu$, there is a probability strictly bounded away from zero that some k of the first ν_2 servers first complete service, and at least one of these servers is in the set $\{\nu_1,\ldots,\nu_2\}$. This takes the system to a state with $\{b_1(z+1)=1,\ b_2(z+1)=0\}$ with a probability strictly bounded away from zero. Thus, the fraction of time f in which $b_1(\cdot)>b_2(\cdot)$ is lower bounded by the product of the aforementioned probabilities, and is hence strictly bounded away from zero. Thus the distribution of the system occupancy in S_2 is strictly dominated by that of S_1, and the average latency under S_2 is strictly smaller than that in system S_1. ■