Token Redundancy in Distributed JIQ

Harte, Jesse

Award date:
2021

Technische Universiteit **Eindhoven** University of Technology

Department of Mathematics and Computer Science
Stochastic Operations Research
Postbus 513, 5600 MB Eindhoven
The Netherlands
www.tue.nl

Author
Jesse Harte (1258125)

Supervisor
dr. S.C. Borst

Date
August 22, 2021

Token Redundancy in Distributed JIQ

Bachelor Thesis

Jesse Harte (1258125)
`j.harte@student.tue.nl`

Table of contents

Title
Token Redundancy in Distributed JIQ

Technische Universiteit
Eindhoven
University of Technology

Table of contents

Title
Token Redundancy in Distributed JIQ

1 Introduction

1.1 Load balancing and applications

Load balancing is the act of distributing jobs among servers that process these jobs, and this distribution is done by a dispatcher. For each incoming job, the dispatcher must choose a server that will process the job, or in other words, assign the job to a particular server. This job will take a certain amount of time to be processed by the server. Meanwhile, other jobs assigned to the same server will form a queue at this server. Once a job is done processing, the server will start processing the next job in its queue, if there are any. Ultimately, the goal of the dispatcher is to assign jobs in such a way that they can be completed as quickly as possible.

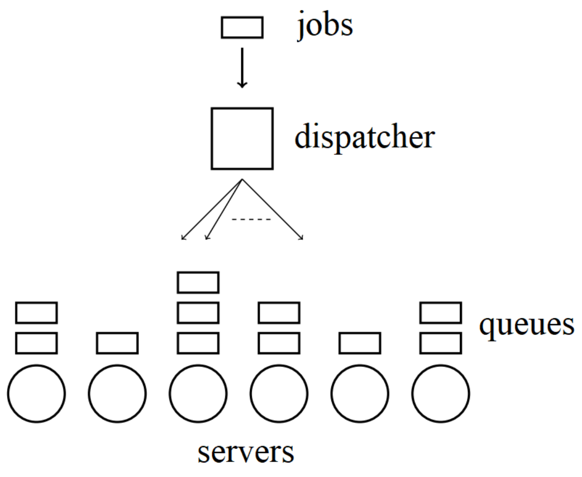


Figure 1.1: Schematic of a load balancing system. Borrowed from [1].

1.2 Current algorithms

The vital factor in the performance of a load balancing system is the algorithm used at the dispatcher to determine which server should process the job. The algorithm of randomly selecting a server is simple, but causes large delays. Another natural algorithm is selecting the server with the smallest queue at the time the job arrives, called Join-the-Shortest-Queue (JSQ) [2], [3]. While jobs experience very little delay with this algorithm, it requires probing all servers for their queue size every time a job arrives. In settings with a large number of servers, like data centers, JSQ causes a significant communication overhead, which becomes a burden on the network of the data center. Hence, in large-scale systems, the goal of the dispatcher is to cause as little delay as possible, but subject to a manageable communication overhead.

Join-the-Idle-Queue (JIQ) [4], [5] is an algorithm that keeps this communication overhead to a minimum. When a load balancing system employs JIQ, a server will advertise its availability to the dispatcher by sending a token once it is done processing all its jobs, i.e. it has an empty queue. The dispatcher will collect these tokens in an I-queue (short for idle-queue). Once a job arrives at the dispatcher, the dispatcher will remove a token from the I-queue and send the job to the server corresponding to the token. If there is no token to remove from the I-queue, the dispatcher will randomly select a server instead.

Note that JIQ only generates a message when a server enters the idle state, which happens at most once per job arrival. Therefore, the communication overhead is practically negligible compared to the communication overhead of JSQ. In addition, JIQ allows all communication to be off the critical path.

To clarify, the critical path is the path the job takes through the system, from entering at a dispatcher to being completed by the server. In JIQ, the dispatchers can immediately forward jobs to the servers, both when it has no tokens and when it does. On the other hand, JSQ imposes that jobs have to wait at the dispatchers for all the queue sizes to be probed. In practice, this adds to the delay the job experiences, on top of the delay from waiting in the queue and being processed.

1.3 Issues with JIQ

In the single-dispatcher setting, both JSQ and JIQ will assign a job to an idle server if there is one. The only time a job experiences delay under both policies is when there are no idle servers. In the case of no idle servers, JSQ will dispatch jobs to the shortest queue, and JIQ will dispatch randomly. Hence, when the probability of at least one idle server at a a certain time epoch is close to one, like in the large-capacity (fluid scale) and the Halfin-Whitt (diffusion scale) [6] regimes, JIQ and JSQ have similar performance. On the other hand, when this probability is not close to one, like in the classical heavy-traffic and non-degenerate slow-down regimes, JIQ will perform much worse.

Some systems receive too many jobs for a single dispatcher to handle, and will instead have multiple dispatchers. In this case, the system will have one I-queue for each dispatcher. This imposes a new challenge; the server must choose the receiving dispatcher for its token each time the server becomes idle. It is essential for the system to have a reasonable distribution of tokens across the I-queues, because an empty I-queue will cause the dispatcher to dispatch randomly. However, the probability that a dispatcher has no tokens in its queue now not only increases with the load, but with the number of dispatchers as well. Hence, the multiple-dispatcher setting amplifies the performance degradation of JIQ under high loads. In contrast, the performance of JSQ does not change with the number of dispatchers, because the policy does not require memory at the dispatchers and each dispatcher will simply probe all servers at each job arrival.

1.4 Scope of thesis

Previous literature has attempted to solve the shortage of tokens by increasing the threshold at which a server should advertise its availability, as we will see in Chapter 2. Instead, we will exploit the crucial insight that one can usually expect a non-zero number of idle servers at any point in time, and these should be given priority in receiving the next jobs. We do this by introducing the novel JIQ-Redundancy policy, which as one might expect, uses elements from Redundancy Scheduling to preserve JIQ's performance under high loads without jeopardizing its attractive property of having low communication overhead. We will first provide the model in which we evaluate the policy in Chapter 3, and then continue with the definition of JIQ-Redundancy in Chapter 4. We will evaluate the policy through analysis in Chapter 5 and simulation in Chapter 6. Additionally, the latter will be used to evaluate the robustness of the policy under variations of our model, including heterogeneous loads at the dispatchers and message delay.

2 Related work

In this chapter we will introduce the dominant JSQ and JSQ(d) load balancing algorithms. Next, we will abandon the traditional push-based paradigm, and introduce the Join-the-Idle-Queue (JIQ) algorithm. We continue with the bottlenecks associated with JIQ, and the existing variations on JIQ to counter these bottlenecks. We will then present the results on JIQ and its variations in the multiple-dispatcher setting, including the extension to asymmetric loads at the dispatchers.

2.1 Asymptotic scaling regimes

Existing literature makes use of various asymptotic scaling regimes to analyze and compare different algorithms because of the mathematical tractability these regimes provide. Within such a regime, the performance measures of expected waiting time and the probability of waiting time can be estimated. The popular regimes have been summarized in [7].

2.2 Join-the-Shortest-Queue (JSQ)

The Join-the-Shortest Queue (JSQ) load balancing algorithm was introduced in [2] and [3]. Upon the arrival of a job, the dispatcher in the JSQ algorithm queries all servers for their queue size, and forwards the job to the server with the shortest queue. [3] considers the case of a finite number of servers and jobs with exponentially-distributed service times, and proves that the throughput under JSQ is maximized, i.e. the discounted number of jobs to have been completed at any given time epoch is stochastically maximized.

Furthermore, the JSQ algorithm causes $\mathcal{O}(N)$ messages per job arrival. As a result, the traditional JSQ algorithm is unfit to satisfy the requirements of large-scale load balancing systems where N is large. Accordingly, [8] and [9] introduce the JSQ(d) algorithm. The diversity parameter d represents the number of servers that should be queried for their queue size. Hence, upon the arrival of a job, the dispatcher queries d randomly selected servers, and directs the job to the server with the shortest queue among the queried servers. Although fixing $d \geq 2$ provides significantly better results than directing jobs to randomly-selected servers, the performance of JSQ(d) remains constant when the number of servers, N, grows [9]. Therefore, the expected queuing delay is strictly larger than zero for a fixed d in the fluid limit. The result on JSQ(d) in the fluid limit is in line with the results by [10], which proves that the number of messages per job arrival must grow with N in the fluid limit to achieve vanishing queuing delay for any policy that does not use memory at the dispatcher.

2.3 Join-the-Idle-Queue (JIQ)

The aforementioned basic load balancing algorithms can be categorized as push-based algorithms, as explained in [4]. In short, the push-based paradigm cover algorithms where jobs are imposed by the dispatchers on the servers. In contrast, the authors of [4] introduce the concept of pull-based load balancing algorithms, where servers request the jobs from the dispatcher. Most pull-based load balancing algorithms use memory at the dispatcher, such that the conditions for vanishing queuing delay in [10] are satisfied without a communication overhead that grows with N. The authors of [4] propose a simple but general algorithm, where the dispatcher places the job in a FCFS queue. Once a server meets a certain condition, like having idle resources, the server requests a job from the dispatcher's queue.

The Join-the-Idle-Queue (JIQ) algorithm was introduced in [5]. Comparable to the algorithm in [4], JIQ also places the initiative at the servers. In the single-dispatcher setting, a server sends a token to the dispatcher once the server completes the last job in its queue. This token is an indication of availability to the dispatcher, and is placed in a FCFS I-queue at the dispatcher. Upon the arrival of a job at the dispatcher, the dispatcher removes a token from this I-queue, and directs the job to the associated server. However, if the token-queue is empty, the dispatcher will direct the job to a server selected uniformly at random. This distinguishes JIQ from the algorithm in [4], since JIQ will not delay the jobs at the dispatcher if there are no tokens available. Hence, JIQ is the first algorithm where all information exchange between dispatchers and servers is off the critical path, which is the path the job takes through the system, from entering at a dispatcher to being completed by a server. In JIQ, the dispatcher does not need to communicate with servers when a job arrives, because it either uses a token or dispatches the job randomly. This is in contrast with JSQ, where the dispatcher communicates with the servers to find the shortest queue at the time of a job arrival. In practice, this will add to the delay experienced by a job, because the communication in JSQ is on the critical path.

Lastly, JIQ causes at most one message per job, since a server will only generate a message when it enters the idle state. However, the number of messages per job arrival tends to one in the fluid limit, because nearly all job completions will cause a server to become idle. Motivated by this observation, the authors of [11] introduce the Join-the-Open-Queue (JOQ) policy. The novelty of JOQ is that the dispatcher infers the idleness of a server when the dispatcher does not receive a message at predefined time epochs. Most notably, this mechanism allows the expected queuing delay to converge to zero in the fluid limit of JOQ, while the expected number of messages remains strictly less than one.

2.3.1 Varying the reporting treshold

To counter the performance degradation under high loads, as mentioned in Section 1.3, the authors of [5] and [12] suggest to increase the reporting threshold. With this variation, a server generates a token when it has k jobs left in its queue. In this case, the performance of JIQ in comparison to JSQ remains similar, given that the probability that there is at least one server with a queue size of at most k is close to one. Naturally, increasing k will increase the relative load λ at which JIQ starts to significantly perform worse than JSQ.

Given these suggestions, [13] proposes Join-Below-Threshold (JBT-d). This policy is inspired by the suggestion mentioned above and provides an adaptive mechanism to repeatedly set k to a suitable value. In the discrete-time model, the threshold is set to the shortest queue length of d randomly sampled servers, every T time units. The servers then send a token once their queue size is not larger than the new threshold for the first time. The authors of [13] propose two conditions to form a class of load balancing policies that are both throughput and heavy-traffic delay optimal. As a result, they are able to prove that JBT-d has these properties, while having a similar communication overhead as JIQ, which is not heavy-traffic delay optimal.

2.4 Distributed JIQ

In their paper introducing JIQ, the authors of [5] recognize that for an algorithm to be applicable in practice, it must be able to operate in a multiple-dispatcher setting, as explained in [14]. In the case of JIQ, multiple dispatchers in the load balancing system results in multiple I-queues. Therefore, in [5] the distinction is made between two challenges in token-based load balancing:

- Primary load balancing, which concerns the distribution of jobs across the servers.
- Secondary load balancing, which concerns the distribution of tokens across the dispatchers.

As such, the authors consider two algorithms for the secondary system: Random and JSQ(d). This results in the following notation for the whole system: JIQ-Random and JIQ-SQ(d). Unsurprisingly, JIQ-SQ(2) is significantly better at preventing empty I-queues than JIQ-Random. However, the benefits of fair token distribution diminish once there are less idle servers than I-queues (which happens at $\lambda > \frac{N-M}{N}$ with N and M the number of servers and dispatchers, respectively). In this case, there must be empty I-queues by the pigeonhole principle, and the performance degradation explained in Section 1.3 sets in.

2.4.1 Fluid limit of distributed JIQ

The same paper analyzes the JIQ policy in the fluid limit (large-capacity regime) [5]. Two simplifying assumptions are made: there is exactly one token of each idle server in the I-queues, and all tokens in the I-queues represent idle servers. The authors first analyze the secondary system by letting $r = \frac{N}{M}$ with $N, M \to \infty$. The authors prove that the expected proportion of occupied I-queues, denoted ρ, satisfies the equation

$$\frac{\rho}{1-\rho} = r(1-\lambda) \tag{2.1}$$

for JIQ-Random, and

$$\sum_{i=1}^{\infty} \rho^{\frac{d^i-1}{d-1}} = r(1-\lambda) \tag{2.2}$$

for JIQ-SQ(d). They then move on to the primary system, and let $s = \lambda(1-\rho)$. With the assumptions at hand, they prove

$$\bar{T} = \frac{q_s}{s} \tag{2.3}$$

where $\bar{T}$ is the mean response time, q_s is the mean queue size of a $M/G/1$ server at reduced load s with the same service time distribution and service discipline.

The author of [12] uses another approach to analyse JIQ in the fluid limit, without the simplifying assumptions made by [5]. He derives limiting differential equations for the states of the I-queues (the secondary system) and the states of the server queues (the primary system). The derivatives are then set to zero to represent the system in equilibrium, and the equilibrium can be calculated numerically through a binary search on a single variable. Although no proof has been given that the computed value is unique in satisfying all equations, the computed equilibrium does accurately estimate the average response time with a relative error of 0.1%.

The limiting equations for several variations on JIQ are also given in [12], including JIQ with a reporting threshold of one, JIQ with LCFS I-queues and JIQ-SQ(2).

2.4.2 Heterogeneous dispatcher loads

Although JIQ is insensitive to server heterogeneity [15], it is sensitive to heterogeneous loads at the dispatchers [1]. The analyses in [5] and [12] implicitly assume equal loads at the dispatchers. In contrast, [1] lets these loads be different. This paper makes the distinction between the blocking and queuing scenario, and shows that JIQ-Random, with an arbitrarily small degree of skewness in the dispatcher loads, does not result in zero blocking or zero wait respectively in the fluid limit.

Two extensions are proposed to counter this performance degradation. In the first extension, a dispatcher is selected by the server to receive its token with the same probability that a job enters the system at this dispatcher. We note that this extension requires knowledge of the load distribution at the servers. However, the extension does allow JIQ to achieve zero blocking or zero wait in the fluid limit. In the second extension, tokens are exchanged by the dispatchers at an exponential rate. The author of [1] proves that zero blocking and zero wait are achieved in the fluid limit for a finite token-exchange rate.

3 Model description and notation

In our model, the load balancing system consists of N servers and M dispatchers, similar to [5], [12]. We assume that jobs arrive according to a Poisson process, such that the inter-arrival times are exponentially distributed with mean λN. We will refer to λN as the total load on the system, and we will refer to λ as the relative load on the system.

Moreover, we let the service times of the jobs be independent and exponentially distributed with mean 1. We are interested in the behaviour of the system in its steady state, so we impose $\lambda < 1$, which is a necessary condition for the system to stabilize. Furthermore, we assume that the load balancing systems considered in this thesis will converge to a steady state.

The Poisson arrival distribution and the exponentially distributed service times will make our model Markovian. As a result, we are able to formulate two birth-death processes in Chapter 5, such that we can approximate the behaviour of the system in its steady state.

3.1 Notation

Below is a comprehensive list of all notation used throughout this thesis.

- λ. The relative load on the system.
- M. The number of dispatchers in the system.
- N. The number of servers in the system.
- r. The fraction $\frac{N}{M}$.
- d. The diversity parameter for the JIQ-Redundancy(d) policy.
- S. One arbitrary but fixed server in the system, used to refer to the server under consideration while arguing about the effects of other servers and dispatchers on server S.
- D. One arbitrary but fixed dispatcher in the system, used to refer to the dispatcher under consideration while arguing about the effects of other servers and dispatchers on dispatcher D.
- s. The vector representing the approximate distribution of jobs at server S when the system has entered its steady state in the fluid limit. s_j will represent the approximate fraction of time that server S will have j jobs in its queue in the fluid limit. We will argue that s_j also approximates the fraction of servers of the whole system with j jobs in the queue.
- q. The vector representing the approximate distribution of tokens at dispatcher D when the system has entered its steady state in the fluid limit. q_i will represent the approximate fraction of time that dispatcher D will have i tokens in the fluid limit. We will argue that q_i also approximates the fraction of dispatchers of the whole system with i tokens.
- p. The probability that a job enters the system at a dispatcher without any tokens. We will approximate p in a finite system with a fixed-point equation, where we denote the n^{th} iteration by p_n.
- y. The average delay (either through analysis or simulation) in a finite system when the system has entered its steady state.
- m. The average number of messages per job arrival (either through analysis or simulation) in a finite system when the system has entered its steady state.
- α, β. The vector representing the relative load distribution on the dispatchers and the degree of skewness of this load distribution, respectively. We will elaborate on these definitions in Section 6.5.
- δ. The message delay, such that jobs and tokens arrive δ time units later than the time they have been sent. In our standard model, $\delta = 0$, but in Section 6.6 we will set δ to a positive value.

4 JIQ-Redundancy(d)

In this chapter we will introduce JIQ-Redundancy(d). The policy is based on the traditional JIQ algorithm introduced in [5], and further inspired by the mechanics of Redundancy-Scheduling.

4.1 Algorithm description

We can make a similar distinction between the secondary and primary load balancing system as in [5]. Both systems are characterized by two distinct types of events.

For the secondary system,

- When a server becomes idle, it advertises its availability by sending a token to d dispatchers ($d \leq M$), selected uniformly at random without repetition. Each dispatcher will save this token in a table. The server will save the addresses of the selected dispatchers.
- Once an idle server receives a job, it will send a cancel message to the d dispatchers it previously sent a token to. If the token is still present in its table, a dispatcher receiving a cancel message will delete the token corresponding to the server from its table.

For the primary system,

- When a job arrives at a dispatcher with a non-empty table, the dispatcher will *randomly* remove a token from its table, and dispatch the job to the server corresponding to the removed token. These jobs will be referred to as directly-dispatched.
- When a job arrives at a dispatcher with an empty table, the dispatcher will dispatch the job to a randomly selected server. These jobs will be referred to as randomly-dispatched.

Note that the novelty of JIQ-Redundancy is the fact that tokens do not have to be used, and can instead be cancelled. As a result, servers can advertise to multiple dispatchers, which decreases idle time without running the risk of receiving multiple jobs.

4.2 Imposed system requirements

The policy imposes several requirements on the load balancing system. First of all, the table at the dispatcher can have at most N tokens, such that it requires at most $\mathcal{O}(N)$ of memory space. We will later find that a dispatcher will have $\frac{(1-\lambda)Nd}{M}$ tokens on average. Furthermore, an idle server needs to keep track of the dispatchers keeping a token, which requires $\mathcal{O}(d\log(M))$ of memory space. Lastly, a single job may cause one idle server to send d cancel messages, and may cause another batch of d tokens if the same server becomes idle again, resulting in at most $2 \cdot d$ messages per job arrival. Hence, the additional burden on the network is $\mathcal{O}(d)$ messages per job arrival. The last requirement will be further refined in Section 5.4.

4.3 Design decisions

The design decisions made for JIQ-Redundancy can be formatted as answers to the following questions.

- *Why does a server need to send a token to multiple dispatchers?*

  The core disadvantage of traditional distributed JIQ is that the performance decreases significantly under high loads, because of an increased number of empty I-queues, which in turn causes more random dispatching. The authors of [5] propose to increase the reporting threshold, such that a server generates a token when it has k (or at most k, depending on the implementation) jobs left in its queue.

  JIQ-Redundancy lets servers advertise to multiple dispatchers in order to avoid random dispatching. However, it also recognizes the fact that there are roughly $(1 - \lambda) \cdot N$ idle servers at any point in time. Therefore, its goal is to direct jobs to idle servers as much as possible by generating multiple tokens at the same time, and only when a server becomes idle.

- *Why does a server need to send cancel messages?*

  The cancel messages are necessary to ensure that each token in a table corresponds to an idle server. The extension in [5] does not have this invariant, such that their reporting threshold parameter introduces a trade-off (directly quoted from [5]):

  > Decreasing the reporting threshold will increase the rate of random arrivals, which results in a larger queue size. On the other hand, increasing the reporting threshold will attract too many arrivals at once to an idle server and result in unbalanced load.

  As a result of the cancel messages, we can increase d without making a server attract multiple arrivals at once. Hence, the trade-off for JIQ-Redundancy is simpler: the communication overhead is inversely proportional to the delay. We will provide a formal analysis of the number of messages in Section 5.4.

  Lastly, we can decrease the number of cancel messages because we assume that a job contains information on the dispatcher that assigned the job to the server. If the address of the dispatcher matches one of the addresses that the server has saved, the server does not have to send a cancel message to this dispatcher, because the server is aware that its token in the dispatcher's table has already been removed. For small d, this is a notable improvement. Therefore, we use this optimized algorithm for the remainder of this thesis.

- *Why does a server sample dispatchers uniformly at random without repetition?*

  The ultimate goal of the secondary system is to reduce the probability that a table is empty. Theoretically, it would be best to probe (all or several) dispatchers for their table sizes, and then direct a token to the dispatchers with the least amount of tokens, analyzed in [5]. However, this would imply at least one additional message per generated token, additional complexity, and non-instantaneous token distribution in practice. Since we simply need non-empty tables, we prefer to increase d (which we can do without further inducing performance degradation besides communication overhead).

  In addition, servers sample without repetition. If a dispatcher was to keep more than one token from a single server, then the usage of one of those tokens would cause a cancel message for the other tokens, rendering them useless. It also introduces the invariant that each token at one dispatcher is unique.

- *Why does a dispatcher save tokens in a table?*

  We refer to the data structure keeping the tokens as a table. Distributed JIQ keeps an I-queue that either returns tokens in a FCFS or a LCFS manner, the latter of which has been proven to provide a small performance improvement in [12], since newer tokens are less likely to have been affected by random arrivals. This difference is irrelevant to the JIQ-Redundancy implementation, since each token will correspond to an idle server anyway.

Nonetheless, we still impose that dispatchers sample randomly from their table. This improves the average delay in case there is a communication delay between the dispatchers and the servers, which will be elaborated upon in Section 6.6. Furthermore, it enables our analysis on the distribution of jobs at a single server in Subsection 5.2.1 because we do not have to keep track of the ordering at which tokens arrived.

Therefore, the data structure only needs to provide efficient insertion and deletion. For example, a hash table could be a possible implementation of the table. This would provide an average time complexity of $\mathcal{O}(1)$ for both insertion and deletion.

- *Why does the dispatcher select a random server in case of an empty table?*

  We adhere to the convention that tokens are dispatched randomly in case of an empty table. Admittedly, a dispatcher could employ a push-based algorithm like JSQ(d) when it has no tokens, but such a policy is beyond the scope of this thesis.

4.4 Similar concepts in previous literature

After having defined the algorithm ourselves, we found several one-line remarks on the possibility of letting a server cancel a token or advertise to multiple dispatchers [1], [5], [12]. For example, the authors of [5] state that “each idle processor joins only one I-queue to avoid extra communication to withdraw from I-queues”, which is in fact, very similar to JIQ-Redundancy. However, this mechanism was not researched any further. Moreover, the authors propose an extension to JIQ with a varying reporting threshold to solve the same problem we will solve with JIQ-Redundancy.

5 Analytical Results

In this chapter we will approximate the behaviour of a JIQ-Redundancy(d) system in the fluid limit. We will first model the distribution of queue sizes and the distribution of tokens as two birth-death processes. We then use the stationary distributions of these birth-death processes to approximate the average delay and the number of messages per job arrival in finite systems. In Section 6.3, we will demonstrate the accuracy of our analytical results for systems with finite M and N in comparison with the corresponding statistics from simulations.

5.1 Distribution of queue sizes

To model the distribution of queue sizes, we will consider a single server S when the system has converged to its steady state. We let the state of this server S be described as the current number of jobs in its queue, denoted by $j \in \mathbb{N}_0$. We have omitted j's dependence on time in the notation, so that we can intuitively reuse j as an index in the stationary distribution of Subsection 5.1.3, which will be independent of time.

Similar to [5], [12], we let $N, M \to \infty$ with the ratio $r = \frac{N}{M}$ fixed. Since $N \to \infty$, we have that the state of server S is independent of the states of the other servers. Therefore, we model the number of jobs at server S as a birth-death process, and use the fraction of time that server S spends in state j according to the stationary distribution to approximate the fraction of servers in state j in the steady state of the whole system.

Furthermore, we consider the probability that a job enters the system at a dispatcher with an empty table to be p. Reversely, we consider the probability that a job enters the system at a non-empty table to be $1 - p$. Since we let $N \to \infty$, we consider probability p to be independent of the state of server S. We also consider probability p to be fixed to its average value in order to facilitate our analysis. In Section 5.2, we will approximate the average value of p.

In [12], the author states that "analyzing strategies where servers can be on multiple I-queues would necessarily complicate our analysis, as then we need to use variables that track the position of the servers in multiple I-queues". Fortunately, we do not have to track the position of the servers, because we will use probabilities to model when a token is used, which effectively replaces the need to track tokens' position. Since we have the invariant that all tokens correspond to idle servers, and we also have that all idle servers have a token at the dispatchers, we can use the number of idle servers to derive the number of tokens at the dispatchers. Subsequently, we find the average probability that a specific token is randomly sampled from a dispatcher's table.

5.1.1 Asymptotic independence

Note that when determining the upcoming birth-death process to describe the state at a single server, we assume that the states of every subset of the other servers are independent. Therefore, the analysis requires asymptotic independence between the servers. Although we will not formally prove the independence, we believe the validity of this condition to be plausible. The states of two arbitrary servers will be dependent when both servers have a token at the same dispatcher, but the probability that this happens converges to zero when the number of dispatchers grows to infinity, while d remains fixed. Similarly, the states of two arbitrary dispatchers will be dependent when both dispatchers have a token from the same server, but the probability that this happens converges to zero when the number of servers grows to infinity, while d remains fixed.

5.1.2 Construction of birth-death process

We have three transitions that govern the number of jobs in the queue of server S.

- The only death transition in the number of jobs at server S is a job completion. Since service times are assumed to be exponentially distributed with mean 1, this transition fires at rate 1 when $j > 0$.
- The two birth transitions are "filtered" forms of the Poisson arrival process. When a job enters the system, there are two mutually exclusive events that cause a birth transition. The probability that one of the events occurs is independent and identically distributed for each job arrival. Therefore, we will find the rate at which these transitions occur by multiplying the rate at which jobs arrive into the system with the probability that a job causes the event.
  - The first birth transition represents the arrival of a randomly-dispatched job at server S. We have that jobs enter the system at rate λN and are subsequently randomly-dispatched with probability p. Given that a job is randomly-dispatched, the probability that it arrives at server S is $\frac{1}{N}$. Therefore, the rate at which randomly-dispatched jobs arrive at server S is

$$\lambda N p \frac{1}{N} = \lambda p$$

  - The second birth transition represents the arrival of a directly-dispatched (through the usage of a token) job at server S. Note that this transition will only fire when $j = 0$, because S will only have tokens at the dispatchers when it is idle. We have that S distributed its tokens by selecting the receiving dispatchers uniformly at random without repetition. Therefore, there will be d dispatchers holding a token of S, where d is the diversity parameter of JIQ-Redundancy. As a result, the probability that a job arrives at a dispatcher with a token from S is $\frac{d}{M}$.
    We now need to find the probability that a dispatcher uses the token from S, given that a job arrived at this dispatcher and it holds a token from S. In the steady state, any stable load balancing system will have an average of $(1-\lambda)N$ idle servers, because there must be an average of λN busy servers to match the outflux of jobs with the influx. Hence, we have an average of $(1-\lambda)Nd$ tokens. These tokens have been distributed among $(1-p)M$ dispatchers. Note that we need the multiplication by $(1-p)$ because we need to compensate for the fact that we know that each of the d dispatchers has at least one token. Therefore, we expect an average of $\frac{(1-\lambda)Nd}{(1-p)M}$ tokens at each of the d dispatchers. Since a dispatcher will sample tokens randomly from their table, the probability that a dispatcher uses the token from S, given that a job arrived at this dispatcher and it holds a token from S, equals $\frac{(1-p)M}{(1-\lambda)Nd}$.
    Hence, we multiply the rate at which a job enters the system with the probability that it enters the system at one of the d dispatchers holding a token from S, and subsequently multiply with the probability that one of the dispatchers will use the token from S. Finally, the rate at which directly-dispatched jobs arrive at server S when $j = 0$ equals

$$\lambda N \frac{d}{M} \frac{(1-p)M}{(1-\lambda)Nd} = \frac{\lambda(1-p)}{1-\lambda}$$

  We have that the birth transitions are mutually exclusive events, they are fired with an independent probability at each job arrival, and jobs arrive according to a Poisson arrival distribution. Therefore, we can merge the two birth transitions when $j = 0$ into one birth transition with rate

$$\lambda p + \frac{\lambda(1-p)}{1-\lambda}$$

We visualize the birth-death process that describes the number of jobs in the queue of server S in Figure 5.1.

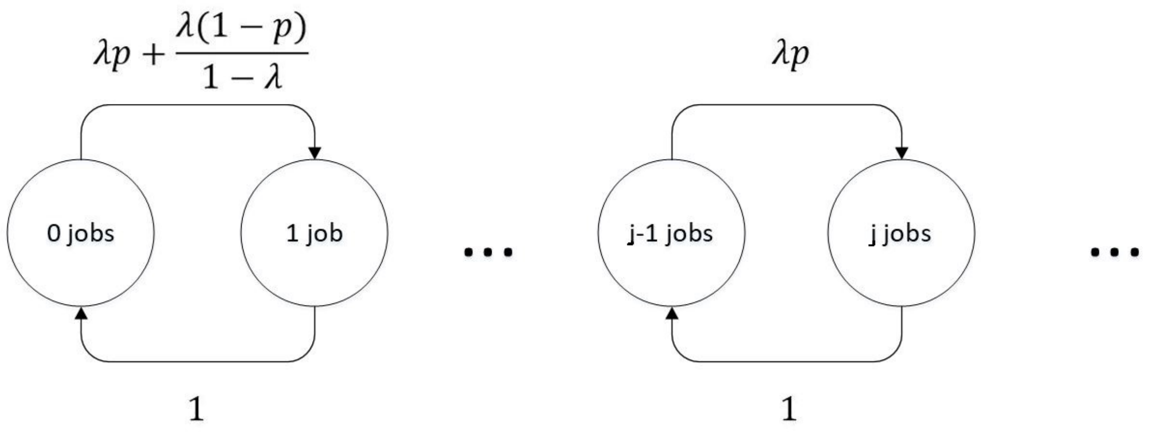


Figure 5.1: The birth-death process to describe the number of jobs in the queue of server S.

5.1.3 Stationary distribution

Now that we have modelled the number of jobs in the queue of server S as a birth-death process, we are interested in the stationary distribution of the birth-death process. We denote this distribution by s, such that s_j represents the fraction of time that server S has j jobs in its queue.

The first global balance equation gives us

$$s_1 = \left(\lambda p + \frac{\lambda(1-p)}{1-\lambda}\right) s_0 \tag{5.1}$$

Furthermore, the other global balance equations give us that for $j \geq 1$

$$s_{j+1} = (\lambda p) s_j \tag{5.2}$$

Hence, for $j \geq 1$ we get

$$s_j = (\lambda p)^{j-1} s_1 = (\lambda p)^{j-1} \left(\lambda p + \frac{\lambda(1-p)}{1-\lambda}\right) s_0 \tag{5.3}$$

Lastly, we impose the condition $\sum_{j=0}^{\infty} s_j = 1$ such that we arrive at a value for s_0

$$s_0 = 1 - \lambda \tag{5.4}$$

Fortunately, this value is to be expected for s_0. When jobs arrive to a load balancing system in its steady state with rate λN, there have to be λN non-idle servers, such that jobs complete at rate λN as well. As a result, $(1-\lambda)$ is the fraction of time that we can expect one server to be idle, because each server is assumed to be identical.

We can now further simplify Equation (5.3) so that for $j \geq 1$

$$s_j = (\lambda p)^{j-1} s_1 = (\lambda p)^{j-1} (\lambda(1 - \lambda p)) \tag{5.5}$$

To conclude, we have found the distribution s that describes the expected fractions of time that server S contains j jobs in the fluid limit. This distribution must be unique, since we have only used algebraic techniques to find the stationary distribution. We also note that server S was chosen arbitrarily, and the fluid limit makes the state of S independent of the state of other servers. As a result, we approximate the fraction of servers in state j in the whole system by s_j. For the remainder of the analysis, we will use s to approximate the number of servers in a system with finite M and N with queue size j by $s_j N$. We do this to facilitate further analyses, despite the fact that we have used a fluid limit to find s.

5.2 Probability p

Now that we have derived the distribution s in terms of p, we will approximate the average value of p in terms of λ, d and the ratio r using the same fluid limit we used to derive s. We will also adopt a similar approach as in Section 5.1, so we consider a single dispatcher D when the system has converged to its steady state. We let the state of D be described as the current number of tokens in the table, denoted by $i \in \mathbb{N}_0$. Note that the value of i has no upper bound because there may be an infinite amount of tokens at dispatcher D in the fluid limit. The state's dependence on time has again been omitted for the same reason as in Section 5.1. Since $M \to \infty$, we have that the state of dispatcher D is independent of the states of the other dispatchers. We will again use a birth-death process, and approximate the total fraction of dispatchers in state i by the fraction of time that dispatcher D spends in state i according to the stationary distribution of the birth-death process.

5.2.1 Construction of birth-death process

We have four transitions that govern the number of tokens at dispatcher D.

- The only birth transition is the event that a server sends dispatcher D a token. We will refer to this transition as the birth transition. In the steady state, servers complete their last job at rate $s_1 N$, because there are $s_1 N$ servers with one job left in their queue on average, and the service times are exponentially distributed with mean 1. A job completion causes d tokens to be distributed among M dispatchers, selected uniformly at random without repetition. Hence, D receives tokens at rate

$$\frac{s_1 N d}{M} = \lambda(1 - \lambda p) r d$$

.

- Three different transitions decrement the number of tokens at D.
  - The foremost transition is the usage of a token. A job arrival at D will cause exactly one token to disappear when $i > 0$. Note that this is where we use the invariant that all tokens at a single dispatcher are unique. We have that a job enters the system at dispatcher D with probability $\frac{1}{M}$. Therefore, we have that the number of tokens is decremented due to job arrivals at D with rate

$$\frac{\lambda N}{M} = \lambda r$$

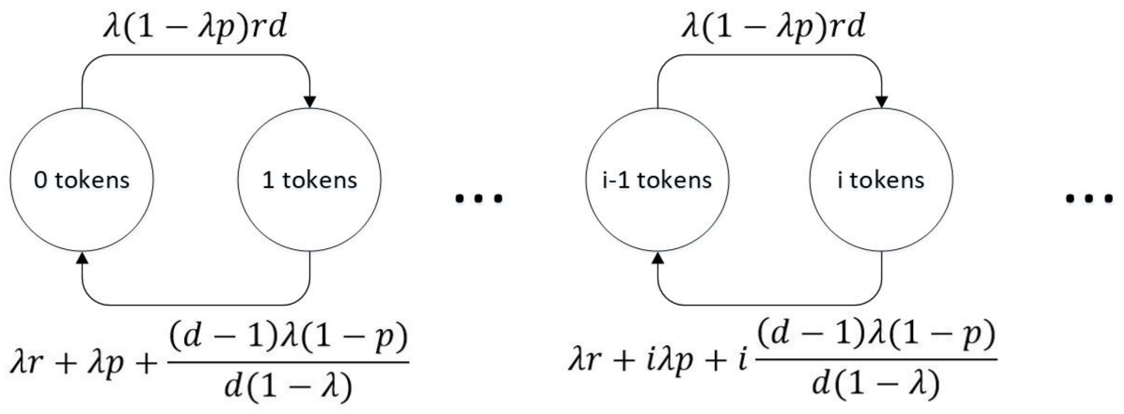
  .

  - In addition, we have that D can receive a cancel message for a token because of a random arrival to the corresponding idle server. We have that each token corresponds to a unique server in the table of D. Therefore, we can use the rate at which randomly-dispatched jobs arrive at a single server to derive the rate at which randomly-dispatched jobs arrive at the servers of the tokens of D. In Section 5.1, we found that randomly-dispatched jobs arrive at a single server at rate λp. We have i distinct servers, so the rate at which the number of tokens is decremented due to randomly-dispatched jobs equals

$$i\lambda p$$

  .

  - Lastly, we have that D can receive a cancel message for a token because another dispatcher used the same token. We can use the same argument as in the previous transition, because each token in the table of D corresponds to a unique idle server. We found in Section 5.1 that directly-dispatched jobs arrive at an idle server at rate $\frac{\lambda(1-p)}{1-\lambda}$. We have with probability



$\lambda(1-\lambda p)rd$
0 tokens
1 tokens
$\lambda r + \lambda p + \frac{(d-1)\lambda(1-p)}{d(1-\lambda)}$
$\lambda(1-\lambda p)rd$
i-1 tokens
i tokens
$\lambda r + i\lambda p + i\frac{(d-1)\lambda(1-p)}{d(1-\lambda)}$

Figure 5.2: The birth-death process to describe the number of tokens at dispatcher D.

$\frac{d-1}{d}$ that this directly-dispatched job was forwarded by an other dispatcher than dispatcher D, which would cause the cancel message at dispatcher D. We have i distinct idle servers, so the rate at which the number of tokens is decremented due to directly-dispatched jobs equals

$$i\frac{(d-1)\lambda(1-p)}{d(1-\lambda)}$$

Similar to Section 5.1, we can now aggregate the death transitions for all $i \geq 1$ into one death transition with rate

$$\lambda r + i\lambda p + i\frac{(d-1)\lambda(1-p)}{d(1-\lambda)}$$

We visualize the birth-death process that describes the number of tokens in the table of dispatcher D in Figure 5.2.

5.2.2 Stationary distribution

Now that we have constructed the birth-death process that describes the number of tokens at one dispatcher, we can again calculate the stationary distribution of the birth-death process. Although we would have enjoyed denoting this distribution by d in order to complete the parallel with Section 5.1, we have reserved this notation for our diversity parameter. Therefore, we denote the stationary distribution of our birth-death process for the number of tokens at dispatcher D by q.

The global balance equations for $i \geq 1$ are

$$((\lambda(1-\lambda p))rd)q_{i-1} = \left(\lambda r + i\lambda p + i\frac{(d-1)\lambda(1-p)}{d(1-\lambda)}\right) q_i$$

By induction we then have for $i \geq 1$

$$q_i = \left(\prod_{k=1}^{i} \frac{(\lambda(1-\lambda p))rd}{\lambda r + k\lambda p + k\frac{(d-1)\lambda(1-p)}{d(1-\lambda)}}\right) q_0$$

The condition $\sum_{i=0}^{\infty} q_i = 1$ then gives us a formula for q_0.

$$q_0 = \left(1 + \sum_{i=1}^{\infty}\left(\prod_{k=1}^{i} \frac{(\lambda(1-\lambda p))rd}{\lambda r + i\lambda p + i\frac{(d-1)\lambda(1-p)}{d(1-\lambda)}}\right)\right)^{-1} \tag{5.6}$$

Hence, we have found the distribution q that describes the expected fraction of time that dispatcher D contains i tokens. We repeat the logic of Subsection 5.1.3; we have chosen dispatcher D arbitrarily, and the fluid limit makes the state of D independent of the state of other dispatchers. As a result, the value of q_i should also approximate the fraction of dispatchers with i tokens in the steady state. Therefore, we have that $p = q_0$, because p represents the average probability that a job enters the system at a dispatcher with an empty table, and jobs see time averages because of the PASTA property (in combination with homogeneous dispatcher loads).

5.2.3 Approximating p in the finite case

We note that the formula for p $(= q_0)$ in Equation (5.6) is a fixed-point equation because it includes p itself, such that algebraically calculating the value of p is infeasible. In addition, it contains an infinite sum, but we have not found a closed-form solution of this expression. To approximate p with M and N finite, we define the sequence p_n and use the fixed-point iteration method

$$p_{n+1} = \left(1 + \sum_{i=1}^{N} \left(\prod_{j=1}^{i} \frac{(\lambda r(1-\lambda p_n))d}{\lambda r + i\lambda p_n + i\frac{(d-1)\lambda(1-p_n)}{d(1-\lambda)}} \right)\right)^{-1} \tag{5.7}$$

Although we were unable to formally prove that Equation (5.6) with $p = q_0$ is a contraction, we have experimented with various settings and $p_0 = 0.5$, and found that the sequence by Equation (5.7) converged in all cases. Several experiments are visualized in Figure 5.3. Therefore, we will use p_{100} as the value for p in finite system for the remainder of this thesis.

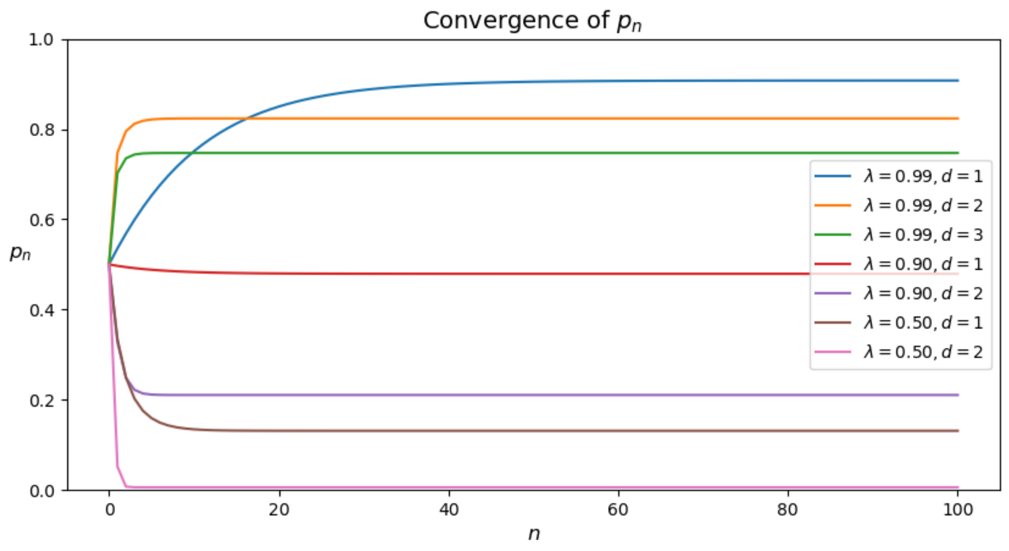


Figure 5.3: The sequence p_n for various values for λ and d with $M = 100$ and $N = 1000$.

5.2.4 Balls into Bins

Our initial approach was to consider all dispatchers at once, and model the fraction of empty tables as a variation on the Balls into Bins problem. We then planned to use the fraction of empty tables as an approximation for p. However, the Balls into Bins model does not incorporate time as a relevant factor; it implicitly assumes that all balls are distributed into the bins at once. As a result, the probability that a dispatcher receives x tokens when k tokens are distributed, does not adequately represent the fraction of time that this dispatcher contains x tokens. We need the latter, because the job arrivals see time averages by the PASTA property. Hence, the Balls into Bins model is not suitable to model the fraction of empty tables. We will see that the Balls into Bins model severely undershoots the probability that a table is empty for low λ and finite M and N in Figure 5.4.

5.2.5 Non-numerical evaluations

Now that we can calculate an approximation for p, we can perform several non-numerical evaluations for finite M and N. First of all, we have an approximate $s_0 rd$ number of tokens per dispatcher, which should correspond to $\sum_{i=1}^{N} iq_i$. Fortunately, all our experiments show that this sum does indeed equal the expected number of tokens per dispatcher.

In addition, we can use [5]. This paper analyzes the fluid limit of JIQ under the simplifying assumptions that an idle server has exactly one token at a dispatcher, and that all tokens correspond to idle servers. These assumptions are in fact invariants in our case, so that we can expect that their analytical results will show similar behaviour to our results with $d = 1$. For JIQ-Random and the assumptions mentioned, the authors proved that $\frac{\rho}{1-\rho} = r(1-\lambda)$. The parameter ρ is defined to be the proportion of occupied I-queues, such that $\rho = 1 - p$, because p is equal to the proportion of empty tables under the model assumption of homogeneous dispatcher loads. Therefore, we check whether the following approximation holds

$$p \approx \frac{1}{r(1-\lambda)+1} \tag{5.8}$$

We have found that Equation (5.8) is a good approximation when $d = 1$, and becomes more accurate for higher λ. We have included this approximation in Figure 5.4. The small difference in performance comes from the fact that the servers under JIQ-Redundancy will cancel their token when they receive a randomly-dispatched job. Naturally, we attempted to extend the result by [5] to include d with Equation (5.9). Unfortunately, the analysis in [5] does not account for cancel messages. Since cancel messages become more common as d grows, the approximation worsens with d.

$$p \approx \frac{1}{dr(1-\lambda)+1} \tag{5.9}$$

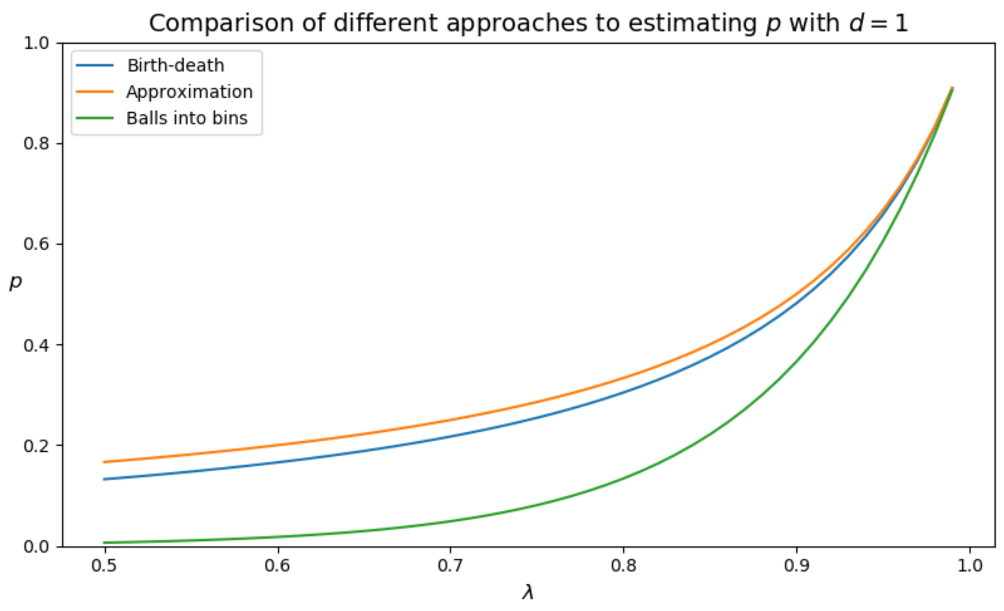


Figure 5.4: Several approaches to approximating p with $d = 1$ and $M = 100$ and $N = 1000$, including the approximation by the birth-death process, the approximation by [5] and the approximation by the Balls into Bins model.

5.3 Delay

We can approximate the average delay in a finite system in its steady state using the stationary distribution s and p with

$$y = (1 - p) + p \sum_{i=0}^{\infty} (i + 1)s_i \tag{5.10}$$

The first term of the equation covers all directly-dispatched jobs. We approximate the probability that a job arrives at a non-empty table with $(1 - p)$. These jobs will certainly be dispatched to an idle server, and hence have an expected delay of 1.

In addition, we approximate the probability that a job arrives at an empty table with p. In this case, the job will be randomly dispatched, and the probability that this job will be dispatched to a server with queue size i can be approximated with s_i. In that case, the job will have to wait $i + 1$ time units on average.

Filling in the value of s_i for each i, we can simplify Equation (5.10) to

$$y = \frac{1}{1 - \lambda p} \tag{5.11}$$

5.4 Number of messages

A server causes two types of messages: tokens and their cancel messages. The former is generated when a server becomes idle, and the latter is generated when a server leaves this idle state. In the steady state, we have that the expected number of idle servers has converged, such that servers enter and leave the idle state at the same rate.

In the steady state, we expect approximately $s_1 N$ servers to each have one job left in their queue. Since we have assumed the service times to be exponential with mean 1, we expect $s_1 N$ servers to become idle in one time unit. Each of these completions will cause d tokens. In the same time unit, $s_1 N$ servers will leave the idle state. We assume that an arriving job contains information on the dispatcher that assigned the job to the server. Therefore, if a directly-dispatched job arrives at a server, the server does not have to send a cancel message to the dispatcher that sent the job.

Hence, we will split the rate $s_1 N$ at which servers leave their idle state based on whether they received a randomly-dispatched job or a directly-dispatched job. In Section 5.1 we found that idle servers receive randomly-dispatched jobs at rate λp, and directly-dispatched jobs at rate $\frac{\lambda(1-p)}{1-\lambda}$ in the fluid limit. Therefore, the probability that a job arriving at an idle server was randomly dispatched in a finite system can be approximated by

$$\frac{\lambda p}{\lambda p + \frac{\lambda(1-p)}{1-\lambda}} = \frac{p - \lambda p}{1 - \lambda p}$$

Moreover, the probability that a job arriving at an idle server was directly dispatched can be approximated by

$$\frac{\left(\frac{\lambda(1-p)}{1-\lambda}\right)}{\lambda p + \frac{\lambda(1-p)}{1-\lambda}} = \frac{1-p}{1-\lambda p}$$

Additionally, we have that $ds_1 N$ tokens are distributed in one time unit. Finally, we have that the expected number of messages in one time unit equals

$$ds_1 N + ds_1 N \frac{p - \lambda p}{1 - \lambda p} + (d-1)s_1 N \frac{1-p}{1-\lambda p} = 2d(\lambda N(1-\lambda p)) - \lambda N(1-p) \tag{5.12}$$

To adhere to the convention of presenting the number of messages per job arrival, we divide the value in 5.12 by λN, the expected number of arrivals per time unit. We arrive at an approximation for the average number of messages per job arrival, denoted by m

$$m = 2d(1-\lambda p) - (1-p) \tag{5.13}$$

Note that without the optimization we would have $2d(1-\lambda p)$ messages per job arrival, such that the communication overhead would be inversely proportional to the delay. This was our initial approximation for m, but we found the term $(1-p)$ to be too significant to leave out. In fact, we find that this optimization causes $m < 1$ for $d = 1$ and $\lambda > 0.5$.

6 Simulation Results

In this chapter we will simulate JIQ-Redundancy with various settings for the input parameters λ, M, N, d. We will first explain the implementation of our simulation, in addition to how we extracted the relevant statistics from the simulation. We will follow with a comparison between our approximations for y, p, m and s derived in Chapter 5 and the corresponding statistics from the simulations. We will then test the robustness of the algorithm with variations on the classical setting, including heterogeneous loads at the dispatchers and message delay.

6.1 Event-based scheduling

The simulation is implemented with event-based scheduling. A central component of this technique are events. Each event will contain the timestamp at which it should occur, and information on what should happen once this event occurs. In our simulation, we have the following events:

- Job arrival at a dispatcher. This will cause the dispatcher to use the load balancing policy to forward the job towards a server, so that it will generate an event for the job arrival at the server.
- Job arrival at a server. This will cause the server to place the job in its queue and possibly perform other actions based on the load balancing policy.
- Job departure at a server. This will cause the server to remove the job from its queue and possibly perform other actions based on the load balancing policy. In addition, if there are jobs left in the queue of the server, the server will generate the departure event for the next job.

It is important to see that job arrivals at dispatchers basically create a chain of events so that each event will generate the next one. Furthermore, note that in our basic model, there is no time difference between the job arrival at the dispatcher and the job arrival at the server. However, we will later introduce message delay, such that these events should be separated.

Each event will be added to and stored in a heap data structure called the Future Event Set (FES), which will allow us to efficiently retrieve the next event. The current time in the system will then be the timestamp of the event that is currently being handled. We can simply use the timestamp of an event, in addition to a sample from a particular distribution, to calculate the timestamp of the event that should be generated from the former event.

6.1.1 Correctness

The correctness of this technique relies on the condition that each event will have a timestamp equal or higher than the event that generated this event. In other words, the sample from the distribution must be non-negative. In short, at the time of an event, the state of all dispatchers and servers is fully determined by all previous events, such that each event will be correctly handled with the current state of the dispatchers, the servers, and the load balancing policy.

To give an example where this condition would be violated, suppose event B was generated by event A. When event B were to have a timestamp lower than the timestamp of event A, the FES would still first return event A, because event B has not yet been generated. Afterwards, event B would be returned by the FES. As a result, the simulation would not go through time sequentially, and event B might not be correctly handled because the state of the dispatchers and servers has already been affected by event A.

6.1.2 The alternative: discrete-time simulation

Another option was to implement the simulation with the popular technique of discrete-time simulation. With this approach, the simulation is governed by a clock. Each time interval, a certain number of jobs will arrive at a dispatcher according to the arrival distribution, and jobs will be forwarded according to the state of the dispatcher at that time interval. In addition, a certain number of jobs will finish at a server according to the service time distribution. Note that this technique requires to make a choice on the time interval between each step. In fact, the time interval for a particular settings of input parameters is crucial for acquiring accurate results. If the interval is too large, the simulation will not accurately represent the system. If it is too small, the simulation itself will take longer than needed. In contrast, in event-based scheduling the accuracy of the results is only affected by floating point errors, and the running time of the simulation is only dependent on the aforementioned input parameters, and the number of jobs to be simulated.

6.2 Collection of simulation statistics

Naturally, the goal of the simulation is to produce relevant statistics, which need to be extracted from the simulation. In Subsection 6.2.1 we conclude that each simulation should take $\frac{200N}{1-\lambda}$ jobs, of which the second half will be used to extract the relevant statistics. In Subsection 6.2.2 we explain our approach to calculating the number of simulations.

6.2.1 Warm-up period

An important issue within the field of load balancing and stochastic processes as a whole is the correct identification of the warm-up period. It takes time for a system to converge to its steady state, but we are only interested in the performance measures of the system once it has reasonably converged to this steady state. Otherwise, the results of our simulation would depend on the starting state. Moreover, the length of a warm-up period will be negligible compared to the active time of the load balancing system in practice, such that this should also be reflected in our simulation results.

To determine when we should start collecting the statistics within the simulation, we visually inspect the running average of the job delay, y. Note that we implicitly assume here that a stabilization in the running average of the job delay would indicate the convergence of the system towards its steady-state, including all other reported performance measures, including p, m and s. For each set of input parameters, we inspected the running average of the job delay over time, and came to the conclusion that the warm-up period can be avoided by starting the collection after $\frac{100N}{1-\lambda}$ jobs. Once collection starts, we continue the simulation for another $\frac{100N}{1-\lambda}$ jobs. We realize that $\frac{100N}{1-\lambda}$ is not a tight upper bound for the warm-up period, but it does avoid the warm-up period in all cases we have considered during the simulations. By making the number of jobs to be simulated dependent on the input parameters, we save time running the simulations while still maintaining accurate results.

6.2.2 Number of simulations

In addition to averaging the output parameters within one simulation, we also average the outcomes between simulations. This is to ensure that we also average out any possible effects within one simulation that may not let the output parameter converge to the true average.

We keep running simulations until the relative error in the reported output parameters is at most 1% with 90% confidence. The confidence interval for the average of an output parameter, denoted by $\bar{x}$, is $[\bar{x} - \dfrac{z \cdot \sigma}{\sqrt{n}}, \bar{x} + \dfrac{z \cdot \sigma}{\sqrt{n}}]$, where σ is the standard deviation between the averages, n is the number of simulations, and z is the appropriate z-value from the standard normal distribution. Hence, we simulate until $\frac{z \cdot \sigma}{\bar{x} \cdot \sqrt{n}} < 0.01$. Since we want 90% confidence, $z = 1.645$.

6.3 Comparison with analytical results

We evaluate the accuracy of our analytical results by simulating a finite system with the JIQ-Redundancy policy and comparing the approximations for y, p, m and s derived in Chapter 5 with the corresponding statistics from the simulations. We will constructively consider settings with a low, high and extremely high load. The results can be found in Table 6.1.

Input				**Simulation results**					**Analytical results**				
λ	M	N	d	y	p	m	s_1	s_2	y	p	m	s_1	s_2
0.5	100	1000	1	1.07	0.132	1.00	0.466	0.031	1.07	0.131	1.00	0.467	0.031
0.9	100	1000	1	1.78	0.483	0.61	0.508	0.221	1.76	0.479	0.62	0.512	0.221
0.9	100	1000	2	1.25	0.215	2.44	0.725	0.140	1.23	0.211	2.45	0.729	0.138
0.9	100	1000	3	1.10	0.098	4.57	0.819	0.072	1.09	0.093	4.59	0.825	0.069
0.99	100	1000	1	9.92	0.907	0.11	0.100	0.091	9.85	0.908	0.11	0.101	0.090
0.99	100	1000	2	5.53	0.825	0.56	0.182	0.148	5.41	0.823	0.56	0.183	0.149
0.99	100	1000	3	3.96	0.749	1.30	0.255	0.188	3.84	0.747	1.31	0.258	0.191

Table 6.1: The simulation statistics for y, p, m and s from a simulated system with $M = 100$ and $N = 1000$ and the corresponding approximations derived in Chapter 5.

We have fixed $M = 100$ and $N = 1000$ for these simulations, and found that our approximations are very accurate for $\lambda \leq 0.9$ with a relative error of at most 1%. At $\lambda = 0.99$ we find that our approximations are off with a relative error of at most 3%. However, a small number of simulations with larger M, N indicate that our approximations become more accurate for $\lambda = 0.99$, but we would have to run more simulations to provide the true averages of the simulation statistics.

6.4 Comparison with other policies

This section is intended to compare JIQ-Redundancy with other policies in the setting of $M = 100$, $N = 1000$ and various λ. More specifically, we compare JIQ-Redundancy with $d = 1$ and $d = 2$ with the pull-based JIQ and JIQ with a reporting threshold of one, denoted JIQ(1) [5], [12]. In addition, we compare with the popular push-based policy JSQ(d). We choose to report the performance of JSQ(2), as this policy will cause 4 messages per job arrival, which upper bounds the average number of messages per job arrival in JIQ-Redundancy(2).

We note that several push-based policies exist that employ memory [16]–[19]. However, these policies have been developed for a single dispatcher. In the multiple-dispatcher setting, [20], [21] introduce push-based policies with memory that estimate the queue sizes of the servers, but these policies assume a time-slotted system. As a result, we find that these policies are not suitable for comparison with JIQ-Redundancy.

We have visualized the average delay by various policies in Figure 6.1, and the number of messages per job arrival in Figure 6.2.

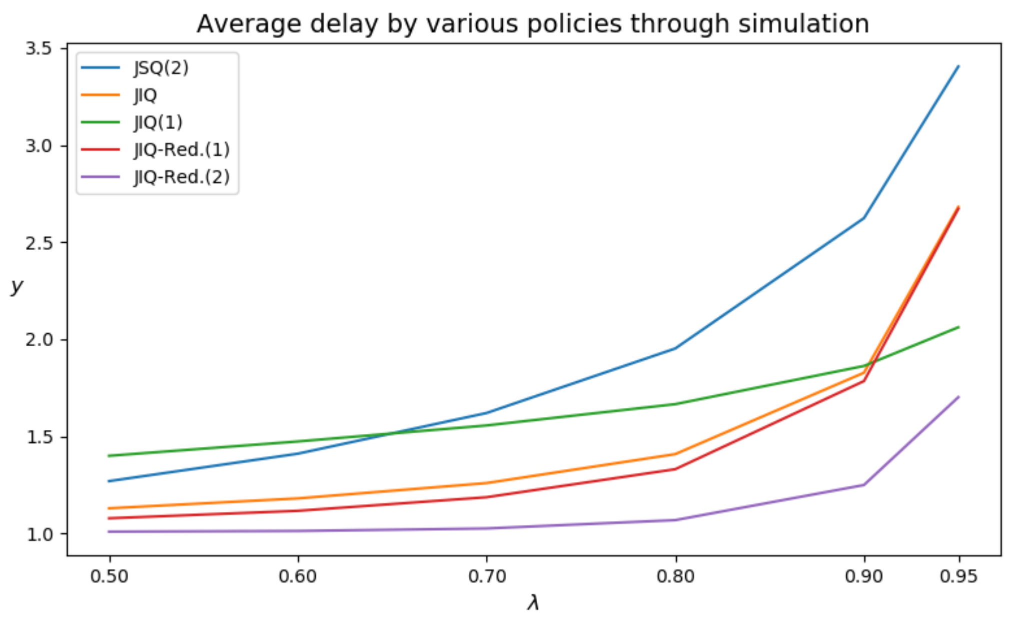


Figure 6.1: Average delay by various policies. The results were acquired by simulating a system with $M = 100$ and $N = 1000$.

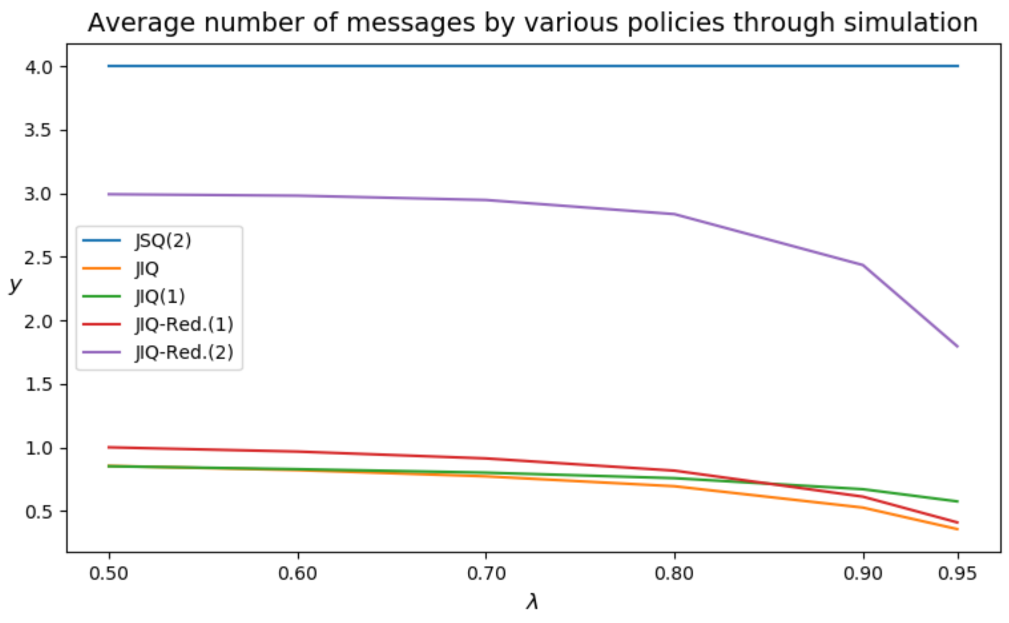


Figure 6.2: Average number of messages per job arrival by various policies. The results were acquired by simulating a system with $M = 100$ and $N = 1000$.

Unsurprisingly, JSQ(2) performs worse than the policies that use memory at the dispatchers. Furthermore, it is interesting to note that the performance improvement of JIQ-Redundancy(1) compared to JIQ decreases when λ increases. When $(1 - \lambda)Nd \leq M$, there are less tokens than dispatchers on average. As a consequence, cancelling a token because the server has become non-idle has a negative effect, because forwarding a job to a server with one job would have been better than random dispatching. For the same reason, JIQ with a reporting threshold of one performs better than JIQ in this setting.

For JIQ-Redundancy, we can set $d = 2$ to "postpone" this effect to higher λ. In fact, we find that JIQ-Redundancy(2) has a better average delay than JIQ(1) in the range of λ's we considered. However, the communication overhead is considerably larger. In JIQ(k), the number of messages per job arrival will always be less than one, because it can be approximated by the fraction of servers with a queue size equal to or smaller than k. In contrast, we approximated the number of messages in Section 5.4, and found that it grows with d. One must therefore evaluate whether the performance improvement is worth the additional communication overhead when considering to implement JIQ-Redundancy(d) instead of JIQ(k).

6.5 Heterogeneous dispatcher load

We also test the performance of the JIQ-Redundancy policy with heterogeneous dispatcher loads. The papers [5], [12] and [15] all assume that the loads at the dispatchers are equal. Instead, [1] assumes the load across the dispatchers to be asymmetrical. Surprisingly, the author shows that the expected delay in the fluid limit is strictly larger than one for an arbitrarily small degree of skewness, and an arbitrarily low load. This is because all tokens will eventually have accumulated at the least-loaded dispatcher, such that the performance of the system degrades to the Random policy. The author proposes two different approaches to solving this problem, which can be found in Subsection 2.4.2. This section is intended to empirically verify our intuition that JIQ-Redundancy also solves the problem of token accumulation at the least-loaded dispatcher.

We will examine the behaviour of the system with JIQ and JIQ-Redundancy under heterogeneous dispatcher loads with $M = 100$ and $N = 1000$. Similar to [1], we denote the probability that a job enters the system at dispatcher i to be α_i, where the dispatchers are indexed such that $\alpha_1 \leq \alpha_2 \leq \ldots \leq \alpha_M$. Needless to say, $\sum_{i=1}^{M} \alpha_i = 1$. Additionally, we will vary the degree of skewness with parameter β, such that $\alpha_M = \beta\alpha_1$. Lastly, we assign values to the α_i's such that they will form an evenly-spaced sequence between α_1 and α_M. For example, we have that

$$\beta = 3 \implies \alpha = (0.005, 0.00510101, \ldots, 0.1489899, 0.015)$$

We present the results for heterogeneous dispatcher loads with the JIQ policy in Table 6.2. The results for JIQ-Redundancy can be found in Table 6.3.

Input				**Simulation results**		
β	λ	M	N	y	p	s_1
1	0.5	100	1000	1.13	0.160	0.442
3	0.5	100	1000	1.39	0.489	0.356
10	0.5	100	1000	1.71	0.801	0.289

Table 6.2: JIQ performance under heterogeneous dispatcher loads.

Input					Simulation results		
β	λ	M	N	d	y	p	s1
3	0.5	100	1000	1	1.14	0.239	0.441
3	0.5	100	1000	2	1.01	0.023	0.494
3	0.5	100	1000	3	1.00	0.001	0.500
10	0.5	100	1000	1	1.21	0.346	0.414
10	0.5	100	1000	2	1.04	0.071	0.482
10	0.5	100	1000	3	1.00	0.005	0.499

Table 6.3: JIQ-Redundancy performance under heterogeneous dispatcher loads.

Evidently, JIQ-Redundancy also solves the problem of token accumulation at the least-loaded dispatcher. Even with $d = 1$ we find that the performance degradation is greatly mitigated when compared to JIQ. We ascribe this robustness to the mechanism that tokens can be cancelled. When the least-loaded dispatcher throttles a token, a single randomly-dispatched job to the corresponding idle server is enough to "free" the server. In contrast, a token in JIQ must wait for multiple job arrivals at the least-loaded dispatcher until its turn to be used has come. Hence, the time spent at the least-loaded dispatcher will be considerably less with JIQ-Redundancy. As a result, the token utilization in JIQ-Redundancy is much higher, such that more jobs are dispatched to an idle server. Moreover, increasing d will reduce the effect of having an asymmetric dispatcher load. When $d > 1$, a throttled token can be cancelled through the usage of its copies, such that cancelling the token does not require a costly randomly-dispatched job. In addition, increasing d will also directly increase the probability that a highly-loaded dispatcher has a token. Given these results, it is plausible that the expected waiting time will indeed vanish in the fluid limit for JIQ-Redundancy.

We also find that our analytical results remain accurate when using the average p from the simulations results. Filling in Equation (5.5) and Equation (5.11) with the value for p returned by the simulations will provide an accurate approximation for s and y respectively. As a result, we may conclude that the variation of heterogeneous dispatcher loads only affects the system through the parameter p. One reason that our formula has become incorrect is that we know that the first death transition in the birth-death process of Subsection 5.2.1 has become invalid. Jobs do not arrive at rate $\frac{\lambda N}{M}$ at dispatcher D anymore, but rather at rate $\alpha_i \lambda N$ where i is the index of dispatcher D. Evidently, our approximation for p in Equation (5.7) is not linear in α_i. Therefore, the average of the approximations for p using the α_i of each individual dispatcher does not equal the approximation for p using the average value of α_i. As a consequence, our approximation for p has become inaccurate. Besides, more jobs arrive at highly-loaded dispatchers anyway, which implies that Equation (5.7) should be adapted to calculate a weighted average based on α_i.

6.6 Message delay

The following simulations are intended to test our intuition that letting dispatchers sample randomly from their table should provide a small performance improvement when compared to using a LCFS or FCFS queue at the dispatchers. This is relevant for bringing the policy to practice, since dispatchers and servers are separate components in load balancing systems, either as different software or hardware components. As a result of this separation, message delay occurs.

We evaluate JIQ-Redundancy under the same model as in described in Chapter 3, but with the key difference that messages are delayed with δ time units. This includes that jobs will arrive at the server δ time units after they have been dispatched. For JIQ-Redundancy, this also includes that tokens and cancel messages will arrive δ time units after they have been sent. This breaks the invariant that all tokens in the table of a dispatcher correspond to idle servers.

We believe that the FCFS and LCFS sampling policies, that order the tokens based on their time of arrival, cause a higher probability that a dispatcher uses a token of which a copy has already been used by another dispatcher. We will refer to such an event as a clash. Randomly sampling from the table does not take the arrival times of the tokens into account, such that the probability of a clash will solely depend on the average number of tokens in the table.

We present our results with $\delta = 0.1$ and $d = 3$ in Figure 6.3. We use the quantity $y - \hat{y} - \delta$ as our measure for the performance degradation. The value $\hat{y}$ represents the average delay of the system in the same setting, but with $\delta = 0$ (which also implies that the sampling policy at the dispatcher is irrelevant). We also subtract the value δ, since it takes δ time units for a job to travel from the dispatcher to the server.

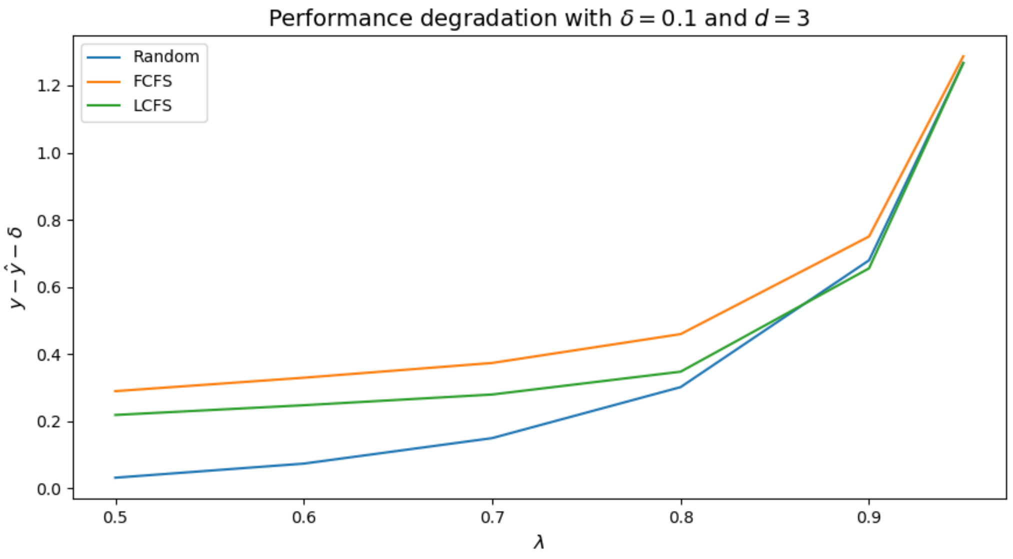


Figure 6.3: The average performance degradation for a JIQ-Redundancy system with message delay ($\delta = 0.1$) with diversity parameter $d = 3$.

Figure 6.3 shows that, indeed, randomly sampling tokens from the table is better than FCFS and LCFS when message delay is present. However, the effect of the sampling policy disappears for higher λ. This is to be expected, because the sampling policy becomes irrelevant when a table contains at most one token. Besides clashes, we also found that the performance degradation is partly due to a decrease in the total number of tokens in the system. Moreover, we found that LCFS performs better in this setting than FCFS. Intuitively, variations in queue sizes should decrease the probability of a clash in FCFS because it decreases the probability that two copies of the same token are in front of the queue at the same time. In LCFS, copies will be at the front of the queue immediately, which should increase the probability of a clash. Hence, there must be other factors that degrade the performance of the system, but we defer this research to future work. Although we fixed $\delta = 0.1$ and $d > 1$ in Figure 6.3, we found that randomly sampling remained better in various experiments with $\delta > 0$ and $d > 1$.

7 Conclusion

In this thesis, we have analyzed and simulated JIQ-Redundancy in the setting of Poisson arrivals and exponentially distributed service times. JIQ-Redundancy exploits the fact that we can always expect idle servers in the system, and increases the odds that new jobs are forwarded to these idle servers. Throughout this thesis we have supported all decisions in the design of the policy either through argumentation, analysis, or empirical evidence.

We found that the policy is mathematically tractable as a result of the two invariants it establishes; each token in a table corresponds to an idle server, and each token in a table is unique. We were then able to approximate the distribution of queue sizes and the distribution of tokens in the fluid limit. As a corollary, we could approximate the performance measures of average delay and the average number of messages per job arrival for finite systems.

In the analysis we focused on finding the means of the performance measures. Although we found that our results were accurate in Section 6.3, we believe that the simplicity of the algorithm allows for more elaborate analyses, like deriving distributional properties of the performance measures. This could be of interest when bringing the policy to practice. For instance, it is relevant for the network of a data center to know the maximum communication overhead it should be able to handle at any point in time, and possibly the amount of time it should be able to withstand such overhead. Another example could be that a service provider is interested in the maximum delay a customer might experience, or how often it exceeds a particular value, which might be the vital factor in the retention of the customer for future services. As a consequence, approximations for the averages of the performance measures might not always be the values of interest to the implementers of JIQ-Redundancy. Therefore, we suggest such analyses for future work.

Naturally, the model used in the analysis limits the relevancy of our analytical results as well. The Poisson arrival distribution, exponentially-distributed service times, and the fluid limit allowed us to model the system behaviour in birth-death processes, which ultimately enabled us to approximate the relevant time averages. Furthermore, the PASTA property allowed us to employ the derived time-average system to make conclusions on how the job arrivals would encounter the dispatchers, specifically on how many dispatchers would have at least one token. Subsequently, we assumed instantaneous dispatchment, such that randomly-dispatched jobs would also encounter the servers in their time-averages in terms of number of jobs queued. We would be interested in extending the analytical framework to cover all arrival distributions, including batch arrivals, but it might require other techniques to describe the system at the time of a job arrival.

After our analytical results, we provided our approach to simulating the load balancing system and demonstrated the accuracy of our results. It appeared that our approximations are accurate for suitably large M and N. For $M = 100$ and $N = 1000$ we found that the relative error in the delay was at most 3% for the extremely high load of $\lambda = 0.99$, and we expect our results to become more accurate as M and N grow. Furthermore, the effects of letting $d > 1$ are promising. In practice, one can easily control the parameter to balance the average communication overhead with the average delay in order to satisfy the requirements of the system. Certainly, the load on the load balancing system will not be stable in practice, and a fixed d would either imply too much communication overhead for a low relative load or too much delay for a high relative load. Therefore, we suggest the construction of JIQ-Redundancy systems that are adaptive to variations in λ through varying d for future work.

Lastly, we note that Chapter 6 leaves room for more future work. Natural variations that come to mind when considering a new load balancing policy are heterogeneous jobs, heterogeneous servers, general service time distributions, batch arrivals etc., but we were unable to consider such variations due to time constraints. Furthermore, we focused on comparing different policies, instead of rigorously researching why certain variations have the effects that they do. To illustrate, in Section 6.5 we find that JIQ-Redundancy is more robust against heterogeneous dispatcher loads compared to JIQ. We provide an intuitive explanation on the results, but omit a rigorous analysis to prove our conclusion. However, we believe that our analytical results can be extended to incorporate heterogeneous dispatcher loads in order to prove that the expected waiting time does indeed vanish under the variation. In Section 6.6, we provide empirical evidence for our design decision of randomly sampling tokens from the tables, but it would be useful to model the system with message delay analytically. As a result of time constraints, we defer the aforementioned variations and analyses to future work.

Bibliography

[1] M. van der Boor, S. Borst, and J. van Leeuwaarden, "Load balancing in large-scale systems with multiple dispatchers," *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, 2017.

[2] A. Ephremides, P. Varaiya, and J. Walrand, "A simple dynamic routing problem," *IEEE Transactions on Automatic Control*, vol. 25, no. 4, pp. 690–693, 1980.

[3] W. Winston, "Optimality of the Shortest Line Discipline," *Journal of Applied Probability*, vol. 14, no. 1, pp. 181–189, 1977.

[4] R. Badonnel and M. Burgess, "Dynamic pull-based load balancing for autonomic servers," *NOMS 2008 - IEEE/IFIP Network Operations and Management Symposium: Pervasive Management for Ubiquitous Networks and Services*, no. May 2008, pp. 751–754, 2008.

[5] Y. Lu, Q. Xie, G. Kliot, A. Geller, J. R. Larus, and A. Greenberg, "Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services," *Performance Evaluation*, vol. 68, no. 11, pp. 1056–1071, 2011.

[6] S. Whitt and W. Halfin, "Heavy-Traffic Limits for Queues with Many Exponential Servers," *Operations Research*, vol. 29, no. 3, pp. 567–588, 1981.

[7] M. van der Boor, S. C. Borst, J. S. van Leeuwaarden, and D. Mukherjee, "Scalable load balancing in networked systems: A survey of recent advances," *arXiv*, pp. 1–66, 2018.

[8] M. Mitzenmacher, "The power of two random choices in randomized load balancing," Ph.D. dissertation, University of California, Berkeley, 1996.

[9] N. Vvedenskaya, R. Dobrushin, and F. Karpelevich, "Queuing System with Selection of the Shortest of Two Queues: An Asymptotic Approach," *Problemy Peredachi Informatsii*, vol. 32, no. 1, pp. 20–34, 1996.

[10] B. D. Gamarnik, J. N. Tsitsiklis, and M. Zubeldia, "Delay, memory and messaging tradeoffs in distributed service systems," *Proc. ACM SIGMETRICS 2016*, pp. 1–12, 2016.

[11] M. van der Boor, M. Zubeldia, and S. Borst, "Zero-wait load balancing with sparse messaging," *Operations Research Letters*, vol. 48, no. 3, pp. 368–375, 2020.

[12] M. Mitzenmacher, "Analyzing distributed Join-Idle-Queue: A fluid limit approach," *54th Annual Allerton Conference on Communication, Control, and Computing, Allerton 2016*, pp. 312–318, 2017.

[13] X. Zhou, F. Wu, J. Tan, Y. Sun, and N. Shroff, "Designing low-complexity heavy-traffic delay-optimal load balancing schemes: Theory to algorithms," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 1, no. 2, pp. 1–30, 2017.

[14] R. Gandhi, H. Liu, Y. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang, "Duet: Cloud scale load balancing with hardware and software," *Computer Communication Review*, vol. 44, no. 4, pp. 27–38, 2015.

[15] A. L. Stolyar, "Pull-based load distribution among heterogeneous parallel servers: the case of multiple routers," *Queueing Systems*, vol. 85, no. 1-2, pp. 31–65, 2017.

[16] M. Mitzenmacher, B. Prabhakar, and D. Shah, "Load balancing with memory," *Annual Symposium on Foundations of Computer Science - Proceedings*, pp. 799–808, 2002.

[17] J. Anselmi and F. Dufour, "Power-of-d-Choices with Memory: Fluid Limit and Optimality," *Mathematics of Operations Research*, vol. 45, no. 3, pp. 862–888, 2020.

[18] T. Hellemans and B. Van Houdt, "Performance Analysis of Load Balancing Policies with Memory," *ACM International Conference Proceeding Series*, pp. 27–34, 2020.

[19] M. Van Der Boor, S. Borst, and J. Van Leeuwaarden, "Hyper-Scalable JSQ with Sparse Feedback," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 1, pp. 1–37, 2019.

[20] S. Vargaftik, I. Keslassy, and A. Orda, "LSQ: Load balancing in large-scale heterogeneous systems with multiple dispatchers," *IEEE/ACM Transactions on Networking*, vol. 28, no. 3, pp. 1186–1198, 2020.

[21] X. Zhou, N. Shroff, and A. Wierman, "Asymptotically optimal load balancing in large-scale heterogeneous systems with multiple dispatchers," *ACM SIGMETRICS Performance Evaluation Review*, vol. 48, no. 3, pp. 57–58, 2020.