Load Balancing in Compute Clusters With Delayed Feedback

Anam Tahir, Bastian Alt, Amr Rizk, *Senior Member, IEEE*, and Heinz Koeppl

Abstract—Load balancing arises as a fundamental problem, underlying the dimensioning and operation of many computing and communication systems, such as job routing in data center clusters, multipath communication, Big Data and queueing systems. In essence, the decision-making agent maps each arriving job to one of the possibly heterogeneous servers while aiming at an optimization goal such as load balancing, low average delay or low loss rate. One main difficulty in finding optimal load balancing policies here is that the agent only partially observes the impact of its decisions, e.g., through the delayed acknowledgements of the served jobs. In this paper, we provide a partially observable (PO) model that captures the load balancing decisions in parallel buffered systems under limited information of delayed acknowledgements. We present a simulation model for this PO system to find a load balancing policy in real-time using a scalable Monte Carlo tree search algorithm. We numerically show that the resulting policy outperforms other limited information load balancing strategies such as variants of Join-the-Most-Observations and has comparable performance to full information strategies like: Join-the-Shortest-Queue, Join-the-Shortest-Queue(d) and Shortest-Expected-Delay. Finally, we show that our approach can optimise the real-time parallel processing by using network data provided by Kaggle.

Index Terms—Parallel systems, load balancing, partial observability

1 Introduction

As the growth rate of single-machine computation speeds started to stagnate in recent years, parallelism seemed like an effective technique to aggregate the computation speeds of multiple machines. Since then, parallelism has become a main ingredient in compute cluster architectures [1], [2] as it incurs less processing and storage cost on any one individual server [3].

Beyond the aggregation of capacity, a major difficulty in the operation of parallel servers is optimizing for low latency and loss. The key to this optimization is the mapping of the input, that we denote as *jobs*, to the different and possibly heterogeneous serving machines, denoted *servers*, of time-varying capacity and finite memory (buffers). This mapping of input to servers is carried out by a decision-making agent we refer to as the *load-balancer*.

Classical results prove the optimality of the Join-the-Shortest-Queue (JSQ) algorithm in terms of the expected job delay[4], [5] when the servers are homogeneous, have infinite buffer space and the job service times are independent, identically (i.i.d) and exponentially distributed. For the case when the service times are exponentially distributed but with different service rates, Shortest-expected-delay (SED) has been shown to minimize the mean response time of jobs, especially in the case of heavy traffic limits [6], [7].

Note, however, that such types of algorithms assume that the decision-making agent has accurate, timely and synchronized information of all servers and their queues. *In practice this assumption does not hold*, e.g., in data center clusters it cannot be assumed that a load-balancer has timely and synchronized information of all available servers but rather observes some event or time-triggered server feedback.

The goal of this work is to model and optimize the load balancing decision-making over parallel and heterogeneous servers having finite buffers and each following first-in-first-out (FIFO) order, where these servers provide randomly delayed acknowledgements back to the load-balancer. These acknowledgments are sent by the servers to inform the load-balancers of the *number of jobs* the servers have processed in the last epoch. This model captures that the load-balancer is only able to partially observe the server states at the decision time points. The load-balancer does not directly observe the server queues, rather it receives acknowledgements for completed jobs that are randomly delayed on the way back to it. Our contributions are:

- We present a model for controlled load balancing in parallel buffered systems with randomly delayed acknowledgements.
- We find a control law by optimizing a predefined objective function subject to the stochastic dynamics induced by the model.
- We present *POL* - a Partial Observability Load-Balancer, which maps incoming jobs to the parallel servers. POL estimates the unknown parameters of the

- *Anam Tahir, Bastian Alt, and Heinz Koeppl are with Self-Organizing Systems Lab (SOS), Department of Electrical Engineering and Information Technology, Technische Universität Darmstadt, 64289 Darmstadt, Germany. E-mail: {anam.tahir, bastian.alt, heinz.koeppl}@tu-darmstadt.de.*
- *Amr Rizk is with Communication Networks and Systems Lab, Universität Duisburg Essen, 47057 Duisburg, Germany. E-mail: amr.rizk@uni-due.de.*

Manuscript received 15 February 2022; revised 6 September 2022; accepted 23 September 2022. Date of publication 19 October 2022; date of current version 10 May 2023.
This work was supported by the German Research Foundation (DFG) as part of sub-projects C3 and B4 within the Collaborative Research Center (CRC) 1053 – MAKI.
(Corresponding author: Anam Tahir.)
Recommended for acceptance by E. Bini.
Digital Object Identifier no. 10.1109/TC.2022.3215907

parallel system at runtime and despite partial observations achieves a job drop rate and response time comparable with full information load-balancers. The performance evaluation is carried out in a simulation environment. Note that POL can be used for any number of servers and for any kind of inter-arrival and service time distributions, and we have tested a few of them.

- Lastly, we also show how POL can optimize parallel processing in clusters by using real network data provided by Kaggle [8], [9].

The rest of this paper is organized as follows: In Section 2 we first discuss the related work. Then, in Section 3 we outline the system model and give some background on the key topics of this work. In Section 4, we give our contributions starting from the modelling of the partially observable queueing system to our Monte Carlo approach for finding a near-optimal load balancing policy. In Section 5, we give our simulation results and discuss the inference of unobserved system parameters in Section 4.5 along with an experiment with real world data. And then we conclude the paper in Section 6.

2 Related Work

Dynamic load balancing for the performance optimization of parallel systems has fueled numerous seminal algorithms such as Join-the-shortest-queue (JSQ), and Shortest-expected-delay (SED), and more generally Power-of-d policies [10], [11], [12]. JSQ provides optimal decisions, for minimizing the mean response time of a job, when the servers are homogeneous and the service times are independent and identically exponentially distributed [4], [5]. A similar approach for heterogeneous servers is the SED algorithm, which implicitly considers the server rates and maps an incoming job to the server which provides the smallest expected response time for the job at hand. SED is known to perform well for heterogeneous servers, especially in the case of heavy traffic [6], [7].

However, when the number of parallel systems N becomes large, the assumption of knowing the state of the entire system before every decision becomes too strong. The state may be the queue length or the required cumulative service times for the waiting jobs at each system. Depending on the type of system this information, e.g., the service times for servers of random varying capacity, is not known in advance. Control theory and the recently attractive machine learning approaches have also been extensively applied to stochastic queuing networks in order to analyse their performance as self-adaptive software systems, see [13], [14] and references therein.

Power-of-d policies provide a remedy to the problem of the decision-making agent not being able to know all system states at decision time. Here, a number of servers, specifically $d < N$, is repeatedly polled at random at every decision instant and hence, JSQ(d) or SED(d) is performed on this changing subset [15]. This policy is enhanced by a short term memory that keeps knowledge of the least filled servers from the last decision instant [16], [17]. Hence, instead of choosing d servers at random for every job, the decision is based on a combination of new randomly chosen d servers and the least filled servers known from the last decision. For our evaluation purposes we have chosen to compare our algorithm with SED, JSQ and JSQ(d) since the comparison is with respect to the classes of full and limited information at the load-balancers.

The strong assumption behind the different variants of JSQ, SED and the Power-of-d policies is that at every decision instant the load-balancer is aware of the current system state of all or some of the servers. The main difference in this work is that we hypothesize that this assumption of instant (and full) knowledge is often not realistic, as due to the distributed nature of the system the load-balancer may only observe the impact of a decision that maps a job to a server after some non-deterministic feedback time. This feedback time may arise, e.g., due to the propagation delay or simply because the decision-making agent receives feedback only after the job has been processed. The impact of this non-deterministic and heterogeneous feedback time on the decision-making process is significant, as the consequence of a decision on performance metrics such as job response times or job drop rates is only partially observed at the decision instant.

Markov decision processes, MDP, have been used to model queuing systems and achieve optimal control under static and dynamic environments [18], [19]. In an MDP the current feedback, delayed or not, is known to the agent [20]. The concept of transitions between mechanisms to achieve better overall performance in a dynamic communication system, modelled as an MDP, has recently been proposed in [21]. The authors of [5] used an MDP formulation together with a stochastic ordering argument to show that JSQ maximizes the discounted number of jobs to complete their service for a homogeneous server setting. The authors of [22] study the problem of allocating customers to parallel queues. They model this problem as an MDP with the goal of minimizing the sojourn time for each customer and produce a 'separable rule', which is a generalization of JSQ, for queues with heterogeneous servers (in rates and numbers). Note, however, that it is assumed that the queue filling is known and available without delay to the decision-making agent.

In [23] the authors assume that the decision-making agent receives the exact queue length information but with a delay of k steps. They formulate their system as a Markov control model with *perfect state information* by augmenting to the state space the last known state (exact queue length) and all the actions taken until the next known state. In their work, they solve the flow control problem by *controlling the arrival* to a single server queue and show for $k = 1$ that the optimal policy is of threshold type and depends on the last action. In [24], along with the single server flow control problem with similar results as [23], they also showed that when $k = 1$, the optimal policy for minimizing the discounted number of jobs in a system of two parallel queues is join-the-shortest-expected-length. In [25], the decision-making agent at time n, *knows the number of jobs* that were present in each (infinite) queue at time *n-1*, such that it takes decisions at a deterministic delay of one time slot. The state space at any time n is augmented and contains the actual queue filling at time *n-1*, the action taken at time n and if there was an arrival at time n.

In our work, we assume that not only is the information received by the load-balancer randomly delayed, but it is also not the state of the buffers, rather it is in the form of acknowledgements of the number of jobs processed. This makes the system *partially observable* and complex to solve, i.e., in the sense of a partially observable Markov decision process (POMDP) model. Several online and offline algorithms have been introduced to solve a POMDP [26], but there is little work on optimizing queuing systems as a POMDP. Standard solutions, for a POMDP, that do full-width planning [27], like Value iteration and Policy iteration, perform poorly when the state space grows too large. This can easily be the case in queuing systems, due to the curse of dimensionality and the curse of history [28], [29]. For such large state problems, the POMCP algorithm [30], which uses the Monte Carlo tree search (MCTS) approach, is a fast and scalable algorithm for solving a POMDP. Our algorithm also makes use of the MCTS algorithm specifically designed to simulate a parallel queeing system with finite buffers. In addition, we also designed a Sequential Importance Resampling (SIR) particle filter to deal with the delayed feedback acknowledgements in the queueing system. We also use this MCTS approach for solving a Partially observable semi Markov decision process (POSMDP) [31], which is needed when the time between decision epochs is no more exponential.

3 System Model

We consider a queuing system with N parallel servers each having its own finite FIFO queue. The queue filling is denoted $b_i \in \mathcal{B}_i, i = 1, \ldots, N$, where $\mathcal{B}_i = \{0, \ldots, \bar{b}_i\}$ and $\bar{b}_i$ is the buffer space for the i-th queue. We define the vector of queue sizes as $\mathbf{b} = [b_1, \ldots, b_N]^\top$. In the following, we will use boldface letters to denote column vectors.

We consider the general case of heterogeneous servers where the service times, $V_i^{(1)}, V_i^{(2)} \ldots$, of consecutive jobs at the ith server[1] are independent and identically distributed (i. i.d) according to a probability density f_{V_i}. Homogeneous jobs arrive to the load-balancer, as depicted in Fig. 1, according to some renewal process described by the sequence $(T^{(n)})_{n\in\mathbb{N}}$, where the job inter-arrival times $U^{(j)} := T^{(j+1)} - T^{(j)}$ are drawn i.i.d from a distribution F leading to an average arrival rate λ. Each arriving job is mapped by the load-balancer to exactly one server, where the service rate for that job is random and we denote the average service rate of the i-th server as μ_i. A job that is mapped to a full buffer is lost. When a job leaves the system, the server sends an acknowledgement back to the load-balancer, to inform it of a slot getting free in its associated queue. The load-balancer then uses this feedback acknowledgement from each server to calculate the current buffer fillings, b_i, of each queue.

A major challenge for deciding on the job routing arises when the acknowledgments from the server are delayed. Here, we incorporate three main delay components, i.e., (i) the job waiting time in the queue it was assigned to, (ii) the job processing time, and (iii) the propagation delay of the acknowledgement back to the load-balancer. The third component *makes the decision problem particularly hard* as a decision does not only impact the current state of the system but also future states due to its delayed acknowledgement feedback. Note that the load-balancer makes the decision based only on these observed acknowledgments, thus making the system state partially observable (PO). We denote by y_i the number of acknowledgements from the i-th server that are observed by the load-balancer in one inter-arrival time. And we denote by x_i the delayed feedback, which is the number of acknowledgments that are on the way back to the load-balancer but have not reached it yet. Since the load-balancer is PO, it does not directly observe the (1) queue states b_i, (2) the job service times v_i, or (3) the delayed feedback x_i. See Fig. 1 for further visualization.

1. We denote random variables by uppercase letters and their realizations as lowercase letters.

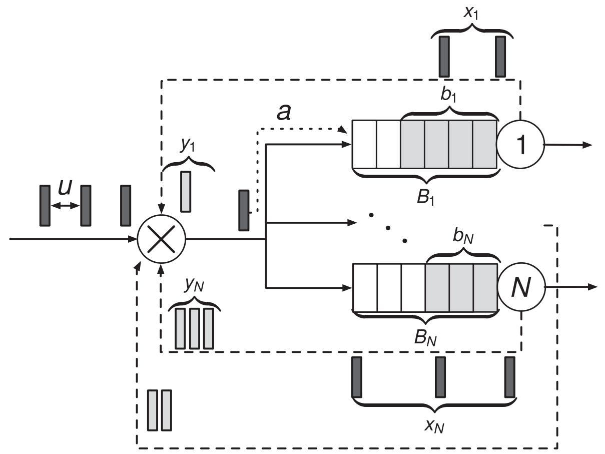

Fig. 1. A parallel queuing system with a load-balancer that maps jobs to servers. The load-balancer observes the inter-arrival times u and the feedback, i.e., the number of acknowledgements, from each server y_i. The load-balancer *does not observe* the queue states b_i, the job service times, or the delayed feedback x_i, i.e., the number of acknowledgements on the way back.

Depending on the distribution type of the inter-arrival times, U, and service times, V, we model the decision-making process in this PO queuing system as a *partially observable Markov decision process*, POMDP, or a *partially observable semi Markov decision process*, POSMDP. A POMDP has an underlying Markov decision process (MDP), where the actual state of the system is not known to the agent, i.e., the load-balancer in our case. This modelling can be used when both U and V are exponentially distributed, keeping the system Markovian. For non-exponential U and/or V, POSMDP formulation can be used, where the underlying process is now semi-Markov and the actual state of the system is still not known to the load-balancer.

3.1 Markov Decision Process With Partial Observability

An MDP [20] is defined as a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$, where $\mathcal{S}$ is the countable state space, $\mathcal{A}$ is the countable action space, $\mathcal{T} : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is the transition function, $\mathcal{R} : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function. Throughout this work we assume time homogeneity for the transition function $\mathcal{T}$, observation function $\mathcal{O}$ and reward function $\mathcal{R}$. An MDP with partial observations is also a controlled Markov process, where the exact state of the process is latent. It is in addition defined using; $\mathcal{Z}$, the countable observation space, and $\mathcal{O} : \mathcal{Z} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$, the observation function. Consider the case of discrete epochs at time points $t \in \mathbb{N}_0$. Note that the clock given by t is an event clock and not a wall-clock time. If the decision epochs t are exponentially distributed, then this PO process can be modelled as a POMDP [28]. However, if t is not exponentially distributed then under the condition that the decision-making is done only at the epochs t, it can be modelled as a POSMDP [31].

We consider a latent process $S^{(t)} \in \mathcal{S}$ that can be controlled by actions $A^{(t)} \in \mathcal{A}$. Since, the state is latent, only observations $Z^{(t)} \in \mathcal{Z}$ are available. The transition function $\mathcal{T}(s', s, a) := \mathbb{P}(S^{(t+1)} = s' \mid S^{(t)} = s, A^{(t)} = a)$ is the conditional probability of moving from state s under action a to a new state s'. The observation function $\mathcal{O}(z, s, a) := \mathbb{P}(Z^{(t+1)} = z \mid S^{(t+1)} = s', A^{(t)} = a)$ denotes the conditional probability of observing z under the latent state s' and action a. An agent receives a reward $R^{(t)} = \mathcal{R}(S^{(t+1)}, S^{(t)}, A^{(t)})$, which it tries to maximize over time.

Since the current state is not directly accessible by the agent, it has to rely on the action-observation history sequence, $\mathcal{H}^{(t)} = \{A^{(0)}, Z^{(0)}, A^{(1)}, Z^{(1)}, \ldots, A^{(t)}, Z^{(t)}\}$, up to the current time point t. A policy $\pi_t(a, h) := \mathbb{P}(A^{(t)} = a \mid \mathcal{H}^{(t)} = h)$ is the conditional probability of choosing action a under action-observation history h. The solution then corresponds to a policy which maximizes an objective over a prediction horizon. The policy is defined as a function of the observation-action history of the agent, which makes it very challenging, since a naive planning algorithm requires to evaluate an exponentially increasing number of histories in the length of the considered time horizon. For this reason, different solution techniques are required.

Since keeping a record of the entire history, h, is not feasible, one way is to represent this history in terms of the belief state, $\boldsymbol{\rho}^{(t)} \in \Delta^{|S|}$, where $\Delta^{|S|}$ is an $\mathcal{S}$ dimensional probability simplex, $\boldsymbol{\rho}^{(t)} = [\rho_1^{(t)}, \ldots, \rho_{|S|}^{(t)}]^\top$ and the components $\rho_s^{(t)}$ are the filtering distribution $\rho_s^{(t)} = \mathbb{P}(S^{(t)} = s \mid \mathcal{H}^t = h)$. If the state space is huge, this will be a very high dimensional vector. So, in order to break the curses of *history* and *dimensionality*, a certain number of particles can be used to represent the belief state $\boldsymbol{\rho}^{(t)}$ of the system at time t, see [30] for further details. These particles represent the belief state $\boldsymbol{\rho}^{(t)}$ of the system and are updated using Monte Carlo simulations based on the action taken and observations received. This is the approach that we build upon in this paper. We consider an infinite horizon objective, where the optimal policy π^* is found by maximizing the expected total discounted future reward $\pi^* = \operatorname{argmax}_\pi \sum_{t=0}^{\infty} \mathsf{E}_\pi[\gamma^t R^{(t)}]$, with $\gamma < 1$. Note that our work can easily be reformulated into finite horizon objectives.

4 Load Balancing in Parallel Queuing Systems With Delayed Acknowledgments

In this section, we explain how we model our partially observable (PO) system and then propose our solution.

4.1 Modelling Load Balancing in the Partially Observable Queuing System

In order to model the load balancing decision in a system with N parallel finite buffer servers (cf. Fig. 1), as a PO process, we define the system state $\mathbf{s} \in \mathcal{S}$, using $\mathbf{s} = [\mathbf{s}_1, \ldots, \mathbf{s}_N]^\top$. Here, $\mathbf{s}_i$ is the augmented state of the i-th queue that is defined as $\mathbf{s}_i = [b_i, x_i, y_i]^\top$, where, $b_i \in \mathcal{B}_i$ denotes the current buffer filling at queue i, $x_i \in \mathcal{B}_i$ denotes the number of delayed acknowledgements for the jobs executed by the

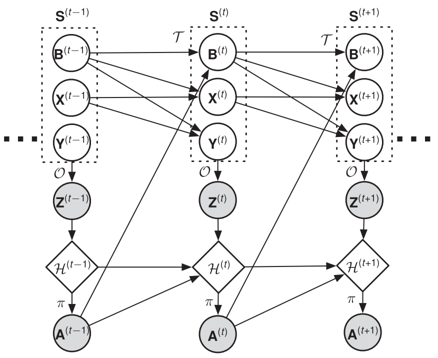

Fig. 2. Probabilistic graphical model of the partially observable queuing system with delayed acknowledgements. Shown are three time slices, where grey nodes depict observed quantities and diamond shaped nodes denote deterministic functions.

server i but not observed by the load-balancer in the current epoch[2], and $y_i \in \mathcal{B}_i$ denotes the number of acknowledgements actually *observed* by the load-balancer in the current epoch. Hence, the state space is $\mathcal{S} \subseteq \mathbb{N}_0{}^{3N}$ and an action $a \in \mathcal{A}$, with $|\mathcal{A}| = N$ corresponds to sending a job to the a-th server. An observation $\mathbf{z} \in \mathcal{Z}$ is the vector of observed acknowledgements at the load-balancer, with the observation space being $\mathcal{Z} \subseteq \mathbb{N}_0{}^N$.

4.2 The Dynamical Model

Next, we describe the dynamics of the underlying processes of the PO model. The corresponding probabilistic graphical model is depicted in Fig. 2. In case of the POMDP model, the time t in Fig. 2 is exponentially distributed, while for POSMDP it can be random (non-exponential). As we are using Monte Carlo simulations to solve the PO system, the transition probabilities do not have to be defined explicitly. Therefore, we define the transition function indirectly as a generative process.

We consider a load-balancer (our decision-making agent) that makes a mapping decision at each job arrival, where the inter-arrival times $U^{(j)}$ are i.i.d. In order to characterize the stochastic dynamics, we determine the random behaviour of the number of jobs $\tilde{k}_i$ that leave the i-th queue during an inter-arrival time. Note that the number of jobs leaving the queue is constrained by the current filling of the queue, b_i, of the ith server, hence we use $\tilde{k}_i = \min(k_i, b_i)$, where k_i is the number of jobs that *can be served*, which is determined by the inter-arrival time and the service times of the i-th server. with maximum capacity, $\bar{b}_i$. Therefore, we define the generative model for the queuing dynamics as:

$$\mathbf{b}' = \min(\max(\mathbf{b} - \mathbf{k}, \mathbf{0}) + \mathbf{e}_a, \bar{\mathbf{b}}), \tag{1}$$

where $\mathbf{b}$ denotes the queue size vector of the queuing system at some arrival time point and $\mathbf{b}'$ is the queue size vector of the queuing system at the next epoch. By $\mathbf{k}$ we denote the non-truncated vector of number of jobs that can be served and $\mathbf{e}_a$ is a vector of all zeros, except the a-th position is set to one to indicate a mapping of the incoming job to the a-th server. We use $\bar{\mathbf{b}}$ as the vector of maximum buffer sizes, and $\min(\cdot, \cdot)$ and $\max(\cdot, \cdot)$ denote the element-wise minimum and maximum operation.

2. An epoch corresponds here to one inter-arrival time.

As the *load-balancer only observes the job acknowledgements*, we update the augmented state space, $\mathbf{s}$ (as defined in Section 4.1), using the following stochastic update equation:

$$\mathbf{s}' = \begin{bmatrix} \mathbf{b}' \\ \mathbf{x}' \\ \mathbf{y}' \end{bmatrix} = \begin{bmatrix} \min(\max(\mathbf{b}-\mathbf{k},\mathbf{0}) + \mathbf{e}_a, \bar{\mathbf{b}}) \\ \min(\mathbf{b},\mathbf{k}) + \mathbf{x} - \mathbf{l} \\ \mathbf{l} \end{bmatrix}, \tag{2}$$

where $\mathbf{l}$ is the vector containing the number of jobs which are observed by the load-balancer for each queue at the current epoch. The number of unacknowledged jobs from the previous epoch is denoted $\mathbf{x}$, which is updated by removing the observed jobs $\mathbf{l}$ and adding the newly generated acknowledgments given by $\min(\mathbf{b},\mathbf{k})$. The delay model according to which $\mathbf{l}$ is calculated is given next.

4.3 Delay Model

We assume that the number of jobs that can be served in one inter-arrival time is distributed as $K_i \sim f_{K_i}(k_i)$ and we choose a delay model, where

$$L_i \,|\, b_i, k_i, x_i \sim \mathrm{Bin}\left(\min(b_i, k_i) + x_i, p_i\right). \tag{3}$$

Here, $\min(b_i, k_i)$ is the number of jobs leaving the i-th queue at the current epoch, x_i is the number of jobs from the i-th queue for which no acknowledgements have been previously observed by the load-balancer, p_i is the probability that an acknowledgement is received by the load-balancer in the current epoch and L_i is the distribution from which l_i is sampled for Eq. (2).

We have chosen the binomial distribution because it generally captures the fact that only a subset of the sent, $\min(b_i, k_i) + x_i$, acknowledgements are successfully observed at the load-balancer in the current epoch. At the cost of simplifying the usually correlated delays of jobs, this model helps to obtain tractable results. For example, $p = 0.6$ would mean that out of all the acknowledgements sent, only 60% are expected to be received by the load-balancer in that epoch while 40% are expected to be delayed. Similarly, $p_i = 1$ would then represent the case of no delay. Note that any other distribution describing the delay model can be numerically evaluated. The number of acknowledgements from all servers that are not observed in this epoch are accounted for in the next epoch in $\mathbf{x}'$. The previously introduced observations $\mathbf{z}$ are essentially the received acknowledgements, i.e., $\mathbf{z} = \mathbf{y}' = \mathbf{l}$. Since only the vector $\mathbf{y}'$ of the state $\mathbf{s}'$ (Eq. (2)) is observed by POL, a partial observability is established. We note that one limitation of this model is due to the delay independence assumption that lies below the used Binomial distribution.

4.4 Job Acknowledgement Distribution

Next, we discuss how to quantify the distribution of the number of jobs k_i that can be served at the i-th queue in one inter-arrival time. The marginal probability

$$f_{K_i}(k_i) = \int_0^\infty f_{K_i\,|\,U}(k_i \,|\, u) f_U(u) du, \tag{4}$$

can be computed by noting [32]

$$f_{K_i\,|\,U}(k_i \,|\, u) = \mathbb{P}\left(\bar{V}_i^{(k_i)} \le u\right) - \mathbb{P}\left(\bar{V}_i^{(k_i+1)} \le u\right), \tag{5}$$

with $\bar{V}_i^{(k_i)} = \sum_{m=1}^{k_i} V_i^{(m)}$. For the POMDP model, with exponentially distributed inter-arrival times $U \sim \mathrm{Exp}\,(\lambda)$, with rate parameter λ, and exponential service times for all servers $V_i \sim \mathrm{Exp}\,(\mu_i)$, with rate parameter μ_i, the distribution $f_{K_i}(k_i)$ can be calculated in closed form. Since the service process corresponds to a Poisson process, we find the conditional distribution $f_{K_i\,|\,U}(k_i \,|\, u)$ as $K_i \,|\, u \sim \mathrm{Pois}\,(\mu_i u)$. Carrying out the integral, in Eq. (4), we find the number of jobs that can be served in one inter-arrival time follow a Geometric distribution $K_i \sim \mathrm{Geom}\,(\frac{\lambda}{\mu_i+\lambda})$, with $\frac{\lambda}{\mu_i+\lambda}$ denoting success probability, [32].

For the POSMDP model with general inter-arrival and service time distributions closed form expressions for the marginal probability above are often not available, since, this would require closed form expressions for a k-fold convolution of the probability density function (pdf) of the service times. Note that some corresponding general expressions exist as Laplace transforms where the difficulty is passed down to calculating the inverse transform. Therefore, in such cases we will resort to a sampling based scheme for the marginal distribution $f_{K_i}(k_i)$, i.e.,

$$\begin{aligned} &U \sim f_U(u) \\ &V_i^{(m)} \sim f_{V_i}(v_i), \quad m = 1, 2, \ldots \\ &\mathcal{K}_i = \left\{ j \in \mathbb{N}_0 : \sum_{m=1}^{j} V_i^{(m)} \le U \right\} \\ &K_i = \max(\mathcal{K}_i). \end{aligned} \tag{6}$$

In this numerical solution, we draw a random inter-arrival time U and count the number K_i of service times that fit in the inter-arrival interval. Note that the number of jobs $K_i^{(t)}$ at time t is not necessarily independent for the number of jobs $K_i^{(t+1)}$ in the next interval. The impact of this effect can be well demonstrated when the service time distribution is, e.g., heavy tailed. Also note that the exact modelling of this behaviour would, in general, require an extended state space incorporating this memory effect. Therefore, the sampling scheme can be seen as an approximation to the exact system behaviour.

4.5 Inferring Arrivals and System Parameters

For POL to be deployed in an unknown environment, we may require an estimate of the inter-arrival and/or service rate densities. For this purpose, we will resort to a Bayesian estimation approach to infer the densities $f_U(u)$ and $f_{V_i}(v_i)$. We select a likelihood model $f(\mathcal{D} \,|\, \theta)$ for the data generation process and a prior $f(\theta)$, with model parameters Θ. We assume we have access to data $\mathcal{D} = \{d^{(1)}, d^{(2)}, \ldots, d^{(n)}\}$, where $d^{(j)}$ is inter-arrival times between the j-th and $j+1$-st job arrival event that is observed by the load-balancer or the service times for each server. For inference-based load balancing in POL we use the inferred distribution of the inter-arrival times as in the posterior predictive $f(d^* \,|\, \mathcal{D}) = \int f(\theta \,|\, \mathcal{D}) f(d^* \,|\, \theta) d\theta$, of a new data point d^*, which is then used in the sampling simulator, see Eq. (6). The same can be done with the data for the service times. We will now describe some models of different complexities for the data generation. Since, most models do not admit a closed form solution, we resort to a Monte Carlo sampling approach to sample from the posterior predictive.

4.5.1 Inference for Exponential Inter-Arrival Times

Here, we briefly show the calculation for the posterior distribution and posterior predictive distribution for renewal job arrivals with exponentially distributed inter-arrival times. For the likelihood model we assume

$$D^{(j)} \mid m \sim \mathrm{Exp}\,(m), \quad j = 1, \ldots, n,$$

where m is the rate parameter of the exponential distribution. We use a conjugate Gamma prior $M \sim \mathrm{Gam}\,(\alpha_0, \beta_0)$. Hence, the posterior distribution is

$$M \mid \mathcal{D} \sim \mathrm{Gam}\,(\alpha_0 + n, \beta_0 + \sum_{j=1}^{n} d^{(j)}).$$

And the posterior predictive distribution is found as

$$D^{*} \mid \mathcal{D} \sim \mathrm{TP}\,(\alpha_0 + n, \beta_0 + \sum_{j=1}^{n} d^{(j)}),$$

where TP (α, β) denotes the translated Pareto distribution.

4.5.2 Inference for Gamma Distributed Inter-Arrival Times

In case of a gamma likelihood of the form

$$D^{(j)} \mid \alpha, \beta \sim \mathrm{Gam}\,(\alpha, \beta), \quad j = 1, \ldots, n$$

we use independent Gamma priors for the shape and the rate, with $A \sim \mathrm{Gam}\,(\alpha_0, \beta_0)$ and $B \sim \mathrm{Gam}\,(\alpha_1, \beta_1)$. Finally, we sample from the posterior predictive using Hamiltonian Monte Carlo (HMC) [33], which can be implemented using a probabilistic programming language, e.g. using PyMC3 [34].

4.5.3 Service Times Distributed as an Infinite Gamma Mixture

Here, we present a framework to non-parametrically infer the posterior distribution. We use an approximate Dirichlet process mixture model which can be regarded as an infinite mixture model [35]. We use a gamma distribution for the observation model

$$\phi(d \mid \theta_i) \propto d^{a_i - 1} e^{-b_i d}, \tag{7}$$

with mixture parameters a_i and b_i. For the base measure G_0, i.e., the prior distribution of the mixture parameters, we use $G_0 = F_{A_i} \times F_{B_i}$, with $A_i \sim \mathrm{Gam}\,(1, 1)$ and $B_i \sim \mathrm{Gam}\,(1, 1)$. The truncated stick-breaking approximation is then given by

$$M \sim \mathrm{Gam}\,(1, 1), \quad B_i \mid m \sim \mathrm{Beta}\,(1, m), \quad i = 1, \ldots, c-1$$

$$W_i = \beta_i \prod_{j=i-1}^{i} (1 - B_j), \quad i = 1, \ldots, c-1$$

$$W_c = 1 - \sum_{j=1}^{c-1} W_j, \quad \Theta_i \sim G_0$$

$$D^{(j)} \mid w_1, \ldots, w_c, \theta_1, \ldots, \theta_c \sim \sum_{i=1}^{c} w_i \phi(d \mid \theta_i), \; j = 1, \ldots, n,$$

which corresponds to a mixture of Gamma pdfs. Here too, samples from the posterior predictive can be efficiently generated using HMC. For the truncation point, the number of components c in the formula above can be assessed using

$$c = \lceil 2 - \mathsf{E}\,[M] \log\,(\epsilon) \rceil = \lceil 2 - \log\,(\epsilon) \rceil, \tag{8}$$

where ϵ is an upper bound on the total variation distance between the exact and truncated approximation. For example, we can choose $\epsilon = 10^{-12}$, which corresponds to $c = 30$ components.

Algorithm 1. POL Load-Balancer With Delayed Feedback

```
1: Input: $N, \lambda, \mu_1 \ldots \mu_N, \mathcal{G}, \eta, \boldsymbol{\rho}^{(0)}, \mathcal{R}, \mathbf{s}_0, \kappa, b, T_m, T_e, Q_s$
2: Output: $R_{\text{avg}}$, average reward for each time step $T_e$
3: Initialize $R_m$
4: for $t = 0, 1, \ldots, T_m$ do
5:     Initialize $\mathbf{u}, \mathbf{v}^{(1)} \ldots \mathbf{v}^{(N)}$, tree $\boldsymbol{\Psi}_\mathbf{0}, R_e, Q_s$    ▷$Q_s$ is the real world representation of the queueieng network.
6:     for $t = 0, 1, \ldots, T_e$ do
7:         $\boldsymbol{\Psi}_{\boldsymbol{t+1}}$ = SimulateTree($\boldsymbol{\Psi}_{\boldsymbol{t}}, \mathcal{G}$)  ▷See the pseudocode given in [30]
8:         $a_{t+1} \rightarrow \text{argmax}_a \mathcal{R}(\mathbf{s}, a)$
9:         $o_{t+1}, s_{t+1}, r_{t+1}$ = $Q_s(a_{t+1})$
10:        $R_e \rightarrow R_e \cup r_{t+1}$    ▷Collect reward for each epoch
11:        $\boldsymbol{\rho}^{(t+1)}$ = UpdateBeliefandTree($\boldsymbol{\Psi}_{\boldsymbol{t+1}}, o_{t+1}, a_{t+1}, \mathcal{G}$)
12:    end for
13:    $R_m \rightarrow R_m + R_e$
14: end for
15: $R_{\text{avg}} = \frac{R_m}{T_m}$
16: Return $R_{\text{avg}}$
17: function UpdateBeliefandTree$\boldsymbol{\Psi}, o, a, \mathcal{G}$
18:     Initialize $K_s = \{\}, W_s == \{\}$    ▷Set of particles and their weights
19:     repeat
20:         $s', o', r' \sim \mathcal{G}(s, a), \quad s \sim \boldsymbol{\Psi}(\text{root})$
21:         $w_s = p(s'|o', a)$
22:         $K_s \rightarrow K_s \cup s', W_s \rightarrow W_s \cup w_s$
23:     until Timeout()
24:     Generate particles from $K_s$ according to weights $W_s$ ▷SIR particle filter
25:     Update root and prune tree $\boldsymbol{\Psi}$
26: end Function
```

4.6 Reward Function Design

A main difference of our approach to *explicitly* defining a load balancing algorithm is that we provide the algorithm designer with the flexibility to set different optimization objectives for the load-balancer and correspondingly obtain the optimal policy by solving the POMDP or the POSMDP. This is carried out through the design of the reward function $\mathcal{R}$ as defined in Section 3.1. The optimal policy π^* maximizes the expected discounted reward as $\pi^* = \text{argmax}_\pi \sum_{t=0}^{\infty} \mathsf{E}_\pi[\gamma^t R^{(t)}]$ where $R^{(t)} = \mathcal{R}(S^{(t+1)}, S^{(t)}, A^{(t)})$, with $\gamma < 1$. In the following, we discuss several reward functions $\mathcal{R}$ in the context of mapping incoming jobs to the parallel finite queues, see Fig. 1.

Minimize queue lengths: A reward function which aims to minimize the overall number of jobs waiting in the system. This objective that can be formalized as

$$\mathcal{R}(\mathbf{s}', \mathbf{s}, a) = -\sum_{i=1}^{N} b_i' \tag{9}$$

as it takes the sum of all queue fillings. Similarly, a polynomial or an *exponential* reward function, such as

$$\mathcal{R}(\mathbf{s}', \mathbf{s}, a) = -\sum_{i=1}^{N} \chi^{b'_i} \quad (10)$$

For a fixed overall number of jobs in the system and $\chi > 1$, this objective tends to balance queue lengths, e.g., if total jobs in the system are 10 and $N = 2$ then an allocation of [5,5] jobs will have much higher reward than [9,1] allocation. Using the *variance* amongst the current queue fillings also balances the load on queues. The reward function is then given as:

$$\mathcal{R}(\mathbf{s}', \mathbf{s}, a) = \mathrm{Var}(b_i, \ldots, b_n) \quad (11)$$

Note, however, that balancing queue lengths does not necessarily lead to lower delays if the servers are heterogeneous. Hence, *proportional allocation* provides more reward when jobs are mapped to the faster server as

$$\mathcal{R}(\mathbf{s}', \mathbf{s}, a) = -\sum_{i=1}^{N} \frac{b'_i}{\mu_i} \quad (12)$$

Minimize loss events: To prevent job losses, we can also formulate a reward function that penalizes actions that lead to fully filled queues, i.e.,

$$\mathcal{R}(\mathbf{s}', \mathbf{s}, a) = -\sum_{i=1}^{N} \mathbb{1}(b'_i = \bar{b}_i) \quad (13)$$

The indicator function evaluates to one only when the corresponding queue is full.

Minimize idle events: One might also require that the parallel system remains work-conserving, i.e., no server is idling, as this essentially wastes capacity. Hence, in the simplest case we can formulate a reward function of the form

$$\mathcal{R}(\mathbf{s}', \mathbf{s}, a) = -\sum_{i} \mathbb{1}(b'_i = 0) \quad (14)$$

Note that some of the reward functions above can be combined, e.g., in a weighted form such as the following.

$$\mathcal{R}(\mathbf{s}', \mathbf{s}, a) = -\left[\sum_{i=1}^{N} b'_i + \kappa \mathbb{1}(b'_i = \bar{b}_i)\right], \quad (15)$$

where, b'_i is the buffer state after taking action a and the constant weight $\kappa > 0$ is used to scale the impact of the events of job drops to the impact of the buffer filling on the reward.

4.7 Partial Observability Load-Balancer: A Monte Carlo Approach for Delayed Acknowledgements

In this section, we outline our approach to solve the partially observable system for the job routing problem in parallel queuing systems with delayed acknowledgements. Our solution is an alternate technique to Dynamic Programming and is based on a combination of the Monte Carlo Tree Search (MCTS) algorithm [30] and Sequential Importance Resampling (SIR) particle filter.

The reason for choosing an MCTS algorithm is that load balancing problems, like the one presented in this work, can span to very large state spaces. In these scenarios, solution methods based on dynamic programming [36] often break due to the *curse of dimensionality*. MCTS solves this problem by using a sampling based heuristic approach to construct a search tree to represent different states of the system, the possible actions in those states and the expected value of taking each action. In recent years, these techniques have been shown to yield exceptional results in solving very large decision-making problems [37],[38].

One main contribution of the work at hand lies in the design of a simulator[3], $\mathcal{G}$, which incorporates the properties of the queuing model discussed above, into the algorithm. The simulator $\mathcal{G}$, provides the next state $\mathbf{s}^{(t+1)}$, the observation $\mathbf{z}^{(t+1)}$ and the reward $R^{(t+1)}$, when given the current state $\mathbf{s}^t$ of the system and the taken action a^t as input,

$$\mathbf{s}^{(t+1)}, \mathbf{z}^{(t+1)}, R^{(t+1)} \mid \mathbf{s}^t, a^t \sim \mathcal{G}(\mathbf{s}, a). \quad (16)$$

This simulator $\mathcal{G}$, is used in the MCTS algorithm to rollout simulations of different possible trajectories in the search tree. Each trajectory is a path in the search tree starting from the current belief state of the system and expanding (using $\mathcal{G}$) to a certain depth. While traversing through the search tree, the trajectories (actions) are chosen using the Upper Confidence bounds for Trees algorithm (UCT), which is an improvement over the greedy-action selection [39]. In UCT the upper confidence bounds guide the selection of the next action by trading off between exploiting the actions with the highest expected reward up till now and exploring the actions with unknown rewards.

At every decision epoch POL starts with a certain belief on the state of the system, $\boldsymbol{\rho}^{(t)}$, see Section 3.1 for a formal definition, which is represented with particles and also used as the root of the search tree. Starting from the root, i.e., the current belief, the search tree uses UCT to simulate the system for a given depth (fixed here to tree depth of 10), after which the action a with has the highest expected reward is chosen. The trajectories for all other actions are then pruned from the search tree since they are no longer possible. This is done to avoid letting the tree grow infinitely large.

Once the action a is taken and the job is allocated to a certain queue, the load-balancer receives real observations, $\mathbf{z} = \mathbf{l}$, from the system. These observations are the randomly delayed acknowledgements from the servers. The load-balancer then uses the received observations as an input to the SIR particle filter, in order to update its belief of the state the system is in now. The weights given to each particle (state), $w(s_i) = p(z_i|s_i)$, while *resampling* in this SIR particle filter were designed to incorporate our queuing system and its delay model, Eq. (3). After applying the SIR filter, we will have the new set of particles representing the current belief state of the system, $\boldsymbol{\rho}^{(t)}$. These particles are then used to sample the states for simulating the search tree and finding the optimal action at the next epoch. [4] It is shown in Theorem 1 of [30] for a POMDP and in Theorem 2 of [40] for a POSMDP, that the MCTS converges to an optimal policy.

3. https://github.com/AnamTahir7/Partially-Observable-Load-Balancer

4. Note that in POL receiving observations, action selection and belief update all happen at each decision epoch, which is why it is possible to model the system as a POSMDP [31] as well.

To summarize, at every job arrival, POL simulates the tree from the root. The root contains the current set of belief particles. POL then acts on the real environment using the action which maximizes the expected value at the root. On taking the action, POL gets a real observation. This observation is the acknowledgement that is subject to delays. Using this observation and an SIR particle filter, POL updates its set of particles (belief state) of the system for the next arrival. The pseudocode of the working of POL is given in Algorithm 1.

5 Simulation Results

In the following, we show numerical evaluation results for the proposed Partial Observability Load-Balancer (POL), under randomly delayed acknowledgements. Recall that if the acknowledgement is not observed in the current inter-arrival time, it is not accumulated into the future observations. In order to evaluate the impact of delayed observations, we consider in our simulations a probability of $p_i = 0.6 \, \forall i$ in Eq. (3), if not stated otherwise. This means that an acknowledgement is delayed until the next epochs with probability: $1 - p_i = 0.4$. We set the buffer size, b_i, for all queues to 10 jobs. This value for b_i was chosen arbitrarily, and any other value can be used. Further, if not explicitly given, we use the *combined reward function* given in Eq. (15) with $\kappa = 100$, since we aim to avoid job drops in the system. We consider the system depicted in Fig. 1 for both cases of heterogeneous and homogeneous servers. In particular, we show numerical results comparing POL to different variants of load balancing strategies (with and without full system information) with respect to:

- the log complementary cumulative distribution function (CCDF) of the empirical job response time (measured from the time a job enters the queue until it completes service and leaves, lower is better). *This is done only for the jobs which are not dropped*,
- the empirical distribution of the job drop rate (measured over all simulation runs where for each run we track the number of jobs dropped out of all jobs received per run, lower is better),
- and the cumulative reward (higher is better).

The evaluation box plots are based on $T_m = 100$ independent runs of $T_e = 5 \cdot 10^3$ jobs with whiskers at $[0.5, 0.95]$ percentiles. For every independent run, a new set of inter-arrival and service times are sampled based on the chosen distributions. These sampled times are then used by all load-balancing policies in that run in order to do variance reduction, according to the Common Random Numbers (CRN) technique [41]. The plotted results are an average of 100 such Monte carlo simulations.

The chosen inter-arrival and service time distributions are mentioned with the figures, with unit of measurement req/sec. The offered load ratio ($\eta := \lambda / \sum_i \mu_i$) is used to describe the ratio between arrival rate and the combined service rate. The higher the value of η, the higher is the job load on the system.

5.1 Overview of Compared Load-Balancers

We compare POL to the following load balancing strategies:

5.1.1 Full Information (FI) Strategies

These strategies have access to the exact buffer length of queues at the time of each job arrival, and also know the arrival rate and the service rates of the servers.

- JSQ-FI: Join-the-Shortest-Queue assigns the incoming job to the server with the smallest buffer filling.
- DJSQ-FI: Join the shortest out of d randomly selected queues. If not stated otherwise, $d = 2$ has been used for our experiments.
- SED-FI: Shortest-Expected-Delay assigns the incoming job to the server with the minimum fraction of the current buffer filling divided by the average service rate.

5.1.2 Limited Information (LI) Strategies

These strategies, similar to POL, only have access to the randomly delayed acknowledgements.

- JMO: Join-the-Most-Observations maps an incoming job to the server that has generated the most observations, i.e., received acknowledgements, in the last inter-arrival epoch. This might lead to servers becoming and remaining idle (stale).
- JMO-E (with Exploration): with probability 0.2 randomly chooses an idle server and with probability 0.8 performs JMO.

For all strategies, ties are broken randomly.

5.2 Numerical Results

We first consider a system with $N = 2$ heterogeneous servers with exponentially distributed service times with rates $\mu_1 = 4$ and $\mu_2 = 2$. The inter-arrival times are also exponentially distributed with rate $\lambda = 5$. Fig. 3 shows the numerical comparison of POL with other load-balancers. Observe that, even though POL does not have access to the exact state of parallel systems and also the acknowledgements from the different systems are randomly delayed, it still achieves comparable results to *full information strategies*, while it outperforms the other *limited information strategies*. This is because in the other *limited information strategies*, the initially chosen queues play a key role. Since the queue to which more jobs are sent, will also give back more observations (acknowledgements), and JMO and JMO-E will keep sending to those queues, resulting in job drops. The overlap in response time indicates the similarity of the policies of the strategies, especially for high offered load when most of the finite queues will be full. The heatmap in, Fig. 3d, represents the policy of POL at each buffer filling state. It can be seen that higher priority is given to the faster server, μ_1, having buffer filling b_1. The light(dark) regions in the heatmap corresponds to the state where jobs are allocated to server 1(2). *This heatmap shows that for very possible state of the two queues $s = \{b_1, b_2\}$, even with limited and delayed information, POL is able to allocate more jobs to the queue with lower filling or faster servers, similar to JSQ and SED.* Hence, it is able to perform almost as good as the FI strategies.

Fig. 4 shows the performance of POL for $N = 50$ heterogeneous servers. The service and arrivals rates of this setup are kept that the offered load is ($\eta \approx 1$). This experiment

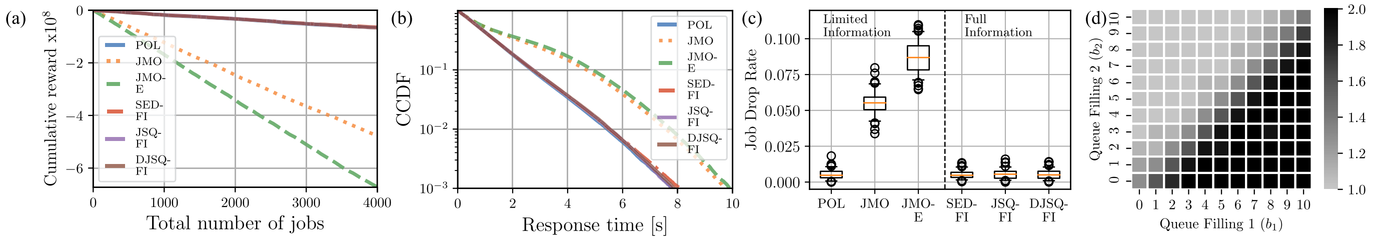

Fig. 3. $N=2$ heterogeneous servers with exponential service rates $\mu_1=4$, $\mu_2=2$. Job inter arrival times are exponentially distributed with rate $\lambda=5$. POL outperforms the algorithms with limited information, JMO and JMO-E, in terms of (a) cumulative reward (higher is better), (b) response time (lower is better), and (c) job drops (lower is better). Although POL *only observes* the randomly delayed job acknowledgements, while SED-FI, DJSQ-FI and JSQ-FI know the *exact buffer fillings and service times / rates*, POL still has comparable performance. The heatmap (d) shows the allocation preference of POL based on how filled the queue is $b_1(b_2)$, i.e., the label bar 1.0(2.0) denotes the allocation to queue 1(2), respectively.

shows that our load-balancer POL is scalable to perform well for large number of queues. Next, we remain with the case of $N=50$ heterogeneous servers, however with inter-arrival times that are gamma distributed while the service times follow a heavy tailed Pareto distribution, with the offered load ($\eta \approx 1$). Fig. 5 shows that here too, POL is able to outperform both the LI strategies and the FI strategies, DJSQ-F. The other two FI strategies have better performance because they always have timely and exact information of the queues, which is unrealistic. Note that as we consider heavy-tailed distributions in this example the prediction of job acknowledgments by POL suffers, because of reasons discussed at the end of Section 4.4.

5.3 Sensitivity Analysis

Next, we discuss the impact of the limited observations on POL under different acknowledgement delays, p_i. Recall, that POL is not able to observe the buffer fillings, but rather receives the randomly delayed acknowledgements of the served jobs. These delayed acknowledgements are used by the SIR filter of POL to keeps its belief of the state of the environment updated. Fig. 6 depicts sample runs showing the actual evolution of the job queue states (red solid line) and the belief (in shaded region) that POL has on each queue state at each time step, under different acknowledgement delays. Observe that increasing delays (i.e., lower p_i) increases the uncertainty in belief of each state. However, POL is still able to track the system state for different delays for each server, which shows the efficiency of the SIR particle filter and also *justifies the performance of POL to be as good as FI strategies*. Having different delays in acknowledgements from each server reflects a distributed system, where network conditions may be different for each server and may lead to different delays in acknowledgements from different servers. Fig. 7 visualizes the belief of POL on the state of each queue after 1000 epochs, for different delays in acknowledgements in each queue.

In Fig. 8 we analyse the performance of POL under varying offered loads ranging from $\eta = 0.2$ to $\eta = 1.2$. It can be seen that POL has almost no job losses up to a load of $\eta = 1$. Note the qualitative change of the response time distribution as the offered load reaches $\eta = 1$ and beyond. For lower offered load, the response time distribution resembles an exponentially tailed distribution which changes with $\eta = 1$ and beyond. In Fig. 9 we show the performance comparison of different LI and FI strategies for the high offered load case of $\eta = 1.2$. This is done to show that even though POL has high job drops and response times, it outperforms the LI strategies and has comparable performance to the FI strategy, especially DJSQ-FI.

For the sake of completeness, Fig. 10 compares the performance of POL using different reward functions from Section 4.6, while keeping all the other parameters the same.

Time Analysis of POL: POL consists of two main components: (i) Tree simulator for action evaluation and (ii) SIR particle filter for belief update. Both steps need to be done at every decision epoch, i.e. on every job arrival. Since we assume no queue at the load-balancer, POL needs to allocate the incoming job to one of the queues, before the next arrival. Note that MCTS is a very successful online algorithm. Hence, POL first takes a portion of the time between arrivals to simulate the tree and take a decision for the current arrival. Then takes that action on the real environment and based on the received delayed acknowledgement performs the belief update using the SIR particle filter, until the next job arrives. As can be seen from the simulation results, POL is scalable in terms of number of servers and is able to handle high offered loads, η. Note that the code used to run POL here in the system simulation is the same that would be used in a

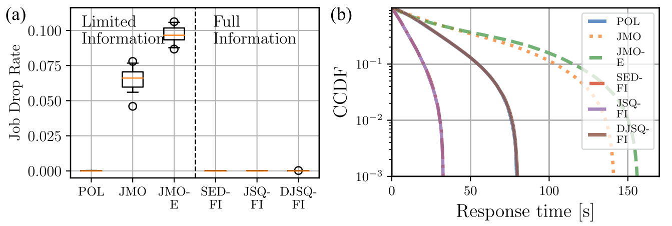

Fig. 4. $N=50$ heterogeneous servers with job inter-arrival times and service times described by an exponential distribution, with the offered load ($\eta \approx 1$). POL outperforms the other algorithms with limited information, JMO and JMO-E, in terms of both (a) job drops and (b) response time. And has comparable performance to the full information strategies, SED-FI, DJSQ-FI and JSQ-FI.

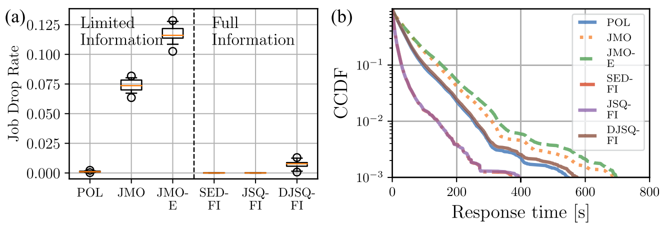

Fig. 5. $N=50$ heterogeneous servers with gamma arrivals and Pareto service times, with offered load ($\eta \approx 1$). The effect of the heavy tailed Pareto distribution can be seen in the response plot (b). In terms of job drops, POL outperforms limited information (LI) strategies as well as FI strategy, DJSQ-FI.

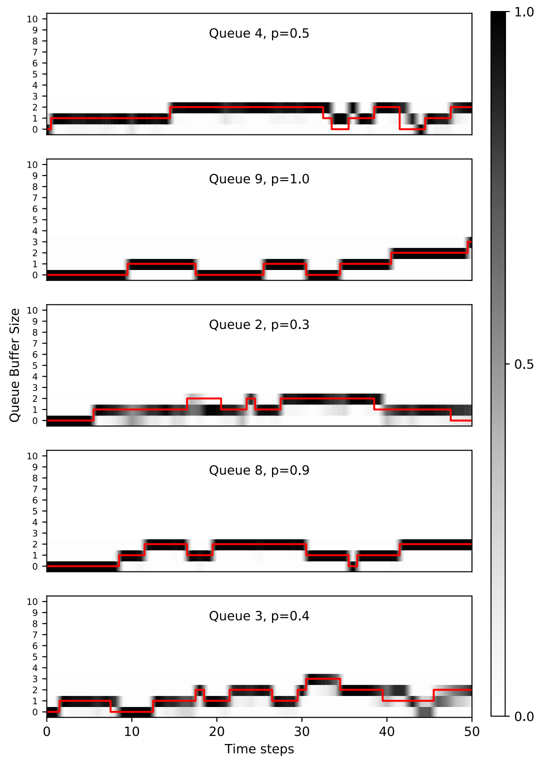

Fig. 6. $N = 10$ homogenous servers from which 5 were chosen at random to see their sample run for 50 time steps. The delay p for the acknowledgements from each of the server was also allocated randomly, ranging from 0.1 to 1.0. In each subplot is given the queue number and its delay, p. The solid red line trajectory is the true sample path for each queue (*not known to POL*), while the shaded region around it is the belief probability that POL has for each state at each time step. Exponentially distributed inter-arrival and service times were used, with the offered load $\eta \approx 1$. POL, with the help of SIR particle filter, is able to track the real state of the system, as long as the delay is not too high, which is why it is able to perform as good as FI strategies.

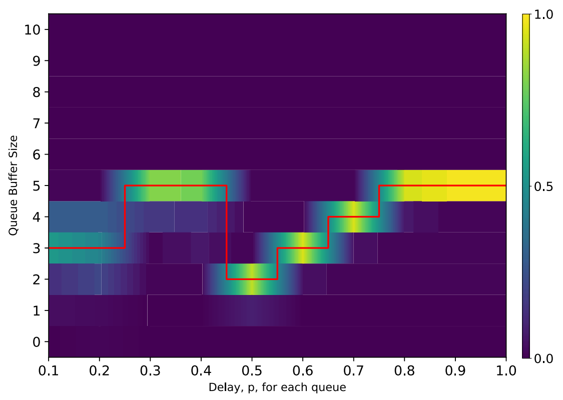

Fig. 7. Visualization of belief state, based on the particles, of $N = 10$ queues after 1000 epochs. The x-axis gives the delay probability of each queue, ranging from $p = 0.1$ (worse delay) to $p = 1.0$ (no delay). The solid red line trajectory is the true state of each queue (not known to POL), while the shaded region around it is the belief probability that POL has for each queues' state after 1000 time steps. It can be seen that as the acknowledgements become less delayed (going from $p = 0.1$ to $p = 1.0$)), the belief of POL gets closer to the true state of the queue. Exponentially distributed inter-arrival and service times were used, with the offered load, $\eta \approx 1$.

deployment scenario. Next, we investigate the impact on the inter-arrival time between jobs on the load balancer performance as the inter-arrival time needs to be sufficient for POL to perform the above two steps at every job arrival.

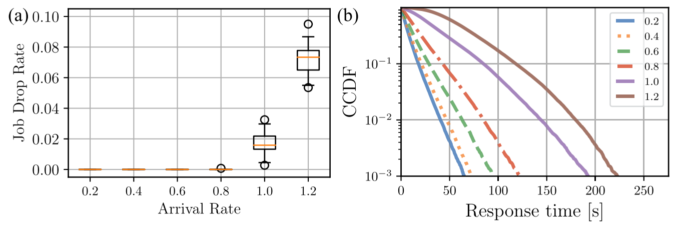

Fig. 8. Varying offered load for a setup as in Fig. 4. As long as the offered load is $\eta < 1$, POL has no job losses. For loads $\eta \geq 1$ we observe a load dependent exponential tail of the response time distribution.

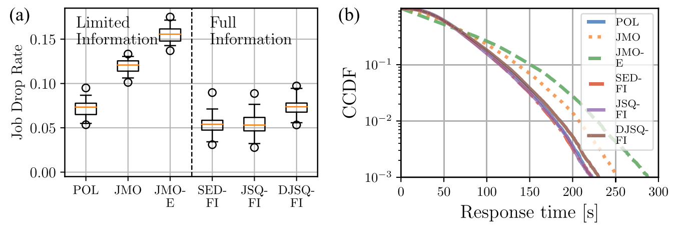

Fig. 9. For the setup from Fig. 8 with an offered load $\eta = 1.2$: POL shows a comparable performance to full information (FI) strategies, while outperforming other limited information (LI) ones.

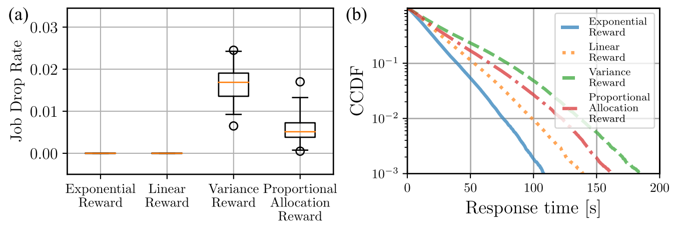

Fig. 10. Comparison of different reward functions for 50 homogeneous servers, with exponentially distributed inter-arrival and service times and offered load $\eta \approx 1$.

In Fig. 11, the number of homogeneous servers N was increased from 10 to 90, while keeping the offered load fixed, i.e., $\eta = 0.99$. The average service rate μ_i of the servers is kept fixed in all experiments, i.e. the increase in N results in an increase in the sum of service rates, $\sum_i \mu_i$. Hence, to keep the offered load fixed with scale the job arrival rate accordingly. Firstly, this experiment demonstrates the scalability of POL in terms of number of servers. Although the state space of the system increases with the number of servers, hence, queues, POL manages to deal well with the increased state space. In POL we use MCTS adapted from POMCP [30], so instead of considering the entire state space we have a fixed set of particles to represent the state based on our belief of the state, thus tapering the curse of dimensionality and space complexity. As the inter-arrival time is the decision epoch we observe that the time given to POL to simulate the tree and do the belief update reduces, the effect of which can be seen as the slight decrease in performance as N increases. It can be seen in Fig. 12, that for $N = 90$, lowering the load again, i.e. giving POL more time to decide, improves the performance of the system. This shows the trade-off between the load balancing performance, e.g. in terms of the response time and drop rate vs. the load which directly impacts the time provided to POL to make a decision.

We believe this to be the current limitation of POL, however in future, step (i) can be further optimized using MCTS

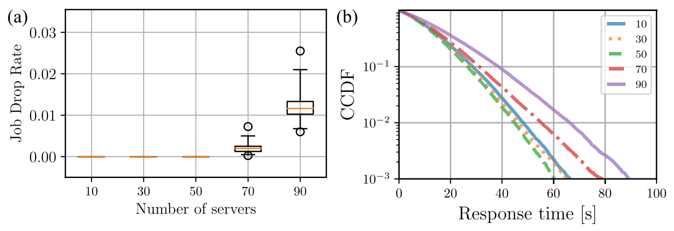

Fig. 11. Homogeneous servers with exponentially distributed inter-arrival and service times. The number of servers are increased in intervals of 20 servers, while keeping the offered load always $\eta = 0.99$. With a higher number of servers, less time is available for POL to simulate the tree and do belief update, resulting in a slight deterioration in the performance.

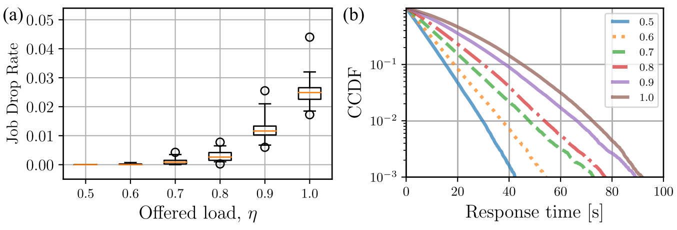

Fig. 12. Performance of POL for $N = 90$ homogeneous servers for different loads.

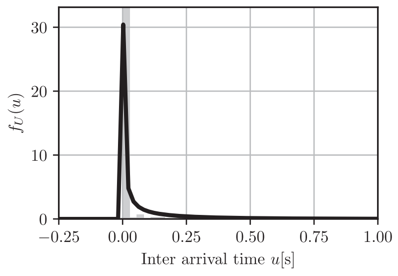

Fig. 13. The density estimate of the job inter-arrival times indicates that it is exponentially distributed.

parallelization [42]. Note that the computational resources used also have a strong impact on the performance of POL. Here, we use a dedicated machine with an Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz for all our experiments.

In the next section, we discuss the scenario when some system parameters are not known and need to be inferred from the available data.

5.4 Experiments With Trace Data

For the results of this section, we make use of *Labeled Network Traffic Flow* data, provided by Kaggle in 2019 [8], [9]. We used the frameworks given in Section 4.5 to infer the underlying distributions of the inter-arrival and service times provided in this data set. The inferred distribution based on data of the inter-arrival times of the chosen source is given in Fig. 13, it can be seen to follow an exponential distribution with high arrival rate. We then selected 20 heterogeneous servers from the available data such that they all followed the Gamma Mixture distribution. Gamma Mixture was selected to show the performance of POL with yet another type of service time distributions. The empirical distribution as a histogram as well as the posterior mean estimate for some of these selected servers is given in Fig. 14. The hyperparameters used are: $c = 3$, $a_i, b_i = 1$, $m = 1$. POL makes use of the samples generated using the posterior predictions.

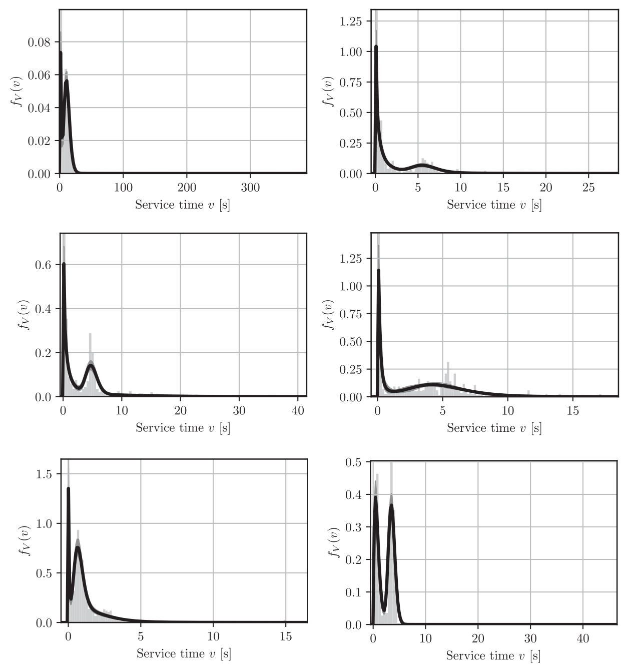

Fig. 14. The density estimate of the job service times using a Gamma Mixture Likelihood model indicates that the servers are heterogeneous with high service times.

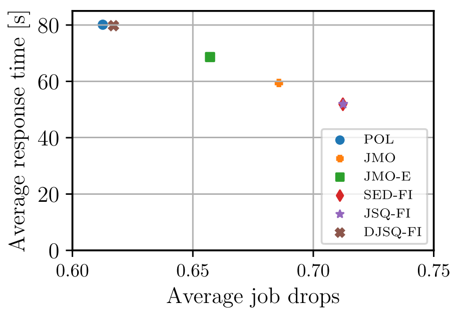

Fig. 15. Tradeoff between response time and job drop: In terms of job drops, POL has comparable performance to DJSQ-FI, while it outperforms the other FI and all LI strategies. The additionally allocated jobs increase the overall average response time.

We assume that the servers have a finite buffer of size $B = 10$ (arbitaily chosen) and a delay in acknowledgements of $p = 0.6$. Fig. 15 shows that even with limited information, POL has the lowest average job drops. However, due to the limited information available to POL and the reward function it is using, its average response time is as high as the full information strategy DJSQ-FI. POL is able to allocate more jobs to the servers (due to fewer drops), which can come at a cost of higher response time for some jobs, which will be allocated to the slower servers. Note that the reward function we used, (15), focuses on avoiding job drops and not on minimizing the response time.

6 Discussion & Conclusion

In this work, we analyzed online algorithms for mapping incoming jobs to parallel and heterogeneous processing systems under partial observability constraints. This partial observability is rooted in the assumption that the entity controlling this mapping, denoted load-balancer, takes decisions only based on *randomly delayed feedback* of the parallel systems. Unlike classical models that assume full knowledge of the parallel systems, e.g., knowing the queue lengths (Join-the-Shortest Queue - JSQ) or additionally the job service times (Shortest Expected Delay - SED) this model is particularly suited for large distributed processing systems that only provide an acknowledgement-based feedback.

In addition to presenting a partially observable (semi-) Markov decision process model that captures the load balancing decisions in this parallel queuing system under delayed acknowledgements, we provide a Partial Observable Load-Balancer (POL) - to find near-optimal solutions online. A particular strength of POL is that it allows to *define the objectives of the system and lets it find the appropriate load balancing policy instead of manually defining a fixed one*. It can also be used for any kinds of inter-arrival and service time distributions and is scalable to a large number of queues. We numerically show that the POL load balancing policies obtained under partial observability are comparable to fixed policies such as JSQ, JSQ(d) and SED which have full information. This is the case even though POL receives less, and in addition randomly delayed, informative feedback.

One future direction is to parallelize the tree search part of POL to enable more trajectory simulations in a limited time. Use of GPUs can also be tested to investigate the system performance for a higher number of queues. Another direction for extending this work is to include the memory effect of the last served job in the model simulator, e.g., through state space extension. Heterogeneous and batch jobs can also be considered. This work can also be extended to more than one load-balancer, working in parallel in a multi-agent manner.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[2] I. Polato, R. Ré, A. Goldman, and F. Kon, "A comprehensive view of Hadoop research—A systematic literature review," *J. Netw. Comput. Appl.*, vol. 46, pp. 1–25, 2014.

[3] A. Fox et al., "Above the clouds: A berkeley view of cloud computing," Dept. Electrical Eng. Comput. Sciences, Univ. California, Berkeley, Tech. Rep. UCB/EECS, 2009.

[4] A. Hordijk and G. Koole, "On the optimality of the generalized shortest queue policy," *Probability Eng. Inf. Sci.*, vol. 4, no. 4, pp. 477–487, 1990.

[5] W. Winston, "Optimality of the shortest line discipline," *J. Appl. Probability*, vol. 14, no. 1, pp. 181–189, 1977.

[6] S. A. Banawan and J. Zahorjan, "Load sharing in heterogeneous queueing systems," in *Proc. 8th Annu. Joint Conf. IEEE Comput. Commun. Societies*, 1989, pp. 731–732.

[7] J. Selen, I. Adan, S. Kapodistria, and J. van Leeuwaarden, "Steady-state analysis of shortest expected delay routing," *Queueing Syst.*, vol. 84, no. 3–4, pp. 309–354, 2016.

[8] J. S. Rojas, "Labeled network traffic flows," Accessed: Sep. 02, 2022. https://www.kaggle.com/jsrojas/labeled-network-traffic-flows-114-applications

[9] J. S. Rojas, A. Pekar, Á. Rendón, and J. C. Corrales, "Smart user consumption profiling: Incremental learning-based OTT service degradation," *IEEE Access*, vol. 8, pp. 207 426–207 442, 2020.

[10] S. A. Banawan and N. M. Zeidat, "A comparative study of load sharing in heterogeneous multicomputer systems," in *Proc. 25th IEEE Annu. Simul. Symp.*, 1992, pp. 22–31.

[11] 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," 2018, *arXiv:1806.05444*.

[12] W. Whitt, "Deciding which queue to join: Some counter-examples," *Operations Res.*, vol. 34, no. 1, pp. 55–62, 1986.

[13] D. Arcelli, "Exploiting queuing networks to model and assess the performance of self-adaptive software systems: A survey," *Procedia Comput. Sci.*, vol. 170, pp. 498–505, 2020.

[14] S. Shevtsov, M. Berekmeri, D. Weyns, and M. Maggio, "Control-theoretical software adaptation: A systematic literature review," *IEEE Trans. Softw. Eng.*, vol. 44, no. 8, pp. 784–810, Aug. 2018.

[15] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 10, pp. 1094–1104, Oct. 2001.

[16] D. Shah and B. Prabhakar, "The use of memory in randomized load balancing," in *Proc. IEEE Int. Symp. Inf. Theory*, 2002, Art. no. 125.

[17] K. Psounis and B. Prabhakar, "Efficient randomized web-cache replacement schemes using samples from past eviction times," *IEEE/ACM Trans. Netw.*, vol. 10, no. 4, pp. 441–454, Aug. 2002.

[18] B. Liu, Q. Xie, and E. Modiano, "Reinforcement learning for optimal control of queueing systems," in *Proc. 57th Annu. Allerton Conf. Commun. Control Comput.*, 2019, pp. 663–670.

[19] U. Ayesta, "Reinforcement learning in queues," *Queueing Syst.*, vol. 100, pp. 497–499, 2022.

[20] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.

[21] B. Alt et al., "Transitions: A protocol-independent view of the future internet," *Proc. IEEE*, vol. 107, no. 4, pp. 835–846, Apr. 2019.

[22] K. Krishnan, "Joining the right queue: A Markov decision-rule," in *Proc. 26th IEEE Conf. Decis. Control*, 1987, pp. 1863–1868.

[23] E. Altman and P. Nain, "Closed-loop control with delayed information," *ACM Sigmetrics Perform. Eval. Rev.*, vol. 20, no. 1, pp. 193–204, 1992.

[24] J. Kuri and A. Kumar, "Optimal control of arrivals to queues with delayed queue length information," *IEEE Trans. Autom. Control*, vol. 40, no. 8, pp. 1444–1450, Aug. 1995.

[25] D. Artiges, "Optimal routing into two heterogeneous service stations with delayed information," in *Proc. 32nd IEEE Conf. Decis. Control*, 1993, pp. 2737–2742.

[26] I. Chadès, L. V. Pascal, S. Nicol, C. S. Fletcher, and J. Ferrer-Mestres, "A primer on partially observable markov decision processes (POMDPs)," *Methods Ecol. Evol.*, vol. 12, no. 11, pp. 2058–2072, 2021.

[27] S. Ross, J. Pineau, S. Paquet, and B. Chaib-Draa, "Online planning algorithms for POMDPs," *J. Artif. Intell. Res.*, vol. 32, pp. 663–704, 2008.

[28] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artif. Intell.*, vol. 101, no. 1–2, pp. 99–134, 1998.

[29] J. Pineau, G. Gordon, and S. Thrun, "Anytime point-based approximations for large POMDPs," *J. Artif. Intell. Res.*, vol. 27, pp. 335–380, 2006.

[30] D. Silver and J. Veness, "Monte-carlo planning in large POMDPs," in *Proc. Adv. Neural Inf. Process. Syst.*, 2010, pp. 2164–2172.

[31] H. Yu et al., "Approximate solution methods for partially observable Markov and semi-Markov decision processes," Ph.D. dissertation, Massachusetts Inst. Technol., Cambridge, MA, USA, 2006.

[32] S. M. Ross, *Introduction to Probability Models*. Cambridge, MA, USA: Academic Press, 2014.

[33] M. D. Hoffman and A. Gelman, "The No-U-Turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1593–1623, 2014.

[34] J. Salvatier, T. V. Wiecki, and C. Fonnesbeck, "Probabilistic programming in Python using PyMC3," *PeerJ Comput. Sci.*, vol. 2, 2016, Art. no. e55.

[35] S. Ghosal, and A. Van der Vaart, *Fundamentals of Nonparametric Bayesian Inference*. Cambridge, U.K.: Cambridge Univ. Press, 2017.

[36] R. Bellman, "Dynamic programming," *Science*, vol. 153, no. 3731, pp. 34–37, 1966.

[37] D. Silveretal ., "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[38] D. Silveretal ., "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[39] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *Proc. Eur. Conf. Mach. Learn.*, 2006, pp. 282–293.

[40] N. A. Vien and M. Toussaint, "Hierarchical Monte-Carlo planning," in *Proc. 29th AAAI Conf. Artif. Intell.*, 2015, pp. 1–7.

[41] P. L'Ecuyer, "Efficiency improvement and variance reduction," in *Proc. IEEE Winter Simul. Conf.*, 1994, pp. 122–132.

[42] G. M.-B. Chaslot, M. H. Winands, and H. Herik, "Parallel Monte-Carlo tree search," in *Proc. Int. Conf. Comput. Games*, 2008, pp. 60–71.

Anam Tahir received the BSc degree in electrical engineering and information technology from the National University of Science and Technology, Pakistan, and the MSc degree in electrical engineering and information technology from Technische Universität Darmstadt, in 2018. Since November 2018, she is working as a research associate within the Self-Organizing Systems Lab, Technische Universität Darmstadt. She is interested in planning and performance analysis of large queuing and other networked systems, particularly in uncertain environments.

Bastian Alt received the BSc and MSc degrees in electrical engineering and information technology from Technische Universität Darmstadt, in Nov. 2013 and Dec. 2016, respectively, and the doctoral degree (Dr.-Ing.) from the Department of Electrical Engineering and Information Technology, Technische Universität Darmstadt, in 2022. Since January 2017, he is working as a research associate within the Self-Organizing Systems Lab, Technische Universität Darmstadt. He is interested in the modelling of network systems using tools from machine learning, control and optimization.

Amr Rizk (Senior Member, IEEE) received the doctoral degree (Dr.-Ing.) from the Leibniz Universität Hannover, Germany, in 2013. After that he held postdoctoral positions with the University of Warwick, UMass Amherst, and the TU Darmstadt, Germany. From 2019 to 2021 he was an assistant professor with Ulm University, Germany. Since 2021 he is a professor with the Department for Computer Science, University of Duisburg-Essen, Germany. He is interested in performance evaluation of communication networks, stochastic models of networked systems and their applications.

Heinz Koeppl received the MSc degree in physics from Karl-Franzens University Graz, in 2001, and the PhD degree in electrical engineering from the Graz University of Technology, Austria, in 2004. After that he held postdoctoral positions with UC Berkeley and Ecole Polytechnique Federalede Lausanne (EPFL). From 2010 to 2014 he was an assistant professor with the ETH Zurich and part-time group leader at IBM Research Zurich, Switzerland. Since 2014 he is a full professor with the Department of Electrical Engineering and Information Technology, Technische Universität Darmstadt, Germany. He received two awards for his PhD thesis, the Erwin Schrödinger Fellow-ship, the SNSF Professorship Award and currently holds an ERC consolidator grant. He is interested in stochastic models and their inference in applications ranging from communication networks to cell biology.