

Multi-Agent Reinforcement Learning for Network Load Balancing in Data Center

Zhiyuan Yao, Zihan Ding, Thomas Clausen

▶ **To cite this version:**

Zhiyuan Yao, Zihan Ding, Thomas Clausen. Multi-Agent Reinforcement Learning for Network Load Balancing in Data Center. 31st ACM International Conference on Information and Knowledge Management (CIKM '22), Oct 2022, Atlanta, GA, United States. 10.1145/3511808.3557133 . hal-03753203

HAL Id: hal-03753203

https://polytechnique.hal.science/hal-03753203

Submitted on 18 Aug 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Multi-Agent Reinforcement Learning for Network Load Balancing in Data Center

Zhiyuan Yao
zhiyuan.yao@polytechnique.edu
École Polytechnique & Cisco Systems
Paris, France

Zihan Ding
zihand@princeton.edu
Princeton University
Princeton, New Jersey, USA

Thomas Clausen
thomas.clausen@polytechnique.edu
École Polytechnique
Paris, France

ABSTRACT

This paper presents the network load balancing problem, a challenging real-world task for multi-agent reinforcement learning (MARL) methods. Conventional heuristic solutions like Weighted-Cost Multi-Path (WCMP) and Local Shortest Queue (LSQ) are less flexible to the changing workload distributions and arrival rates, with a poor balance among multiple load balancers. The cooperative network load balancing task is formulated as a Dec-POMDP problem, which naturally induces the MARL methods. To bridge the reality gap for applying learning-based methods, all models are directly trained and evaluated on a real-world system from moderate- to large-scale setups. Experimental evaluations show that the independent and "selfish" load balancing strategies are not necessarily the globally optimal ones, while the proposed MARL solution has a superior performance over different realistic settings. Additionally, the potential difficulties of the application and deployment of MARL methods for network load balancing are analysed, which helps draw the attention of the learning and network communities to such challenges.

CCS CONCEPTS

• **Computing methodologies** → **Multi-agent reinforcement learning**; • **Networks** → **Network resources allocation**; *Cloud computing*;

KEYWORDS

MARL; load balancing; distributed systems

ACM Reference Format:
Zhiyuan Yao, Zihan Ding, and Thomas Clausen. 2022. Multi-Agent Reinforcement Learning for Network Load Balancing in Data Center. In *Proceedings of the 31st ACM International Conference on Information and Knowledge Management (CIKM '22), October 17–21, 2022, Atlanta, GA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3511808.3557133

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CIKM '22, October 17–21, 2022, Atlanta, GA, USA.
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9236-5/22/10…$15.00
This work is, in part, supported by the Cisco endowed "Internet Technologies and Engineering" chaire at École Polytechnique.
https://doi.org/10.1145/3511808.3557133

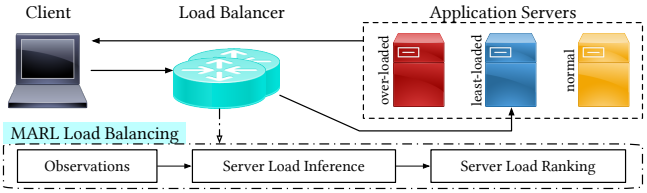


Figure 1: Network load balancing in DC networks and the scope of study of this paper.

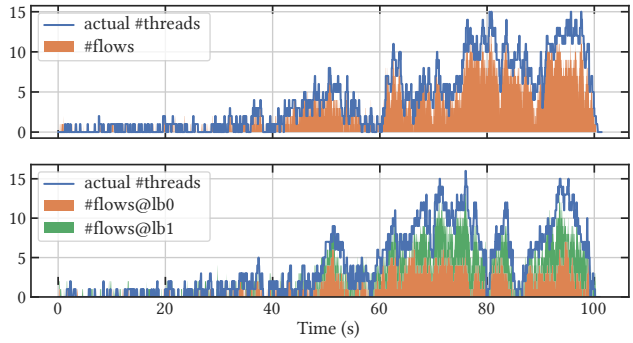


Figure 2: Comparison of the observed number of on-going flows (#flow) between single- (top) and double-load-balancer (bottom) environment.

1 INTRODUCTION

In data centers (DCs), network load balancers (LBs) play a significant role to distribute time-sensitive requests from clients across a cluster of application servers and provide scalable services [9]. The network topology of load balanced system in real-world DCs is depicted in Figure 1. With the advancement of virtualisation technology and elastic data centers [8], application servers can be instantiated on heterogeneous architectures [15] and have different processing capacities. This requires network LBs to make fair workloads distribution decisions based on instant server load states to optimise resource utilisation, so that less application servers can be provisioned to provide better quality of service (QoS)–lower job completion time (JCT)–with reduced operational costs. However, there are **2 challenges** for network LBs to make fair workloads distribution decisions in real-world systems.

Network LBs have limited observations to make informed decisions. Operating at the Transport Layer, network LBs are agnostic to Application-Layer protocol and do not inspect the Application-Layer headers or payloads in network packets, in order to generalise to, and stay universal for all types of network applications [9]. However, this makes LBs also agnostic to the

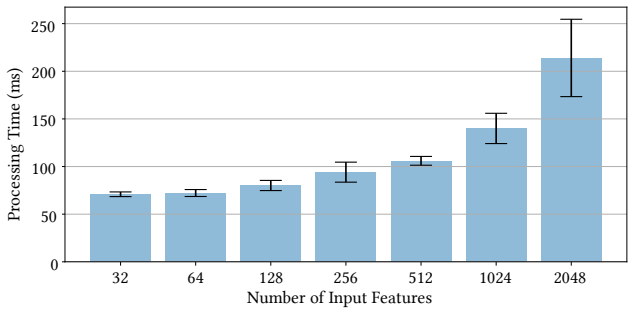


Figure 3: With a 3 layer fully connected neural network with 4865 parameters built with Keras/Tensorflow, assuming input datapoints have 32 features, it takes more than 50ms to obtain a singal output, when using single CPU core (Intel Xeon CPU E5-2690 v3 at 2.60GHz).

information of received requests and network flows (jobs)–*e.g.*, expected JCT, computation intensity, required database–when making load balancing decisions, which can lead to overloaded servers dealing with multiple heavy network flows [11, 29]. Besides, to avoid single-point-of-failure and improve system reliability, multiple LBs are deployed in modern DCs so that the service stays available when a LB fails. To improve workload distribution fairness, heuristic LBs [11, 29] make informed load balancing decisions based on simple features (*e.g.*, the number of on-going connections) extracted from network packets. However, the presence of multiple LBs makes them have only partial observations on the network traffic and workloads distributed among servers. An example of partial observation of the number of on-going flows is depicted in Figure 2. In presence of 2 LBs at the same time, the counted number of flows (`#flow`) no longer reflect the actual number of busy threads (`#threads`) on the server. This indicates that more features–especially those that are not affected by partial observations (*e.g.*, latency-related features)–should be taken into account when making load balancing decisions.

Network LBs deal with high flow arrival rates (higher than 500 flows/s)–load balancing decisions have to be made within sub-ms or micro-second level [7]. Machine learning (ML) and reinforcement learning (RL) approaches are able to make inferences and informed decisions based on multi-modal features extracted from dynamic environments, and they have shown performance gains in various system and networking problems and help avoid error-prone manual configurations [1, 4, 17, 24, 26]. However, Figure 3 shows that it is computationally intractable to apply ML/RL algorithms on network load balancing problems to make more than 500 load balancing decisions per second even using just a minimal size of neural network. Therefore, state-of-the-art network LBs rely on heuristics for decisions on where to place workloads [2, 7, 10].

This paper formulates the network load balancing problem as a cooperative game to bridge the gap between the networked computing system community and MARL community. A new load balancing method RLB is proposed, allowing both (i) taking advantage of the learning and inference capacity of RL algorithms given only partial observations, and (ii) making–at micro-second level–data-driven load balancing decisions that bring performance gains. Experimental evaluations are conducted by running real-world networking traffic to compare and contrast the proposed mechanism with state-of-the-art (SOTA) load balancing algorithms.

The contribution of this paper are summarised as follows: (i) This paper formally defines the network load balancing problem as a cooperative game in Dec-POMDP [20] framework and presents a real-world system implementation for realistic performance evaluations. (ii) This paper proposes a new mechanism that allows benefiting from MARL algorithms and make sub-ms level data-driven load balancing decisions. (iii) This paper implements 3 different learning agents–one based on the MARL algorithm QMIX [21] and two other based on independent agents using RL algorithm soft actor-critic (SAC) [12] for solving load balancing tasks with multiple LBs, and evaluates their performance on different scenarios, and compares with 5 SOTA heuristic load balancing algorithms. (iv) By analysing the experimental results, this paper implies the potential challenges (*e.g.* scalability, synchronisation among distributed agents) for MARL to solve the network load balancing problem, and suggests future work directions.

2 RELATED WORK

Network Load Balancing: Network LBs in modern DCs follow the distributed design as in [10], where multiple LBs randomly assign servers to incoming tasks using Equal-Cost Multi-Path (ECMP). In case where servers have different processing capacities, servers are assigned with probabilities proportional to their weights using Weighted-Cost Multi-Path (WCMP). However, as available server processing capacities change with time in DCs [8], these statically configured weight may not correspond to the actual processing capacities of servers, and they fail to follow and adapt to the dynamic networking environment [1, 11]. As a variant of WCMP, active WCMP (AWCMP) periodically updates server weights using regression algorithm [1] or threshold-based algorithm [2], by actively probing resource utilisation information (CPU, memory, or IO usage). However, AWCMP requires modifications on every servers to manage communication channels that allow collecting observations. This incurs additional control messages and management overheads, especially in large and scalable DCs. Local shortest queue (LSQ) counts the number of distributed jobs on each server [11, 29], yet it assumes that all servers share the same processing capacity. The proposed method in this paper requires no modification in the distributed system, yet it is able to passively extract features from network traffic, which help infer server load states and make adaptive and fair load balancing decisions.

MARL for Cooperative Games in Networked Computing Systems: RL has been applied on scheduling problems [4, 17, 25], which is similar yet different from network load balancing problems. Agents in scheduling problems know *a priori* the information of workloads–including the expected job durations, job dependencies, etc. –to be distributed before assigning workloads to different processing queues. However, network LBs have limited observations on only the subset of network flows they distribute–only the number of flows distributed on each server and the elapsed time of on-going network flows. Comparing with the networking problems, the jobs in scheduling problems also arrive at lower rates (at second level) and have longer completion time. The inaccurate observation

and highly-frequent decision making process make network load balancing a challenging problem to solve. Previous work [30] has explored single-agent RL for load balacing in a network systems, with a close-to SOTA heuristic performance evaluated in simulations. For the case with multiple LBs working at the same time, this paper applies and studies MARL algorithms on the network load balancing problem in real-world systems to evaluate server load states based only on local observations extracted from network packets, and to dynamically adapt to time-variant environments. Researches on centralised training, decentralised execution (CTDE) frameworks show performance gain in multi-agent setups [13, 21, 27]. [13] assumes that agents are homogeneous and therefore interchangable, which does not apply in DC networks where load balancers can be deployed on different hardware infrastructure with topologically different distances from servers. This paper adapts QMIX [21]–which achieves similar performance as in [27]–in an asynchronous mechanism to make highly frequent load balancing decisions, and to learn from partial observations and solve the network load balancing problem as a cooperative game. RL-based algorithm has also been applied on other load balancing problems [3, 14, 16, 19]. However, the network load balancing problem studied in this paper is different from link load balancing problems studied in [14, 16], where link utilisation is to be maximised and load balancers have observations on link utilisations. As discussed in [32], in network load balancing problems–more precisely, Layer-4 server load balancing problem is studied in this paper–load balancers have no direct observation on server utilisations. The network load balancing problem studied in this paper is also different from mobility load balancing problem studied in [3, 19], where load balancers operate in celular networks instead of DCs and they do not correspond to all resourcese. The action and optimisation goal (link failure) are entirely different as well. Furthermore, these works [3, 14, 16, 19] conduct evaluations based on simulations while this paper implements and evaluates MARL-based load balancing algorithms in a real-world testbed.

3 MARL NETWORK LOAD BALANCING

This section defines the network load balancing problem and formulates it into a cooperative game.

3.1 Multi-Agent Load Balancing Problem

Network load balancing can be defined as allocating a Poisson sequence of network flows with different workloads $w \in \mathcal{W}$–whose unit can be, *e.g.*, amount of time to process–on a set of n servers, to achieve the maximal utilisation of the computational capacity of all servers. The workload $w_j(t)$ assigned on the j-th server at time t follows an exponential distribution in practical experiments [22]. Multi-agent load balancing problem considers workload distribution through a number of m LBs, which provide high availability and reliability in modern DCs [10].

Deterministic Case. The load balancing method for each LB $i \in [m]$ can be a pure strategy $\pi_i \in \Pi_i$: $\mathcal{W} \to [n]$. Therefore, a deterministic workload assignment function is: $\mathcal{W} \times [m] \to [n]$. The ensembled policy for the whole multi-agent load balancing system is thus $\boldsymbol{\pi} = [\pi_1, \ldots, \pi_m] \in \boldsymbol{\Pi} = \Pi_1 \times \cdots \times \Pi_m$. The processing speed for each server is $v_j, j \in [n]$, *i.e.*, the amount of workloads that can be processed per unit time. The remaining workloads on the j-th server ($j \in [n]$) during a time interval $t \in [t_0, t_n)$ is thus:

$$l_j = \frac{\sum_{i \in [m]} \sum_{t \in [t_0, t_n)} w_{i,j}(t)}{v_j}, \tag{1}$$

where $w_{i,j}(t)$ indicates the workload at time t assigned to the j-th server via the i-th LB. l_j represents the expected time to finish processing all the workloads on the j-th server.

Stochastic Case. Since modern DCs have fan-out topology and $m < n$, using deterministic strategy during a time interval will flood n servers under heavy traffic rate (*e.g.*, higher than 500 flows/s), therefore the stochastic load balancing strategies are more often used in practice [10, 18]. The stochastic workload assignment function α is defined as: $\mathcal{W} \times [m] \times [n] \to [0, 1]$, representing the probability of the event that the workload is assigned by a specific LB to a specific server. The expected time to finish all workloads on j-th server during the time interval $t \in [t_0, t_n)$ is, $\forall j \in [n]$:

$$l_j = \frac{\sum_{i \in [m]} \sum_{t \in [t_0, t_n)} w_i(t) \alpha_{i,j}(t)}{v_j}, \sum_{j=1}^{n} \alpha_{i,j}(t) = 1, \tag{2}$$

$\alpha_{i,j}(t)$ denoting the probability that LB i chooses server j for load w at time t.

Objective. The objective for the whole load balancing system can be defined as finding the optimal ensemble policy:

$$\boldsymbol{\pi}^* = \min_{\boldsymbol{\pi} \in \boldsymbol{\Pi}} c(\boldsymbol{l}), \boldsymbol{l} = \{l_j\}, j \in [n] \tag{3}$$

where c is a cost function depending on the expected task finishing time for all servers $j \in [n]$. The definition of *makespan* is $c(l_1, \ldots, l_j) = \max_{j \in [n]}\{l_j\}$. However, in the practical network load balancing problem, LB agents have no observation over the theoretical makespan for the following reasons:

(1) operating at the Transport Layer, LB agents are agnostic to application-level information, thus they cannot estimate the remaining workload on each server but counting only the amount of ongoing jobs;
(2) the expected JCT of networking requests follow long-tail distribution [22], which makes it difficult to estimate remaining workload only based on the number of ongoing jobs;
(3) in multi-agent setups for the sake of reliability, LB agents only observe partial networking traffic, which makes their counted number of ongoing jobs partial and inaccurate;
(4) the estimation of makespan uses the max operator, which may produce large variances when facing dynamic traffic.

This paper thus proposes a different cost function–fairness index–which is proved to be equivalent of the makespan as objective.

Definition 3.1. (Fairness) For a vector of task completion time $\boldsymbol{l} = [l_1, \ldots, l_n]$ on each server $j \in [n]$, the linear product-based fairness for workload distribution is defined as:

$$F(\boldsymbol{l}) = F([l_1, \ldots, l_n]) = \prod_{j \in [n]} \frac{l_j}{\max(\boldsymbol{l})} \tag{4}$$

Proposition 3.2. *Maximising the linear product-based fairness is sufficient for minimising the makespan:*

$$\max F(\boldsymbol{l}) \Rightarrow \min \max(\boldsymbol{l}) \tag{5}$$

Proof. For a vector of task completion time $\boldsymbol{l} = [l_1, \ldots, l_n]$ on each server $j \in [n]$, by the definition of fairness,

$$\max F(\boldsymbol{l}) = \max \frac{\prod_{j\in[n]} l_j}{\max_{k'\in[n]} l_{k'}} \tag{6}$$

WLOG, let $l_k = \max_{k'\in[n]} l_{k'}$, then,

$$\max F(\boldsymbol{l}) = \max \prod_{j\in[n], j\neq k} l_j \tag{7}$$

By means inequality,

$$\left(\prod_{j\in[n],j\neq k} l_j\right)^{\frac{1}{n-1}} \leq \frac{\sum_{j\in[n],j\neq k} l_j}{n-1} = \frac{C-l_k}{n-1}, C = \sum_{j\in[n]} l_j. \tag{8}$$

with the equivalence achieved when $l_i = l_j, \forall i, j \neq k, i, j \in [n]$ holds. Therefore,

$$\max F(\boldsymbol{l}) \Rightarrow \max \frac{C-l_k}{n-1} \tag{9}$$

$$\Leftrightarrow \min l_k \tag{10}$$

$$\Leftrightarrow \min \max_{j\in[n]} l_j \tag{11}$$

The inverse may not hold since $\max \frac{C-l_k}{n-1}$ does not indicate $\max F(\boldsymbol{l})$, so maximising the linear product-based fairness is sufficient but not necessary for minimising the makespan. This finishes the proof. □

To sum up, the network load balancing can be formulated as a constrained optimisation problem:

$$maximise \prod_{j\in[n]} \frac{l_j}{\max(\boldsymbol{l})} \tag{12}$$

$$subject\ to \quad l_j = \frac{\sum_{i\in[m]} \sum_{t\in[t_0,t_n)} w_i(t)\alpha_{i,j}(t)}{v_j} \tag{13}$$

$$\sum_{j=1}^{n} \alpha_{i,j}(t) = 1 \tag{14}$$

$$\sum_{i=1}^{m} w_i \leq \sum_{j=1}^{n} v_j \tag{15}$$

$$\alpha_{i,j} \in [0,1], w_i, v_j \in (0, +\infty). \tag{16}$$

The optimisation cost c in Eq. (3) is transformed to be the product-based fairness $F(\boldsymbol{l})$ due to the Proposition 3.2. Constraints (13) and (14) are from Eq. (2) for stochastic network load balancing. Constraint (15) is a necessary condition to have bounded queue backlog (stability).

3.2 MARL Methods

The multi-agent load balancing problem defined in Eq. (12)-(16) can be viewed as a multi-agent cooperative game, where each agent needs to coordinate their behaviour to maximise the common payoff. Specifically, each agent acts independently according to their local observations, the common payoff is improved as long as each agent improves their local policies.

Dec-POMDP: MARL for cooperative games can be formulated as decentralised partially observable Markov decision process (Dec-POMDP) [20], which can be represented as $(\mathcal{I}, \mathcal{S}, \mathcal{A}, R, O, \mathcal{T}, \gamma)$. $\mathcal{I}$ is the agent set, $\mathcal{S}$ is the state set and $\mathcal{A} = \times_i \mathcal{A}_i, i \in \mathcal{I}$ is the joint action set, $O = \times_i O_i, i \in \mathcal{I}$ is the joint observation set, and R is the global reward function $R(s,a)$: $\mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ for current state $s \in \mathcal{S}$ and action $\boldsymbol{a} \in \mathcal{A}$. The state-transition probability from current state and action to a next state $s' \in \mathcal{S}$ is defined by $\mathcal{T}(s'|s,\boldsymbol{a})$: $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0,1]$. $\gamma \in (0,1)$ is a reward discount factor. The goal of the RL algorithm is optimising the joint policy $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ to maximise their expected cumulative rewards: $\max_{\boldsymbol{\pi}\in\boldsymbol{\Pi}} \mathbb{E}_{\boldsymbol{\pi}}[\sum_t \gamma^t r_t]$.

To solve the above Dec-POMDP problem, this paper implements and compares 3 different RL schemes: (i) CTDE, (ii) centralised training and execution (single agent), and (iii) independent agents.

QMIX: QMIX [21] algorithm is implemented in the proposed method, in a CTDE manner. Specifically, QMIX estimates a total Q-value function Q_{tot} as a nonlinear combination of the Q_i-value for each agent $i \in \mathcal{I}$, as long as the monotonic dependence relationship is satisfied: $\frac{\partial Q_{tot}}{\partial Q_i}, \forall i \in [\mathcal{I}]$. $Q_{tot}(\boldsymbol{\tau}, \boldsymbol{a}, s)$ is a function of joint action-observation history $\boldsymbol{\tau}$, joint action $\boldsymbol{a}$ and the state s, while $Q_i(\tau_i, a_i)$ for each agent is a function of agent observed history τ_i and its own action a_i. The update rule of QMIX follows:

$$\min L = \min \sum [Q_{tot}(\boldsymbol{\tau}, \boldsymbol{a}, s) - (r + \gamma \max_{\boldsymbol{a}'} Q_{tot}(\boldsymbol{\tau}', \boldsymbol{a}', s'))]^2.$$

Each LB agent using QMIX algorithm has a stochastic policy.

SAC: For single-agent game, soft actor-critic (SAC) [12] follows the maximum entropy reinforcement learning framework, which optimises the objective $\mathbb{E}[\sum_t \gamma^t r_t + \alpha \mathcal{H}(\pi_\theta)]$ to encourage the entropy $\mathcal{H}(\cdot)$ of the policy π_θ during the learning process. Specifically, the critic Q network is updated using the gradients $\nabla_\phi \mathbb{E}_{s,a}\left[\left(Q_\phi(s,a) - R(s,a) - \gamma \mathbb{E}_{s'}[V_{\tilde{\phi}}(s')]\right)^2\right]$, where $V_{\tilde{\phi}}(s') = \mathbb{E}_{a'}[Q_{\tilde{\phi}}(s',a') - \alpha \log \pi_\theta(a'|s')]$ and $Q_{\tilde{\phi}}$ is the target Q network; the actor policy π_θ is updated using $\nabla_\theta \mathbb{E}_s[\mathbb{E}_{a\sim\pi_\theta}[\alpha \log \pi_\theta(a|s) - Q_\phi(s,a)]]$. Single-agent SAC (S-SAC) method is implemented as the second RL scheme.

Independent Learning. Apart from QMIX, independent learning agents treat the objective in Eq. (3) from an independent view, where the optimal ensemble policy is factorised as the optimization over each individual policy for each LB agent:

$$\pi_i^* = \min_{\pi_i\in\Pi_i, i\in[m]} c(l_i), i \in [m] \tag{17}$$

where $l_i = \{l_{i,j}\}, l_{i,j} = \frac{\sum_{t\in[t_0,t_n)} w_i(t)\alpha_{i,j}(t)}{v_j}, j \in [n]$ for the stochastic case. The objective achieved with Eq. (17) is different from Eq. (3) unless the workloads going to each LB are the same at all time, which is impossible in practice. SAC is used for each independent LB agent, which gives the independent-SAC (I-SAC) method.

3.3 MARL for Multi-Agent Load Balancing

The network load balancing problem belongs to multi-commodity flow problems and is NP-hard, which makes it hard to solve with trivial heuristic solution at a micro-second level speed [23]. In real-world systems, limited observations on system states and changing environments require LB agents to continuously approximate server load states. This section describes the network load balancing problem mathematically as a cooperative Dec-POMDP under realistic constraints. The overview of the MARL framework is depicted in Figure 4. The pseudo-code of MARL framework for network load balancing is shown in Algorithm. 1.

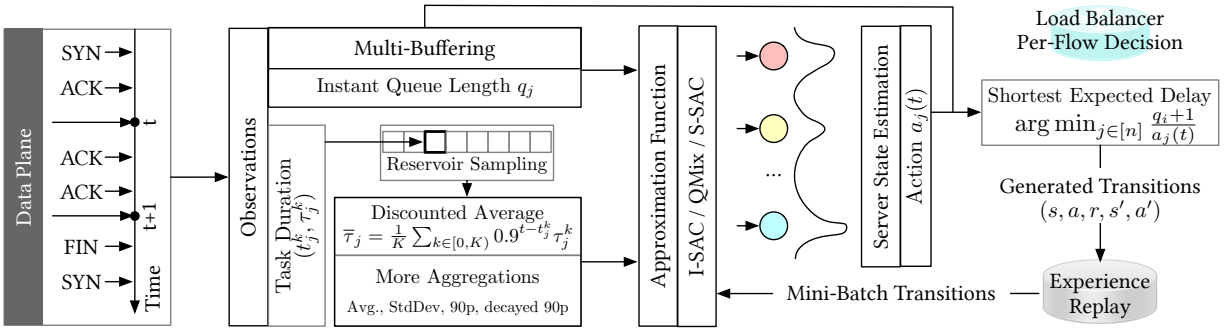


Figure 4: Overview of the proposed MARL framework for network LB: A distributed learning framework with multiple LB agents is implemented to interact with network LB devices and allocate tasks on different servers. Each LB agent contains a replay buffer and can learn using 1 of the 3 different RL algorithms–independent SAC (I-SAC), QMix, or single-agent SAC (S-SAC). The 3 RL algorithms consume the network features (on-going flows and flow duration statistics on each server collected based on the framework proposed in [28]) as well as the actions from last time step, and they generate server load state estimations as the next time-step action for making fair per-flow-level decision based on the shortest expected delay algorithm.

Algorithm 1 MARLLB

Initialise:
  replay buffer $\mathcal{B}$
  learning agents parameterised by $\boldsymbol{\theta} = \{\theta_i\}, \forall i \in [m]$
  reinforcement learning algorithm $\mathcal{P}$
  server processing speed function $v_j, \forall j \in [n]$
  initial observed instant queue length on server k by the i-th LB: $q_{i,k} = 0, \forall i \in [m], k \in [n]$
while not converge **do**
  Reset:
    server load state $X_j(1) \leftarrow 0, \forall j \in [n]$
    observations $\boldsymbol{o}(1)$ for LB agents
  for $t = 1, \ldots, N$ **do**
    for LB agent i **do**
      $w_{i,j}(t) \leftarrow 0, \forall j \in [n]$
      $a_i(t) \leftarrow \{a_{i,j}(t)\}_{j=1}^n = \pi_{\theta_i}(\boldsymbol{o}_i(t))$
    for job $\tilde{w}$ arrived at LB i between timestep $[t, t+1)$ **do**
      LB i assigns $\tilde{w}$ to server $j = \arg\max_{k\in[n]} \frac{q_{i,k}+1}{a_{i,k}(t)}$
      $w_{i,j}(t) \leftarrow w_{i,j}(t) + \tilde{w}$
    for each server j **do**
      $X_j(t+1) \leftarrow max\{X_j(t) + \sum_{i=1}^m w_{i,j}(t) - v_j, 0\}$ ▷ update workload
    Receive reward $r(t)$
    Collect observation $\boldsymbol{o}(t+1)$
    $\mathcal{B} = \mathcal{B} \bigcup (\boldsymbol{o}(t), \{a_i(t)\}, r(t), \boldsymbol{o}(t+1))$ ▷ Update replay buffer
  $\boldsymbol{\theta} \leftarrow \mathcal{P}(\mathcal{B})$ ▷ Update agents with RL/MARL algorithms
return $\boldsymbol{\theta}$

Agent Set. There is a set of homogeneous LB agents $\mathcal{I}$ ($|\mathcal{I}| = m$) distributing workloads among the same set of n application servers. Each agent only distributes and observes over a subset of workloads that arrive at the system.

State and Observation Set. The state set is defined as $\mathcal{S} = \mathcal{W} \times \mathcal{V}$, where $\mathcal{W} = \boldsymbol{w} : \boldsymbol{w} \in (0, \infty)^m$ is a set of incoming network traffic (workloads) to be distributed among servers, and $\mathcal{V} = \boldsymbol{v} : \boldsymbol{v} \in (0, \infty)^n$ is a set of server processing speeds. The observation set $\mathcal{O} = (\boldsymbol{q}, \boldsymbol{\tau}) : \boldsymbol{q}, \boldsymbol{\tau} \in (0, \infty)^n$, where $\boldsymbol{q}$ is a vector of counting numbers, each represents the number of on-going task on each server, and $\boldsymbol{\tau}$ is a vector of statistical evaluations (mean, standard deviation, 90th-percentile, and discounted mean and 90th-percentile over time) of task elapsed time on each server.

Action Set. $\mathcal{A} = \times_i \mathcal{A}_i$ is the action set containing the individual action set $\mathcal{A}_i$ for each agent $i \in \mathcal{I}$. In the discrete action set $\mathcal{A}_i \subset {\mathbb{R}_+}^n$, an action a_i is a vector representing the weights for the current workload to be allocated to n application servers by LB agent i. To make hundreds or thousands of load balancing decisions per second while incorporating RL intelligence, this paper adopts the form of the shortest expected delay (SED)[1] to assign server $\arg\min_{j\in[n]} \frac{q_{i,j}+1}{a_{i,j}(t)}$ to the newly arrived flow, where $q_{i,j}$ is the number of on-going flows on server j observed by LB i, and $a_{i,j}(t)$ is the weight assigned to server j by LB i at timestep t.

Conventionally, LB agents make actions on receipt of each networking requests, assigning a server to new-coming requests. However, it is not possible for RL models to make such decisions under extremely high traffic rates in practice–sub-ms interval between two consecutive decisions. Using the form of the SED, this paper converts the action of LB agents from assigning servers for each request to periodically (250ms) and dynamically estimating server processing speed $a_{i,j}(t)$. By tracking the number of on-going tasks on receipt of every network flow, this allows the proposed LB agents to make load balancing decisions at the pace of task arrival rates and guarantees high-throughput, while adapting to the ever-changing and dynamic server load states and networking environments.

State and Observation Transition. The state transition probability function is defined as $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$, following Markov decision process. More specifically, $\mathcal{T}(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t) = Pr(\boldsymbol{s}_{t+1}|\rho(\boldsymbol{s}_t, \boldsymbol{a}_t))$, where $\boldsymbol{a}_t \in \mathcal{A}$, $\boldsymbol{s}_t, \boldsymbol{s}_{t+1} \in \mathcal{S}$, $\rho(\boldsymbol{s}_t, \boldsymbol{a}_t) \mapsto \boldsymbol{s}_{t+\delta t}$ represents the response of servers given the updated workloads distribution, and $Pr(\boldsymbol{s}_{t+1}|\boldsymbol{s}_{t+\delta t})$ represents the change of incoming traffic rates and server processing speeds. The time interval between two actions is denoted as $\Delta t = 250$ms and $\delta t \ll \Delta t$.

[1] http://www.linuxvirtualserver.org

Algorithm 2 Reservoir sampling

```
K ← reservoir buffer size
p ← probability of gathering samples
buf ← [(0,0),...,(0,0)]                ▷ Size of K
M ← 1/p
for each observed sample v arriving at t do
    randomId ← rand()
    if randomId%M == 0 then
        idx ← randomId%N               ▷ randomly select one index
        buf[idx] ← (t, v)              ▷ register sample in buffer
```

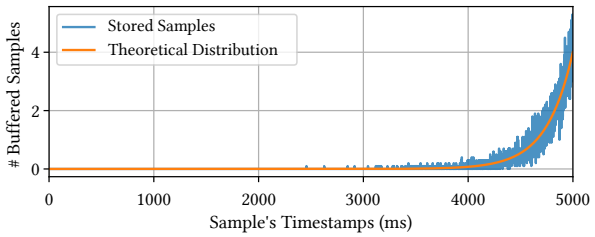


Figure 5: An example of reservoir samples' timestamp distribution with $\lambda = 80$, $p = 0.05$, $K = 10000$, $n \leq 50000$.

Observation Probability Function and Feature Collection. The observation probability function $\Omega : \mathcal{S} \times \mathcal{A} \times O \rightarrow [0, 1]$ describes the measurement errors that can occur when extracting and collecting features and statistics from network packets on LB agents. The counters of on-going network flows $\boldsymbol{q}$ are tracked based on connection states (*e.g.*, identified by TCP SYN, FIN packets). These counters are subject to partial observations in presence of multiple LB agents. To reduce the impact of partial observation, this paper proposes to use the flow duration (elapsed time since the connection establishment) to indicate the server load state. The intuition in behind is that, for the same service provided by the server cluster, heavy-loaded or less powerful servers yield longer flow duration than less-loaded or more powerful servers. Using reservoir sampling (Algorithm 2), an exponentially-distributed number of flow duration samples are collected over time. For a Poisson stream of events with an arrival rate λ, the expectation of the amount of samples that are preserved in buffer after T steps is $E = \lambda p \left(\frac{K-p}{K}\right)^{\lambda T}$, where p is the probability of gathering sample and k is the size of reservoir buffer. An example reservoir samples distribution over time is shown in Figure 5. Flow duration samples can be gainfully used to infer server load state and reduce the impact of partial observations in presence of multiple LB agents.

Reward Function. Since LB agents have limited observations over the actual server load states $\boldsymbol{s}$, the paper uses the flow duration to approximate the sum of queuing delay and task workload over the underlying processing speed of a server. To give more credits to the latest observations, given the set of samples $\{(t_j^k, \tau_j^k) | k \in [K]\}$ of server j, the discounted average of flow elapsed time (an estimatiion of l_j in Eq. (2)) at time t is computed as $\overline{\tau}_j(t) = \frac{1}{K} \sum_{k \in [0,K)} \gamma^{t - t_j^k} \tau_i^k$, where $\gamma = 0.9$ in this paper. Then, based on the Proposition 3.2, the reward function is defined as the fairness index of the exponentially weighted average of $\overline{\boldsymbol{\tau}}$ for all servers:

$$r_{t+1} = \begin{cases} F(\overline{\boldsymbol{\tau}}_t) & \text{if } t = 0 \\ F((1-\gamma)\overline{\boldsymbol{\tau}}_t + \gamma \overline{\boldsymbol{\tau}}_{t+1}) & \text{otherwise.} \end{cases} \tag{18}$$

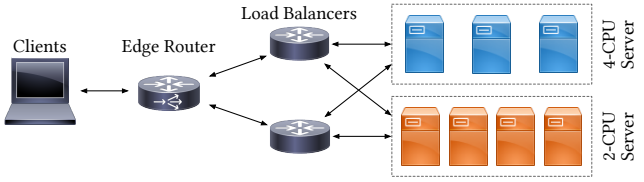


Figure 6: The moderate-scale testbed topology consisting of 1 traffic generator representing clients, an edge router, 2 LBs and 7 application servers with different processing capacities.

Objective Function. The objective is to maximise their expected cumulative rewards: $\max_{\boldsymbol{\pi} \in \boldsymbol{\Pi}} \mathbb{E}_{\boldsymbol{\pi}}[\sum_t \gamma^t r_t]$, through optimising over the parameterised joint policy $\boldsymbol{\pi} \in \boldsymbol{\Pi}$, $\boldsymbol{\pi} = \times_{i \in \mathcal{I}} \pi_i, \pi_i : \tilde{O}_i \times \mathcal{A}_i \rightarrow [0, 1]$ is the stochastic policy for agent i. $\tilde{O}_i$ is a concatenation of historical observations and actions for agent i. This paper uses gated recurrent units (GRU) [6] for both QMIX and SAC agents to handle the sequential history information.

4 IMPLEMENTATION

To evaluate the performance of MARL LB algorithms in different realistic setups, experiments are conducted in a real-world system with real network traces deployed on physical servers. The experimental platform consists of client nodes, an edge router nodes, LB agents, and Apache HTTP servers providing Web services, virtualised as Kernel-based Virtual Machines (KVMs) as in real-world cloud environments, with the same topology as in Figure 6.

4.1 System Platform

The KVMs are virtualised on 4 UCS B200 M4 servers, each with one Intel Xeon E5-2690 v3 processor (12 physical cores and 48 logical cores), interconnected by UCS 6332 16UP fabric. Operating systems are `Ubuntu 18.04.3 LTS` (`GNU/Linux 4.15.0-128-generic x86_64`). The programmable software network stack `VPP v20.05` is used to implement the network layer (the data plane) of LB agents for feature collection and policy updates. The KVMs are deployed on the same layer-2 link, with statically configured routing tables.

4.2 Apache HTTP Servers

Apache HTTP servers share the same VIP address on one end of GRE tunnels with the load balancer on the other end. The Apache servers use `mpm_prefork` module to boost performance. Each server has max 32 worker threads. The TCP backlog is configured as 128. The `tcp_abort_on_overflow` flag is set, so that, in the Linux kernel, when the TCP connection backlog is full, a TCP RST is sent directly to signify the termination of the connection, instead of silently dropping the packet and waiting for a SYN retransmit. This configuration allows measuring flow completion time as the application response delays without taking into account additional TCP SYN retransmission delays.

4.3 24-Hour Wikipedia Replay Trace

In order to evaluate MARL algorithms using real-world environments, this paper creates replicas of Wikipedia servers using the instance of MediaWiki[2] of version 1.30. On each application server

[2]https://www.mediawiki.org/wiki/Download

Table 1: Two testbed configurations

Testbed Configuration n	Moderate Scale	Large Scale
Server Group 1	$4 \times$ 2-CPU	$12 \times$ 4-CPU
Server Group 2	$3 \times$ 4-CPU	$12 \times$ 8-CPU
LB Agents	$2 \times$ 8-CPU	$6 \times$ 8-CPU
Network Trace	Wikipedia Replay	Poisson Traffic
Traffic Rates	[518.8, 796.3]	[391.5, 436.7]
JCT Distribution	Real-world distribution	exp(200ms)

instance, a MySQL server and the `memcached` cache daemon are installed. To populate MySQL databases, this paper uses the *WikiLoader* tool and a copy of the English version of Wikipedia database. The 24-hour trace, for privacy reasons, does not contain any information that exposes user identities.

4.4 PHP for-Loop Trace

Besides the 24-hour Wikipedia replay trace which is based on MySQL, a PHP `for`-loop script is created to study CPU-bound applications. The number of `for`-loop iterations `#iter` directly determines the expected workload for each request. The data size that is transmitted for each task is also proportional to the number of iterations, which follows an exponential distribution. This allows to generate a distribution of flow durations and transmitted bytes that preserve the long-tail characteristic of network traffic [22].

4.5 Network Settings

Two configurations are implemented to study both moderate- and large-scale DC network environments, which is noted in Table 1. In the moderate-scale configuration, network trace samples are extracted and replayed from a real-world 24-hour replay [7], which consists of requests for CPU-intensive Wiki pages[3] and IO-intensive static pages. In the large-scale configuration, a synthesised Poisson traffic of CPU-intensive network flows is applied. The traffic rates of the two network traces in both configurations are selected to consume 80% $\sim$ 95% average provisioned computational resources.

4.6 MARL Settings

To apply the QMIX algorithm, the action space is discretised so that the action set for each LB agent i is $\mathcal{A}_i = \{1.0, 1.2, 1.4, 1.6, 1.8, 2.0\}^n$. QMIX follows a CTDE manner, with each LB having a Q network for specifying the action choice. In order to implement the centralised training of the QMIX algorithm, TCP sockets are maintained among LB agents. Through these TCP sockets, a master LB agent orchestrates the periodic process (every 250ms) of making actions and collecting observations for all LB agents to create synchronised trajectories for training. During each episode, each LB agent collects their locally observed system states and rewards. At the end of each episode, their collected trajectories are merged on the master LB agent for centralised training. A global reward–the mean of rewards on all LB agents–is computed for each time step as.

4.7 Benchmark LB Methods

In experimental evaluations, the QMIX-based MARL (RLB-QMIX) is evaluated and compared against other methods, namely independent SAC (I-SAC) agents, single-agent SAC (S-SAC), and SOTA heuristic methods including Equal-Cost Multi-Path (ECMP) [18], Weighted-Cost Multi-Path (WCMP) [10], active WCMP (AWCMP) [1], Local Shotest Queue (LSQ) [11], and SED. Among these heuristics, WCMP and SED configure server weights proportional to their provisioned CPU power. AWCMP relies on TCP channels to periodically probe server resource utilisation information (number of busy Apache threads) to update server load state estimation and recompute server weights. For I-SAC, each LB node has an independent SAC agent with local observations. There is no communication among LB agents. Each LB agent follows the independent learning procedure as introduced in Section 3.2. For S-SAC, a single LB node is deployed to distribute and balance all the workloads across the server cluster. This single LB agent has global observation on system states and it is trained using a SAC algorithm. The original SAC algorithm works for continuous action spaces only. Modifications are made based on [5] to support discrete action space.

[3]Wiki pages are identifiable by the string `/wiki/index.php/` in URLs.

Table 2: Hyperparameters in MARL-based LB.

Hyperparameter	Moderate-Scale	Large-Scale
Learning rate	1×10^{-3}	3×10^{-4}
Hidden units	128	512
Batch size	12	12
Replay Buffer Size	3000	3000
Episodes	72	72
Episode Length	60s	30s
Step Interval	0.25s	0.25s
Update Iterations	25	25
Target Entropy (SAC)	$-\lvert\mathcal{A}\rvert$	$-\lvert\mathcal{A}\rvert$

4.8 Hyperparameters and Training Details

The hyperparameters for each learning agent (QMIX, I-SAC, S-SAC) and different experimental setups are provided in Table 2. RL-based load balancing methods are trained in both moderate- and large-scale testbed setups for 72 episodes. When replay buffer gathers enough samples (more than 25 episodes of trajectory samples), the LB agents train and update RL models for 25 iterations before running the next episode. QMIX and SAC models use the the same neural network architecture for both Q networks and policy networks–2 fully-connected layers, followed by 1 GRU layer, followed by 2 other fully-connected layers. The hidden dimension for all layers is 128. The activation function of all fully-connected layers is ReLU. Given the total provisioned computational resource, the traffic rates of network traces for training are carefully selected so that the RL models can learn from sensitive cases where workloads should be carefully placed to avoid overloaded less powerful servers. The traffic rates for large-scale setup is lower than the one for moderate-scale setup (see Table 1), because the synthesised Poisson traffic has heavier per-job workloads than the real-world Wikipedia Web trace.

5 EVALUATION AND RESULTS

5.1 Moderate-Scale Testbed Evaluations

As depicted in Figure 7, MARL-based LB methods show improved performance after 600 iterations of updates during training while the single agent RLB-S-SAC struggles to learn. Trained RL-based LB methods are then compared with all the heuristic LB methods

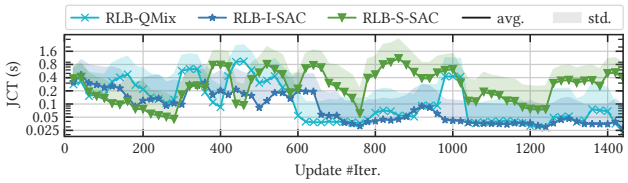


Figure 7: JCT distribution during training using 3 different RL algorithms.

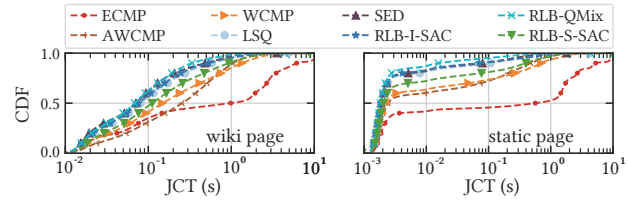


Figure 8: JCT comparison using different load balancing algorithms under different traffic rates (more than 600 queries/s, 5 runs per traffic rate).

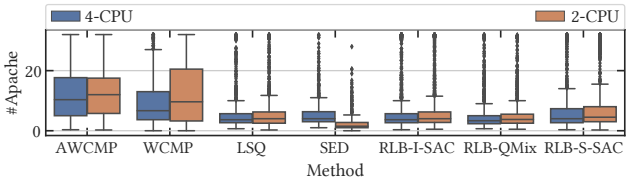


Figure 9: Comparison of number of busy Apache threads on two groups of application servers with different processing capacities.

on 4 unseen network traces, which covers a various range of traffic rates from 518.8 to 796.3 flows/s. As shown in in Tables 3 to 6, RLB-QMIX achieves superior performance over all other methods in most scenarios, including the SOTA heuristic method (SED) and other learning agents (I-SAC and S-SAC). Only when the system is subject to the highest traffic rate (796.3 flows/s), the SED for Wiki pages and the I-SAC agents for static pages win over RLB-QMIX by a slight margin. Figure 8 depicts the overall performance comparisons by aggregating the JCTs over the 4 tested scenarios. RLB-QMIX is 1.44× and 5.11× faster than SED at 90th-percentile, which is an important QoS metric. Figure 9 shows the distribution of the number of busy Apache threads on two groups of servers. With manually configured server weights, SED assigns 2.329× more workloads on more powerful servers while RLB-QMIX maintains the equivalence between the two groups of servers.

5.2 Large-Scale Testbed Evaluations.

As shown in Table 7, although the best performances are achieved with LSQ for 398.5 flows/s traffic rate and SED for 419.3 flows/s traffic rate, MARL methods (QMIX and I-SAC) both have a very close performance to the superior method, which demonstrates a certain level of scalability for these learning-based methods to work in real-world large-scale systems.

Among all the heuristic LB methods, SED has the best performance since it takes both the queue occupation and server processing capacity information into account. However, when there are multiple LB agents, SED will be mis-guided because of the partially observed numbers of on-going flows. In the large-scale setup, as depicted in Figure 10, SED assigns 2.67× and 2.25× more workloads to more powerful servers under 398.5 and 419.3 flows/s traffic rates, while the capacity ratio between the two groups of servers is 2.

Table 3: Comparison under traffic rate 518.8 flows/second.

Method	Traffic Type	
	Wiki	Static
ECMP	529.4 ± 110.5	258.6 ± 94.4
AWCMP	140.9 ± 4.1	27.6 ± 4.0
WCMP	92.6 ± 16.2	12.7 ± 8.1
LSQ	78.7 ± 29.9	8.4 ± 11.0
SED	67.2 ± 5.1	6.7 ± 3.1
RLB-I-SAC	84.3 ± 16.9	9.4 ± 7.4
RLB-QMix	**63.4 ± 3.9**	**3.1 ± 0.1**
RLB-S-SAC	84.2 ± 13.0	14.4 ± 14.4

Table 4: Comparison under traffic rate 690.9 flows/second.

Method	Traffic Type	
	Wiki	Static
ECMP	3178.9 ± 615.9	2835.3 ± 542.8
AWCMP	430.7 ± 154.5	153.3 ± 112.5
WCMP	443.6 ± 268.1	201.0 ± 188.5
LSQ	236.6 ± 164.4	69.3 ± 105.7
SED	189.3 ± 118.4	47.3 ± 48.6
RLB-I-SAC	349.6 ± 397.8	152.9 ± 260.5
RLB-QMix	**166.9 ± 62.3**	**22.0 ± 14.1**
RLB-S-SAC	397.6 ± 258.2	156.3 ± 155.9

Table 5: Comparison under traffic rate 696.5 flows/second.

Method	Traffic Type	
	Wiki	Static
ECMP	2748.5 ± 371.1	2424.6 ± 388.3
AWCMP	348.3 ± 80.3	101.3 ± 48.1
WCMP	530.7 ± 411.7	287.1 ± 355.9
LSQ	207.9 ± 67.6	40.3 ± 41.0
SED	182.6 ± 85.7	40.0 ± 35.1
RLB-I-SAC	146.4 ± 54.7	19.8 ± 16.9
RLB-QMix	**88.0 ± 10.4**	**4.0 ± 0.7**
RLB-S-SAC	169.1 ± 56.4	27.0 ± 24.1

Table 6: Comparison under traffic rate 796.3 flows/second.

Method	Traffic Type	
	Wiki	Static
ECMP	3018.5 ± 837.3	2636.8 ± 859.7
AWCMP	539.1 ± 152.4	203.6 ± 103.2
WCMP	466.8 ± 269.4	192.5 ± 181.5
LSQ	208.8 ± 117.5	50.8 ± 38.0
SED	**150.9 ± 69.2**	22.8 ± 18.5
RLB-I-SAC	155.0 ± 97.0	**17.5 ± 21.9**
RLB-QMix	188.8 ± 104.7	38.2 ± 32.1
RLB-S-SAC	398.9 ± 367.3	163.4 ± 212.3

Table 7: Comparison under different traffic rates (398.5, 419.3 flows/second) with synthesised CPU-intensive Poisson traffic for large-scale system setup.

Method	Traffic Rate (flows/second)	
	398.5	419.3
ECMP	5907.2 ± 550.1	7841.1 ± 484.7
AWCMP	467.7 ± 5.5	595.4 ± 6.7
WCMP	629.9 ± 25.0	1027.5 ± 65.5
LSQ	**332.7 ± 1.6**	420.2 ± 2.0
SED	338.6 ± 0.7	**410.3 ± 2.3**
RLB-I-SAC	344.8 ± 2.0	425.3 ± 1.8
RLB-QMix	340.7 ± 1.5	419.7 ± 2.8
RLB-S-SAC	353.0 ± 3.5	454.6 ± 8.4

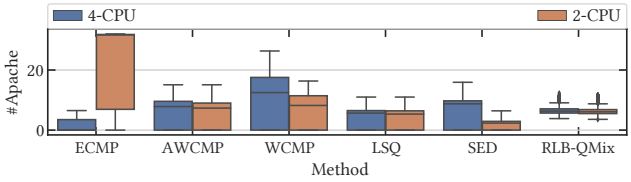


Figure 10: Comparison of the distribution of busy Apache threads on two groups of servers with different processing speeds in the large-scale scenario.

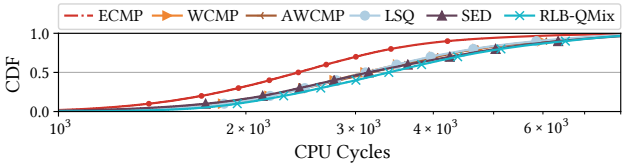


Figure 11: Load balancing decision making latency for each network flow measured under 800 flows/s traffic rate.

This behavior will lead to overloaded powerful servers, which is the reason why LSQ performs better than SED with low traffic rates. Future studies need to be conducted to learn and adapt to use different strategies under different scenarios.

5.3 Decision Making Latency

The decision making latency is compared among all load balancing methods by computing CPU cycles required on the LB node for dispatching every single network flow in the data plane. As depicted in Figure 11, the RLB-QMIX method has 3.6% and 8.6% additional processing latency than SED and LSQ respectively. The average number of CPU cycles required for each flow is 4326.27, which consumes 1.66μs on 2.6GHz-CPU devices. This allows handling high-throughput network traffic (more than 600M packet per second) for real-world systems in production. Therefore, the proposed RL framework for network load balancing problem is able to incorporate intelligence while making high-frequent decisions.

5.4 Centralised Training and Communication Overhead.

QMIX adopts the centralised training scheme, which is challenging to implement in real-world distributed system. This paper synchronises all the LB agents by way of TCP connections among all agents. Only one master agent takes the responsibility of orchestrating the actions of the other agents so that the interactions between agents and the environments are synchronised and the gathered trajectories follows the Dec-POMDP specification. This implementation shows good performance in this paper, especially in the moderate-scale setup. However, in the large-scale setup, RLB-QMIX is outperformed by SED and LSQ with a small margin. One of the reasons is that the increased communication overhead (latency) and delayed actions at the presence of more LB agents. As depicted in Figure 12, in the large-scale setup, the time interval between two consecutive controls (actions) is 1.24× larger than in the moderate-scale setup. This additional communication delay fails to effectuate the latest action in time, which deteriorate performance especially in dynamic environments. Future studies need to be conducted to alleviate this issue.

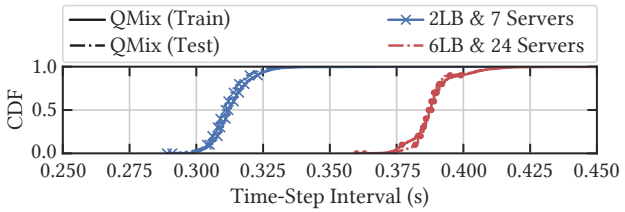


Figure 12: Time-step interval is incremented when using QMIX because of the synchronisation process.

6 CONCLUSIONS AND FUTURE WORK

This paper presents a MARL framework for network load balancing problem, and evaluates different methods for the cooperative game in a real-world system. The learning-based methods including QMIX, independent-SAC and single-agent SAC are tailored for this application and compared with SOTA heuristic methods. Experiments show that in moderate-scale system with different traffic rates and types, the MARL method RLB-QMIX achieve superior performance in most settings. While for large-scale system, learning agents like RLB-QMIX and I-SAC also achieve close performance to the best heuristic methods. This verify the scalability of the proposed MARL methods for real-world large-scale load balancing system. Although promising results are achieved, limitations exist in current work: (1) the QMIX algorithm makes additional structural assumption that the joint-action value is monotonic in individual agent value, which may be restrictive for the load balancing problem; (2) reducing the communication cost among agents during training and decision making latency is important for application in real-world load balancing; (3) there are other types of scoring mechanism other than the linear-product fairness for load balancing system, like maxinising Jain's fairness, which does not suffice as minimising the makespan yet still worths exploring since it has been used to evaluate load balancing performances [7]. Future work includes (1) evaluating more different types of MARL algorithms [31] on the current system as well as a simulation system in diverse and flexible settings, and (2) the contribution of each proposed component (*e.g.*, with or without the GRU), to further improve the performance of MARL solutions.

REFERENCES

[1] Ashkan Aghdai, Cing-Yu Chu, Yang Xu, David H Dai, Jun Xu, and H Jonathan Chao. 2018. Spotlight: Scalable Transport Layer Load Balancing for Data Center Networks. *arXiv preprint arXiv:1806.08455* (2018).

[2] Ashkan Aghdai, Michael I-C Wang, Yang Xu, Charles H-P Wen, and H Jonathan Chao. 2018. In-network Congestion-aware Load Balancing at Transport Layer. *arXiv preprint arXiv:1811.09731* (2018).

[3] Ghada Alsuhli, Hassan A Ismail, Kareem Alansary, Mahmoud Rumman, Mostafa Mohamed, and Karim G Seddik. 2021. Deep reinforcement learning-based CIO and energy control for LTE mobility load balancing. In *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, 1–6.

[4] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. 2018. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 191–205.

[5] Petros Christodoulou. 2019. Soft actor-critic for discrete action settings. *arXiv preprint arXiv:1910.07207* (2019).

[6] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).

[7] Yoann Desmouceaux, Pierre Pfister, Jérôme Tollet, Mark Townsley, and Thomas Clausen. 2018. 6LB: Scalable and Application-Aware Load Balancing with Segment Routing. *IEEE/ACM Transactions on Networking* 26, 2 (2018), 819–834.

[8] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*. Springer, 195–216.

[9] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A fast and reliable software network load balancer. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 523–535.

[10] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer.. In *NSDI*. 523–535.

[11] Guy Goren, Shay Vargaftik, and Yoram Moses. 2020. Distributed Dispatching in the Parallel Server Model. *arXiv:2008.00793 [cs]* (Aug 2020). arXiv: 2008.00793.

[12] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*. PMLR, 1861–1870.

[13] Xiaotian Hao, Weixun Wang, Hangyu Mao, Yaodong Yang, Dong Li, Yan Zheng, Zhen Wang, and Jianye Hao. 2022. API: Boosting Multi-Agent Reinforcement Learning via Agent-Permutation-Invariant Networks. *arXiv preprint arXiv:2203.05285* (2022).

[14] Omar Houidi, Djamal Zeghlache, Victor Perrier, Pham Tran Anh Quang, Nicolas Huin, Jérémie Leguay, and Paolo Medagliani. 2022. Constrained Deep Reinforcement Learning for Smart Load Balancing. In *2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, 207–215.

[15] Adithya Kumar, Iyswarya Narayanan, Timothy Zhu, and Anand Sivasubramaniam. 2020. The Fast and The Frugal: Tail Latency Aware Provisioning for Coping with Load Variations. In *Proceedings of The Web Conference 2020*. 314–326.

[16] Tianle Mai, Haipeng Yao, Zehui Xiong, Song Guo, and Dusit Tao Niyato. 2020. Multi-agent actor-critic reinforcement learning based in-network load balance. In *GLOBECOM 2020-2020 IEEE Global Communications Conference*. IEEE, 1–6.

[17] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2018. Learning scheduling algorithms for data processing clusters. *arXiv preprint arXiv:1810.01963* (2018).

[18] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, 15–28. event-place: Los Angeles, CA, USA.

[19] Amin Mohajer, Maryam Bavaghar, and Hamid Farrokhi. 2020. Mobility-aware load balancing for reliable self-organization networks: Multi-agent deep reinforcement learning. *Reliability Engineering & System Safety* 202 (2020), 107056.

[20] Frans A Oliehoek and Christopher Amato. 2016. *A concise introduction to decentralized POMDPs*. Springer.

[21] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. 2018. Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning. In *International Conference on Machine Learning*. PMLR, 4295–4304.

[22] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, 123–137. https://doi.org/10.1145/2785956.2787472 event-place: London, United Kingdom.

[23] Siddhartha Sen, David Shue, Sunghwan Ihm, and Michael J Freedman. 2013. Scalable, optimal flow routing in datacenters via local link balancing. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. 151–162.

[24] Viswanath Sivakumar, Tim Rocktäschel, Alexander H Miller, Heinrich Küttler, Nantas Nardelli, Mike Rabbat, Joelle Pineau, and Sebastian Riedel. 2019. Mvfst-rl: An asynchronous rl framework for congestion control with delayed actions. *arXiv preprint arXiv:1910.04054* (2019).

[25] Jun Wu, Xin Xu, Pengcheng Zhang, and Chunming Liu. 2011. A novel multi-agent reinforcement learning approach for job scheduling in grid computing. *Future Generation Computer Systems* 27, 5 (2011), 430–439.

[26] Yue Xu, Wenjun Xu, Zhi Wang, Jiaru Lin, and Shuguang Cui. 2019. Load Balancing for Ultra-Dense Networks: A Deep Reinforcement Learning Based Approach. *IEEE Internet of Things Journal* 6, 6 (Dec 2019), 9399–9412. https://doi.org/10.1109/JIOT.2019.2935010 arXiv: 1906.00767.

[27] Yaodong Yang, Jianye Hao, Ben Liao, Kun Shao, Guangyong Chen, Wulong Liu, and Hongyao Tang. 2020. Qatten: A general framework for cooperative multiagent reinforcement learning. *arXiv preprint arXiv:2002.03939* (2020).

[28] Zhiyuan Yao, Yoann Desmouceaux, Juan-Antonio Cordero-Fuertes, Mark Townsley, and Thomas Clausen. 2022. Aquarius-Enable Fast, Scalable, Data-Driven Service Management in the Cloud. *IEEE Transactions on Network and Service Management* (2022).

[29] Zhiyuan Yao, Yoann Desmouceaux, Juan-Antonio Cordero-Fuertes, Mark Townsley, and Thomas Clausen. 2022. HLB: Toward Load-Aware Load Balancing. *IEEE/ACM Transactions on Networking* (2022).

[30] Zhiyuan Yao, Zihan Ding, and Thomas Heide Clausen. 2021. Reinforced Workload Distribution Fairness. *arXiv preprint arXiv:2111.00008* (2021).

[31] Chao Yu, Akash Velu, Eugene Vinitsky, Yu Wang, Alexandre Bayen, and Yi Wu. 2021. The surprising effectiveness of ppo in cooperative, multi-agent games. *arXiv preprint arXiv:2103.01955* (2021).

[32] J. Zhang, F. R. Yu, S. Wang, T. Huang, Z. Liu, and Y. Liu. 2018. Load Balancing in Data Center Networks: A Survey. *IEEE Communications Surveys Tutorials* 20, 3 (2018), 2324–2352. https://doi.org/10.1109/COMST.2018.2816042