Public Review for

Towards Declarative Self-Adapting Buffer Management

Pavel Chuprikov, Sergey Nikolenko, Kirill Kogan

The paper introduces a novel machine learning based approach to buffer management. The idea is to provide a queue management infrastructure that automatically adapts to traffic changes and identifies the policy that is hypothetically best suited for current traffic patterns.

To do so, the authors take a declarative approach, where the queue management is provided with a set of policy instances and metrics to compare their performance. The idea is to adopt a multiarmed bandits model, where the best policy has to be chosen based on incomplete information, while minimizing the total cost of learning. Recognising that different objectives and assumptions lead to different bandit algorithms, the authors discuss and explore the design space while providing an experimental evaluation that validates their recommendations.

The reviewers appreciated the timeliness and the rigorousness of the paper. However, they also found that the narrow set of experiments and the lack of in-depth discussion around the main takeaways made it fall short in being convincing about the proposed solution. The authors were therefore asked for a revision that fixed the aforementioned concerns. Together with their revised version, the authors provide a GitHub repository that allows for the reproducibility of their result through the NS-2 simulator.

Public review written by
Gianni Antichi
Queen Mary University of London, UK

Towards Declarative Self-Adapting Buffer Management

Pavel Chuprikov
IMDEA Networks Institute,
Universitá della Svizzera italiana
pavel.chuprikov@usi.ch

Sergey Nikolenko
National Research University
Higher School of Economics,
Steklov Institute of Mathematics at
St. Petersburg
sergey@logic.pdmi.ras.ru

Kirill Kogan
Ariel University
kirillk@ariel.ac.il

ABSTRACT

Buffering architectures and policies for their efficient management are one of the core ingredients of network architecture. However, despite strong incentives to experiment with and deploy new policies, opportunities for changing or automatically choosing anything beyond a few parameters in a predefined set of behaviors still remain very limited. We introduce a novel buffer management framework based on machine learning approaches which automatically adapts to traffic conditions changing over time and requires only limited knowledge from network operators about the dynamics and optimality of desired behaviors. We validate and compare various design options with a comprehensive evaluation study.

1 INTRODUCTION AND MOTIVATION

The growing complexity of network management has been the subject of many debates. There are two major factors impacting the complexity of network operations: size and structure of a manageable state and how often it changes. Networks should be as autonomous as possible and adjust their operation to changing traffic patterns, allowing to better exploit network infrastructure, ideally without manual intervention.

In this work, we study the design principles of a self-adjusting queueing module (QM), which is an essential building block in every network element. Traditional network management allows only to deploy a fixed set of buffer management policies optimizing predefined objectives and incorporating specific traffic parameters. Finding the "perfect" policy settings is a complicated task that requires high qualifications of network operators. Moreover, well-chosen parameters reflect only the current understanding about traffic patterns: usually, there is no *closed loop* between network operators and the QM, and as a result the performance of even well-chosen policy parameters will deteriorate when traffic patterns change over time.

Instead of implementing a complex closed loop, we propose a declarative approach, where the QM is provided with a set of "implementable" policy instances and metrics to compare their performance. Each policy instance is a "black box", and network operators can supply to the QM multiple "reasonable" policy instances. The QM infrastructure exploits machine learning techniques and deploys currently best suitable policy candidate. This approach is based on the multiarmed bandits model, where the best policy (arm) has to be chosen based on incomplete information, minimizing the total cost of learning (regret). However, to make multiarmed bandits useful in real settings, one has to choose which model is best suited: different objectives and assumptions lead to different bandit algorithms. One also has to solve problems related to discretization and representation of performance indicators, e.g., choose the measurement interval; in this work, we discuss these issues and show an experimental evaluation that validates our recommendations. Moreover, various programming abstractions [32, 36, 38] have recently defined atomic primitives that can be used to compose buffer management policies. In this case, a network operator can supply to QM instances of such atomic primitives (instead of policy instances) and the QM infrastructure will build a set of candidate policies by making the QM even less dependent on network operator skills in the design of new policies.

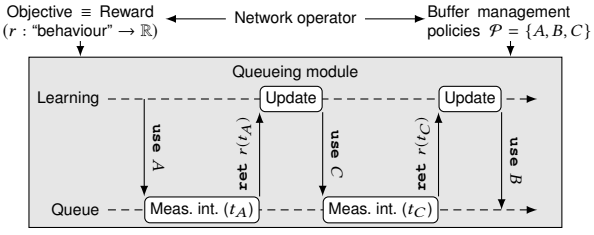

Figure 1: System structure: the network operator defines an objective and a set of policies. The learning algorithm interacts with the queue to identify the best policy.

2 PROBLEM SETTING

We assume that a network operator configures the QM in a *declarative* way, by specifying a set of *candidate policies* $\mathcal{P}$ and a *reward function* r, $0 \le r(t) \le 1$, that expresses a desired objective (more is better) and allows to evaluate QM performance over any time interval t; sample candidate policies might include *RED* [17] or *CoDel* [23] with various parameters, and r could be, for instance, the network power [44], average packet delay, or utilization over respective time interval. Each policy instance is an implemented algorithm with fixed set of parameters; the QM knows nothing about their internal operation except that they halt and can run at line rate.

Let $\mathcal{T}$ denote the time period of QM operation. The currently prevalent approach is to use a fixed policy configured by the network administrator for the entire $\mathcal{T}$. This static approach has limitations. First, the administrator must have a deep understanding of traffic patterns to make the right choice; even

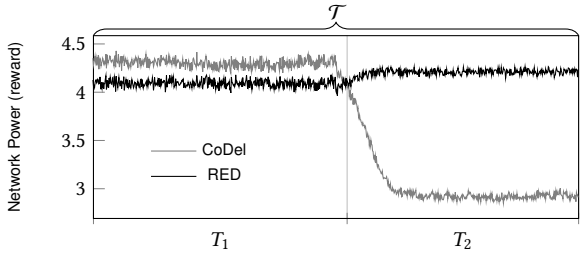

Figure 2: Sample behavior of two policies; shaded areas show the loss incurred by using a single policy over $\mathcal{T}$.

choosing the best policy by data collected about the incoming traffic is a very hard problem in general, let alone trying to predict future traffic patterns. Even more importantly, the network traffic patterns are likely to vary during $\mathcal{T}$, and as a result the performance of each individual policy will vary too. As an example, Fig. 2 (which is a real evaluation result, an excerpt from Fig. 4) shows a time interval $\mathcal{T}$ divided into T_1 and T_2, and each of the two policies *CoDel* [23] and *RED* [17] yields the best reward during its own interval. Thus, in order to achieve optimal performance we should run different policies on different subintervals T_i; the shaded area on Fig. 2 shows losses incurred by using a single policy throughout $\mathcal{T}$.

Our main goal is to design the queue management infrastructure in such a way that would *automatically* adapt to traffic changes and identify the policy that is hypothetically best suited for current traffic patterns. The optimization objective is to choose such a sequence of policies over $\mathcal{T}$ that would maximize the overall performance $r(\mathcal{T})$. The QM infrastructure knows neither what kind of traffic is expected during $\mathcal{T}$ nor the performance of buffer management policies on that traffic.

We measure the performance of each policy over smaller *measurement intervals* and then choose the policies to use on future intervals. We denote the reward of a policy P active on a measurement interval t by $r_P(t)$ and divide $\mathcal{T}$ into *stationary* subintervals $T_1, \ldots, T_q$ on which the best policy does not change, i.e., for every j a single policy $P_j^* \in \mathcal{P}$ shows optimal expected performance over T_k; e.g., Fig. 2 shows three subintervals with optimal policies P_A, P_C, and P_B respectively. Optimal $r(\mathcal{T})$ can be achieved by running P_j^* during each T_j.

Unfortunately, the QM cannot switch policies perfectly for two reasons: it know neither when each T_j begins nor what P_j^* is. The main goal of this work is to present approaches to approximate this desirable behavior by providing a solution to the *stationary optimization* problem: identifying and running the best policy P^* over a stationary interval T. We leave *non-stationarity detection*, i.e., identifying a switch between two stationary intervals, for further study.

Note that the reward value over a given time interval depends not only on the policy's behaviour but also, e.g., on how it has been discretized. This raises a separate question of how to choose the duration of measurement intervals and implies a separate high-level infrastructural component that tests the necessary properties and chooses hyperparameters for the optimization process. The closed-loop control makes the choice of the measurement interval especially difficult: end hosts may need some time to pick up a change in buffer management policy and adjust they behavior. If not accounted for, these reaction delays may cause the reward to include irrelevant behavior on the interval prefix, if the interval is short, represent just noise. There are different countermeasures: use a larger interval so that the true policy behavior dominates, skip the prefix from the reward, and, a more advanced is to make the interval dynamic, i.e., start the measurement only when the reward has stabilized. In Section 4 we show that even the simplest approach helps.

3 STATIONARY OPTIMIZATION

The principal objective of stationary optimization is to run the best policy P^* for the largest possible fraction of T. In order to identify P^*, some part of T has to be spent on exploration through trial and error. There are two general approaches:

(a) *evaluate then exploit*: dedicate an *exploration interval* Δ in the beginning of T for the evaluation of different policies, and then use the resulting best policy for the rest of T;
(b) *continuous exploration*: combine evaluation with exploitation of the currently best policy; this approach keeps incurring exploration costs but can converge on the best policy with probability 1 and can handle dynamic policy rewards.

To apply known algorithms with proven results, we need to make assumptions regarding the properties of the process. First, we review several different cases that may lead to completely different problems. The simplest case occurs if we are able to find the best policy after trying each policy once, i.e., if the best policy is *always* better than any other, on every time interval.

ASSUMPTION 1 (DISJ(Δ)). *For a set $\Delta = t_1 \sqcup \ldots \sqcup t_n$ of measurement intervals and any $P \in \mathcal{P}$, $P \neq P^*$, we have $r_{P^*}^{\min} > r_P^{\max}$, where $r_P^{\min} = \min_i r_P(t_i)$, $r_P^{\max} = \max_i r_P(t_i)$, and P^* is the best policy as defined above.*

The DISJ(Δ) assumption, illustrated on Fig. 3a, allows for a very simple yet perfectly accurate LOCALGREEDY policy identification algorithm: try every policy once and choose the one that has performed best. It requires only $|\mathcal{P}|$ measurement intervals, so it is essentially independent of the duration of Δ. Note that LOCALGREEDY may rank suboptimal policies (e.g., P_A and P_B in Fig. 3) differently depending on the way policies are ordered, but the best one is always identified correctly. However, DISJ(Δ) is as unrealistic as it is powerful. In practice, incoming traffic is always random, and numerous unknown factors, which may differ from interval to interval, can affect the arrivals. Thus, random fluctuations can easily lead to even the best overall policy losing on a specific time interval.

One way to relax DISJ(Δ) is to assume some (unknown) probability distribution over possible traffic behaviors. The stationarity assumption here would be that the expected reward for every policy remains constant, and rewards on different time intervals are independent random variables.

ASSUMPTION 2 (AVGEQ(Δ)). *For a set $\Delta = t_1 \sqcup \ldots \sqcup t_n$ of measurement intervals, any $P \in \mathcal{P}$, and any $1 \leq i < j \leq n$, $\mathbb{E}[r_P(t_i)] = \mathbb{E}[r_P(t_j)]$, and $r(t_i)$ and $r(t_j)$ are independent.*

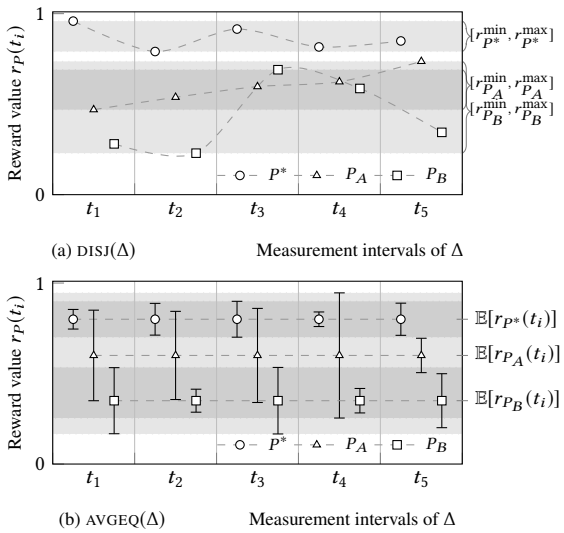

Figure 3: Possible reward values at different subintervals of Δ satisfying different assumptions.

The objective now becomes $\mathbb{E}[\sum_i r_P(t_i)] = n\mathbb{E}[r_P(t_i)]$, i.e., it is also stochastic and there is a certain probability of error. We minimize either the probability of error (if we need to choose a single best policy) or the total *regret*, i.e., difference between its expected reward and the expected reward of the clairvoyant algorithm that always chooses the best policy. Independence is also an important part of AVGEQ(Δ): without it the setting would be essentially equivalent to making no assumptions at all. In particular, AVGEQ(Δ) (illustrated on Fig. 3b) means that there are no long-lasting (spanning several measurement intervals) effects that substantially change the reward distributions.

Still, in practice long-term dependencies often do exist, so one must look for an even more relaxed (but still hopefully tractable) setting. The next setting assumes only that the more often we try a policy, the closer the average reward gets to its true performance; this is a strict relaxation of AVGEQ(Δ).

ASSUMPTION 3 (LIMIT(Δ)). *For a set $\Delta = t_1 \sqcup \ldots \sqcup t_n$ of measurement intervals, any $P \in \mathcal{P}$, and any sequence $1 \le i_1 < \ldots < i_k \le n$, there exists the limit* $\lim_{n,k\to\infty} \frac{1}{k} \sum_j r_P(t_{i_j})$.

Finally, we introduce the most general setting ADV(T), where the inputs can be completely arbitrary and even adversarial.

ASSUMPTION 4 (ADV(Δ)). *$r_P(t_i)$ are arbitrary.*

Note that in reality these assumptions pertain to properties of the traffic distribution. Let us now see how these assumptions map to the two basic approaches of stationary optimization.

Evaluate then exploit. In this setting, we assume that a special evaluation period Δ, a prefix of T, is reserved for pure exploration; this problem is known in machine learning as *best arm identification*. The main assumption here is that Δ is representative of the entire T, and the rewards of policies will not change over time, or at least the best policy will stay the same. The reward obtained during Δ is irrelevant for the learning algorithm's objective, but it does lead to a loss in the total regret, so the length of Δ represents a fundamental

Assumption	Constraints	Method	Accuracy
FINDBEST			
DISJ(Δ)	$r^{\min}_{P^*} > r^{\max}_{P}$	LG	exact
AVGEQ(Δ)	$\mathbb{E}[r_P(t_i)] = \mathbb{E}[r_P(t_j)]$	SR	$1 - e^{O(n)}$
LIMIT(Δ)	$\exists \lim_{n,k\to\infty} \frac{1}{k} \sum_{j=1}^{k} r_P(t_{i_j})$	SH	> RR
MAXREWARD			
DISJ(T)	$r^{\min}_{P^*} > r^{\max}_{P}$	LG	0
AVGEQ(T)	$\mathbb{E}[r_P(t_i)] = \mathbb{E}[r_P(t_j)]$	UCB1	$O(\log n)$
ADV(T)	—	EXP3.1	$O(\sqrt{n})$

LG = LOCALGREEDY, SR = SUCCESSIVEREJECTS,
SH = SUCCESSIVEHALVING, RR = ROUNDROBIN

Table 1: FINDBEST and MAXREWARD summary.

trade-off between exploration and exploitation, i.e., between the accuracy of identification and the fraction of time when suboptimal policies are active. We assume that Δ contains n reward measurement intervals $t_1, t_2, \ldots, t_n$.

PROBLEM 1 (FINDBEST). *For $\Delta = t_1 \sqcup \ldots \sqcup t_n$, find a policy $P \in \mathcal{P}$ that maximizes $\sum_{i=1}^{n} r_P(t_i)$.*

We now see a natural tradeoff between assumptions and performance: we would like to relax the restrictions as much as possible, but more general methods might perform worse. Under DISJ(Δ) there is no problem: LOCALGREEDY is perfect.

Under AVGEQ(Δ), there is a natural way to reduce the variance of estimates: average over several reward measurements. A straightforward UNIFORM strategy tries every policy $\lfloor n/|\mathcal{P}| \rfloor$ times and then chooses the policy with the highest average reward. It is known that under AVGEQ(Δ) the error probability of UNIFORM decreases exponentially with n [10]. However, intuitively, if a policy has already performed very poorly several times, the rest of Δ is probably better spent distinguishing between other policies. In [4], this intuition was formalized as the SUCCESSIVEREJECTS algorithm that has a provably better performance than UNIFORM while still being parameter-free.

Under LIMIT(Δ), one cannot guarantee that by the end of any given Δ the correct policy is chosen with high probability: it is always possible that n was not large enough. Still, the SUCCESSIVEHALVING algorithm [21] needs provably fewer measurements than the round-robin algorithm under LIMIT(Δ). Finally, under ADV(Δ) there is nothing we can do: the exploration part of the interval has no bearing on what happens next. Table 1 contains a short summary of the results.

Continuous exploration. Making decisions based on a prefix Δ is a bad idea if Δ does not accurately represent the entire period T. At the end of the day, best arm identification is merely an improved version of UNIFORM, and it falls prey to the same issues: large error probability under AVGEQ(Δ) or insufficient rate of convergence under LIMIT(Δ). To address this, we allow to evaluate any policy at any moment, getting the classical *exploration vs. exploitation* tradeoff between gathering new information and using the currently best hypothesis.

PROBLEM 2 (MAXREWARD). *For a set $T = t_1 \sqcup \ldots \sqcup t_n$ of measurement intervals, assign a policy from $\mathcal{P}$ to each interval so that $\mathbb{E}\left[\sum_{i=1}^{n} r(t_i)\right]$ is maximized.*

In MAXREWARD, the exploration/exploitation trade-off is implemented by policy selection instead of choosing Δ. This is a more flexible approach: decisions can be made online,

on the basis of already collected data. In what follows we extend the same assumptions from Δ to the entire interval T. DISJ(T) is again trivial: LOCALGREEDY is still perfect. Under AVGEQ(T), average rewards of each policy are assumed to be equal among all measurement intervals $t_1, \ldots, t_n$. A policy's performance can still be evaluated by averaging over multiple reward measurements. This is the default setting for the multi-armed bandits problem, and classical results in this field apply [42]. For example, the ϵ-GREEDY algorithm (run the currently best policy with probability $(1-\epsilon)$ and a random policy with probability ϵ) yields expected regret $O(\epsilon n)$, linear in $n = |T|$. Better algorithms are motivated by "optimism under uncertainty": assume the best if little is known. The UCB1 algorithm [6] has expected regret $O(\log n)$; on t_i, UCB1 chooses $P = \arg\max_P \left[\hat{\mu}_{P,i} + \sqrt{2\log i / n_{P,i}}\right]$, where $\hat{\mu}_{P,i}$ is the current estimate of μ_P (empirical average reward), and $n_{P,i}$ is the number of times policy P has been chosen. Its variance-aware version UCB-V [5] uses priorities $\hat{\mu}_{P,i} + \sqrt{\frac{2V_{i,n_{P,i}}E_{n_{P,i},i}}{n_{P,i}}} + c\frac{3bE_{n_{P,i},i}}{n_{P,i}}$, where $V_{i,n_{P,i}}$ is the empirical variance estimate for arm i, $E_{n_{P,i},i}$ is the exploration function, usually $E_{n_{P,i},i} = \zeta \log i$, and c, ζ are constants. If there are no restrictions at all, ADV(T), an optimal solution to MAXREWARD may have to use several different policies during T, which was not the case for DISJ(T) or AVGEQ(T). There can be no general performance guarantees, but since we know that T is stationary we can restrict ourselves to comparing only against single-policy solutions, i.e., with $\max_P \sum_{i=1}^{n} r_P(t_i)$. Unlike FINDBEST, now it becomes possible to outplay even an adversary (at least on average) with a randomized algorithm. One possible such algorithm in this case is called EXP3.1 [7], shown to guarantee expected regret $O(\sqrt{n})$ [7], again sublinear, so per-interval loss still tends to zero, although slower than for UCB1 under AVGEQ(T). Note that learning algorithms should be fast since they should not slow down network elements; fortunately, all algorithms above have very low complexity: e.g., UCB1 or UCB-V need to keep a priority queue of policies and store a few numbers per policy; the number of different policies is low in practice and does not depend on the network load or traffic.

Table 1 presents a brief summary of the results mentioned above. It shows the natural tradeoff between the strength of assumptions and guaranteed results: stronger assumptions lead to weaker guarantees. Theoretical guarantees, however, may be overly pessimistic, and in realistic settings the learning cost (regret) might be significantly lower. In the next section, we proceed to a practical evaluation of our proposed framework.

4 EVALUATION

Evaluation setup. We used the NS2 simulator [1] with the dumbbell topology similar to [23]. Two types of traffic are routed through a single bottleneck link with 3 Mbit/s bandwidth: batch transfer TCP connections that each transmit 10^8 B of data and web traffic (TCP) generated using the model [20] built into NS2. Access delay is set to 20 ms, and bottleneck delay to 10 ms, totalling to 100 ms RTT. While 3 Mbit/s bandwidth is quite low by modern standards, we wanted to use proven parameters of CoDel, which was originally evaluated at 3 Mbit/s [23]. The input is defined by the number of batch FTP transfers N_{ftp}, request rate of web connections R_{web}, and start times of FTP and web transfers, T_{ftp} and T_{web}. Start times for individual FTP transfers are chosen randomly between T_{ftp} and T_{ftp} + 100 s. The code is available at *GitHub* [12].

Policies. We use the SFQCoDel (Stochastic Fair Queueing with CoDel) policy [23] implemented in NS2 with two parameters: interval int and target tgt; we set tgt = 5 ms and vary int. The second policy is RED [17] with four main parameters: min and max queue thresholds $\min_{\text{th}}$ and $\max_{\text{th}}$, max drop probability p_{drop}, and queue length weighting coefficient w_q. We have tested several configurations for both CoDel and RED.

As the reward we used *network power* (see also [44]) defined as $\frac{1}{|F|}\sum_{f\in F}\log\left(\text{throughput}_f/\text{delay}_f\right)$, combining *throughput*—the total number of bytes sent by f divided by the time it takes to complete f and *delay*—the average RTT of successfully transmitted packets. We approximate this measure over a measurement interval δ by considering queueing delay instead of RTT and per-interval flow throughput instead of true throughput. We compared several policies with learning based on this reward: greedy, uniform exploration, UCB, and SR (successive rejects); see Section 3. The learning process begins at time T_{learn} to give warm-up time to TCP flows.

Experimental results. Selected results of our experimental study are shown in Fig. 4. Fig. 4a compares explore-then-exploit algorithms, showing that SUCCESSIVEREJECTS works best. On Fig. 4b and 4c, mean rewards for the policies are very different, and simple explore-then-exploit strategies behave better since they decide faster. Fig. 4d demonstrates an important case where due to changing traffic patterns a wrong policy starts out as the best. We see that UCB-V rewards do not fall all the way down to the level of the now-worst RED variant; moreover, UCB-V quickly starts to converge to the new leader. Another important point is that UCB-V remembers the previous behavior of suboptimal policies and hence does not fall back to full exploration but rather prefers the next-best policy. As a result, the worst policy is unlikely to be used until all other show an even worse behavior. Fig. 4e shows the case of similar mean rewards, so there is a relatively large error probability for explore-then-exploit policies, while continuous exploration gradually reduces the probability of making the wrong choice and overcomes explore-then-exploit. Fig. 4f shows the case when the difference between RED and CoDel grows rapidly due to a large number of FTP flows appearing 150s into the simulation; naturally, in this case UCB-V can adapt better. Fig. 4g shows the case of many policies, when there is little time for FindBest approaches to identify the best option while multi-armed bandits still converge to the best policy. Fig. 4h shows the case of 100 Mbit/s bandwidth, where there is only a small difference between approaches and due to reward bias (explained next) UCB follows a suboptimal policy.

Measurement interval. In Section 2, we saw that under closed-loop transport protocol, the reward over an interval may not reflect actual policy behavior. Indeed, Fig. 5 shows that for a too small measurement interval the reward observed by learning may be quite different from the reward obtained by the policy

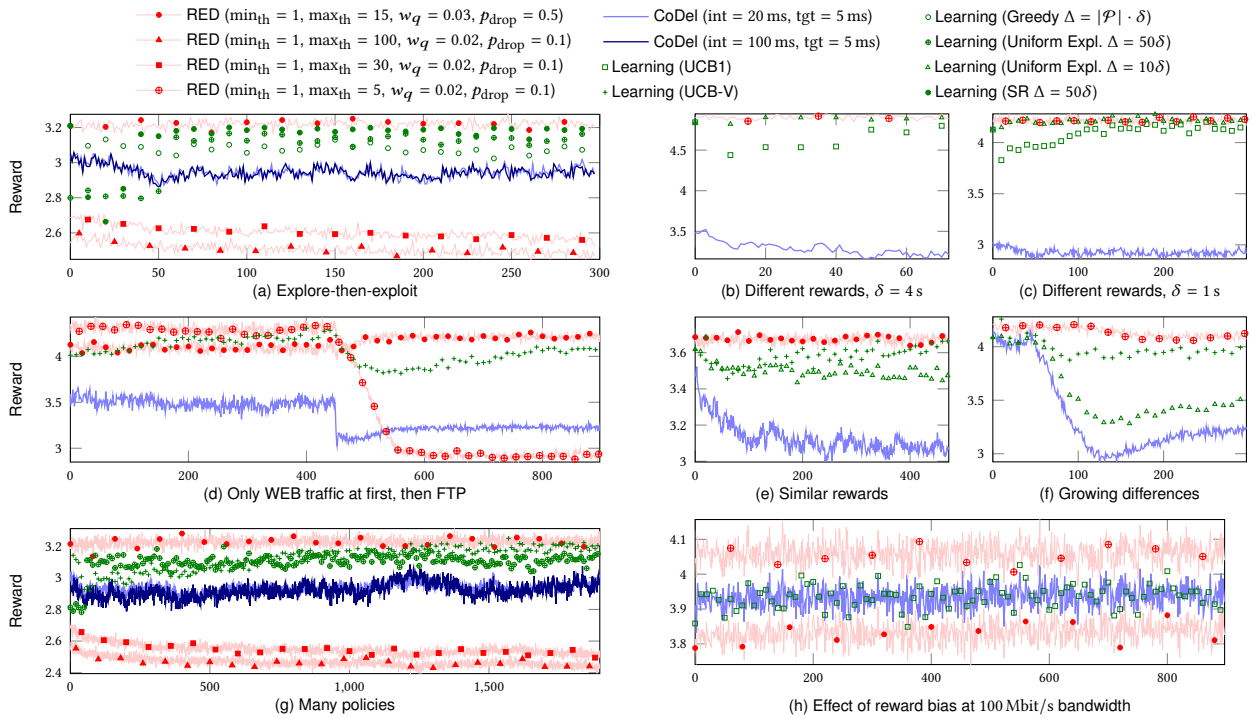

Figure 4: Evaluation results. Parameters: $T_{\text{learn}} = 100\delta$**,** $T_{\text{web}} = 0$ s**;** R_{web} = **(a-c,e-h)** 30**, (d)** 40**;** N_{ftp} = **(a,d)** 14**, (b,c,e-h)** 15**;** T_{ftp} = **(a-c,f-h)** 0 s**, (d)** 150 s**, (e)** 550 s**;** $|T|$ = **(a-c,f)** 400 s**, (d,g,h)** 1000 s**, (e)** 2000 s**;** δ = **(a,c-h)** 1 s**, (b)** 4 s**.**

in isolation; e.g., the first column shows how network power under learning is biased towards higher values due to a better policy affecting traffic. Note that learning algorithms see the same *relative order* of policies since every policy experiences roughly the same "pulling effect", and the choice of the next policy does not depend on the current policy. The difference between policies, however, may diminish, as we see in the 100 Mbit/s columns of Fig. 5, increasing convergence time, so the choice of the right measurement interval remains important.

5 WHITE-BOX AND DISCUSSION

So far learning models took as input a *reward function* and a set of *candidate policies*, which should contain a policy that performs well for every "representative" traffic pattern. The search space here contains essentially all possible policies, and usually there is little prior knowledge of which policy is good for a given objective (complex theoretical analysis may be required to obtain such knowledge). This makes the network operator's choice very hard and likely to miss good policies for specific arrival patterns. In this section we propose a method to automatically compose policies from a set of characteristics that can significantly relax requirements for operator qualification.

We begin with the building blocks. A single-queue buffer management policy is defined by three elements [32]: (1) *admission priority* `admPrio` : $\mathbb{P}\times\mathbb{P} \to \mathbb{B}$ defining the order of packets in admission and when dropping them on congestion; (2) *processing priority* `procPrio` : $\mathbb{P}\times\mathbb{P} \to \mathbb{B}$ defining the order of processing the packets; (3) *congestion condition* `isCongested` : $\mathbb{B}$,

Algorithm 1 POLICY(`admPrio`, `procPrio`, `isCongested`)

```
Let B be a set of packets in a buffer
on arrival of a packet p do
    if isCongested() then
        p_worst ← (B ∪ {p}).maxBy(admPrio)
        B ← (B ∪ {p}) \ {p_worst}
    else if not B.isFull() then
        B ← B ∪ {p}
on choose next packet for processing do
    if B = ∅ then
        return ∅
    p ← B.maxBy(procPrio)
    return {p}
```

where $\mathbb{P}$ is a set of all possible packets, and $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$. Once `admPrio`, `procPrio`, and `isCongested` are known the policy operates as shown in Algorithm 1. This policy structure allows to replace the set of policies $\mathcal{P}$ (which are difficult to design) with three "simpler" sets: a set of admission priority comparators $\mathcal{P}_a$, a set of processing priority comparators $\mathcal{P}_p$, and a set of congestion conditions $\mathcal{P}_c$. The set of policies is now $\mathcal{P} = \{\text{POLICY}(a,p,c) : a \in \mathcal{P}_a, p \in \mathcal{P}_p, c \in \mathcal{P}_c\}$. Note that only the congestion condition may have complex structure; admission and processing priorities are pure comparators.

To choose a priority one has to choose a packet characteristic and a sign (are higher values better or worse). We believe that it is much easier to decide whether a certain characteristic is relevant to the objective than to understand whether a given policy optimizes that same objective. $\mathcal{P}_a$ and $\mathcal{P}_p$ can even be generated exhaustively from all available characteristics.

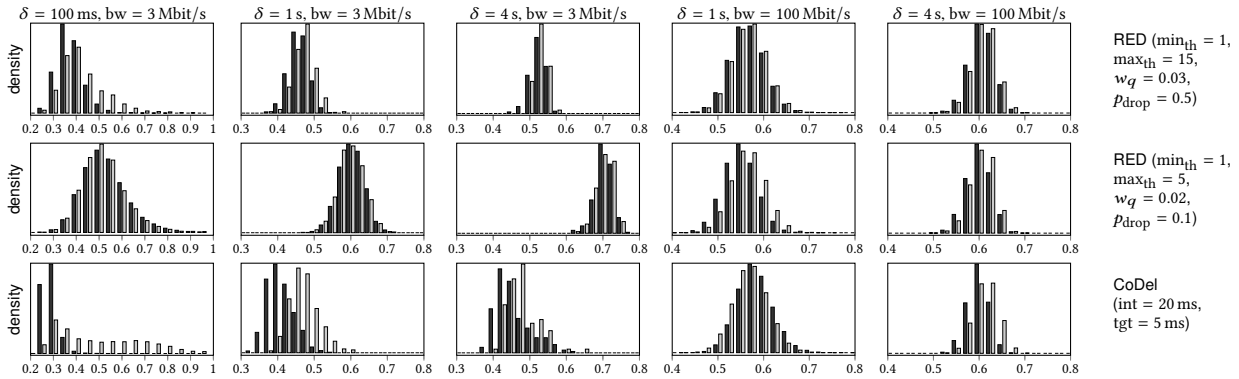

Figure 5: Rewards distribution: CoDel and RED under various conditions. Dark bars show rewards for an isolated policy, while lighter ones—rewards as seen by learning. Common parameters: $R_{\text{web}} = 30$, $N_{\text{ftp}} = 15$, $T_{\text{web}} = T_{\text{ftp}} = 0$ s, $|T| = 1000$ s.

The choice of congestion conditions $\mathcal{P}_c$ cannot be easily reduced to packet characteristics, and in general all buffered packets have to be input to `isCongested`, perhaps together with state information. But here we can use *push-outs*, allowing to drop admitted packets on congestion, and let `isCongested`() be equal to B.`isFull`(). Intuitively, this greedy approach should not lose much to a custom `isCongested` since we can ignore p_{worst} until the buffer is full and throw it away only then.

After replacing the set of candidate policies with two sets of comparators, we can take another simplification step. We expect `procPrio` to prefer better packets and, intuitively, we want to keep better packets on admission too, so we can let `admPrio`(p_1, p_2) = **`not`** `procPrio`(p_1, p_2) (this might fail either when non-tail drops are prohibited or when admission and processing have different sets of packet characteristics).

The next step, which we suggest for future research, is to learn queue management policies in a fully automated way, without network operators, given only objective and packet characteristics available to generated policies. Based on this, the QM infrastructure could automatically produce candidate policies and then exploit the layer designed for switching among "black-box" policies. We believe that the "black-box" approach we develop here can also in the future be extended to learning policies directly in a "white-box" manner.

6 RELATED WORK

There are two main components of congestion control: control at the end hosts and buffer management. Prior work has concentrated on using online learning and reinforcement learning for congestion control at the end hosts [14, 22, 45]; in this work, we introduce machine learning techniques for buffer management. In other related work, we note the research into flexible *adaptable* networks [19, 24] and highlight the work [8] that proposes an approach similar to ours. The difference is that the work [8] leverages data collected by solving computationally hard problems in networking and uses it to reduce the search space for the algorithms, while we use performance to choose among several policies. Specification of buffer management policies in various settings with provable guarantees is a complex task [2, 3, 11, 13, 15, 25–31]. Frenetic [18], Pyretic [35], and Maple [43] among others have proposed abstractions to express management policies in packet networks, focusing on service abstractions based on flexible *classifiers*, but not addressing management of buffering architectures. Other approaches [16, 40] allow only for a predefined set of buffer management parameters, limiting expressivity. Works such as [9, 33, 39] abstract representations of the southbound API (e.g., OpenFlow) in the data plane, while languages such as P4 [9] are very successful in representing packet classifiers but less suited to express buffer management policies. [37, 38] expressed policies by one priority and one calendar queue, leaving the language specification as future work; Mittal et al. attempt to build a universal packet scheduling scheme [34]. Recently, [36, 41] explore the ways to improve efficiency of the QM. In difference from the current prior art that is mostly dealing with efficient APIs to express buffer management policies, this work adds an additional dimension allowing to choose on demand preferable candidate policies for arrival patterns.

7 CONCLUSION

In this work, we have shown the main design principles of a declarative self-adaptive QM infrastructure with black-box approach allowing the QM to reduce dependency on the design skills of network operators. In addition we discuss about directions to exploit the developed QM infrastructure for the white-box approach, where a network operator provides only a desired objective and a set of potentially involved characteristics to build candidate policies automatically reducing dependency on qualification skills of network operators even more. This is only the first step towards a more ambitious goal of declarative autonomous data planes.

ACKNOWLEDGMENTS

The work of Sergey Nikolenko shown in Sections 1, 2, and 3 was supported by the Russian Science Foundation grant no. 17-11-01276. The work of Kirill Kogan was supported in part by the Ariel Cyber Innovation Center in cooperation with the Israel National Cyber Directorate in the Prime Minister's Office and in part by the Data Science and Artificial Intelligence Research Center at Ariel University.

REFERENCES

[1] [n.d.]. The network simulator — ns2. http://www.isi.edu/nsnam/ns/.

[2] S. Albers, M. Charikar, and M. Mitzenmacher. 2001. On delayed information and action in online algorithms. *Information and Computation* 170, 2 (2001), 135–152. https://doi.org/10.1109/SFCS.1998.743430

[3] N. Andelman, Y. Mansour, and A. Zhu. 2003. Competitive queueing policies for QoS switches. In *SODA*. 761–770.

[4] J. Audibert and S. Bubeck. 2010. Best Arm Identification in Multi-Armed Bandits. In *COLT*. https://www.microsoft.com/en-us/research/publication/best-arm-identification-multi-armed-bandits/

[5] J. Audibert, R. Munos, and C. SzepesvÃąri. 2009. Exploration–exploitation tradeoff using variance estimates in multi-armed bandits. *Theoretical Computer Science* 410, 19 (2009), 1876 – 1902. https://doi.org/10.1016/j.tcs.2009.01.016 Algorithmic Learning Theory.

[6] P. Auer, N. Cesa-Bianchi, and P. Fischer. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning* 47, 2-3 (2002), 235–256. https://doi.org/10.1023/A:1013689704352

[7] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. Schapire. 2002. The Nonstochastic Multiarmed Bandit Problem. *SIAM J. Comput.* 32, 1 (2002), 48–77. https://doi.org/10.1137/S0097539701398375

[8] A. Blenk, P. Kalmbach, W. Kellerer, and S. Schmid. 2017. O'zapft is: Tap Your Network Algorithm's Big Data!. In *Big-DAMA*. New York, NY, USA, 19–24. https://doi.org/10.1145/3098593.3098597

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[10] S. Bubeck, R. Munos, and G. Stoltz. 2009. Pure Exploration in Multi-armed Bandits Problems. In *Algorithmic Learning Theory*. Berlin, Heidelberg, 23–37. https://doi.org/10.1007/978-3-642-04414-4_7

[11] P. Chuprikov, S. Nikolenko, and K. Kogan. 2015. Priority queueing with multiple packet characteristics. In *INFOCOM*. 1418–1426.

[12] Code for simulations [n.d.]. Code for simulations. https://github.com/ccrsubmission/learning.

[13] A. Davydow, P. Chuprikov, S. Nikolenko, and K. Kogan. 2017. Throughput Optimization with Latency Constraints. In *INFOCOM*. 1–9. https://doi.org/10.1109/INFOCOM.2017.8057015

[14] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, B. Godfrey, and M. Schapira. 2018. PCC Vivace: Online-Learning Congestion Control. In *NSDI*. 343–356. https://www.usenix.org/conference/nsdi18/presentation/dong

[15] P. Eugster, A. Kesselman, K. Kogan, S. Nikolenko, and A. Sirotkin. 2018. Admission control in shared memory switches. *J. Scheduling* 21, 5 (2018), 533–543. https://doi.org/10.1007/s10951-018-0564-2

[16] A. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. 2013. Participatory Networking: An API for Application Control of SDNs. In *SIGCOMM*. New York, NY, USA, 327–338. https://doi.org/10.1145/2486001.2486003

[17] S. Floyd and V. Jacobson. 1993. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking* 1, 4 (Aug 1993), 397–413. https://doi.org/10.1109/90.251892

[18] N. Foster, R. Harrison, M. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. 2011. Frenetic: A Network Programming Language. *SIGPLAN Not.* 46, 9 (Sept. 2011), 279–291. https://doi.org/10.1145/2034574.2034812

[19] M. He, A. M. Alba, A. Basta, A. Blenk, and W. Kellerer. 2019. Flexibility in Softwarized Networks: Classifications and Research Challenges. *IEEE Communications Surveys Tutorials* 21, 3 (thirdquarter 2019), 2600–2636. https://doi.org/10.1109/COMST.2019.2892806

[20] J. Cao, W. S. Cleveland, Y. Gao, K. Jeffay, F. D. Smith, and M. Weigle. 2004. Stochastic models for generating synthetic HTTP source traffic. In *INFOCOM*, Vol. 3. 1546–1557 vol.3. https://doi.org/10.1109/INFCOM.2004.1354568

[21] K. Jamieson and A. Talwalkar. 2016. Non-stochastic Best Arm Identification and Hyperparameter Optimization. In *AISTATS*, Vol. 51. Cadiz, Spain, 240–248. http://proceedings.mlr.press/v51/jamieson16.html

[22] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar. 2019. A Deep Reinforcement Learning Perspective on Internet Congestion Control. In *ICML*, Vol. 97. 3050–3059. http://proceedings.mlr.press/v97/jay19a.html

[23] N. Kathleen and V. Jacobson. 2012. Controlling Queue Delay. *Queue* 10, 5, Article 20 (May 2012), 15 pages. https://doi.org/10.1145/2208917.2209336

[24] W. Kellerer, P. Kalmbach, A. Blenk, A. Basta, M. Reisslein, and S. Schmid. 2019. Adaptable and Data-Driven Softwarized Networks: Review, Opportunities, and Challenges. *Proc. IEEE* 107, 4 (April 2019), 711–731. https://doi.org/10.1109/JPROC.2019.2895553

[25] I. Keslassy, K. Kogan, G. Scalosub, and M. Segal. 2011. Providing performance guarantees in multipass network processors. In *INFOCOM*. 3191–3199. https://doi.org/10.1109/INFCOM.2011.5935167

[26] A. Kesselman, K. Kogan, and M. Segal. 2010. Packet mode and QoS algorithms for buffered crossbar switches with FIFO queuing. *Distributed Computing* 23, 3 (2010), 163–175. https://doi.org/10.1007/s00446-010-0114-4

[27] A. Kesselman, K. Kogan, and M. Segal. 2012. Improved Competitive Performance Bounds for CIOQ Switches. *Algorithmica* 63, 1-2 (2012), 411–424. https://doi.org/10.1007%2Fs00453-011-9539-9

[28] K. Kogan, A. López-Ortiz, S. Nikolenko, G. Scalosub, and M. Segal. 2016. Large profits or fast gains: A dilemma in maximizing throughput with applications to network processors. *J. Network and Computer Applications* 74 (2016), 31–43. https://doi.org/S1084804516301540?via%3Dihub

[29] K. Kogan, A. López-Ortiz, S. Nikolenko, A. Sirotkin, and D. Tugaryov. 2012. FIFO Queueing Policies for Packets with Heterogeneous Processing. In *Design and Analysis of Algorithms*. Berlin, Heidelberg, 248–260. https://doi.org/10.1007/978-3-642-34862-4_18

[30] K. Kogan, A. L'opez-Ortiz, S. I. Nikolenko, G. Scalosub, and M. Segal. 2014. Balancing work and size with bounded buffers. In *COMSNETS*. 1–8. https://doi.org/10.1109/COMSNETS.2014.6734878

[31] K. Kogan, A. L'opez-Ortiz, S. I. Nikolenko, and A. V. Sirotkin. 2013. Multi-queued network processors for packets with heterogeneous processing requirements. In *COMSNETS*. 1–10. https://doi.org/10.1109/COMSNETS.2013.6465538

[32] K. Kogan, D. Menikkumbura, G. Petri, Y. Noh, S. Nikolenko, A. Sirotkin, and P. Eugster. 2017. A programmable buffer management platform. In *ICNP*. 1–10. https://doi.org/10.1109/ICNP.2017.8117533

[33] C. Kozanitis, J. Huber, S. Singh, and G. Varghese. 2010. Leaping Multiple Headers in a Single Bound: Wire-Speed Parsing Using the Kangaroo System. In *INFOCOM*. 1–9. https://doi.org/10.1109/INFCOM.2010.5462139

[34] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker. 2016. Universal Packet Scheduling. In *NSDI*. Santa Clara, CA, 501–521. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/mittal

[35] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker. 2013. Modular SDN Programming with Pyretic. *login Usenix Mag.* 38, 5 (2013). https://www.usenix.org/publications/login/october-2013-volume-38-number-5/modular-sdn-programming-pyretic

[36] A. Saeed, Y. Zhao, N. Dukkipati, E. Zegura, M. Ammar, K. Harras, and A. Vahdat. 2019. Eiffel: Efficient and Flexible Software Packet Scheduling. In *NSDI*. Boston, MA, 17–32. https://www.usenix.org/conference/nsdi19/presentation/saeed

[37] A. Sivaraman, S. Subramanian, A. Agrawal, S. Chole, S. Chuang, T. Edsall, M. Alizadeh, S. Katti, N. McKeown, and H. Balakrishnan. 2015. Towards Programmable Packet Scheduling. In *HotNets*. New York, NY, USA, Article 23, 7 pages. https://doi.org/10.1145/2834050.2834106

[38] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *SIGCOMM*. New York, NY, USA, 44–57. https://doi.org/10.1145/2934872.2934899

[39] H. Song. 2013. Protocol-Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane. In *HotSDN*. New York, NY, USA, 127–132. https://doi.org/10.1145/2491185.2491190

[40] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E.G. Sirer, and N. Foster. 2014. Merlin: A Language for Provisioning Network Resources. In *CONEXT*. New York, NY, USA, 213–226. https://doi.org/10.1145/2674005.2674989

[41] B. Stephens, A. Akella, and M. Swift. 2019. Loom: Flexible and Efficient NIC Packet Scheduling. In *NSDI*. Boston, MA, 33–46. https://www.usenix.org/conference/nsdi19/presentation/stephens

[42] R. Sutton and A. Barto. 1998. *Reinforcement Learning: An Introduction* (first ed.). MIT Press. https://mitpress.mit.edu/books/reinforcement-learning

[43] A. Voellmy, J. Wang, Y. Yang, B. Ford, and P. Hudak. 2013. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *SIGCOMM*. New York, NY, USA, 87–98. https://doi.org/10.1145/2486001.2486030

[44] K. Winstein and H. Balakrishnan. 2013. TCP Ex Machina: Computer-generated Congestion Control. In *SIGCOMM*. New York, NY, USA, 123–134. https://doi.org/10.1145/2486001.2486020

[45] D. Zarchy, R. Mittal, M. Schapira, and S. Shenker. 2019. Axiomatizing Congestion Control. *POMACS* 3, 2 (2019), 33:1–33:33. https://doi.org/10.1145/3341617.3326148