Practical and Dynamic Buffer Sizing using LearnQueue

Nader Bouacida, Student Member, IEEE, and Basem Shihada, Senior Member, IEEE

Abstract—Wireless networks are undergoing an unprecedented revolution in the last decade. With the explosion of delay-sensitive applications usage on the Internet (i.e., online gaming, VoIP and safety-critical applications), latency becomes a major issue for the development of wireless technology since it has an enormous impact on user experience. In fact, in a phenomenon known as bufferbloat, large static buffers inside the network devices results in increasing the time that packets spend in the queues and, thus, causing larger delays. Concerns have arisen about designing efficient queue management schemes to mitigate the effects of over-buffering in wireless devices. In this paper, we advocate the exploitation of machine learning techniques for dynamic buffer sizing.

We propose LearnQueue, a novel reinforcement learning design that can effectively control the latency in wireless networks. LearnQueue adapts quickly and intelligently to changes in the wireless environment using a sophisticated reward structure. The latency control is performed dynamically by tuning the buffer size. Adopting a trial-and-error approach, the proposed scheme penalizes the actions resulting in longer delays or hurting the throughput. In addition, the scheme parameters are designed for an optimized operation depending on different applications requirements. Using the latest generation of WARP hardware, we investigated LearnQueue performance in various network scenarios employing different transport protocols. The testbed results prove that LearnQueue can grantee low latency while preserving throughput under various congestion situations. We also discuss the feasibility and possible limitations of large-scale deployment of the proposed scheme in wireless devices.

Index Terms—bufferbloat, dynamic buffer sizing, latency, wireless networks, Active Queue Management, Reinforcement Learning.

1 INTRODUCTION

With the advent of mobile devices, both in terms of their applications and affordable prices, users generate ever-increasing demand on wireless connectivity, resulting in excessive latency and poor Internet performance. In the last few years, experts [1] point out “bufferbloat” as the main suspect for the observed long delays into today’s networks. To avoid packet dropping, several Internet service providers force vendors to supply their hardware with larger buffers. The manufacturers do not find any embarrassment to meet the requirements of their clients since the memory costs have dropped significantly. For instance, the average RAM memory cost per gigabyte [2] falls down from 859.375 $ in 1985 to only 4.37 $ in 2015. Although large buffers managed to absorb traffic bursts and limit packets dropping, their adoption into network devices without sufficient studies fails to take into consideration the nature of congestion-avoidance algorithms of the Internet’s most common transport protocol, TCP. In fact, TCP protocol maintains a congestion window to control its sending rate. The TCP sender continuously increases its congestion window. When there is a disproportionate congestion, then, one or more buffers along the path will overflow, resulting in a datagram containing a TCP segment to be dropped. Hence, the dropped packet induces a loss event, which is perceived by the sender as an indication of congestion along the path. However, with large static buffers initially designed to limit packets dropping, the sender will not be able to perceive the congestion signal on time, causing an unacceptable queuing delay.

To prove the impact of bufferbloat on network latency, Gettys et al. [1] recorded a smoke ping while moving 20 GB of data to a nearby server. They observed latencies in the order of seconds. In 2007, a study of 2000 hosts connected through cable and DSL companies in Europe and North America showed that upstream queues for DSL were frequently in excess of 600 ms, and those for cable generally exceeded one-second [3]. Bufferbloat is not only restricted to broadband. In [4], authors carried out extensive measurements over the 3G/4G networks of all four major carriers in the United States as well as the largest cellular carrier in Korea. Their experiments span more than two months and consume over 200GB of 3G/4G data. According to their measurements, extremely long delays (up to 10 seconds) are detected.

The situation will be even much worse in a wireless environment, since packet drops are more frequent and data rates are limited compared to wired networks. The excessive long delays caused by bufferbloat problem represent an overkill for many real-time applications. For example, online gaming requires a latency around 30 ms. A delay exceeding 100 ms will affect the user experience [5]. However, customers are likely to abandon a game when they experience a network delay of 500 milliseconds. VoIP applications cannot tolerate delays larger than 150 ms to ensure the intended level of QoS. Amazon observed that they lose 1% of sales for each additional 100 ms of latency [6], which literally translates to more than one billion dollars loss.

Multiple challenges are facing buffer management in wireless devices, the most common mechanism for limiting bufferbloat. To prevent latency from going arbitrary large, the naive solution is to shrink the buffer size or the
congestion windows resulting in unnecessary dropping of packets. An early congestion signal following the packets dropping will lead TCP to back off and reduce its rate resulting in link under-utilization. Hence, the low latency should be achieved without affecting high link utilization. Moreover, the wireless environment usually causes random bit errors to occur in short bursts, thus, leading to a higher probability of multiple random packet losses. In TCP protocol, such packets losses are wrongly inferred to be the result of network congestion and would mistakenly trigger the TCP sender to reduce its sending rate unnecessarily. Hence, efficient buffer management methods should be able to deal with the interference of different factors that influences the congestion signal in wireless links. Furthermore, any designed system should guarantee agility to sudden changes in the network and ensure system stability for various stream types, network topologies, and data rates. This requirement is crucial for wireless systems where links rates change dynamically, and the spectrum is a shared resource between a set of neighboring nodes.

In this paper, we propose to leverage a novel reinforcement learning based buffer management scheme for wireless networks, called LearnQueue. It dynamically adjusts the buffer size to maintain the desired Quality of Service (QoS). The congestion detection relies on a reward function that accounts for both latency trends and dropping rates in the queue. Our testbed results demonstrate that LearnQueue can control the latency around a reference value using the previously learned experience and the immediate feedback from the network without impacting the link utilization. By determining the optimal level of balance between delay and dropping, LearnQueue has the ability to service both delay-sensitive and dropping-sensitive applications. Furthermore, LearnQueue does not incur per-packet extra-processing and adapts rapidly to various network conditions. Testbed results demonstrates that our scheme outperforms other well-known techniques in the literature and achieves at least one order of magnitude reduction in network latency compared to the default buffering method.

The rest of the paper is organized as follows. In Section 2, we survey the related work and we highlight different challenges and requirements. In Section 3, we introduce the system design and implementation details. Then, we present a mathematical interpretation of LearnQueue design and discuss useful techniques for performance analysis. In Section 4, we report the performance evaluation. Finally, we discuss the incentives and limitations behind LearnQueue in Section 5 and we conclude the paper in Section 6.

2 Related Work

Buffers are essential to smooth the bursty data traffic over the time-varying channels in wireless networks, but overly large and unmanaged buffers cause excessive delays that affect end-user experience. Active Queue Management schemes (AQM) have come to the rescue of bufferbloat problem. RFC 2309 [7] strongly recommended the testing, the standardization, and the widespread deployment of AQM methods in the network to improve the performance of today’s Internet. One of the earliest AQM algorithms is Random Early Detection (RED) [8]. RED computes the average queue length and drops packets based on a dropping probability. This dropping probability follows the same trends of the queue occupancy. Many variations of RED were proposed after that, but none of them succeeded to gain traction because they require complicated manual configuration and also due to their slow response to quick changes in the network environment.

In 2012, one of the most famous AQM scheme called CoDel [9] was proposed. CoDel monitors the packet sojourn time in the queue. For a given interval, the algorithm finds the lowest queuing delay experienced by all packets. If the lowest packet sojourn time over a predefined interval exceeds the target, the packet is dropped, and the interval will be shortened. Otherwise, the packet is forwarded, and the interval is reset to 100 milliseconds (initial default value). This algorithm was essentially designed to detect bad queues, which are defined as the queues that last longer than one RTT resulting in a constantly high buffering latency. CoDel is a self-configurable algorithm and has shown good performance over traditional AQM solutions [10]. However, it requires per-packet timestamps and drops packets after spending some time in the queue, wasting all resources used for maintaining and processing those packets. Unlike CoDel, LearnQueue does not require per-packet extra processing. Moreover, it is highly responsive to sudden changes in the network conditions. Later, PIE (Proportional Integral controller Enhanced) [11] was introduced. Similar to CoDel, PIE randomly drops a packet when experiencing congestion. The drop probability is computed based on the deviation of the queuing delay from a predefined target and the direction where the delay is moving. However, PIE considers only latency trends without looking at how much packets are dropped in the queue. Our scheme avoid unnecessary dropping of packets by considering both metrics. Neither CoDel nor PIE was originally designed for operation in wireless networks. However, they have shown their ability to cope with such networks. In 2014, authors of [12] proposed Wireless Queue Management (WQM), a dynamic buffer sizing scheme that addressed the unique challenges of wireless networks [13] [14]. WQM adjusts the buffer size according to the queue draining rate. While their scheme is lightweight and simple, it adjust the buffer size naively based on a synthetic mathematical model with parameters chosen to fit a certain wireless standard. LearnQueue tunes the buffer size using real-time feedback from the network and is not dependent on the wireless standard of the testbed.

In a systemic evaluation of the bufferbloat effect, Cardozo et al. [15] suggested that bufferbloat might be resulting from different layers of buffering. They shed lights on the impact of varying the size of various buffers in the network transmit stack, especially the ring buffer, which is a circular buffer associated with the network card in Unix-based architecture. Their speculative claim aims to turn the efforts of mitigating the bufferbloat impact to focus more on ring buffers sizing instead of the conventional transmit buffers. However, the obtained results cannot be taken for granted since the experiments are conducted in a very limited scope of scenarios. In Linux kernel 3.3, a new functionality known as Byte Queue Limits (BQL) has been introduced [16]. BQL is a self-regulating algorithm that intends to estimate the limits of packets data which can be put in the transmission queue.
of a network interface. Using BQL, we can control and reduce the number of packets sent to the ring buffer, shifting the queuing to upper layers. The use of BQL combined with queue disciplines like CoDel can drastically reduce the latency \[15\]. A serious weakness within those methods is that the ring buffers lack flexibility, which makes their access and control a complicated task. For this purpose, LearnQueue operates on transmit buffers.

We should highlight that AQM based solutions are not the only proposals aiming at fighting the bufferbloat phenomenon. In the transport layer, some proposed methods keep the network core unaltered. The adaptation of such solutions is restricted to the end hosts. Those solutions come into sight with the engineering of end-to-end flows and congestion control alternatives to best-effort TCP. Congestion and flow control may have different objectives than AQM techniques, such as controlling the streaming rate over TCP connections as done by YouTube or Netflix, or aggressively protecting user QoE as done by Skype over UDP. In their seminal paper of 2012, authors proposed Dynamic Receive Window Adjustment (DRWA) \[4\], a delay-based congestion control algorithm that modifies the existing receive window adjustment algorithm of TCP to control the sending rate indirectly. DRWA increases the receive window when the current RTT is close to the minimum RTT observed and decreases it when RTT becomes larger due to larger queuing delay. With proper parameters tuning, DRWA succeeded to keep the queue size at the bottleneck link small enough so that the throughput and the delay experienced by a TCP flow are both optimized. The results show that DRWA reduces the RTT by \(25\%\) to \(49\%\) while achieving similar throughput. Congestion control methods have the advantage to be lightweight and easy to deploy, especially with devices with limited resources such as cellular phones. However, those methods are mainly concerned with congestion handling and give little attention to optimizing latency.

3 LearnQueue: Design and Implementation

In this section, we describe the design of LearnQueue, an AQM algorithm for dynamic buffer sizing in wireless networks. To control the latency, LearnQueue dynamically tunes the buffer size using reinforcement learning. Reinforcement learning methods \[17\] learn by trial-and-error which actions are most valuable in which states. Feedback is provided in the form of a scalar reward. The sequence of actions that maximize the total reward defines a policy, which is simply a mapping from states to probabilities of selecting each possible action \[18\]. The value of a state is the total amount of reward an agent can expect to accumulate over the future starting from that state. The reward function is estimated from the sequences of observations that the agent makes over its entire lifetime. One of the common methods in reinforcement learning is Q-learning. The latter \[19\] \[20\] falls under the scope of Temporal Difference (TD) learning algorithms \[21\], where we explore the environment to observe the value of the next state and the immediate reward. The value of current state is updated by checking the current estimate of state value and the value of next state combined with the reward.

3.1 System Description

LearnQueue scheme is an adapted version of Q-learning algorithm. As illustrated in Fig. \[1\] each buffer size is considered a state (the buffer size is defined in terms of packet blocks). In each state, two actions are possible: either increase or decrease the buffer size by one block. Each block can accommodate a single packet. There is a hard limit on the queue size (equal to 400 packets in Fig. \[1\] example; we cannot allow more than 400 packets to wait in the queue.). Q-learning operates by incrementally updating the expected values of actions in different states and storing them in a table known as Q-table. The policy is enforced by executing the action with the highest expected value. For each state, every possible action is given a value which is a function of both the immediate reward resulting from executing the action and the expected future reward in the new state after taking the action. This is expressed by the one-step Q-update equation:

\[
Q(s_t, a_t) = (1 - \alpha) \times Q(s_t, a_t) + \left( r_{t+1} + \gamma \times \max_a Q(s_{t+1}, a) \right)
\]

(1)

where \(r_{t+1}\) is the reward observed after executing the action \(a_t\) while in state \(s_t\), \(Q\) is the expected value of performing the action, \(\alpha\) is the learning rate and \(\gamma\) is the discount factor.

The system can be interpreted as a Markov decision process \(M = (S, A, R^M, P^M)\) in repeated infinite episodes of interaction. \(S\) is the state space, \(A\) is the action space, \(R^M(s)\) is a probability distribution over reward realized when selecting action \(a\) while in state \(s\), \(P^M(s'|s)\) is the probability of transitioning to state \(s'\) if action \(a\) is selected while at state \(s\).

A deterministic policy \(\mu\) is a function that maps each state \(s \in S\) to an action \(a \in A\). For each MDP \(M = (S, A, R^M, P^M)\) and policy \(\mu\), we define a value function as the following:

\[
V^\mu_M(s) = E_{M,\mu} \left[ \sum_{t=0}^{\infty} \gamma^t R^M_{a_t}(s_t) | s_0 = s \right]
\]

(2)

where \(R^M_{a_t}(s_t)\) denotes the expected reward realized when action \(a\) is selected while in state \(s\).

A policy \(\mu\) is said to be optimal for MDP \(M\) if \(V^\mu_M(s) = \max_{a \in A} V^M(a)\) for all \(s \in S\) and the optimal value function verifies:

\[
V^*(s) = \max_{a \in A} \sum_{s' \in S} P^M_a(s'|s) \left[ r(s, a, s') + \gamma V^*(s') \right]
\]

(3)

where \(r(s, a, s')\) is the reward assigned everytime a transition from \(s\) to \(s'\) occurs due to action \(a\).

We define the regret incurred by a reinforcement learning algorithm \(\pi\) up to time \(T\) to be:

\[
\text{Regret}(T, \pi) = \sum_{k=1}^{T} \Delta_k
\]

(4)

where \(\Delta_k\) denotes regret over the \(k\)th episode, defined with respect to the true MDP \(M^*\) by:

\[
\Delta_k = \sum_{s \in S} V^M_{\mu^*}(s) - V^M_{\mu_k}(s)
\]

(5)
with \( \mu^* = \mu^M \) and \( \mu_k \sim \pi_k(H_{t_k}) \). The regret is not deterministic since it can depend on the random MDP \( M^* \), the algorithm’s internal random sampling and, through the history \( H_{t_k} \), on previous random transitions and random rewards. A reinforcement learning algorithm can be assessed in terms of regret and cumulative reward \([22]\).

Given that \( V^*(s) = \max_a Q^*(s,a) \), we define the optimal \(Q\)-function, \( Q^* \) as follows:

\[
Q^*(s,a) = \sum_{s' \in S} P^M_a(s'|s) [r(s,a,s') + \gamma \max_{b \in A} q(s', b)]
\]

\[(6)\]

The algorithm determines the optimal \(Q\)-function using point samples defined by the update rule \([1]\).

The optimal \(Q\)-function is a fixed point of a contraction operator \( H \), defined for a generic function \( q : S \times A \to \mathbb{R} \) as:

\[
(H q)(s,a) = \sum_{s' \in S} P^M_a(s'|s) [r(s,a,s') + \gamma \max_{b \in A} q(s', b)]
\]

\[(7)\]

This operator is a contraction in the \( \infty \)-norm, i.e.,

\[
\| H q_1 - H q_2 \|_\infty \leq \gamma \| q_1 - q_2 \|_\infty
\]

\[(8)\]

To prove the inequality (8), we expand the expression of \( \| H q_1 - H q_2 \|_\infty \) as follows:

\[
\| H q_1 - H q_2 \|_\infty =
\]

\[
\| q_1 - q_2 \|_\infty
\]

\[(9)\]

As long as the above conditions are satisfied, the algorithm is guaranteed to converge the optimal \(Q\)-values.
3.2 Learning Process

Specifically, we define our problem as a reinforcement learning problem with the following characteristics. Our algorithm should be completely goal-independent, allowing the mechanics of the environment to be learned independently of the actions that are being undertaken. The algorithm must also be policy-independent. The reward given by moving from one state to another is not predefined and changes over time based on network latency trends and dropping rates. Finally, the algorithm should be called periodically instead of running for a defined number of iterations in order to keep adapting to the queue length to the changing network conditions. Although the Q-learning algorithm is independent of the policy [25], it is not goal-independent. In our application, there exists no static goal state to reach since the optimal state keeps moving over time. We modified the original Q-learning algorithm to meet the requirements mentioned above. The pseudocode of LearnQueue scheme is given in Algorithm 1. To show various steps, we provide the flowchart of the algorithm in Fig. 2. Note that the algorithm was designed to cope with a system with multiple queues being dequeued using a round-robin strategy. The variable “queue_size” controls the queue length in number of packets. It accounts for both enqueued packets and empty slots. For the sake of exploring the environment, our method makes use of adaptive ε-greedy strategy for choosing actions. With a probability ε, we randomly select an action, independently from the Q-values estimates. The exploration probability ε defines the trade-off between discovering new possibilities (exploration) and refining current procedure (exploitation). We start by a high exploration probability ε = 0.9 and decrease it non-linearly with the number of action-state combinations discovered towards its final value ε = 0.1, thus ensuring optimal actions are discovered. We tried to decrease the exploration probability with different rates and choose the one that matches the intended pattern.

3.3 Reward Function

The learning process is governed by a reward function. The key challenge here is to design a reward function that penalizes the action resulting in longer delays than a specified target without causing a significant increase in the dropping rate. Indeed, a delicate balance between achieving low latency and maintaining high enqueue rate rules the learning process. The reward function contains two main components: “delay_reward” which is related to delay and “enq_reward” which is related to dropping. The former measures the deviation of the current queuing delay “curr_delay” from the reference latency value “delay_ref” and calculated as:

\[ \text{delay\_reward} = \delta \ast (\text{delay\_ref} - \text{curr\_delay}) \]  
(11)

The latter presents the enqueuing rate scaled to the delay and is calculated as:

\[ \text{enq\_reward} = \eta \ast (\text{max\_delay} - \text{delay\_ref}) \ast \text{enq\_rate} \]  
(12)

where “enq\_rate” is computed by dividing the number of enqueued packets by the total number of both enqueued and dropped packets over the update period of the algorithm. “max\_delay” is the maximum time required to drain the queue as follows:

\[ \text{max\_delay} = \frac{\text{packet\_size} \ast \text{queue\_size}}{R_{\text{basic}}} \]  
(13)

where “packet\_size” represents the packet length in bytes and \( R_{\text{basic}} \) is the basic physical rate. Finally, as mentioned

---

**Algorithm 1 LearnQueue Algorithm pseudocode**

```plaintext
1: // Call periodically every 15 ms
2: Loop over the queues of associated stations
3: for i = 1 to active_bss_info members.length do
4: curr_state ← queue_size[i];
5: Epsilon-greedy exploration
6: rnd ← random();
7: if rnd < $\epsilon_i$ then
8: curr_action ← random() (mod 2);
9: else
10: if $Q_i[curr_state][0] > Q_i[curr_state][1]$ then
11: curr_action ← 0;
12: else
13: curr_action ← 1;
14: end if
15: end if
16: Take action and observe the immediate reward
17: if curr_action = 0 then
18: if curr_state = 1 then
19: next_state ← curr_state;
20: reward ← −max_delay[i];
21: else
22: next_state ← curr_state − 1;
23: queue_size[i] ← next_state;
24: reward ← delay_enqueue_reward(i);
25: end if
26: else
27: if curr_state = max_size then
28: next_state ← curr_state;
29: reward ← −max_delay[i];
30: else
31: next_state ← curr_state + 1;
32: queue_size[i] ← next_state;
33: reward ← delay_enqueue_reward(i);
34: end if
35: end if
36: Tune exploration probability and discount factor
37: if the state-action combination is not discovered then
38: if $\epsilon_i > 0.1$ then
39: $\epsilon_i ← 0.995 \ast \epsilon_i$;
40: end if
41: if $\gamma_i < 0.9$ then
42: $\gamma_i ← 1 - (0.995 \ast \gamma_i)$;
43: end if
44: end if
45: Update the Q-value for the current state-action
46: s ← curr_state; a ← curr_action;
47: $Q_i[s][a] ← (1 - \alpha) \ast Q_i[s][a] + (\text{reward} + \gamma_i \ast \text{max}(Q_i[\text{next_state}][0], Q_i[\text{next_state}][1]))$;
48: end for
```
earlier, the total immediate reward will be the sum of the delay component \( (12) \) and the dropping metric \( (13) \):

\[
\text{total\_reward} = \text{delay\_reward} + \text{enq\_reward} \quad (14)
\]

In the above, we introduced two scaling factors \( \delta \) and \( \eta \). Parameter \( \delta \) determines how the deviation of the current queuing delay from the target value affects the total immediate reward. Parameter \( \eta \) measures to what extent the dropping factor influences the reward function. The relative weight between \( \delta \) and \( \eta \) determines the final balance between the latency offset and the enqueue rate. In this context, the quantity “\( \text{max\_delay} \)” aims to make the two components of the reward function on the same scale. As a matter of fact, when the current queuing delay reaches the maximum delay (\( \text{curr\_delay} = \text{max\_delay} \)) and using equal scaling weights (\( \delta = \eta = 0.5 \)), the obtained total reward is equal to zero assuming that \( \text{enq\_rate} = 1 \). In this situation, no packets are dropped from the queue, but, we encounter longer delays. In case there is packet drop accompanied by a high level of latency, we automatically obtain a negative reward. The only exception occurs when the queue size tries to exceed its allowed limits. In this scenario, we instantly penalize the reward by setting it to the negative of the theoretical maximum delay “\( -\text{max\_delay} \)” to move out of this zone.

### 3.4 Implementation

We implemented \textit{LearnQueue} scheme using off-the-shelves WARP v3 boards \cite{24}. The 802.11 Reference Design v1.6 \cite{25} of WARP v3 boards prototypes a MAC layer composed of two levels. The low-level MAC of the reference design handles one packet at a time. The high-level MAC manages many packets at once via a series of queues. In each Access Point (AP), one queue is created per associated node plus one queue for all broadcast traffic and one queue for all management packets. Whenever the low-level MAC finishes the transmission of a packet, the next available packet is dequeued from the appropriate queue and passed to low-level MAC for transmission. The Access Point implements a round-robin dequeuing procedure and alternates between different queues. When a new packet requiring wireless transmission is received, it will be added to the tail of the queue associated with the node to which the packet is addressed. Similar to commercial Wi-Fi devices, WARP v3 boards adopt Drop Tail as default queuing scheme, which means when the queue is filled to its maximum capacity, the newly arriving packets are dropped until the queue has enough room to accept incoming traffic.

Our implementation aims at modifying the default queuing discipline of 802.11 Reference Design for the sake of adopting \textit{LearnQueue}, which manages each queue independently from the others. We do not alter multicast and management queues default behavior since our focus targets only data frames queues, which are vulnerable to bursty traffic. Our scheme is evaluated in a wireless testbed in our lab. The testbed nodes locations are shown in Fig. 3. We tested our scheme through several scenarios. The first one is the two nodes scenario where Node 1 represents an access point (AP), and Node 2 is a station (STA). This scenario mimics the communication of a station with the backbone network via an AP. The second scenario involves one node configured as AP (Node 3), and two nodes (Node 1 and Node 2) set up as STA using 802.11 Reference Design. Node 1 communicates with Node 2 via the AP (Node 3). This scenario represents a typical Wi-Fi network in infrastructure mode. Fig. 4 shows the experimental setup of the three nodes scenario. Further details about our testbed are summarized in Table 1. As shown in the table, we used dual band omni-directional VERT 2450 antennas \cite{26} for operation in the 5 GHz radio band to avoid interfering with our campus production network. The node locations are chosen in accordance with the Wi-Fi standard used. The source code of \textit{LearnQueue} is available along with executable experiment scripts at \cite{27}.
4 SYSTEM EVALUATION

In this section, we evaluate the performance of LearnQueue scheme in both a two node scenario and a three nodes scenario. We measure different network performance metrics such as delay and throughput. For a rigorous analysis, we choose to compare LearnQueue’s performance against a dropping-based AQM method and another dynamic buffer sizing AQM method. The most recent state of the art methods are PIE and WQM discussed earlier in Section 2. The only hardware implementation of those algorithms is available for machines running Linux. Thus, we integrated both PIE and WQM in the software reference design of our hardware. To the best of our knowledge, this is the first work that implements AQM techniques into WARP v3 boards instead of the traditional Linux devices. WARP boards offer a scalable programmable wireless platform to prototype advanced wireless networks and their functionalities. Unless otherwise stated, all latency and queue occupancy statistics are extracted from the sender node, and the experiments are carried out using a physical rate of 26 Mbps. Since the testing traffic is synthetic and not generated from any specific application, we set LearnQueue scaling weights to their standard values $\delta = \eta = 0.5$. Similarly, PIE and WQM parameters are fixed to their default values. However, for the sake of fair comparison, we define a unique reference latency value (target latency value) for all the schemes (equal to 30 ms).

### 4.1 Two Nodes Scenario

To stress the queues inside our devices, we run a fully backlogged UDP flow from the AP to the STA for 100 s using a local Constant Bit Rate (CBR) traffic generator and measure its queuing latency and throughput. We aim here to simulate a heavily congested wireless link. Fig. 5 illustrates the Probability Density Function (PDF) of the throughput for bidirectional UDP flows. LearnQueue manages to keep high link utilization by achieving a throughput sum of 17.36 Mbps, close to the throughput sum of both flows for the default case equal to 17.1 Mbps. However, PIE reduces the total throughput by 6.79%. Moreover, a closer look at Fig. 5 proves that our scheme ensures a better fairness between flows and a lower variation of individual flow throughputs with time.

To demonstrate LearnQueue’s performance under TCP traffic, we repeat the first experiment with a TCP flow generated using Iperf tool [28]. The average queuing latency, depicted in Fig. 6, is controlled around 5.16 ms for LearnQueue, 26.94 ms for PIE and 100.62 ms for WQM. In the default case, the unmanaged static buffers cause long latencies that exceed 400 ms. Again, LearnQueue was able to contain the queuing delay around the reference level regardless of the traffic type. LearnQueue is highly adaptive as illustrated in Fig. 6a. It does not allow the harmful buildup of the queue by controlling the buffer size. To see how LearnQueue affects the dynamics of TCP, we traced the congestion windows variation with time. Fig. 7 clearly shows that LearnQueue prevents the congestion window from growing arbitrarily large in order to provide a steady latency control for varying network conditions without incurring a cost in terms of link utilization. Fig. 8 plots the obtained TCP throughput under different physical rates. Our method regulates the TCP traffic very well so that the wireless link is close to its full capacity. For higher physical rates, LearnQueue succeeded to boost the throughput by 4.1% and 17.1% respectively for 52 Mbps and 65 Mbps rates. In fact, with higher rates, the congestion becomes less severe, and our solution took advantage by allowing more
packets to be injected into the wireless link while remaining within a secure margin of latency. This underlines the LearnQueue ability to reach the best achievable throughput while maintaining the latency unaffected by the increased traffic intensity. If we increase the number of TCP flows to five, similar results despit in Fig. 6d are found. In fact, the resulting average queuing latency is equal to 4.08 ms for LearnQueue, 33.07 ms for PIE, 120.18 ms for WQM and 285.3 ms for Drop Tail. Other methods appear to suffer from the increased number of TCP sessions while our scheme performance remains unchanged.

We know that IEEE 802.11 networks usually use rate adaptation algorithms to determine the optimal data transmission rate most appropriate for current wireless channel conditions [29]. So, the physical rate at which the sender sends its data keeps changing all time. Hence, we were curious how different methods will interact with sudden shifts in the data transmission rates. For this reason, we sent a TCP stream for a period of 200 s, while incrementing the link physical rate each 25 s, starting from the basic rate of 6.5 Mbps until reaching the maximum speed of 65 Mbps. As illustrated in Fig. 6e, the average latency for LearnQueue is equal to 16.83 ms while PIE scored 25.63 ms. Similar to previous results, the network suffers a delay equal to 95.3 ms and 224.47 ms respectively for WQM and Drop Tail. In the worst case, our scheme guarantees one order of magnitude reduction in queuing delay compared to the default buffer management behavior. The average throughput reaches full capacity of 16.53 Mbps. The queue occupancy variation for the first 100 s of the experiment is depicted in Fig. 10b.
Clearly, LearnQueue ensures a precise fine-tuning for the queue size independently from the physical rate employed. The optimal buffer size is estimated with the help of the reward function, that returns an accurate feedback of the congestion level into the network.

We consider the traffic with other characteristics such as bulk and bursty traffic. Indeed, we conduct an experiment where each node acting as a station will generate random traffic with packet inter-arrival time following the exponential distribution. The inter-arrival time distribution of generated packets will alternate between small mean, equal to 0.001 s, and large mean, equal to 0.03 s. Those parameters are experimentally tuned to generate the intended bursty traffic. As a result, the average inter-arrival time of packets generated will oscillate periodically to simulate bursty traffic. The latency results are given in Fig. 9c. Unlike constant bit rate traffic, bursty traffic does not stress the network on a continuous basis (almost half of the latencies observed during the experiment tend to zero) since its sending rate is variable. LearnQueue performs better than WQM and PIE. Latency does not exceed 150 ms for LearnQueue and 250 ms for PIE while it reaches over 500 ms for WQM. One fact that captures our attention here is that WQM behaves even worse than the default buffering mechanism. This highlight the limited ability of WQM to adapt with various traffic patterns. Besides, PIE seems to act optimally with light network congestion. However, with higher congestion level, its performance is suboptimal compared to our scheme. To dig deeper into LearnQueue operation, we provide a complete experimental analysis of the reward function and its interaction with the network performance metrics in the subsection 4.3.

4.2 Three Nodes Scenario

In this part, we experimentally assess the global performance of LearnQueue in a multi-hop situation to confirm the validity of previous results. We run a fully backlogged UDP flow between two STA associated to an AP for a duration of 100 s. The results are given in Fig. 9a. LearnQueue manages to reduce the average queuing latency from 537.1 ms in the default case and 157.67 ms in the case of WQM to only 19.87 ms. PIE results in 47.27 ms of delay, almost greater than the double of the latency achieved by LearnQueue. The queuing delay fluctuates more evenly around the reference level. Our solution does not hurt the throughput since it achieves 8.97 Mbps, almost equal to the default throughput (8.98 Mbps) while PIE slightly reduces the throughput to 8.89 Mbps. Considering bidirectional flows, LearnQueue reduces the average queuing latency from 603.56 ms in the default case and 183.73 ms in the case of PIE to only 60.19 ms. Compared to the default case, this corresponds to 10x reduction, which is a significant improvement. This time, in comparison to the default scheme, a small drop occurs in the total network throughput from 12.79 Mbps to 11.65 Mbps. This reduction may be due to a higher level of unpredictable interference in the wireless link. However, LearnQueue still performing better than both PIE (11.24 Mbps) and WQM (11.44 Mbps). The experiment is repeated using two bidirectional TCP flows. The latency CDF curves are reported in Fig. 9c. In average, the packets experience a queuing delay equal to 5.61 ms for LearnQueue, 16.68 ms for PIE, 48.17 ms for WQM and 56.17 ms for the static buffers. LearnQueue achieves 9.31 Mbps as throughput, almost the same with Drop Tail (9.58 Mbps). Curiously, the latency results are considered lower than expected. We investigate this issue, and we found that there is limited queue buildup even in the default case (see Fig. 10c), which can be explained by a higher packet error rate in the wireless links. Certainly, higher interference will slow down the sending rate of TCP, affecting the congestion level into the network. LearnQueue interacts with the light traffic and allows for more packets to be enqueued (more than 65 packets in some cases as shown in Fig. 10c) without losing its firm control on latency.

4.3 LearnQueue Operation

We aim here to validate the functionalities of LearnQueue, making sure it performs as designed using static traffic sources. We ensure that two nodes are configured as one AP, one STA and that the STA is associated with AP. Then, we start a fully-backlogged locally generated traffic flow from AP to STA for 100 s. After waiting for 1/3 of the trial time, we start another fully-backlogged locally generated traffic flow in opposite direction from STA to AP. We let 1/3 of the trial time elapses and we stop the traffic flow from STA to AP while keeping the other flow running until the end of the experiment. Same experimental conditions in Section 4 apply except that the two nodes are placed within a closer distance (≈ 6 m). Independently from traffic type, the most important thing is to stress the queue limits.

Results in Fig. 11 demonstrate the basic operations of our method. Particularly, Fig. 11a depicts the evolution of queuing delay with time. The queue experiences minimal congestion during the first and the last 33 s when we have a single flow from AP to STA. In fact, it settles around 28 ms. At the moment when another flow in opposite direction is activated, the congestion in the network becomes more severe and the queuing latency jumps to higher values as a result. LearnQueue quickly reacts by bringing down the reward function values, preventing the latency from going arbitrary large. Fig. 11b clearly illustrates the perfect reflexion between the delay curve and the reward function through the x-axis. Every time the traffic intensity changes, the reward function plays the role of a latency compensation engine to restore the control of the queuing latency to be around an equilibrium value. During the high congestion period, the average latency is equal to 65 ms. LearnQueue optimizes the latency with caution towards link throughput. Fig. 11b proves that our scheme does not affect the throughput or the fairness between flows. Whether we have a single flow or bidirectional flows running, the link throughput is always around its maximum capacity of 18 Mbps. LearnQueue provides consistent performance under varying network conditions. To further understand how the scaling factors govern the trade-off between latency and throughput, we plot a 3D representation of the reward function values varying with both time and the δ parameter given that η = 1 − δ. Fig. 11d demonstrates that the reward response to congestion is sharper with higher values of δ which is the factor favoring delay over enqueue rate.
4.4 Experimental Analysis of The Scaling Weights

The scaling factors or scaling weights \( \delta \) and \( \eta \) represent an important aspect of LearnQueue scheme because they directly influence the reward function to control the tradeoff between delay and dropping. As mentioned previously, the first scaling factor \( \delta \) determines how the deviation of the current queuing delay from the target value affects the total immediate reward. The second scaling weight \( \eta \) measures to what extent the dropping factor influences the reward function. The relative weight between \( \delta \) and \( \eta \) defines the final balance between the latency offset and the enqueue rate. We evaluate the effect of changing the scaling weights experimentally and analyze their role in adapting LearnQueue to specific application requirements.

We send a UDP flow from AP to STA with a constant rate of 18 Mbps to saturate the link for a duration of 100 s. The role of buffers is to absorb traffic bursts. We collect statistics on queuing delay and on dropping rates. We focus on the dropping due to queue overflow, and we discard any packets drops due to interference in the wireless link. We tried different values of \( \delta \) and \( \eta \) to test their sensitivity. The results are averaged over five trials and reported in Table 2. When \( \delta = 0.9 \) and \( \eta = 0.1 \), the reward function will be more delay-sensitive. Thus, the algorithm responds

<table>
<thead>
<tr>
<th>Scaling Weights</th>
<th>Queuing Delay</th>
<th>Dropping Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \delta = 0.9, \eta = 0.1 )</td>
<td>23.64 ms</td>
<td>10.23 %</td>
</tr>
<tr>
<td>( \delta = 0.7, \eta = 0.3 )</td>
<td>34.84 ms</td>
<td>8.79 %</td>
</tr>
<tr>
<td>( \delta = 0.5, \eta = 0.5 )</td>
<td>39.51 ms</td>
<td>6.83 %</td>
</tr>
<tr>
<td>( \delta = 0.3, \eta = 0.7 )</td>
<td>42.35 ms</td>
<td>4.98 %</td>
</tr>
<tr>
<td>( \delta = 0.1, \eta = 0.9 )</td>
<td>65.76 ms</td>
<td>2.35 %</td>
</tr>
</tbody>
</table>
by limiting the queue size to gain more rewards originating from delay reduction. As a result, compared with the case where the scaling weights are set to their standard values ($\delta = \eta = 0.5$), the queuing latency will decrease from 39.51 ms to 23.64 ms while the dropping rate will increase from 6.83 % to 10.23 %. This is totally natural because LearnQueue try to maximize its long-term accumulative reward which is significantly more influenced by the delay factor rather than the dropping factor. We observe the opposite effect when we reverse the values of $\delta$ and $\eta$. In the situation where $\delta = 0.1$ and $\eta = 0.9$, the dropping rate declines to only 2.35 % and the queuing latency raise to 65.76 ms. Here, the reward function has a strong bias towards dropping, and those settings can be optimal for video streaming application requirements for example. Besides, we tried some scaling factors values in-between, and the resulting performance confirms previous findings. Fig. 12 illustrates the queuing delay and the dropping rate evolution with time for $\delta = 0.7, \eta = 0.3$. It is apparent that LearnQueue does not allow the delay or the dropping rate to diverge from a reference value and keeps a continuous steady control over time.
5 Discussion: Adoption and Incentives

In this section, we discuss the incentives and the costs for telecommunications equipment manufacturers to deploy LearnQueue in their products. Those factors are key to its wide deployment.

LearnQueue is beneficial for the users as shown in Section 4. It is highly reliable and does not require any user intervention as it seamlessly operates. Its implementation is simple and can be easily deployed via firmware updates. Besides, LearnQueue is generic; that means it can adapt to any kind of Internet traffic, transport protocols or interference levels. This alleviates the need for complex parameter settings as done in RED algorithm. Furthermore, LearnQueue offers a unique opportunity for data services providers to control the delay around a reference value without incurring penalties in terms of link utilization and, thus, resolving the synchronization issues and improving consistency between their primary and secondary servers.

We anticipate that there will be several challenges facing LearnQueue to find its way to deployment into commercial wireless routers. Both time and memory complexity of reinforcement learning methods are usually large [30]. What about LearnQueue? Actually, the memory complexity for our method is $O(m.n.e)$ where $O$ is the big O notation and $m$ is the number of associated queues (excluding broadcast and management queues). We have also $n := |S|$ where $S$ denotes the finite set of states and $e := \sum_{s \in S} |A(s)|$ where $A$ denotes the set of actions that can be executed in the state $s \in S$. For instance, consider that we have five associated STA to an AP with a maximum buffer size of 400 packets for each queue. In this situation, the state space size will be equal to $n = 400$ and the number of associated queues in the AP is $m = 5$. For all states, two actions are possible. Hence, $e = 2$. If we consider storing the $Q$-tables in a signed 32-bit integer array, we need about 16 KB of memory just to store the $Q$-tables (each queue has its own $Q$-table) without considering other variables. This is not negligible for a limited BRAM memory. However, several techniques in the literature provide insights into how to reduce the space complexity of reinforcement learning methods [31]. We can consider such solutions in case we are in shortage of memory. In addition, time complexity is not relevant in our case because our algorithm does not terminate and keeps running in a loop using a scheduler with a resolution of 15 ms. Nevertheless, the computation process includes many operations. For this reason, we avoided any floating point operations in order to optimize the code.

It is unclear how LearnQueue adoption will impact the energy consumption of wireless devices. This is an issue that needs further investigation. The energy challenge for wireless connectivity can only become more stringent in the coming years, and there is a pressing need to develop reliable wireless systems that consume minimum energy [32]. Adapting LearnQueue may increase the utilization of processing resources in the hardware resulting in higher consumed energy. However, power-saving mechanisms can be combined with the queuing routine in the transmission side for a better energy consumption optimization. Typically, a radio is put to sleep only when it has no packets to transmit. The transmit circuitry can be scheduled in coordination with LearnQueue to optimize the sleep time in accordance with the on-going transmissions.

Our method is tested in a typical centralized wireless network scenario in infrastructure mode where all nodes have limited mobility, and all communications should pass through an Access Point. Although LearnQueue was designed for battling bufferbloat phenomenon by altering the buffers dynamics, we believe that our scheme does not interfere with the safe operation of mobile ad hoc network (MANET) that usually has a routable networking environment on top of a Link Layer ad hoc network. In fact, each device in a MANET is free to move independently in any direction, and will, therefore, change its links to other devices frequently. MANETs are equipping each device for the sake of continuously maintain the information required to properly route traffic. We already mentioned that we do not alter multicast or management queues default behavior since our focus targets only data frames queues. Hence, LearnQueue is unlikely to interfere with the routing traffic which resides in the management queue.

Finally, the scaling weights $\delta$ and $\eta$ were presented in this paper as an advantage of LearnQueue in that it allows users to define their own preferences depending on the application type. We thoroughly investigated the behavior of different buffering techniques using various transport protocols such as TCP and UDP. However, we did not focus on their interaction with a specific application. For instance, minimizing the buffer size will bring down the latency, but, it can produce collateral problems for a video streaming application. Streaming video is sensitive to dropping and has strict packet loss requirements. The acceptable packet loss should be between 1% and 2.5%. Higher dropping rates in the transmit queue will automatically result in a poor video quality. Therefore, in such application, we should set the scaling weights of LearnQueue in a way to favor enqueue rate over latency. However, LearnQueue definitely needs an in-depth quantitative study to figure out the optimal weights for each traffic type. This can be done experimentally by trying many scaling weights values and observe the results (as done in part 4.4) or theoretically by considering a model based on TCP fluid-flow and stochastic differential equations originated in [33].

6 Conclusion

In this paper, we introduced LearnQueue, a reinforcement learning based design for controlling bufferbloat in wireless networks. The LearnQueue design periodically tunes the queue size by predicting the reward or the penalty resulting from explored actions along with exploiting the prior experience accumulated. LearnQueue is model-free, and the learning process is fully automated. We have investigated LearnQueue performance in various scenarios with different transport protocols. Experiment results show that our scheme reduces queuing latency drastically while preserving similar throughput in most of the cases. The intellectual merit of this research is to contribute to the efforts of establishing the milestone of next-generation wireless technologies. Our work provides a good starting point for employing artificial intelligence and machine learning techniques to optimize wireless networks performance. As
future work, we intend to investigate the impact of buffering on the energy efficiency of wireless devices. In addition, we plan to perform a sensitivity analysis of LearnQueue scaling weights and find the optimal settings for each application. Finally, we intend to extend LearnQueue by employing more sophisticated exploration/exploitation strategies in correlation with the dynamics of the queue.

ACKNOWLEDGMENTS
This work was funded under grant #AT-35-59 from King Abdulaziz City of Science and Technology.

REFERENCES

Nader Bouacida received his B.Eng. degree in Telecommunications from Higher School of Communication of Tunis (SUPCOM) in 2015. He is currently a MSc student in Computer Science at King Abdullah University of Science and Technology. His research interests include Software-Defined Networking, optimizing wire- less networks performance and developing next- generation wireless network technologies.

Basem Shihada is an Associate and Founding Professor of computer science and electrical engineering in the Computer, Electrical and Mathematical Sciences & Engineering (CEMSE) Division at King Abdullah University of Science and Technology (KAUST). Before joining KAUST in 2009, he was a visiting faculty at the Computer Science Department in Stanford University. His current research covers a range of topics in energy and resource allo- cation in wired and wireless communication net- works, including wireless mesh, wireless sensor, multimedia, and optical networks. He is also interested in SDNs, IoT, and cloud computing. In 2012, he was elevated to the rank of Senior Member of IEEE.