J Netw Syst Manage (2018) 26:486–517
https://doi.org/10.1007/s10922-017-9427-y

A Lightweight Fairness-Driven AQM for Regulating Bandwidth Utilization in Best-Effort Routers

Zawar Hussain[1] · **Ghulam Abbas**[1] · **Zahid Halim**[1]

Received: 6 February 2017 / Revised: 1 September 2017 / Accepted: 5 September 2017 /
Published online: 15 September 2017

Abstract The end-to-end congestion control mechanism of transmission control protocol (TCP) is critical to the robustness and fairness of the best-effort Internet. Since it is no longer practical to rely on end-systems to cooperatively deploy congestion control mechanisms, the network itself must now participate in regulating its own resource utilization. To that end, fairness-driven active queue management (AQM) is promising in sharing the scarce bandwidth among competing flows in a fair manner. However, most of the existing fairness-driven AQM schemes cannot provide efficient and fair bandwidth allocation while being scalable. This paper presents a novel fairness-driven AQM scheme, called CHORD (CHOKe with recent drop history) that seeks to maximize fair bandwidth sharing among aggregate flows while retaining the scalability in terms of the minimum possible state space and per-packet processing costs. Fairness is enforced by identifying and restricting high-bandwidth unresponsive flows at the time of congestion with a lightweight control function. The identification mechanism consists of a fixed-size cache to capture the history of recent drops with a state space equal to the size of the cache. The restriction mechanism is stateless with two matching trial phases and an adaptive drawing factor to take a strong punitive measure against the identified high-bandwidth unresponsive flows in proportion to the average buffer occupancy. Comprehensive performance evaluation indicates that among other well-known

✉ Zawar Hussain
zawar@giki.edu.pk;
http://www.giki.edu.pk/Telecon

Ghulam Abbas
abbasg@giki.edu.pk

Zahid Halim
zahid.halim@giki.edu.pk

[1] Faculty of Computer Sciences and Engineering, GIK Institute of Engineering Sciences and Technology, Topi 23640, Pakistan

AQM schemes of comparable complexities, CHORD provides enhanced TCP goodput and intra-protocol fairness and is well-suited for fair bandwidth allocation to aggregate traffic across a wide range of packet and buffer sizes at a bottleneck router.

Keywords Active queue management · Congestion control · Fair bandwidth allocation · Unresponsive flows

1 Introduction

Internet congestion control comprises two components: an end-to-end transport layer mechanism of TCP, and a router-based AQM scheme. AQM decides how to prevent an impending congestion by providing early warnings to TCP sources before the buffer overflows, while TCP decides how to align its sending rate with the congestion notifications from the AQM [1, 2]. A TCP source is congestion *responsive* since it backs off on receiving a congestion notification and gradually increases its sending rate otherwise. This cooperative behavior of TCP allows similarly situated sources to attain a collective sending rate equal to the capacity of the congestion point and thereby, share the bottleneck bandwidth fairly reasonably [3, 4]. *Fairness* is of a greater concern in the context of the best-effort Internet wherein sources need to compete for the scarce network resources [3–6]. AQM only complements the end-to-end congestion control to increase network utilization and to enable a reasonable degree of fairness among the well-behaved responsive flows. To achieve these objectives though, the Internet depends on sources to supportively deploy end-to-end congestion control mechanisms [7].

However, such a supportive deployment is not always granted by sources since some end-user applications may prefer selfish behavior in order to be more competitive. Such applications generally utilize User Datagram Protocol (UDP), which offers no means of detecting or avoiding the network congestion. As such, UDP-based applications are congestion *unresponsive*—that is, they cannot back off when congestion occurs. Unresponsive flows can quickly capture the bandwidth leftover by responsive flows that back off in response to a congestion. This inflicts unfairness on the well-behaved responsive flows and may also cause congestion collapse [8]. Since the cooperative deployment of congestion control mechanisms universally by all end systems is not practical, the network needs to take control of its own resource utilization. Although, AQM seeks to increase network utilization for responsive flows, it cannot preclude the problem of unfairness without being able to identify and restrict unresponsive flows [7].

To that end, there has been a growing realization that "*fairness-driven*" AQM schemes are inevitable not only to enable a friendly coexistence of aggregate traffic, but also to offer incentives to sources for using congestion control.[1] In the wake of

[1] UDP-based applications can also realize congestion control either through their own application layer mechanism, or by using an alternative transport layer protocol, such as DCCP [9], which is similar to UDP but provides a light-weight congestion control mechanism.

the seminal work of Floyd and Fall [8], and the ensuing calls for more efforts by IETF [3–6], numerous AQMs proposals exist for sharing the scarce network resources among competing aggregate flows in a fair manner. In [7], we have provided a systematic review and the taxonomy of the eminent fairness-driven AQM schemes along with several open issues and design guidelines. Among the important open issues is the fact that the existing fairness-driven AQM schemes are unable to allocate bandwidth fairly and scalably reasonably. The key to the scalability is the accurate identification and restriction of unresponsive traffic without the need to keep too much state information. In addition, most of the existing schemes consider fixed length packets in their designs and evaluations, have large buffer requirements, or cannot cope with intra-protocol and inter-protocol unfairness.

This paper presents an almost stateless fairness-driven AQM, termed CHORD, for best-effort routers to regulate unresponsive flows in wired networks. CHORD is aimed at providing reasonable fairness with the least complexity. For identifying unresponsive flows, CHORD employs a fixed-size cache memory to store the history of recent drops with a state space requirement equal to the size of the cache. For restricting the identified unresponsive flows, CHORD employs the stateless matched-drop framework with two matching trial phases and a drawing factor adapted to take a punitive measure in proportion to the average buffer occupancy. CHORD is light-weight due to its fixed and very small space requirements and is amenable to high-speed implementations in core routers because of its low processing cost. Performance evaluation against eminent AQMs indicates that CHORD provides enhanced fairness for aggregate traffic, ensures intra-protocol and inter-protocol fairness, and retains reasonable performance under packets and buffers of different sizes.

The rest of the paper is organized as follows. Section 2 presents the related work. The proposed CHORD scheme is presented in Sect. 3, along with the complexity analysis of CHORD. Section 4 presents performance evaluation of CHORD, and Sect. 5 concludes the paper.

2 Related Work in Fairness-Driven AQM

An AQM is termed *fairness-driven* if it establishes some form of differential control of responsive and unresponsive traffic for allocating bandwidth fairly. Such a scheme seeks to identify unresponsive traffic on detecting an incipient congestion and applies certain restrictions to confine the bandwidth consumption of unresponsive traffic (see [7] for an overview). Hence, the two crucial components of a fairness-driven AQM scheme are the identification and restriction mechanisms, which compose the *control function* of the scheme. A fairness-driven AQM with a stateful control function (e.g., [10–13]) enforces an exact fairness at the expense of a higher state space and per-packet processing costs. Other AQMs with partial-state (e.g., [14–20]) or with stateless control functions (e.g., [21–29]) seek to achieve reasonable efficiency with moderate or low complexities. The remainder of this

section reviews only the eminent practical schemes that offer a good balance between complexity, fairness, and efficiency.

CHOKe [21] is a stateless AQM with the same parameters as RED [30], i.e., a minimum threshold, min_{th}, a maximum threshold, max_{th}, and the average queue size, avg_q. An arrived packet is queued if $avg_q \leq min_{th}$, and is dropped if $avg_q > max_{th}$. However, when $min_{th} < avg_q \leq max_{th}$, the arrived packet is matched with a randomly drawn (drop-candidate) packet from the buffer. Both the packets are dropped if they have the same *FlowIDs*. This is called the "matched-drop" framework of CHOKe. Otherwise, the arrived packet is marked with probability,p, of RED and the randomly drawn packet is restored in the buffer. This makes CHOKe completely stateless. Nevertheless, CHOKe employs a fixed drawing factor, which is inappropriate since the drawing factor should be adapted in accordance with the congestion level in order to increase the penalty with the growing congestion. The static drawing factor may also hamper the performance of CHOKe in the presence of bursty traffic. The performance of CHOKe may also degrade as the number of unresponsive flows increase [7], due to the reduced likelihood of packets matching and dropping from these flows.

CHOKeR [25] extends CHOKe by incorporating multiple matched drops and using the current instantaneous queue size, q, as the congestion measure. The buffer space is divided into three thresholds, namely, b_{min}, b_{mid}, and b_{max}. The drawing factor, p_0, is updated upon each packet arrival based on q, as follows:

$$p_0 = \begin{cases} max(0, p_0 - pr^-), & if\ b_{min} < q \leq b_{mid} \\ p_0, & if\ b_{mid} < q \leq b_{max} \\ p_0 + pr^+ \cdot \lceil q - b_{max}/b_{max} - b_{mid} \rceil, & if\ b_{max} < q \leq b_{limit} \end{cases}, \quad (1)$$

where b_{limit} is the buffer size and pr^+, pr^- are fixed increase and decrease parameters, respectively. The drawing factor follows an additive decrease, when q is between b_{min} and b_{mid}, but adopts a multiplicative increase to take a more aggressive action when q is larger than b_{max}. The limitation of CHOKeR is that the fixed additive decrease in p_0, and thus the penalty, decreases slowly even when the queue size decreases quickly [7]. Another common limitation to both CHOKeR and CHOKe is that they ignore the location of drop candidates in the queue. To address these limitations, CHORD devises a location and drop history-based adaptive drawing factor to be detailed in the next section.

Other recent extensions of CHOKe include CHOKeD [26], CHOKe-FS [27], CHOKe-RH [28] and LRURC [29]. CHOKeD [26] is a stateless AQM that dynamically divides the queue at each packet arrival into rear and front regions of equal lengths. It then updates its drawing factor as $d_r = round\big(q \cdot \sqrt{B}/(max_{th} - min_{th}) \cdot \ln(B)\big)$, where B is the buffer size, and if $min_{th} < avg_q < max_{th}$, it performs d_r matching trials from the rear region. If the matching trials from the rear region fail, CHOKeD further performs $d_f = round(d_r/2)$ matching trials from the front region. CHOKe-FS [27] divides the queue into four regions. It works in a stateless mode when the congestion level is low to moderate, and in a partial-state mode when the congestion level is severe. In the stateless mode, CHOKe-FS performs matching trials using a drawing factor

whose value increases as the region-wise queue occupancy (instantaneous queue size) increases. The drawing factor is 1 for the queue occupancy in the first region, 2 for the second region, and 3 for the third region. Alternatively, if the queue occupancy is in the fourth region, CHOKe-FS switches to a partial-state mode wherein it estimates a fair-share rate as $f_{share} = B/N_{act}$, and limits high-bandwidth flows to this rate. Here N_{act} is the number of active flows estimated using the direct bitmap technique. CHOKe-RH [28] is an almost stateless algorithm that maintains the history of recently dropped packets in a fixed size cache and uses this history to identify aggressive flows. The scheme consists of initial matching trials and additional trials for the identified unresponsive flows. The drawing factor in the initial trials is set to 3 and is applied on the rear queue if $min_{th} < avg_q < max_{th}$. If any of the initial trials is successful, the dropped flow is searched in the cache. If the flow is found, further d matching trials are performed from the rear queue region. The value of d is initialized to 2 and is doubled if avg_q exceeds half of the buffer size. Different from all other CHOKe-based algorithms, CHOKe-FS offers a location and drop history-based adaptive drawing factor and identification mechanism. LRURC [29] offers a probabilistic control function having five components, namely, Virtual queue, Real queue, Rate check, LRU cache, and Queue manager. LRURC inserts the duplicate of the arrived packet into the Virtual queue in a FIFO manner. According to the arrived packets and their duplicates in the Virtual queue, LRURC uses the matched drop based Rate check and LRU strategy to update the partial flows state in the LRU cache. The Queue manager then uses the LRU cache to manage the Real queue. When $min_{th} < avg_q \leq (min_{th} + max_{th}/2)$, the packet is dropped with probability p_d calculated as $p_d = 0.02 \cdot f_i \Big/ \sum_{j=1}^{n} f_j + 0.005$, where f_i and f_j are the frequencies of identification as a high-bandwidth flow for flow i and j, respectively. This makes high-bandwidth flows have a high dropping probability.

Among the recent non-CHOKe based almost stateless fairness driven AQMs are AFCD [18], Prince [19] and ABC [20]. AFCD [18] employs a synergic approach by forming an alliance between approximated fair queuing and controlled delay queuing. At the enqueue operation, AFCD estimates the sending rate of flows by using a small amount of state information of flows. At the dequeue operation, AFCD calculates a target delay of an individual flow and makes drop decisions for different flows based on the flow's target delay. Heterogeneous flows are shown to be able to acquire an approximated fair bandwidth share in AFCD. Prince [19] adopts a game theoretic approach, where incentive is given to the majority flow by dropping its packets at congestion. In order to find the majority flow, Prince detects the flow with most packets in the queue. By enforcing fair buffer sharing, Prince is shown to achieve fair bandwidth sharing. ABC [20] assigns reference rates to users and their traffic is equipped at the network edge with activity information, which indicates by what factor the transmission rate of a user exceeds its reference rate. ABC uses this activity information to adapt the dropping probabilities inside the network to obtain approximately fair bandwidth allocation.

This paper is a significant extension of our preliminary work [28] and offers an improved identification mechanism, which ensures that only long-standing high

bandwidth unresponsive flows are retained in the cache. This paper also presents a complete description of the proposed algorithm and removes the ambiguities found in [28]. Additionally, this paper presents detailed complexity analysis and extensive performance evaluation of the proposed algorithm, all of which were missing in [28].

3 Active Queue Management by CHORD

As discussed in the previous section, the simplest possible control function is afforded by the matched drop framework of CHOKe that consists of *matching* for identification and *dropping* the matched packets for restricting high bandwidth unresponsive flows. However, performance is largely compromised in the matched drop framework due to the overly simplified control function. The proposed scheme seeks to maximize fair bandwidth sharing among competing responsive and unresponsive flows by complementing both the identification and restriction mechanisms of the matched drop framework, while retaining the simplicity in terms of the minimum possible state space and processing costs. The following sections detail the components of the proposed CHORD scheme and present its complexity analysis.

3.1 Identification Mechanism

For identifying unfair flows more effectively, CHORD employs a small cache memory of a fixed predetermined size δ that seeks to capture the history of recent drops by storing the *FlowIDs* (source–destination address pairs) of the dropped packets. A new entry, i.e., the *FlowID* of the most recently dropped packet is placed at the topmost position, while the oldest entry, i.e., the least recently dropped flow, is placed at the bottom of the cache (hereinafter referred to as the *drop_list*). Note that maintaining a fixed-size *drop_list* is not the same as maintaining a full per-flow state that grows in proportion to the number of active flows being served by a router. The basic purpose of the *drop_list* is to support the restriction mechanism to be detailed in the next subsection, which consists of two matching trial phases. These include an initial phase at each packet's arrival to identify and drop unresponsive flows, and an additional phase to impose an extra penalty on the identified high bandwidth unresponsive flows. The flows recorded in the *drop_list* are the candidates for the extra penalty when congestion occurs. The *drop_list* is used to effectively identify and establish the additional matching trials as follows.

If a successful matched drop occurs in the initial matching trials, the *drop_list* is looked up to determine if the *FlowID* of the dropped flow has already been recorded. If the *FlowID* is present in the *drop_list*, more packets are drawn randomly from the buffer and are compared with the dropped flow to impose an extra penalty. If the *FlowID* is not present, the *FlowID* is added to the *drop_list*. If the *drop_list* is full and there is no space for the new entry, the oldest item is removed to create space for the new entry. Thus, only long-standing high bandwidth unresponsive flows are likely to retain their entries in the *drop_list*. Another

advantage of this identification mechanism is that misbehaving TCP flows, which do not reduce their sending rates despite being dropped, can also be identified. To that end, if a match occurs for a TCP flow in the initial matching trials and its *FlowID* is also present in the*drop_list*, this indicates that the TCP flow is misbehaving and should be restricted in the same way as a high bandwidth UDP flow. This is not possible in other schemes, such as SAC [22] and PUNSI [23], which identify and penalize only UDP flows allowing a misbehaving TCP flow to escape the restriction mechanism.

3.2 Restriction Mechanism

Since unresponsive and misbehaving flows fail to (completely or appropriately) reduce their sending rates, regardless of the congestion notifications from the network, these flows are likely to have more packets in the queue during a congestion epoch. Hence, there is a greater likelihood of matching and dropping of high bandwidth flows under a matched drop framework. In such a framework, a *drawing factor* is used to decide the level of penalty on unresponsive flows by determining the number of drop candidates to be randomly drawn from the buffer to accomplish matched drops. The drawing factor for the restriction mechanism in CHORD consists of multiple matched drops, which involve drawing multiple random drop candidate packets from the buffer for performing the matching trials. The challenge, however, is deciding the appropriate number of drop candidates for the multiple matched drops. With a static drawing factor, such as that of CHOKe, fairness may be hampered in the presence of bursty traffic, and with a fixed-decrease drawing factor, such as that of CHOKeR, the penalty decreases slowly even when the queue size decreases quickly [7]. To overcome these limitations, CHORD uses the *drop_list* and the average buffer occupancy to update its drawing factor dynamically as follows.

As mentioned previously, CHORD consists of two matching trial phases. The drawing factor in the initial matching trial phase is denoted by d_i and its value is set to 3 (the justification for this will be given in the next section). The drawing factor for the additional matching trial phase is denoted by d_x and its value is the function of the average buffer occupancy. The additional matching trial phase comes into play when a matched drop occurs in the initial phase and the dropped flow is also present in the *drop_list*. Thus, d_x signifies an extra penalty on unresponsive flows and its value is set to 4 if the average buffer occupancy exceeds half of the total buffer size and is otherwise halved.

Generally, in the matched drop frameworks, such as those of CHOKe and CHOKeR, drop candidates are drawn randomly from any position in the queue. Thus, there is an equal likelihood for each packet to get selected for a matching trial. However, it is known that high bandwidth unresponsive and aggressive flows tend to accumulate at the rear queue region when congestion occurs [7, 22, 23]. This is because a flow with a higher transmission rate is likely to have its packets clustered at the rear queue end during a congestion epoch. Therefore, as opposed to the whole queue, CHORD preferentially draws drop candidates from the rear half of the instantaneous queue, denoted as q_{rear}. Figure 1 depicts one such instance of q_{rear},

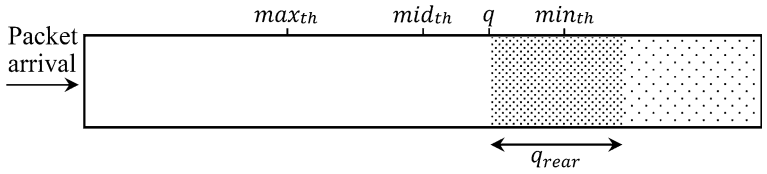

Fig. 1 An instance of the rear and front halves of the instantaneous queue under CHORD

where q is the current instantaneous queue size. In this way, high bandwidth flows can be restricted more effectively since there is a greater likelihood that packets from these flows will get selected for matching trials and will get dropped.

3.3 Complete Algorithm Description

CHORD maintains a buffer at each output port of a router for queueing the packets of the flows that share an outgoing link in a wired network, as shown in Fig. 2. Packets get admitted from the input end when there is space available in the queue and are transmitted on the outgoing link from the output end in a FIFO manner. The only required observables for CHORD are the buffer occupancy and *FlowIDs*, while other observables such as the number of active flows are not required by the algorithm. CHORD employs average buffer occupancy, called the average queue size, avg_q, as a congestion measure. This is because if traffic arrives in bursts, avg_q does not enlarge abruptly as opposed to the current instantaneous queue size, q. Therefore, matched drops are initiated only on the basis of avg_q because the instantaneous queue size is too abrupt to initiate the matched drops. The average queue size is determined using an exponential moving average of q at each packet arrival, as [30]:

$$avg_q = \left(1 - w_q\right) \cdot avg_q + w_q \cdot q, \tag{2}$$

where w_q is the queue weight. CHORD marks three thresholds on the average queue size to determine the level of congestion. These include a minimum threshold, min_{th}, a midpoint threshold, mid_{th}, and a maximum threshold, max_{th}, as depicted in Fig. 1. If the total arrival rate at the queue is less than the outgoing link capacity, avg_q should not build-up to the minimum threshold, indicating that there is no congestion in the network. Thus, for each packet arrival, if the average queue size is less than min_{th}, CHORD admits the arrived packet into the queue. However, if avg_q

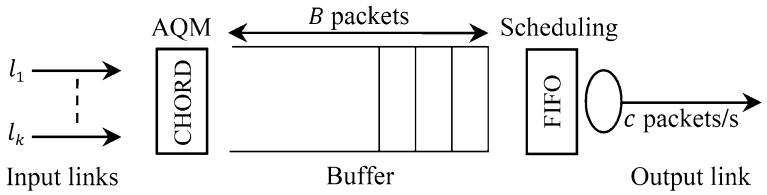

Fig. 2 An output-queued CHORD router with a single buffer stage

exceeds min_{th}, this is an indication of growing congestion. To help reduce the congestion by identifying and restricting unresponsive flows, CHORD draws $d_i = 3$ drop-candidate packets randomly from q_{rear} to perform the initial matching trials. If none of the drop-candidates matches the arrived packet, CHORD handles this packet in the same way as in RED. The arrived packet is marked with probability p and the packets drawn from the buffer are restored at their actual positions. The probability p is a piecewise linear function of the average queue size, computed as in RED.

Alternatively, if the *FlowID* of the arrived packet matches any or all of the drop-candidates, the arrived packet and all drop-candidates are dropped. The *drop_list* is updated, as detailed in Sect. 3.1, and the drawing factor is updated based on the average queue size, as detailed in Sect. 3.2. The update procedure of the complete drawing factor, d, at each packet arrival can be given as:

$$d = \begin{cases} d_i, & \text{if } min_{th} \leq avg_q < max_{th} \\ d_i + d_x/2, & \text{if } Hit \text{ AND } avg_q \leq mid_{th} \\ d_i + d_x, & \text{if } Hit \text{ AND } avg_q > mid_{th} \end{cases}, \tag{3}$$

where *Hit* denotes a successful initial matching trial(s) and an occurrence of a match in the *drop_list*. A *Hit* is an indication of a highly aggressive or high bandwidth unresponsive flow since it is not reducing its transmission rate despite being dropped previously. The identified flow then becomes a candidate for an extra penalty, which is imposed by drawing d random drop candidates from q_{rear} for performing the additional matching trials. If any or all of the drop candidates have the same *FlowID* as that of the arrived packet, all matched packets are dropped. Otherwise, the packets drawn from the buffer are restored at their original positions in the queue.

Though it is of much significance for fairness-driven AQMs to enforce fairness, it is more important for these AQMs to prioritize efficiency over fairness [7]. Hence, when the demand for bandwidth at the output queue is well within the link capacity, no flow is considered unresponsive and CHORD suspends its identification and restriction mechanisms. Alternatively, if $avg_q \geq max_{th}$, every incoming packet is dropped to allow the average queue size to reduce below max_{th} as early as possible. The complete procedure of CHORD is described in Algorithm 1,[2] and its state space and per-packet processing costs are given in the next subsection.

[2] The ns-2 implementation of CHORD is available at https://github.com/fairness-driven-AQM/CHORD.

Algorithm 1. CHORD

// $Draw\ (\cdot)$ – returns a packet sampled from q_{rear}
// $FlowID(\cdot)$ – returns the source destination address pair
// $match$ – flag to denote a successful matching trial
// pk' – a drop candidate
// $Purge\ (\cdot)$ – removes the oldest entry from $drop\_list$

For each packet , pk, arrived at the queue
begin
$match \leftarrow 0,\ d_i \leftarrow 3,\ d_x \leftarrow 4$
$mid_{th} \leftarrow round\ ((min_{th} + max_{th})/2)$
$avg_q \leftarrow (1 - w_q)avg_q + q.w_q$
if $avg_q \geq max_{th}$ **then**
  drop pk
else if $avg_q \geq min_{th}$ **then**
  while $d_i > 0$ **do**
    $d_i \leftarrow d_i - 1$
    $pk' \leftarrow Draw(rand\ (q/2) + q/2)$
    if $FlowID(pk') = FlowID(pk)$ **then**
      drop pk'
      $match \leftarrow 1$
    end if
  end while
  if $match = 0$ **then**
    admit pk with p
  else
    drop pk
    if $(FlowID(pk) \notin drop\_list)$ **then**
      if $drop\_list$ is full **then**
        $Purge(drop\_list)$
      end if
      $drop\_list \Leftarrow FlowID(pk)$
    else
      if $avg_q \leq mid_{th}$ **then**
        $d_x \leftarrow d_x/2$
      end if
      while $d_x > 0$ **do**
        $d_x \leftarrow d_x - 1$
        $pk' \leftarrow Draw(rand\ (q/2) + q/2)$
        if $FlowID(pk') = FlowID(pk)$ **then**
          drop pk'
        end if
      end while
    end if
  end if
else admit pk
end if
end

3.4 Complexity Analysis

An AQM scheme works at the enqueue-end of a router's buffer and is usually actuated for each arrived packet to make a decision based on the level of congestion to either admit or drop the arrived packet [1]. The implementation cost of an AQM is decided by its complexity—that is, the amount of state space required and the amount of processing performed on that state [7]. The complexity of an AQM scheme determines its scalability, which must be taken into account carefully to enable the scheme to retain performance for large-scale flows and be amenable to

high-speed implementations [31]. An AQM with stateless control function does not have any space requirements and has a $O(1)$ cost indicating that, at each packet arrival, it requires a fixed amount of processing irrespective of the number of flows being *served* by the router. On the contrary, the space requirements of the per-active flow (stateful) AQMs, such as [10–13], grow with the growing number of flows being served. Consequently, the per-packet-processing costs of the stateful AQMs are too high, which prevent scalability and limit the deployment of the stateful AQMs in core networks containing a myriad of flows. Similarly, as partial-state AQMs such as [14–20] maintain the state for only a subset of flows, the space complexities of these AQMs grow with the increasing number of flows being *recorded*, whereas the amounts of processing depends on the design of their control functions. Such complexities may also make scaling difficult [7]. The key to the scalability is to have accurate identification and restriction mechanisms without maintaining too much state information. Table 1 compares the control functions and complexities of RED, CHOKe, CHOKeR and CHORD. The complexity analysis of CHORD are given below.

3.4.1 State Space Requirement

CHORD can be deployed in the conventional output queued RED routers by using a small cache memory and a control function at the input end of the FIFO buffer. The control function consists of three steps on a packet arrival when congestion occurs (1) initial matching trials for identification, (2) looking up and updating the *drop_list*, and (3) updating the drawing factor and performing additional matching trials for restricting unresponsive flows. The state keeping for the identification requires a space of $O(\delta)$, where δ is the size of the *drop_list*. Although, this space complexity is higher than a stateless scheme such as CHOKeR, CHORD remains scalable since the size of *drop_list* is limited and the space requirement does not grow with the growing number of active flows. The next section will demonstrate that configuring CHORD with $\delta = 10$ provides reasonable performance. Thus, no more than 10 *FlowIDs* need to be stored at any given time in CHORD. Using 32-bit IP addressing and storing 10 source–destination address pairs impose a *drop_list* size requirement of only a constant 640 bits, which is a very small memory overhead.

3.4.2 Per-Packet Processing Costs

CHORD can enqueue, drop or match drop an arrived packet, based on the level of congestion. When avg_q is less than min_{th}, CHORD admits the arrived packet, and when avg_q is greater than max_{th}, it drops the arrived packet. The decision to enqueue or drop the packet involves a simple operation of checking the average queue size. The per-packet processing cost in both these cases is $O(1)$, which is the best case complexity implying that CHORD will perform the same amount of processing for every incoming packet. When avg_q is greater than min_{th}, CHORD performs the initial matching trials. The average per-packet processing cost in this case is $O(d_i)$,

Table 1 Comparison of complexities

AQM scheme	Control function	State space requirement	Processing performed at	Per-packet processing cost
RED	Uses avg_q as a congestion measure. Admits an arrived packet if $avg_q < min_{th}$ and marks (or drops) the packet if $avg_q \geq max_{th}$. Computes probability, p_a, as follows and marks/drops an arrived packet with p_a if $min_{th} \leq avg_q < max_{th}$. $p_b = max_p(avg - min_{th})/(max_{th} - min_{th}),\ p_a = p_b/(1 - count \cdot p_b)$, where max_p is the maximum value of p_b and *count* is the number of packets since last marked/dropped packet	None	Each packet arrival	$O(1)$
CHOKe	Performs RED queue management along with a single matched drop if $min_{th} < avg_q \leq max_{th}$	None	Each packet arrival	$O(1)$
CHOKeR	Measures congestion with the current queue size, q. Divides the buffer space into three thresholds, b_{min}, b_{mid} and b_{max}. Updates p_0 and performs matched drops when $b_{min} < q \leq b_{limit}$	None	Each packet arrival	$O(p_0)$
CHORD	Performs RED queue management along with initial mating trials when $avg_q \geq min_{th}$, looks up and updates the*drop_list* and updates d for performing additional matching trials	$O(\delta)$	Each packet arrival	$O(d)$

where $d_i$is the drawing factor for the initial matching trials. If any of the initial matching trials is successful, the *drop_list* is looked up for the *FlowID* of the dropped flow. The processing cost of the lookup operation is constant time since the same amount of processing is performed for searching a fixed-size *drop_list*. Managing the *drop_list* (addition and deletion of entries in the cache) also has a constant time complexity, as the *drop_list* size is fixed. The worst case complexity of CHORD is $O(d)$, which represents the total number of matched drops carried out on each packet arrival when an additional penalty is applied to an identified unresponsive flow found in the *drop_list* and when avg_q exceeds mid_{th}.

Note that even the worst case complexity of CHORD is much less than the complexities of the stateful and most partial-state and stateless AQMs (see [7] for an overview of the complexities of well-known AQM schemes). For instance, as shown in Table 1, the per-packet processing cost of CHOKeR is $O(p_0)$, which is higher than that of CHORD's $O(d)$. The value of d ranges between 3 and 7, as shown in Eq. (3), and it never exceeds 7. However, the CHOKeR's drawing factor, p_0, increases multiplicatively at each packet arrival if the congestion is high, as shown in Eq. (1), and it can also exceed 7 in a long standing full queue scenario (e.g., after 800 back-to-back packets arrived at a queue that remains 85% occupied, p_0 will be 8, taking b_{mid} and b_{max} as originally defined in [25] and listed in Table 2 below).

On the one hand, the per-packet processing cost of CHOKeR is higher than that of CHORD. On the other hand, although the space requirement of the proposed CHORD is fixed and very small (640 bits), this requirement is still higher than that of CHOKeR, which is stateless. With this slight tradeoff, CHORD can achieve a performance gain over CHOKeR. The argument of CHORD is hence, that a moderate to significant performance gain can be achieved over CHOKeR with a

Table 2 Simulation parameters

Parameter	Configuration
Packet size	1 Kbyte
Buffer size, B	100 packets
min_{th}	$B/3$
max_{th}	$2min_{th}$
w_q (RED)	0.002
max_p (RED)	0.02
d_i (CHORD)	3
δ (CHORD)	10
b_{min} (CHOKeR)	$0.2B$
b_{mid} (CHOKeR)	$0.25B$
b_{max} (CHOKeR)	$0.35B$
pr^+ (CHOKeR)	0.002
pr^- (CHOKeR)	0.001
Maximum window size	300 segments
Simulation time	500 s

reduced per-packet processing cost and a very small state space overhead, as shall be demonstrated in the next section.

4 Performance Evaluation

This section presents performance analysis of CHORD in comparison with RED, CHOKe and CHOKeR, using ns–2 (version 2.35) simulations. Unless otherwise stated, all simulations are based on the parameters listed in Table 2 and on the topology depicted in Fig. 3. All end-systems are connected to the routers through 1000 Mbps links having a small propagation delay of 1 ms. All sources share a bottleneck link that exists between the routers. The capacity of the bottleneck link is 100 Mbps with a 10 ms propagation delay. The queue thresholds for RED, CHOKe and CHORD are set according to the recommendations in [21, 30]. The queue thresholds of CHOKeR are the same as originally proposed in [25]. Unless otherwise stated, all UDP flows employ CBR traffic with a transmission speed of 100 Mbps, and all TCP flows employ FTP. The performance evaluation metrics include the drop rate of unresponsive flows, throughput, fairness, goodput, intra-protocol and inter-protocol fairness, effects of different packet and buffer sizes, and the effects of the *drop_list* size and the drawing factor on the fairness of CHORD. All results are based on 20 replicated simulation runs for each scenario to obtain a 95% confidence interval. The graphs plot the mean values and omit confidence intervals for clarity.

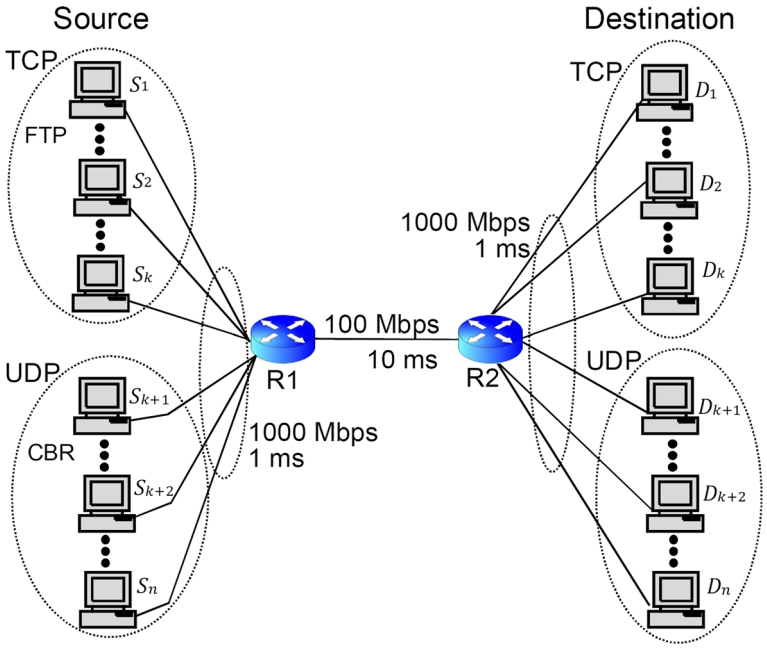

Fig. 3 Dumbbell topology (S_i transmits to D_i, $\forall i = 1, \ldots, n$)

4.1 Drop Rate of Unresponsive Flows

In this subsection, we evaluate CHORD for its ability to restrict unresponsive flows. To that end, we study the drop rate of the UDP packets in three different scenarios by using three different combinations of responsive and unresponsive flows, as shown in Fig. 4. The drop rate of UDP is the lowest under RED. This is because RED does not attempt to implicitly or explicitly identify unresponsive flows. CHOKe drops a comparatively larger number of UDP packets, but its performance degrades with an increasing number of UDP flows. This is because, as the number of UDP flows increases, the likelihood of packets matching and dropping reduces under the single matched drop framework of CHOKe. The performance of both CHOKeR and CHORD is superior to CHOKe. However, as compared to CHORD, CHOKeR drops a large number of UDP packets under all scenarios. This is due to the drawing factor of CHOKeR, which increases multiplicatively causing a much larger number of UDP packet drops at each packet arrival (see Sect. 3.4.2). Additionally, due to the constant decrease in CHOKeR's drawing factor, the penalty decreases slowly even when the queue size reduces quickly. This causes a rather severe punishment of UDP flows under CHOKeR, as shall be demonstrated in the next subsection. On the other hand, the minimum number of matching trials in CHORD is 3 and the penalty increases to a maximum of 7 trials if the average queue size exceeds mid_{th} and the flow is found in the *drop_list*. However, the drawing factor, and hence the penalty, diminishes quickly if the flow was not recently dropped and has vanished from the *drop_list*. This prevents unresponsive flows from getting punished severely under CHORD. The next subsection studies the impact of the UDP drop rate on throughput and link utilization.

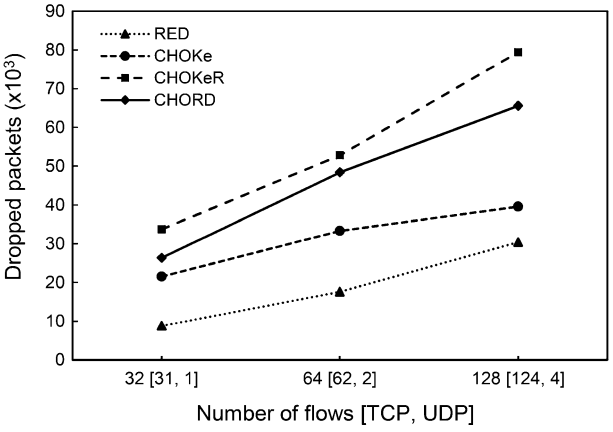

Fig. 4 Drop rate of UDP packets

4.2 Throughput

Throughput is the amount of data successfully received at the destination per unit time [16]. As unresponsive flows may starve well-behaved flows for bandwidth at a bottleneck link, throughput (achieved by different flows) can be used to assess an AQM for its ability to allocate bandwidth fairly. We first evaluate the performance of CHORD for *per-flow* throughput in a network where 31 TCP flows compete for bandwidth with a single unresponsive flow. In the considered scenario, the unresponsive flow captures almost the entire bottleneck link capacity and starves all other responsive flows for bandwidth. In such a case, a fairness-driven AQM should identify the unresponsive flow and restrict it by dropping its packets (according to certain criteria) in order to spare a good part of the bandwidth for responsive flows. The drop rate, however, should be such that *all* unresponsive and responsive flows are made to attain throughputs nearer to the fair share. The drop rate must not be too high for an unresponsive flow that it is completely shutout or is made to suffer unfairness.

Figure 5 presents comparisons of the ideal and actual bandwidth shares received by each flow under all AQM schemes. Flows 1–31 are TCP flows, while Flow 32 is a UDP flow. The ideal fair share in the considered scenario is 3.125 Mbps. Let us consider the scenario with 31 TCP flows and 1 UDP flow in Fig. 4. Since the drop rate of UDP packets is the lowest in case of RED in Fig. 4, the single UDP flow is

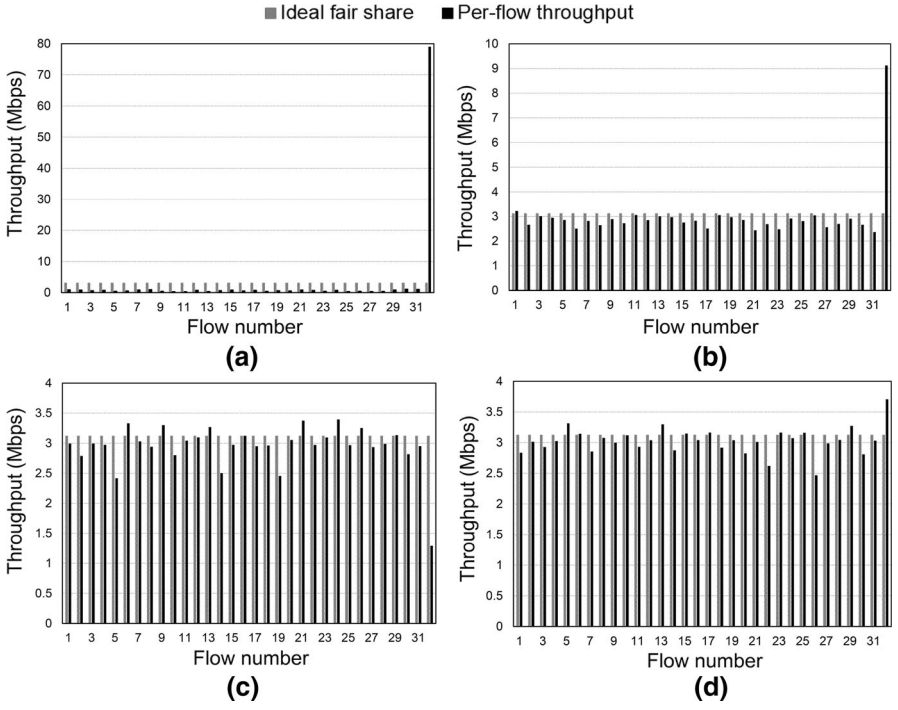

Fig. 5 Per-flow throughputs under **a** RED, **b** CHOKe, **c** CHOKeR, and **d** CHORD

able to capture almost the entire bottleneck bandwidth leaving the TCP flows with a negligible share, as shown in Fig. 5a. For 31 TCP flows and 1 UDP flow in Fig. 4, the drop rate under CHOKe is higher than that under RED. Thus, as compared to RED, CHOKe is able to spare more bandwidth for TCP flows, which are able to achieve throughput of more than 2 Mbps, as shown in Fig. 5b. However, the unresponsive flow still manages to grab a throughput of 9.12 Mbps, which is almost thrice the fair share. Similarly, for 31 TCP flows and 1 UDP flow in Fig. 4, the drop rate under CHORD is higher than that under CHOKe. Due to this higher drop rate of UDP packets, more bandwidth is available for TCP flows. This enables TCP flows to achieve throughputs closer to their fair-share under CHORD, as shown in Fig. 5d. Conversely, the UDP drop rate in case of CHOKeR in Fig. 4 is higher than that under CHORD. This higher-than-required drop rate is undesirable as it results in over-throttling of UDP traffic causing its throughput to fall below the fair share, as shown in Fig. 5c. Throttling unresponsive flows beyond the fair share also causes unfairness, as shall be demonstrated in Sect. 4.3. The drop rate of UDP traffic should be such that *all* TCP and UDP traffic is closer to its fair share rate. The ability to allocate bandwidth closer to fair share will be quantified and compared in the later subsections, where it will be shown that the identification and restriction mechanisms of CHORD are designed such that the drop rate of unresponsive flows is appropriately configured to yield higher fairness.

4.2.1 Multiple Unresponsive Flows

To demonstrate the ability of CHORD to retain its performance in the presence of multiple unresponsive flows, we consider a scenario with 62 TCP flows competing for bandwidth at the bottleneck link with 2 UDP flows of 100 Mbps each. Figure 6 presents a comparison of the cumulative TCP and UDP throughputs as well as the link utilization under all AQM schemes for the scenario considered. Link utilization is the maximum under RED. UDP throughputs under both RED and CHOKe are very high as compared to the ideal throughput. Both CHOKeR and CHORD provide

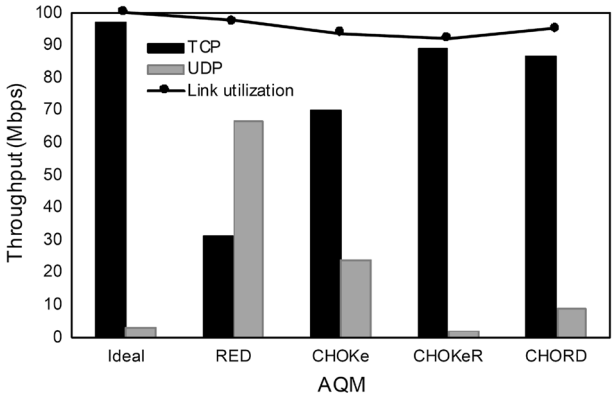

Fig. 6 Throughput under two unresponsive sources with similar sending rates

TCP throughputs closer to the ideal throughput. However, UDP flows again receive a rather severe punishment under CHOKeR. Conversely, the link utilization under CHORD is higher than both CHOKe and CHOKeR.

We further evaluate the performance of CHORD in a network where 30 TCP flows compete for the bottleneck bandwidth with 10 UDP flows of 100 Mbps each. The bottleneck link capacity in this scenario is increased to 400 Mbps to accommodate the larger number of high-bandwidth UDP flows. The results are presented in Fig. 7. As compared to the results presented in Fig. 6, the performances of all AQM schemes have degraded. This happens due to the reduced likelihood of flows match dropping when the number of unresponsive flows is large. TCP flows are deprived of their fair share and receive very low throughput under all AQM schemes, and are almost shut out under RED. CHOKe is unable to deal with the large number of UDP flows due to its static drawing factor. TCP throughputs under both CHOKeR and CHORD are higher than those under RED and CHOKe. However, link utilization under CHORD is higher than that under CHOKeR and lower than those under RED and CHOKe. For the scenarios considered in Figs. 6 and 7, the drop rate of UDP packets is shown in Fig. 8.

To further evaluate the throughput performance of CHORD, we consider scenarios with large numbers of flows. Table 3 presents the results of cumulative TCP and UDP throughputs along with link utilization for 100, 300, and 500 flows. In the scenarios given in Table 3, UDP flows transmit at 15 Mbps and constitute 12% of the overall traffic [16, 32]. The bottleneck link capacity in these scenarios is increased to 1000 Mbps to accommodate the large number of flows. As shown in Table 3, both the TCP throughput and link utilization degrade under all AQM schemes with the increasing number of flows. As in Figs. 6 and 7, the TCP throughputs under both CHOKeR and CHORD are higher than those under CHOKe and RED, but the link utilization is the highest under RED in all scenarios. However, the fairness under RED is the lowest, as shall be seen in Sect. 4.3. Conversely, the link utilization is low under both CHOKeR and CHORD because of

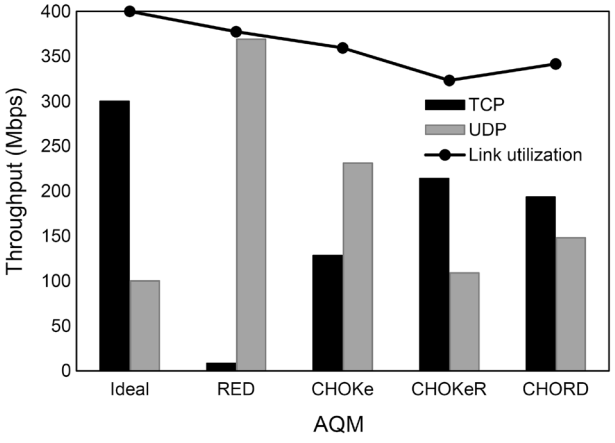

Fig. 7 Throughput under multiple unresponsive sources with similar sending rates

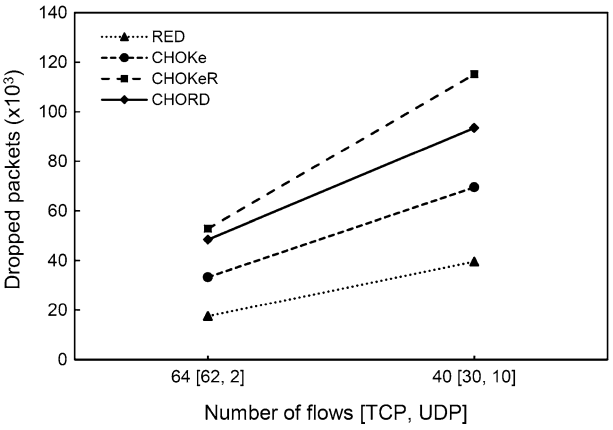

Fig. 8 Drop rate of UDP packets

Table 3 Throughput and link utilization under a large number of flows

AQM	No. of flows	No. of TCP flows	No. of UDP flows	TCP throughput (Mbps)	UDP throughput (Mbps)	Link utilization (Mbps)
RED	100	88	12	296.5	667.8	964.3
CHOKe				689.9	258.7	948.6
CHOKeR				827.0	92.40	919.4
CHORD				810.6	136.1	946.7
RED	300	264	36	132.4	810.1	942.5
CHOKe				488.2	402.5	890.7
CHOKeR				677.1	148.5	825.6
CHORD				653.2	227.1	880.3
RED	500	440	60	72.92	828.5	901.4
CHOKe				241.3	587.8	829.1
CHOKeR				492.1	271.1	763.2
CHORD				561.3	255.6	816.9

the higher drop rates, but fairness under both these algorithms is much higher than RED, as shall be demonstrated in the later subsections. Under CHORD, the link utilization is higher than that under CHOKeR. The UDP throughput under CHORD is also higher than that under CHOKeR in most scenarios. However, the TCP throughput under CHOKeR is higher than that under CHORD in most scenarios. This happens due to the over-throttling of UDP flows under CHOKeR, which leads to low fairness of CHOKeR as compared to CHORD, as shall be demonstrated in Sect. 4.3.

4.2.2 Unresponsive Flows with Different Sending Rates

We next evaluate the performance of CHORD in a network with 30 TCP flows and 2 UDP sources transmitting at 100 and 50 Mbps, respectively. The results are presented in Fig. 9. The performance trend of RED, CHOKe and CHORD is similar to that shown in Fig. 6. In CHOKeR, however, the UDP flow with a higher transmission rate receives more punishment than the one with the smaller transmission rate. The link utilization of CHORD is higher than both CHOKe and CHOKeR. Thus, CHORD can retain its performance in the presence of multiple UDP sources with different sending rates.

4.3 Fairness

Achieving fair bandwidth allocation is the primary goal of fairness-driven queue management that seeks to ensure that *all* responsive and unresponsive flows receive a fair share of the bottleneck capacity when congestion occurs. In Fig. 5d, it is shown that CHORD is able to share bottleneck bandwidth reasonably equitably among TCP and UDP flows. In this subsection, we quantify this ability by means of the Jain's fairness index, *JFI*, and compare it with those of the other AQM schemes. The *JFI* is given as [33]:

$$JFI = \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n \sum_{i=1}^{n} x_i^2}, \tag{4}$$

where x_i is the throughput of flow i. The value of *JFI* ranges from 0 to 1, with 1 denoting a completely fair allocation and 0 indicating a completely unfair allocation.

We first evaluate the fairness of all AQM schemes in a network with only TCP traffic. To that end, a total of six scenarios are considered, each with a different number of TCP flows ranging from 20 to 120. The results are shown in Fig. 10 that

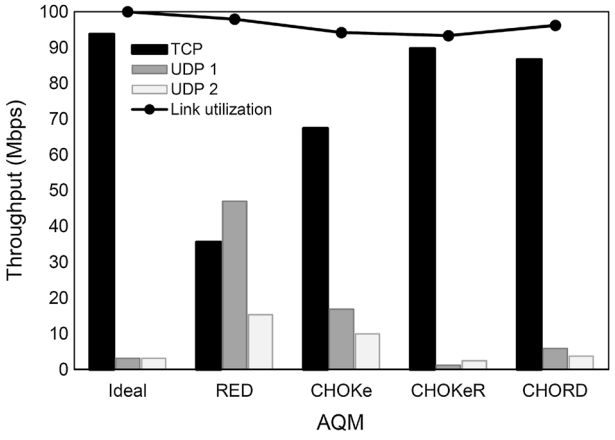

Fig. 9 Throughput under unresponsive sources with different sending rates

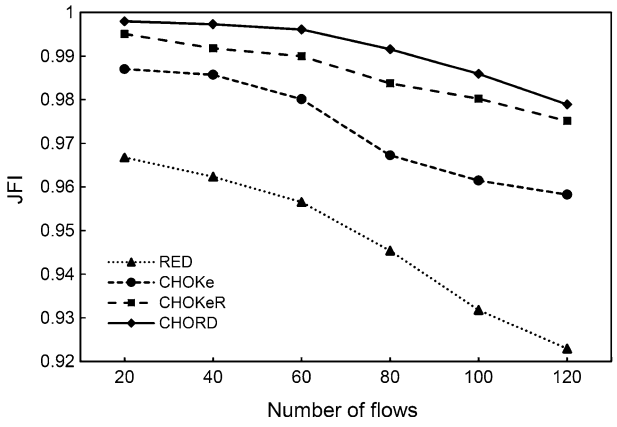

Fig. 10 Fairness for TCP traffic

plots the *JFI* of all AQM schemes for the scenarios considered. As compared to RED and CHOKe, the *JFIs* of CHOKeR and CHORD do not degrade much with the increasing number of flows. However, due to the ability of CHORD to provide all flows with throughputs closer to the fair share, its fairness is also superior to the other AQMs.

To analyze fairness further, we consider a number of scenarios with a traffic mix of TCP and UDP flows, as given in Table 4. The table presents cumulative TCP throughputs as well as *JFI* for all the scenarios considered. As indicated by the results shown in Table 4, the performance of RED has degraded, as compared to that shown in Fig. 10, due to the presence of unresponsive flows. CHOKe is able to provide higher throughput and fairness as compared to RED for a smaller number of UDP flows. However, its performance degrades as the numbers of unresponsive flows increase. Both CHOKeR and CHORD perform better than CHOKe and RED and their performances remain relatively stable. In most scenarios, however, CHORD demonstrates a superior performance than CHOKeR.

4.4 Goodput

Goodput is the amount of useful (non-duplicate) bits received at the destination per unit time [16]. As throughput may contain duplicate bits, goodput becomes convenient in evaluating the network performance for useful bandwidth utilization. We evaluate CHORD in this subsection for goodput under a number of scenarios given in Table 5. For TCP flows, the per-flow goodputs are determined from the per-flow throughputs not including retransmissions. The average TCP goodput is then determined from the aggregate per-flow goodputs divided by the number of the TCP flows. As indicated by the results shown in Table 5, the cumulative TCP goodput is low as compared to the cumulative TCP throughput given in Table 4 (comparing, e.g., the scenarios with 100 flows). The goodputs of all AQM schemes improve as the bottleneck link capacity is increased and when some or all of the UDP flows have sending rates lower than 100 Mbps. As evident from the results in

Table 4 TCP throughput and fairness for traffic mix of TCP and UDP flows

AQM	No. of flows	No. of TCP flows	No. of UDP flows	TCP throughput (Mbps)	Jain's Fairness Index
RED	20	17	3	27.33	0.412
CHOKe				58.62	0.773
CHOKeR				77.27	0.925
CHORD				73.91	0.931
RED	40	35	5	21.82	0.374
CHOKe				55.03	0.601
CHOKeR				76.75	0.815
CHORD				78.83	0.857
RED	60	53	7	18.75	0.319
CHOKe				45.53	0.502
CHOKeR				71.82	0.644
CHORD				75.59	0.721
RED	80	70	10	10.40	0.207
CHOKe				37.39	0.372
CHOKeR				56.45	0.446
CHORD				61.19	0.533
RED	100	88	12	4.796	0.129
CHOKe				32.71	0.291
CHOKeR				48.89	0.392
CHORD				55.17	0.468
RED	120	105	15	2.690	0.071
CHOKe				21.66	0.163
CHOKeR				39.95	0.261
CHORD				43.03	0.373

Table 5, TCP achieves the lowest cumulative goodput under RED and, in most cases, the highest under CHORD.

4.5 Intra-protocol Fairness

Intra-protocol unfairness occurs among TCP traffic when flows having different round-trip times (RTTs) share a bottleneck link. In such a case, TCP flows with a shorter RTTs can receive a larger share of the bandwidth than those with longer RTTs [7]. To evaluate the performance of CHORD in terms of fair bandwidth allocation to TCP flows of diverse RTTs, we generate a range of RTTs using the guidelines given in [34], as follows. We consider four different scenarios, each with the total number of TCP flows ranging from 500 to 2000, respectively. For each scenario, the sources are split into three classes. Each class is then configured with an RTT of 4, 98 and 200 ms, respectively. Figure 11 presents the results of intra-

Table 5 Goodput under different UDP sending rates

AQM	No. of flows	TCP flows	UDP flows	UDP sending rates	Bottleneck capacity	Goodput (Mbps)
RED	100	88	12	100 Mbps	100 Mbps	1.708
CHOKe						18.53
CHOKeR						42.92
CHORD						46.56
RED	300	264	36	25% at 50 Mbps, 75% at 100 Mbps	300 Mbps	32.59
CHOKe						88.44
CHOKeR						145.5
CHORD						138.6
RED	500	440	60	50% at 50 Mbps, 50% at 100 Mbps	300 Mbps	26.27
CHOKe						83.81
CHOKeR						132.3
CHORD						135.4
RED	700	616	84	75% at 50 Mbps, 25% at 100 Mbps	300 Mbps	18.01
CHOKe						67.43
CHOKeR						116.6
CHORD						123.8
RED	1000	880	120	50 Mbps	300 Mbps	4.240
CHOKe						30.05
CHOKeR						78.22
CHORD						87.56

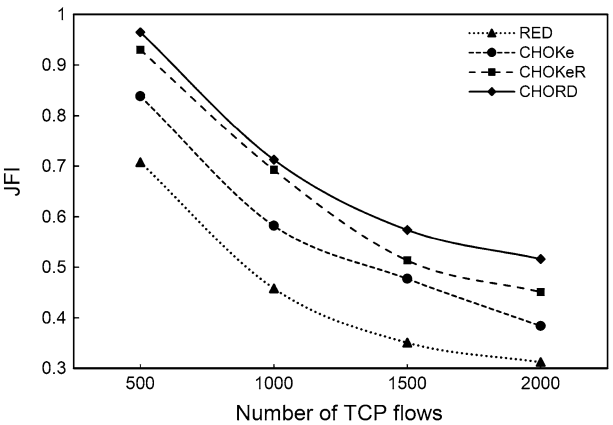

Fig. 11 Intra-protocol fairness among TCP flows of diverse RTTs

protocol fairness. The performances of all AQMs degrade as the number of flows increases. However, as compared to all other AQMs, CHORD demonstrates a higher intra-protocol fairness. The reason for this is because, as flows with shorter RTTs consume larger portions of the buffer space, they are more likely to be recorded in the *drop_list* and are more likely to be dropped.

We now evaluate intra-protocol fairness exclusively among UDP flows having different transmission rates. To that end, we consider four scenarios, each with the total number of UDP flows ranging from 100 to 400, respectively. In each simulation, half of the UDP flows have the sending rate of 100 Mbps, while the remaining half have the sending rate of 50 Mbps. The bottleneck link capacity in this simulation is increased to 1000 Mbps to accommodate the large numbers of high-bandwidth flows. Figure 12 presents the fairness results. Similarly to the intra-TCP fairness, the intra-protocol fairness among the UDP flows is also superior with CHOKeR and CHORD and their performances do not degrade abruptly as the number of flows increases. However, unlike all previous results, the performance of CHOKeR remains superior to CHORD for the intra-UDP fairness for 100 and 400 UDP flows and remains almost similar to CHORD in the case of 200 UDP flows.

4.6 Inter-protocol Fairness

Inter-protocol unfairness occurs when different TCP variants having different congestion control mechanisms coexist with each other [7]. For instance, TCP Reno employs packet loss to determine the available bandwidth, whereas TCP Vegas employs variance between the expected and actual throughputs. Hence, TCP Vegas is more conservative as it enables sources to obtain a proper bandwidth, while TCP Reno is more aggressive as each source grabs the bandwidth until multiple packets are lost. Consequently, when traffic from both the TCP variants coexist, TCP Vegas may suffer inter-protocol unfairness.

To evaluate the performance of CHORD for inter-protocol fairness, we consider a number of scenarios with a traffic mix of TCP Reno and TCP Vegas using the

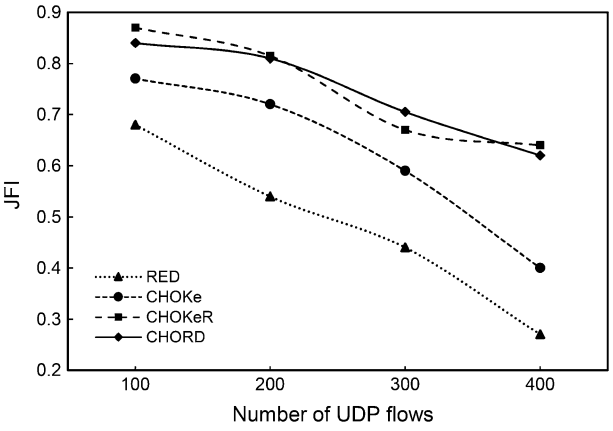

Fig. 12 Intra-protocol fairness among UDP flows with different sending rates

parking lot topology shown in Fig. 13. There are three bottleneck links, namely R1–R2, R2–R3, and R3–R4, each with a buffer size of 200 packets. The bandwidth and delay of the bottleneck and access links are given in Fig. 13. All TCP traffic is from S_i to D_i $(\forall i = 1, 2, \ldots, n)$, and the packet sizes are 1 Kbytes. All TCP sources employ FTP and have a similar round trip propagation delay of 98 ms. Additionally, three UDP flows with a sending rates of 20 Mbps and packet sizes of 500 bytes are established from C_i to C_{i+1} $(i = 1, 2, 3)$, as shown in Fig. 13. In this subsection, the performance of CHORD is also compared with AFCD [18]. All AQM schemes to be compared are deployed in the bottleneck routers. The results are presented in Table 6, which lists the cumulative throughputs achieved by each TCP variant under all AQMs, the average UDP throughput and the average utilization of bottleneck links. Table 6 also presents the Gini index, *GI*, given as Eq. (5) [35], which is widely used in economics and statistics to determine inequality in a society's distribution of wealth to people. Here, we use this index to measure the inequality in an AQM's allocation of bandwidth to various sources.

$$GI = \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} |x_i - x_j|}{2n^2 \bar{x}}, \tag{5}$$

where x_i and x_j denote all possible pairs of per-flow throughputs, $\bar{x}$ is the mean throughput, and n is the total number of flows. Like *JFI*, the value of *GI* also ranges from 0 to 1. However, unlike *JFI*, $GI = 0$ represents a complete equality and $GI = 1$ indicates a complete inequality.

Due to the existence of aggressive flows, queues are built up at bottleneck links and matched drops are triggered. As shown in Table 6, with AFCD, and CHORD, TCP Reno does not show a clear advantage over TCP Vegas. Conversely, with the RED and CHOKe, the sources get the advantage of employing TCP Reno in terms of better throughput than TCP Vegas. However, TCP Reno is rather severely penalized in CHOKeR as the number of flows increases. Thus, unlike all other AQMs, TCP Vegas receive a better throughput than TCP Reno under CHOKeR. The link utilization under RED is the highest, however, the Gini index is also the highest, which represents a high level of inequality under RED. The link utilization under CHORD is the second highest in most scenarios, while the Gini index is the lowest in all scenarios, which shows a high level of equality under CHORD. Thus,

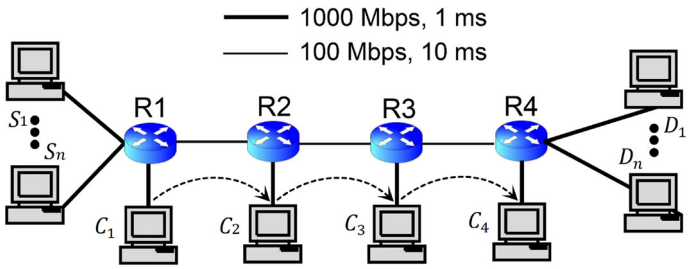

Fig. 13 Parking lot topology (S_i transmits to $D_i(\forall i = 1, \ldots, n)$ and C_i to $C_{i+1}(i = 1, 2, 3)$)

Table 6 Comparison of inter-protocol fairness

AQM	No. of TCP Vegas flows	No. of TCP Reno flows	TCP Vegas throughput (Mbps)	TCP Reno throughput (Mbps)	UDP throughput (Mbps)	Link utilization (Mbps)	Gini index (Mbps)
RED	10	10	38.95	46.18	14.75	99.88	0.317
CHOKe			41.37	47.25	9.94	98.56	0.231
CHOKeR			47.31	45.24	3.43	95.98	0.118
AFCD			44.91	46.89	5.53	97.33	0.115
CHORD			45.93	47.19	5.26	98.38	0.069
RED	20	20	40.32	48.12	11.23	99.67	0.352
CHOKe			41.87	48.55	7.61	98.03	0.266
CHOKeR			48.19	45.61	1.93	95.73	0.161
AFCD			44.82	47.33	4.94	97.09	0.155
CHORD			46.19	47.68	4.24	98.11	0.083
RED	30	30	40.41	48.74	10.37	99.52	0.405
CHOKe			42.88	50.45	4.53	97.86	0.337
CHOKeR			48.78	45.82	0.92	95.52	0.188
AFCD			44.65	47.73	4.61	96.99	0.179
CHORD			46.99	48.78	2.25	98.02	0.106

CHORD offers a more even distribution of bandwidth to TCP Vegas and TCP Reno sources and, as such, offers a better inter-protocol fairness than all other AQMs.

4.7 Effect of Packet Sizes

One of the main reasons for the unfair bandwidth sharing in the Internet is the coexistence of diverse packet sizes. For two flows with similar arrival rates, the one with the larger packet size is likely to attain a higher throughput [7]. Thus, it also becomes imperative for a fairness-driven AQM scheme to retain performance irrespective of different packet sizes. However, most AQM schemes in the literature consider fixed length packets for their evaluations [7]. We study the effect of packet sizes on the performance of CHORD by using 31 TCP flows, 1 UDP flow, a buffer of size 100 KB, and *JFI* as a metric to evaluate performance under packet sizes ranging from 250 bytes to 2 Kbytes. The results are presented in Fig. 14. RED shows a gradual decrease in fairness as the packet sizes increase. The performances of the other AQM schemes improve, in general, as the UDP packet sizes are reduced. This is because smaller UDP packets arrive more numerously increasing the likelihood of matching and dropping in the matched drop frameworks. However, CHORD outperforms CHOKe and CHOKeR in most scenarios and can retain its superior performance irrespective of different packet sizes.

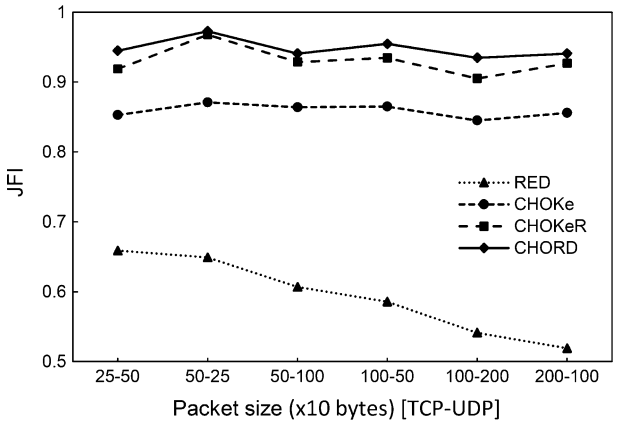

Fig. 14 Fairness under different packet sizes

4.8 Effect of Buffer Sizes

Routers require careful sizing of buffers as under-sizing may induce low link utilization, while oversizing may increase queuing delay [7]. A widely used rule-of-thumb for buffer sizing for TCP traffic is the bandwidth-delay product (BDP) [36]. Recent studies, however, reveal that buffers can be made smaller than BDP in core routers without sacrificing too much link utilization in order to accommodate heterogeneous traffic and to facilitate the development of all-optical routers offering much smaller buffers [37–39]. Therefore, the performance of CHORD has thus far been evaluated using a smaller buffer of 100–200 packets. This also demonstrates that CHORD does not have large buffer requirements to enforce fairness. However, oversized buffers (e.g., BDP or higher) have become commonplace due to the misguided efforts to evade packet loss entirely [40]. Therefore, it is very important for a fairness-driven AQM scheme to retain performance irrespective of the buffer size. In this subsection, we evaluate the compatibility of CHORD with larger buffers. To that end, we consider 1 UDP flow, 31 TCP flows, packet sizes of 500 bytes and *JFI* as a metric to evaluate performance under buffer sizes ranging from 25 to 600 packets. The results are shown in Fig. 15. The performance of RED deteriorates with the increasing buffer sizes. This is because, as the buffer size increases, there is more space available for unresponsive flows to manipulate. Conversely, the performances of the other AQM schemes improve with the increasing buffer sizes. This is because larger buffers increase the likelihood of packets matching and dropping in the matched drop frameworks of CHOKe, CHOKeR and CHORD. As shown in Fig. 15, for the considered scenario, the performance of CHORD remains mostly equal to or slightly below CHOKeR for buffer sizes of less than 200 packets and tends to increase gradually as the buffer sizes increase.

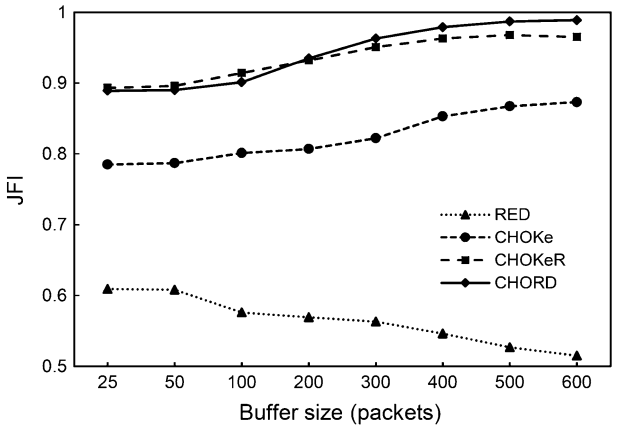

Fig. 15 Fairness under different buffer sizes

4.9 Drawing Factor

In CHORD, the drawing factor, d_i, in the initial matching trials is set to 3. In this subsection, we use a range of values for d_i and demonstrate their effect on the performance of CHORD. For 31 TCP flows and 1 UDP flow, Table 7 presents the *JFI* and UDP throughputs achieved under different drawing factors. As the value of d_i increases from 1 to 3, unresponsive flows are effectively throttled and the fairness improves. Increasing d_i any further unnecessarily causes severe punishment of unresponsive flows, reduces the fairness, and may also reduce link utilization. Similar performance is also observed for the scenario with 88 TCP and 12 UDP flows, as shown in Table 8. In this scenario, however, there is some improvement in the fairness when $d_i = 4$. Nevertheless, the improvement is negligible as compared to the increased processing cost that will be incurred by performing an additional matching trial at the per-packet arrival.

4.10 Size of *drop_list*

The *drop_list* size, δ, plays a significant role in attaining fairness by the proposed scheme. CHORD is configured with δ size of 10. In this subsection, we employ a range of sizes for δ and demonstrate their effect on the achievable throughputs. For 10 UDP and 30 TCP flows and a bottleneck link capacity of 400 Mbps, different δ

Table 7 Effect of the value of d_i on throughput and *JFI* for 1 UDP and 31 TCP flows

d_i	1	2	3	4	6	8	10
UDP throughput (Mbps)	9.23	7.06	3.71	2.62	1.41	1.06	0.34
JFI	0.873	0.959	0.991	0.982	0.945	0.860	0.747

Table 8 Effect of the value of d_i on throughput and *JFI* for 88 TCP and 12 UDP flows

d_i	1	2	3	4	6	8	10
UDP throughput (Mbps)	58.67	43.33	32.40	27.05	19.89	7.56	2.77
JFI	0.276	0.402	0.467	0.473	0.428	0.351	0.216

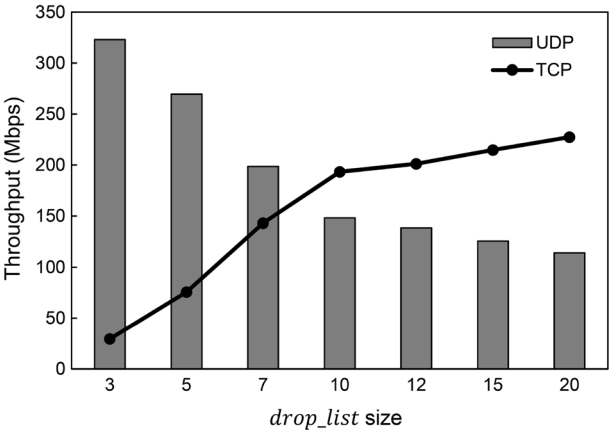

Fig. 16 Effect of *drop_list* size on throughput of 30 TCP and 10 UDP flows

sizes yield different throughputs, as shown in Fig. 16. As δ is increased from 3 to 10, the UDP throughput is throttled and the TCP throughput is enhanced. Increasing δ beyond 10 does not, however, provide a significant performance improvement but will increase the state space and per-packet processing costs due to the complexity associated with the cache memory management. The δ size of 10, therefore, offers a good trade-off between a reasonable performance and manageable computational complexity. A similar effect can also be observed for the scenario with 105 TCP and 15 UDP flows, as shown in Fig. 17.

5 Conclusion

Fair bandwidth allocation is critical to Internet architecture to be more accommodating of the heterogeneity. This paper presents CHORD, a novel fairness-driven AQM scheme for regulating bandwidth utilization in best-effort routers. The identification of unresponsive flows in CHORD consists of a fixed-size cache to store the history of recent drops with state space requirement equal to the size of the cache. For restricting the identified unresponsive flows, CHORD employs a stateless matched drop framework with two matching trial phases, namely, the initial phase at each packet arrival to identify and drop unresponsive flows, and an additional phase to impose an extra penalty on the identified high-bandwidth unresponsive

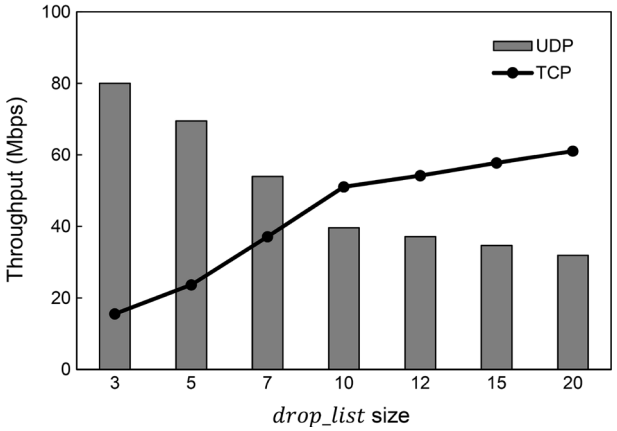

Fig. 17 Effect of *drop_list* size on throughput of 105 TCP and 15 UDP flows

flows. The level of extra penalty is the function of the average buffer occupancy. The per-packet processing cost is proportional to the drawing factor when congestion occurs. The performance of CHORD is evaluated through extensive simulations in comparison with well-known AQMs. The results demonstrate reasonable enhancement in fairness to aggregate traffic. CHORD is also able to improve intra-protocol and inter-protocol fairness and the goodput of responsive flows, and it demonstrates its compatibility with buffers and packets of different sizes without a significant loss in performance. With its low state space and per-packet processing costs, CHORD is lightweight and is well-suited for core routers to regulate bandwidth utilization, and be deployed as an effective tool to promote the use of congestion control mechanisms. Avenues for our future work include modelling and analysis of CHORD for other router architectures, such as the combined-input–output-queued architecture.

References

1. Adams, R.: Active queue management: a survey. IEEE Commun. Surv. Tutor. **15**(3), 1425–1476 (2013)
2. Kushwaha, V., Gupta, R.: Congestion control for high-speed wired network: a systematic literature review. J. Netw. Comput. Appl. **45**, 62–78 (2014)
3. Baker, F., Fairhurst, G.: IETF recommendations regarding active queue management. IETF RFC 7567, BCP 197. https://www.rfc-editor.org/rfc/rfc7567.txt (2015). Accessed 21 Aug 2017
4. Floyd, S.: Congestion control principles. IETF RFC 2914, BCP 41. https://tools.ietf.org/html/rfc2914.html (2000). Accessed 21 Aug 2017
5. Papadimitriou, D. Welzl, M., Scharf, M., Briscoe B.: Open research issues in Internet congestion control. IETF RFC 6077. https://www.rfc-editor.org/rfc/rfc6077.txt (2011). Accessed 21 Aug 2017
6. Braden, B., Clark, D., Crowcroft, J., Davie, B., Deering, S., Estrin, D., Floyd, S., Jacobson, V., Minshall, G., Partridge, C., Peterson, L., Ramakrishnan, K., Shenker, S., Wroclawski, J., Zhang, L.: Recommendations on queue management and congestion avoidance in the Internet. IETF RFC 2309. https://www.rfc-editor.org/rfc/rfc2309.txt (1998). Accessed 21 Aug 2017

7. Abbas, G., Halim, Z., Abbas, Z.H.: Fairness-driven queue management: a survey and taxonomy. IEEE Commun. Surv. Tutor. **18**(1), 324–367 (2016)
8. Floyd, S., Fall, K.: Promoting the use of end-to-end congestion control in the Internet. IEEE/ACM Trans. Netw. **7**(4), 458–472 (1999)
9. Kohler, E., Handley, M., Floyd S.: Datagram congestion control protocol (DCCP). IETF RFC 4340. https://www.rfc-editor.org/rfc/rfc4340.txt (2006). Accessed 21 Aug 2017
10. Anjum, F. M., Tassiulas, L.: Fair bandwidth sharing among adaptive and non-adaptive flows in the Internet. In: Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies, 21–25 March, New York, USA, pp. 1412–1420 (1999)
11. Nossenson, R., Maryuma, H.: Active queue management in blind access networks. In: Third International Conference on Access Networks, 24–29 June, Venice, Italy, pp. 27–30 (2012)
12. Latré, S., Meerssche, W., Deschrijver, D., Papadimitriou, D., Dhaene, T., Turck, F.: A cognitive accountability mechanism for penalizing misbehaving ECN-based TCP stacks. Int. J. Netw. Manag. **23**(1), 16–40 (2013)
13. Hwang, J., Byun, S.-S.: A resilient buffer allocation scheme in active queue management: a stochastic cooperative game theoretic approach. Int. J. Commun Syst. **28**(6), 1080–1099 (2015)
14. Yi, S., Deng, X., Kesidis, G., Das, C.R.: A dynamic quarantine scheme for controlling unresponsive TCP sessions. Telecommun. Syst. **37**, 169–189 (2008)
15. Hanlin, S., Yuehui, J., Yidong, C., Hongbo, W., Shiduan C.: Improving fairness of RED aided by lightweight flow information. In: 2nd IEEE International Conference on Broadband Network & Multimedia Technology, 18–20 October, Beijing, China, pp. 335–339 (2009)
16. Abbas, G., Nagar, A. K., Tawfik, H., Goulermas J. Y.: Pricing and unresponsive flows purging for global rate enhancement. J. Electr. Comput. Eng. Article ID 379652, 1–10 (2010)
17. Alvarez-Flores, E.P., Ramos-Munoz, J.J., Ameigeiras, P., Lopez-Soler, J.M.: Selective packet dropping for VoIP and TCP flows. Telecommun. Syst. **46**(1), 1–16 (2011)
18. Xue, L., Kumar, S., Cui, C., Kondikoppa, P., Chiu, C.-H., Park, S.-J.: Towards fair and low latency next generation high speed networks: AFCD queuing. J. Netw. Comput. Appl. **70**, 183–193 (2016)
19. Tsavlidis, L., Efraimidis, P.S., Koutsiamanis, R.-A.: Prince: an effective router mechanism for networks with selfish flows. J. Internet Eng. **6**(1), 355–362 (2016)
20. Menth, M., Zeitler, N.: Activity-based congestion management for fair bandwidth sharing in trusted packet networks. In: 2016 IEEE/IFIP Network Operations and Management Symposium, 25–26 April, Istanbul, Turkey, pp. 231–239 (2016)
21. Pan, R., Prabhakar, B., Psounis, K.: CHOKe—a stateless active queue management scheme for approximating fair bandwidth allocation. In: Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies, 26–30 March, Tel Aviv, Israel, pp. 942–951 (2000)
22. Jiang, Y., Hamdi, M., Liu, J.: Self adjustable CHOKe: an active queue management algorithm for congestion control and fair bandwidth allocation. In: Eight IEEE International Symposium on Computers and Communication, 30 June–3 July, Kemer–Antalya, Turkey, pp. 1018–1025 (2003)
23. Yamaguchi, Y., Takahashi, Y.: A queue management algorithm for fair bandwidth allocation. Comput. Commun. **30**(9), 2048–2059 (2007)
24. Kesselman, A., Leonardi, S.: Game-theoretic analysis of Internet switching with selfish users. Theor. Comput. Sci. **452**, 107–116 (2012)
25. Lu, L., Du, H., Liu, R.P.: CHOKeR: a novel AQM algorithm with proportional bandwidth allocation and TCP protection. IEEE Trans. Ind. Inform. **10**(1), 637–644 (2014)
26. Manzoor, S., Abbas, G., Hussain, M.: CHOKeD: fair active queue management. In: 15th IEEE International Conference on Computer and Information Technology, 26–28 October, Liverpool, UK, pp. 512–516 (2015)
27. Raza, U., Abbas, G., Hussain, Z.: CHOKe-FS: CHOKe with fair bandwidth share. In: 2015 International Conference on Information and Communication Technologies, 12–13 December, Karachi, Pakistan, pp. 1–5 (2015)
28. Hussain, Z., Abbas, G., Raza, U.: CHOKe with recent drop history. In: Proceedings of 13th IEEE International Conference on Frontiers of Information Technology, 14–16 December, Islamabad, Pakistan, pp. 160–165 (2015)
29. Jiang, X., Jin, G., Yang, J.: LRURC: A low complexity and approximate fair active queue management algorithm for choking non-adaptive flows. IEEE Commun. Lett. **19**(4), 545–548 (2015)
30. Floyd, S., Jacobson, V.: Random early detection gateways for congestion avoidance. IEEE/ACM Trans. Netw. **1**(4), 397–413 (1993)

31. Abbas, G., Nagar, A.K., Tawfik, H.: On unified quality of service resource allocation scheme with fair and scalable traffic management for multiclass Internet services. IET Commun. **5**(16), 2371–2385 (2011)
32. Feknous, M., Houdoin, T., Le Guyader, B., De Biasio, J., Gravey, A., Gijón, J.A.T.: Internet traffic analysis: a case study from two major European operators. In: 2014 IEEE Symposium on Computers and Communications, 23–26 June, Portugal, pp. 1–7 (2014)
33. Jain, R.: The Art of Computer Systems Performance Analysis. Wiley, Hoboken (1991)
34. Andrew, L., Marcondes, C., Floyd, S., Dunn, L., Guillier, R., Gang, W., Eggert, L., Ha, S., Rhee, I.: Towards a common TCP evaluation suite. In: Sixth International Workshop on Protocols for FAST Long-Distance Networks, 5–7 March, Manchester, UK, pp. 1–5 (2008)
35. Gastwirth, J.L.: The estimation of the Lorenz curve and Gini index. Rev. Econ. Stat. **54**(3), 306–316 (1972)
36. Villamizar, C., Song, C.: High performance TCP in ANSNET. ACM SIGCOMM Comput. Commun. Rev. **24**(5), 45–60 (1994)
37. Vishwanath, A., Sivaraman, V., Rouskas G. N.: Considerations for sizing buffers in optical packet switched networks. In: 28th IEEE Conference on Computer Communications, 19–25 April, Rio de Janeiro, Brazil, pp. 1323–1331 (2009)
38. Beheshti, N., Burmeister, E., Ganjali, Y., Bowers, J.E., Blumenthal, D.J., McKeown, N.: Optical packet buffers for backbone Internet routers. IEEE/ACM Trans. Netw. **18**(5), 1599–1609 (2010)
39. Gharakheili, H.H., Vishwanath, A., Sivaraman, V.: Comparing edge and host traffic pacing in small buffer networks. Comput. Netw. **77**, 103–116 (2015)
40. Gettys, J.: Bufferbloat: dark buffers in the Internet. IEEE Internet Comput. **15**(3), 95–96 (2011). doi:10.1109/MIC.2011.56

Zawar Hussain received his M.S. degree in computer system engineering from the GIK Institute of Engineering Sciences and Technology, Pakistan, in 2015. He is currently working as a Research Associate in the Faculty of Computer Sciences & Engineering, GIK Institute, Pakistan. His research interests include active queue management, routing, software defined networks, and Internet of Things.

Ghulam Abbas received his Ph.D. degree in computer networks from the University of Liverpool, U.K., in 2010. Currently, he is serving as Associate Professor at GIK Institute, Pakistan. He is a Fellow of the British Computer Society and a Senior Member of IEEE. His research interests include Internet architecture, congestion control and active queue management.

Zahid Halim received his Ph.D. degree in computer science from the National University of Computer and Emerging Sciences, Pakistan, in 2010. Currently, he is an Associate Professor with the Faculty of Computer Sciences & Engineering, GIK Institute of Engineering Sciences and Technology, Pakistan. His research interests include intelligent and distributed systems.