

2003-12-02

Efficiently and Fairly Allocating Bandwidth at a Highly Congested Link

<https://hdl.handle.net/2144/1523>

"Downloaded from OpenBU. Boston University's institutional repository."

Efficiently and Fairly Allocating Bandwidth at a Highly Congested Link ^{*}

TAO WANG IBRAHIM MATTA AZER BESTAVROS

Computer Science Department
Boston University
Boston, MA 02215

{wtwang, matta, best}@cs.bu.edu

BUCS-TR-2003-027

Abstract. We consider the problem of efficiently and fairly allocating bandwidth at a highly congested link to a diverse set of flows, including TCP flows with various Round Trip Times (RTT), non-TCP-friendly flows such as Constant-Bit-Rate (CBR) applications using UDP, misbehaving, or malicious flows. Though simple, a FIFO queue management is vulnerable. Fair Queueing (FQ) can guarantee *max-min* fairness but fails at efficiency. RED-PD [10] exploits the history of RED's actions [6] in preferentially dropping packets from higher-rate flows. Thus, RED-PD attempts to achieve fairness at low cost. By relying on RED's actions, RED-PD turns out not to be effective in dealing with non-adaptive flows in settings with a highly heterogeneous mix of flows. In this paper, we propose a new approach we call RED-NB (RED with No Bias). RED-NB does *not* rely on RED's actions. Rather it explicitly maintains its own history for the few high-rate flows. RED-NB then *adaptively* adjusts flow dropping probabilities to achieve max-min fairness. In addition, RED-NB helps RED itself at very high loads by tuning RED's dropping behavior to the flow characteristics (restricted in this paper to RTTs) to eliminate its bias against long-RTT TCP flows while still taking advantage of RED's features at low loads. Through extensive simulations, we confirm the fairness of RED-NB and show that it outperforms RED, RED-PD, and CHOKe [11] in all scenarios.

Keywords: Congestion Control; Queue Management; Transport Protocols (TCP, UDP); Performance Evaluation and Simulation.

1 Introduction

Motivation: The past decade has seen considerable change in the role of the Internet. Its design as a best-effort transport medium is showing signs of stress as it is used to support new applications and services (*e.g.*, live streaming, gaming,

^{*} This work was supported in part by NSF grants ANI-0095988, ANI-9986397, EIA-0202067 and ITR ANI-0205294.

and video conferencing) and as it incorporates increasingly heterogeneous infrastructures (*e.g.*, satellite links with very large RTTs and lossy wireless links). This state of affairs suggests that network links must deal with an increasingly complex mix of traffic, including TCP flows with quite different RTTs, malicious flows which insist on some high arrival rate, or legitimate UDP services which do not back off their sending rate even if a highly congested link drops a lot of their packets. Thus one of the main challenges facing link traffic management is how to maintain fairness in bandwidth allocation in the face of such diversity (*e.g.* how to protect TCP-friendly flows from greedy or malicious flows). This challenge is further complicated by the requirement that fairness does not come at the expense of efficiency.

Queue Management Approaches: FIFO (First In First Out) queue management is simple but can not provide reasonable protection. In fact, FIFO distributes increased delay and jitter among all flows. In case of congestion, greedy flows would grab more bandwidth. Fair Queueing (FQ) [5] can guarantee fairness. However, it is not a good option because multiple queues need to be maintained, which is too expensive. RED-PD [10] simply collects statistics from the drop history of RED [6] and then applies preferential drops at a prefilter put ahead of RED. Only flows with high bandwidth are monitored by RED-PD to save cost—since the distribution of the rate of Internet flows is known to obey a power law, a relatively small group of flows are high bandwidth, *i.e.*, they contribute most of the traffic on the Internet. RED-PD takes advantage of the features of RED, which attempts to keep the buffer within a low range by random early notifications or drops. By keeping a short queue, RED attempts to decrease the queueing delay. By making packet drops uniformly distributed, RED avoids the synchronization of TCP flows. However, the design of RED assumes that flows adapt their sending rates in response to RED drops.

Paper Contributions and Outline: We found that RED-PD, by relying on RED dropping history to identify high-rate flows, is not effective in dealing with non-adaptive flows in settings with a highly heterogeneous mix of flows. In this paper, we propose a new approach we call RED-NB (RED with No Bias). RED-NB does *not* rely on RED’s actions. Rather it explicitly maintains its own history for the few high-rate flows. RED-NB then *adaptively* adjusts flow dropping probabilities to achieve max-min fairness by punishing unresponsive, monitored flows. In addition, as a second line of defense, RED-NB helps RED itself at high loads, by tuning RED’s dropping behavior to the flow characteristics (restricted in this paper to RTT) to eliminate its bias against long-RTT TCP flows, while still taking advantage of RED’s features at low loads. Through extensive simulations, we confirm the fairness of RED-NB and show that it outperforms RED, RED-PD, and CHOKe [11] in all scenarios, especially those with a highly heterogeneous mix of flows.

The paper is organized as follows. Section 2 discusses related work and distinguishes RED-NB. Section 3 describes the design of RED-NB in detail. Section 4 evaluates RED-NB by simulation and contrasts its performance against RED, RED-PD and CHOKe. Section 5 concludes the paper with future work.

2 Related Work

To date, there are roughly two classes of efficient algorithms that deal with misbehaving flows at a highly congested router. The first class is stateless. CHOKe[11] is a classic example. The second class takes advantage of the skewed property of traffic in the Internet to monitor and control the small group of high-traffic flows. RED-PD [10] as well as our new proposed approach we call RED-NB, both belong to this second category.

Upon the arrival of a new incoming packet, CHOKe [11] picks a packet from the buffer at random. If they both belong to the same flow (i.e. a hit occurs), CHOKe drops both packets. The idea is to punish the flow with the most traffic. Though simple, CHOKe fails to achieve fairness, especially in the presence of many malicious flows as their packets occupy a large chunk of the buffer space. In addition, dropping a packet from a random position in the buffer adds to the implementation complexity of CHOKe and its variants (e.g. CHOKe+ [1], XCHOKe [2]).

RED-PD [10, 8] uses the dropping history from RED to identify high-rate flows (i.e. flows which experienced more packet drops). A prefilter placed in front of RED uses this information to compute a drop rate for each monitored high-rate flow. As we show in Section 4, RED-PD is not effective in settings with many non-adaptive or highly malicious flows. The reason is that under malicious attacks, as RED-PD relies on RED, a lower packet drop rate is achieved at the expense of adaptive TCP-friendly flows.

Our proposed RED-NB scheme embodies a simple philosophy: *congestion responsibility should be taken by flows with higher-than-average arrival rates*. When congestion happens, those higher-than-average flows should back off first until congestion subsides. This is exactly the definition of *max-min* allocation of a shared resource. When a flow with lower-than-average arrival rate increases its arrival rate and congestion occurs, some bandwidth should be taken from higher-arrival-rate flows to satisfy the new demand. If a misbehaving flow does not back off actively, RED-NB forces it at its max-min share by dropping “extra” packets. Thus, at high load, instead of relying on RED’s dropping behavior as in RED-PD, RED-NB actively tunes RED to achieve max-min fairness.

RED-NB also differs from RED-PD in the design of its prefilter, which imposes drops on high-rate monitored flows. RED-NB collects its own statistics, rather than relying on RED. Thus, RED-NB has a better view of traffic conditions (in this paper, current and average arrival rates and RTT of high-rate monitored flows) and hence provides more precise control. RED-NB’s prefilter also uses a novel adaptive algorithm for probing the right level of drop rate for each monitored flow. Since our approach to fairness at high load is largely decoupled from RED, one can potentially replace RED with other queue management schemes. We do not explore this issue in this paper. We note that RED-NB’s cost of maintaining statistics is comparable to that of RED-PD. The cost of accessing a packet’s header to identify flows is also comparable to that of RED-PD as well as CHOKe-type schemes.

Finally, other schemes (e.g. SRED [12], RIO [3], CSFQ [14]) either have not dealt explicitly with malicious flows or require extensive architectural changes.

3 Design of RED-NB

Our proposed RED-NB consists of a *stream scanner* to identify higher-than-average-rate flows, followed by a *prefilter* placed in front of RED to impose drops on those monitored flows, and finally a *drop tuner* that tunes RED’s dropping behavior so as to eliminate bias against longer-RTT TCP flows at times of heavy load. The overall architecture of RED-NB is shown in Figure 1. We describe each component next. We list all parameters and functions of RED-NB in Tables 1 and 2.

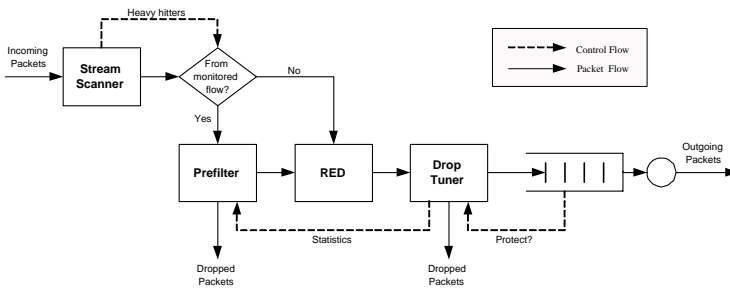


Fig. 1. Architecture of an RED-NB Router

3.1 Stream Scanner: Identifying Heavy Hitters

RED-NB can make use of any efficient stream scanning technique to identify the small subset of flows that consume most of the link’s bandwidth—the heavy hitters. In this paper, we use the lookup algorithm of [4] in the stream scanner. This algorithm uses a small number of counters to identify heavy-hitters with high accuracy.

3.2 Prefilter: Identifying and Punishing Misbehaving Flows

Based on feedback collected regarding past drop statistics (*e.g.* the sum of bytes of arriving packets and the number of dropped packets during a past measurement period), the prefilter identifies misbehaving flows. Namely, when congestion occurs, RED-NB increases its prefilter drop rate for those flows with high bandwidth. Adaptive (*e.g.*, TCP-friendly) flows are expected to react to these losses by reducing their sending rates. For those who don’t, the prefilter’s drop rate is increased until it reaches a certain threshold, beyond which a flow is classified as

non-TCP-friendly and the prefilter adopts and applies an increasingly aggressive drop rate to punish this flow. Notice that unlike RED-PD [10], RED-NB does not rely on RED's poor feedback signal to start dealing with misbehaving flows.

The feedback statistics used in the prefilter are updated periodically every control round interval (RI), which is set to 0.5 second.

Algorithm 1 UpdateRecord(Packet pkt)

```

Flowid fid ← Monitor(pkt)
if fid ≥ 0 then
  ctime ← CurrentTime()
  if ctime − RST ≥ RI then
    Rlover()
  end if
  prob ← GetPrefilterDropRate(fid)
  if prob > FDT then
    if RandomDrop(pkt, prob) then
      return
    end if
  end if
  UpdateInfo(fid, pkt)
end if

```

Algorithm 1 shows the implementation of the front-end prefilter after the stream scanner. At the prefilter, since TCP flows adapt their sending rate in response to RED drops, they won't be subjected to prefilter drops as long as the prefilter drop rate of the TCP-friendly flow remains below a certain Prefilter Drop Threshold (FDT) set to 0.03. Figure 2 illustrates a case where the prefilter drop rate of TCP flows remains under FDT most of the time.

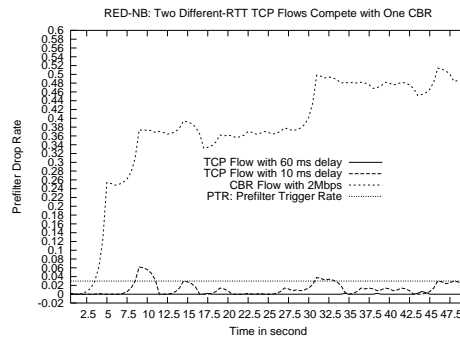


Fig. 2. A Simulation Case on Prefilter Drop Threshold (FDT)

The prefilter drop rate is periodically updated every control round. The prefilter drop rate is increased for those flows with higher arrival rate than the average. Once congestion subsides, the prefilter drop rate is decreased. The prefilter drop rate of a monitored flow is initially set to a very small base number.

Algorithm 2 UpdatePrefilterDropRate(Flowid fid)

```

pdr ← FilterDropRate(fid)
step ← FilterDropRateStep(fid)
if LinkDropRate() > LDT and RfOverRavg(fid) > 1 then
  if pdr == 0 or not(FDRincreasing(fid)) then
    step ← FS
    SetFDRincreasing(fid, true)
  else
    step ← step × 2
  end if
  SetFDRstep(fid, step)
  pdr ← MIN(pdr + step, 1.0)
  SetFDR(fid, pdr)
else
  if pdr == 1.0 or FDRincreasing(fid) then
    step ← FS
    SetFDRincreasing(fid, false)
  else
    step ← step × 2
  end if
  SetFDRstep(fid, step)
  pdr ← MAX(pdr - step, 0)
  SetFDR(fid, pdr)
end if

```

Algorithm 2 implements a binary search for computing the prefilter drop rate for monitored flows. The objective is for the prefilter to adaptively search for the expected bandwidth share for the flows. Since the target is set to the average bandwidth share at the congested link, RED-NB is essentially searching for the max-min bandwidth allocation. Heavy congestion is expressed by a packet drop rate (LDT) set to 0.02. If the packet drop rate is below LDT, the link is considered under control and the prefilter drop rates should decrease. Specifically, prefilter drop rates are increased or decreased by a step which increases or decreases exponentially (as illustrated in Figure 2 for the non-adaptive CBR flow). This increase/decrease step is initially set to 0.002 whenever the search changes its direction by beginning to increase/decrease the prefilter drop rate.¹

¹ This parameter setting reflects a policy of initially dropping about one 1K-byte packet over a control round interval of 0.5 second over link bandwidths of up to 10 Mbps.

3.3 Drop Tuner: Reducing RED’s Bias against long-RTT TCP Flows

As a second line of defense, at high load, RED-NB tunes RED’s drop rate based on the bandwidth and RTT of the monitored flow. This way, RED-NB decreases the probability of dropping packets from low-bandwidth flows.

From the TCP-friendly throughput equation [7], we have

$$f(p, RTT) \approx \frac{\sqrt{1.5}}{RTT\sqrt{p}}$$

where f is the flow throughput, and p is the packet drop rate. Observe that longer-RTT TCP flows receive less throughput. Our goal in this paper is to eliminate this inherent bias by achieving max-min fairness.

Given RTT knowledge (e.g. obtained using efficient “measurements-from-the-middle” techniques [9]), we tune the RED drop probability by

$$\frac{\sum_{i=1}^N RTT_i^2}{(N \times RTT_j^2)}$$

where j denotes the current monitored flow, and N is the number of all monitored flows. Observe that this factor approaches 1 for shorter-RTT flows and approaches 0 for longer-RTT flows in the limit of $N \rightarrow \infty$. If RTT is unknown for a flow, we use 40ms as default, similar to [10].²

However, if congestion happens, and the queue size exceeds the maximum threshold, RED drops *every* incoming packet. This can potentially hurt long-RTT flows as their throughput further degrades compared to short-RTT flows. To avoid this situation, once the queue size approaches RED’s maximum threshold, our approach applies a more aggressive preferential dropping on shorter-RTT or higher-arrival-rate flows. To that end, we tune RED’s computed drop rate taking into account, not only RTT information, but also measured arrival rates of monitored flows. Specifically, RED’s packet drop probability is adjusted by

$$\frac{R_j \times \sum_{i=1}^N RTT_i^2}{R_{sum} \times (N \times RTT_j^2)}$$

where R_j is the measured arrival rate of monitored flow j , and R_{sum} is the total arrival rate of all monitored flows.

Algorithm 3 shows the implementation of tuning RED’s computed drop rate for a monitored flow. Recall that this function is executed right after RED computes its drop probability. The link is considered at heavy load if the total link arrival rate measured over the last control period (RI) exceeds the link bandwidth.

² This default value is used if the RTT measurement is not yet ready, or for one-way flows, e.g. UDP flows, since we can not measure their RTT from the middle of the network without observing a two-way message exchange.

Algorithm 3 REDtoRED-NB_DropRate(Flowid fid, double REDDropRate)

```

if  $fid < 0$  then
    return
end if
 $N \leftarrow \text{NumberOfMonitoredFlows}()$ 
if QueueFull() and HeavyLoad() then
     $REDDropRate \leftarrow RfOverRsum(fid) \times RTTratio(fid)$ 
else
     $REDDropRate \leftarrow REDDropRate \times RfOverRavg(fid) \times RTTratio(fid)$ 
end if
 $REDDropRate = MIN(REDDropRate, 1.0)$ 

```

Table 1. RED-NB Parameters and Variables

Name	Specification
FS	Increase/decrease step of prefilter drop rate for a monitored flow. Set to 0.002
FDT	Drop threshold for prefilter. Set to 0.03
RI	Round interval for periodic computation of statistics. Set to 0.5 second
RST	Starting time of current round interval
LDT	Threshold of the drop rate for the link after the prefilter, set to 0.02. If this drop rate is above this value, the prefilter drop rates may increase for certain flows.

Table 2. RED-NB Functions

Name	Specification
CurrentTime()	Returns current time
FDRincreasing(Flowid fid)	Returns true if the prefilter drop rate of this flow is increasing
NumberOfMonitoredFlows()	Returns the number of all monitored flows
LinkDropRate()	Returns the drop rate at the link for all monitored flows
FilterDropRate(Flowid fid)	Returns the prefilter drop rate for this flow
FilterDropRateStep(Flowid fid)	Returns the probing step on the prefilter drop rate for this flow
HeavyLoad()	Returns true if the total arrival rate to the link exceeds the link's bandwidth
Monitor(Packet pkt)	Runs the monitor algorithm of [4]. Returns the flow id of the packet if it belongs to a monitored flow, otherwise returns -1
QueueFull()	Tests if the queue is approaching its capacity. In the simulation, the threshold is set to 25 packets with the maximum queue length set to 30 packets
RandomDrop(Packet pkt, double prob)	Drops packet with given probability. Returns true if it is a real drop, otherwise returns false
RfOverRavg(Flowid fid)	Returns the ratio of arrival rate of this flow over the average arrival rate at the link
RfOverRsum(Flowid fid)	Returns the ratio of arrival rate of this flow over the total arrival rate at the link
RTTratio(Flowid fid)	Returns $\frac{\sum_{i=1}^N RTT_i^2}{N \times RTT_j^2}$ where N is the number of all monitored flows, RTT_j denotes the RTT for this flow. If this is a UDP flow or its RTT is not ready, use 40ms as default
Rlover()	Computes statistics for this round interval, such as the arrival rate, the drop rate for each monitored flow and over all monitored flows. Calls UpdatePrefilterDropRate(Flowid fid) to compute the prefilter drop rate for each monitored flow. Resets corresponding counters and RST
SetFDR(Flowid fid, double r)	Sets the prefilter drop rate for this flow to given value
SetFDRstep(Flowid fid)	Sets the probing step on the prefilter drop rate for this flow
SetFDRincreasing(Flowid fid, Boolean b)	Sets the flow status: true indicates that the prefilter drop rate of this flow is increasing, otherwise it is set to false
UpdateInfo(Flowid fid, Packet pkt)	Updates local information for this flow and global information over all the monitored flows

4 Performance Evaluation

In this section we show simulation results of five scenarios comparing the performance of RED, RED-PD, CHOKe, and our algorithm RED-NB.

The simulation topology is shown in Figure 3. The bottleneck link R1-R2 has bandwidth of 3Mbps with 8ms delay and queue size of 30 packets. We test different algorithms on this bottleneck. All other links have bandwidth of 10Mbps and employ DropTail for queue management. All sources are connected to router R1 while sinks are connected to router R2. All source-destination connections share only the bottleneck link R1-R2. The delay between R2 and any sink is 1ms while we assign different delay between each source and R1.

We use FTP as application on top of the TCP flows, with packet size of 1000 bytes. Packets on UDP flows follow a Poisson process with constant sending rate, i.e. UDP flows do not back off in response to packet drops.

We show plots of flow throughputs, in Mb per second, measured every 0.5 second over a 50-second interval.

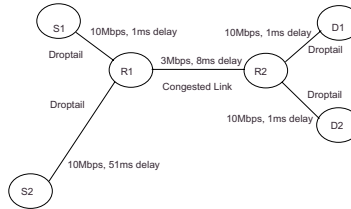


Fig. 3. Topology of Scenario 1

The first scenario (Figure 4) considers two regular TCP flows with different one-way delays of 10ms and 60ms. We observe that under our RED-NB, the throughputs of the two flows converge the closest, however the improvement over other algorithms is small. The reason is that both flows are TCP friendly. In order to converge them, more packets than necessary should be dropped from the short-RTT flow, which will lead to link under-utilization. This illustrates the tradeoff between efficiency in link throughput and fairness in terms of removing the RTT bias. In a more realistic environment with more competing flows, long-RTT TCP flows can be better protected while maintaining high link utilization.

Observe that CHOKe makes the bias even *worse*! The link throughput even degrades unnecessarily due to CHOKe's blind policy of dropping two packets upon a hit, which is excessive in this scenario.

In our second scenario (Figure 5), a third TCP flow with 110ms one-way delay is added. RED-NB is able to converge the flows better than it does in the first scenario. As we mentioned above, the reason is that RED-NB has now more room to apply preferential drops without leading to link under-utilization.

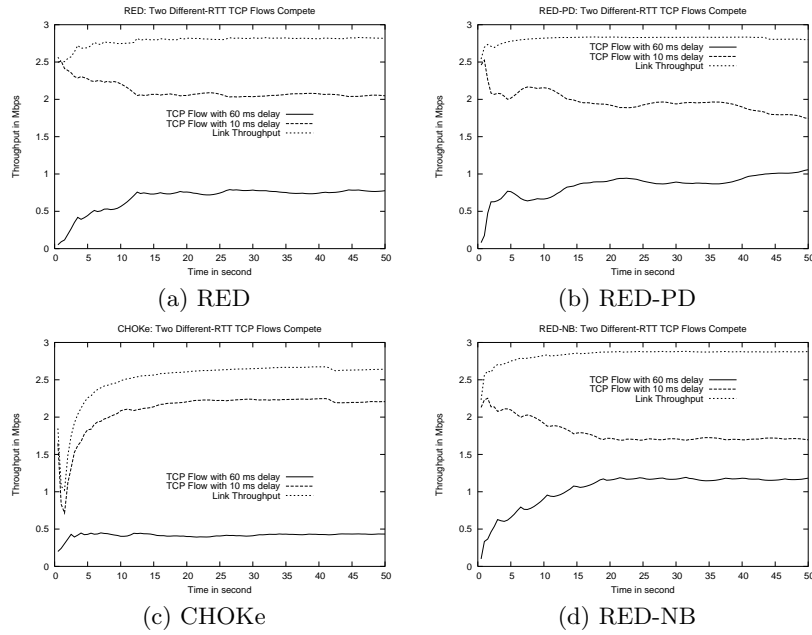


Fig. 4. Scenario 1: Two TCP Flows with Different RTTs Competing

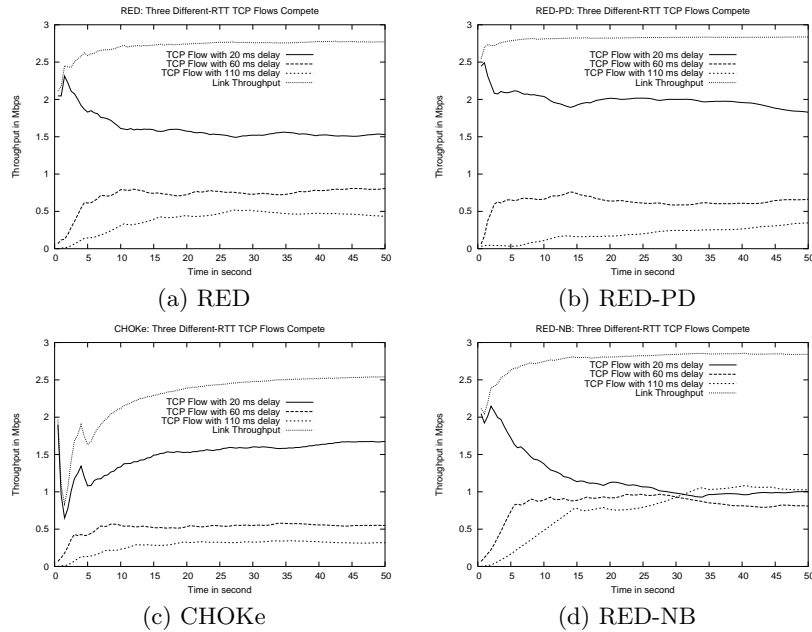


Fig. 5. Scenario 2: Three TCP Flows with Different RTTs Competing

Again, the aggressive action of CHOKe causes the link to be under-utilized. RED-PD performs even worse than RED. The reason is that once RED-PD tags a TCP flow as high-rate, relying on the often misleading history of RED, the prefilter drops packets from these flows. Such drops may help short-RTT flows grab more bandwidth.

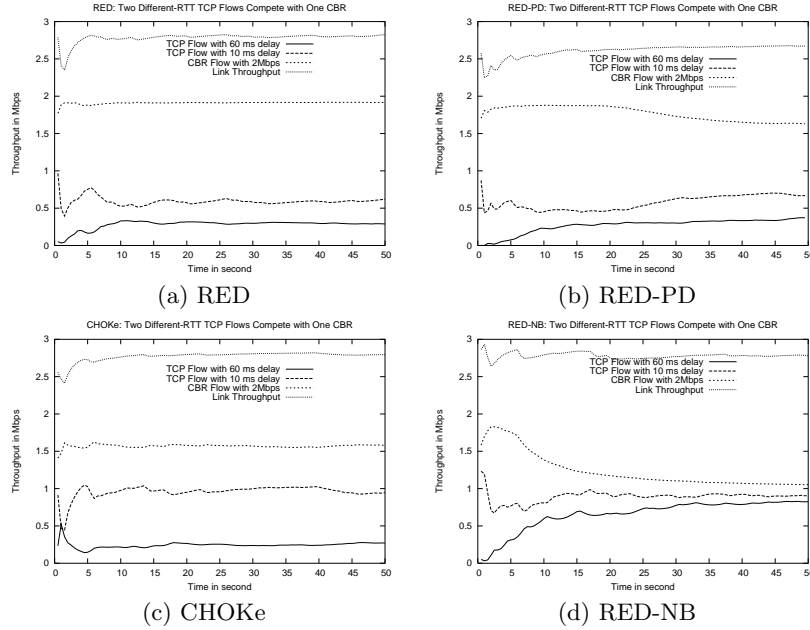


Fig. 6. Scenario 3: Two TCP Flows with Different RTTs Competing with one CBR Flow

Our third scenario (Figure 6) is similar to the second one except that the third flow is a CBR flow with 2Mbps constant demand. Observe that RED-NB converges the three flows very well. RED almost does nothing to the malicious flow. CHOKe protects the short-RTT TCP flow but not the long-RTT TCP flow. RED-PD does not provide enough protection to TCP flows.

Our fourth scenario (Figure 7) presents the competition among three CBR flows. The respective demands are 2Mbps, 1.5Mbps and 0.8Mbps. The expected max-min allocation of bandwidth is 1.1Mbps, 1.1Mbps and 0.8Mbps. The results confirm the effectiveness of RED-NB. RED drops packets from each flow with the same weight. RED-PD takes time until it starts to converge the flows, however at the expense of reduced link utilization.

Our last scenario (Figure 8) considers a mix of three TCP flows with quite different RTT and three CBR flows with high bandwidth demands. The link bandwidth is increased to 6Mbps. RED-NB performs well in forcing flows toward their max-min fair share of the link bandwidth. RED-PD and CHOKe can not

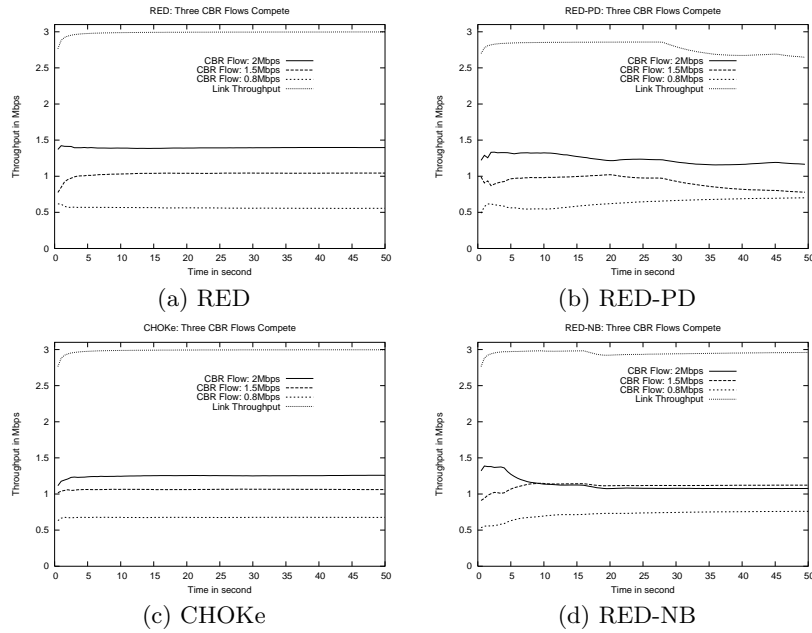


Fig. 7. Scenario 4: Three CBR Flows Competing

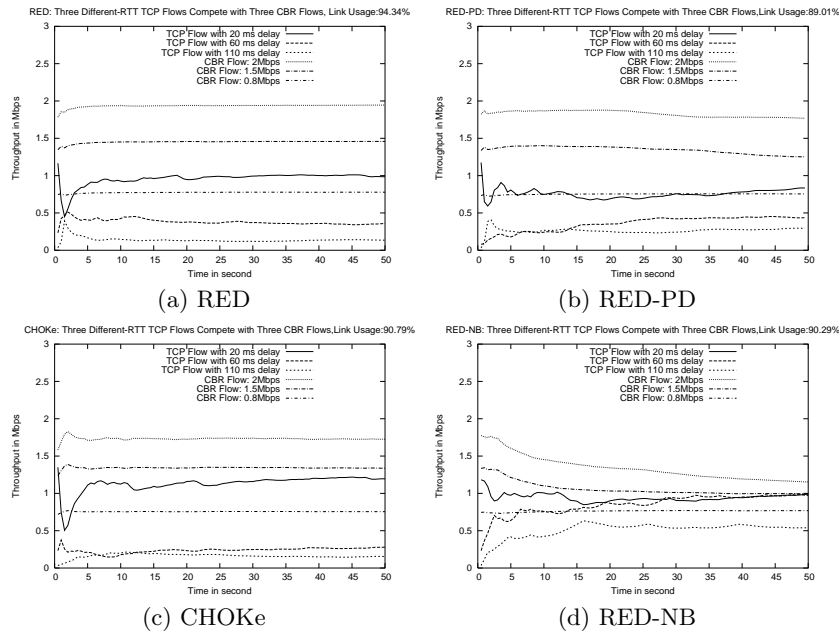


Fig. 8. Scenario 5: Three TCP Flows with Different RTTs Competing with Three CBR Flows

protect TCP flows—they take action too late after TCP flows had already given away their bandwidth share.

5 Conclusion and Future Work

We presented RED-NB to efficiently and fairly allocate bandwidth at a highly congested link. RED-NB can relieve RTT bias and protect TCP friendly flows from losing their max-min bandwidth share to misbehaving flows. RED-NB applies preferential adjustment to RED’s computed drop rate, in addition to a prefilter ahead of RED to shut out misbehaving flows early during heavy congestion. We use an efficient one-pass algorithm [4] to identify high-bandwidth flows and then RED-NB collects statistics on these monitored flows. Our simulation results confirm that RED-NB performs better than RED, RED-PD and CHOCe in all scenarios, even in the presence of multiple misbehaving flows with high bandwidth demands.

The underlying architecture of RED-NB promotes the decoupling of the prefilter and the specific queue management. Although we retained RED for its features at low load, its actions may still interfere with the prefilter’s actions, which are more accurate. We are currently investigating other non-RED queue management algorithms together with a prefilter that adapts its parameters so RTT bias and misbehavior can be completely and quickly removed.

References

1. A. Akella, S. Seshan, R. Karp, S. Shenker, and C. Papadimitriou, "Selfish Behavior and Stability of the Internet: A Game-Theoretic Analysis of TCP," in Proceedings of ACM SIGCOMM, 2002.
2. P. Chhabra, S. Chuig, A. Goel, A. John, A. Kumar, and H. Saran, "XCHOKe: Malicious Source Control for Congestion Avoidance at Internet Gateways," in Proceedings of the ICNP 2002.
3. David D. Clark and Wenjia Fang, "Explicit Allocation of Best-Effort Packet Delivery Service," *IEEE/ACM Transactions on Networking*, 6(4): 362–373, August 1998.
4. E. Demaine, A. Lopez-Ortiz and J. Ian Munro, "Frequency Estimation of Internet Packet Streams with Limited Space," in Proceedings of the 10th Annual European Symposium on Algorithms (ESA 2002).
5. A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," in Proceedings of ACM SIGCOMM, 1989.
6. S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Transactions on Networking*, pp. 397–413, August 1993.
7. S. Floyd and K. Fall, "Promoting the Use of End-to-End Congestion Control in the Internet," *IEEE/ACM Transactions on Networking*, August 1999.
8. S. Floyd, K. Fall, and K. Tieu, "Estimating Arrival Rates from the RED Packet Drop History," <http://www.icir.org/floyd/end2end-paper.html>, April 1998.
9. S. Jaiswal, G. Iannacconne, C. Diot, J. Kurose, and D. Towsley, "Measurement and Classification of Out-of-Sequence Packets in a Tier-1 IP Backbone," in Proceedings of IEEE INFOCOM 2003, April 2003.
10. R. Mahajan, S. Floyd, and D. Wetherall, "Controlling High-Bandwidth Flows at the Congested Router," in Proceedings of ICNP, November 2001.
11. R. Pan, B. Prabhakar, K. Psounis, "CHOKe, a Stateless Active Queue Management Scheme for Approximating Fair Bandwidth Allocation," in Proceedings of the IEEE INFOCOM, 2:942-951, March 2000.
12. T. J. Ott, T.V. Lakshman, and L. Wong, "SRED: Stabilized RED," in Proceedings of INFOCOM 1999.
13. V. Paxson, "End-to-End Routing Behavior in the Internet," *IEEE/ACM Transactions on Networking*, 5(2):601-615, October 1997.
14. I. Stoica, S. Shenker, and H. Zhang, "Core-Stateless Fair Queueing: A Scalable Architecture to Approximate Fair Bandwidth Allocations in High Speed Networks," in Proceedings of ACM SIGCOMM, September 1998.