A Dynamic Buffer Management Scheme Based on Rate Estimation in Packet-Switched Networks

Jeong-woo Cho and Dong-ho Cho

Department of Electrical Engineering and Computer Science
Korea Advanced Institute of Science and Technology (KAIST)
373-1 Kusong-dong, Yusong-gu, Taejeon 305-701, Korea

Abstract—While traffic volume of real-time applications is rapidly increasing, current routers do not guarantee minimum QoS values of fairness and they drop packets in random fashion. If routers provide a minimum QoS, resulting a less delay, reduced delay-jitter, more fairness, and smooth sending rates, TCP-Friendly Rate Control (TFRC) can be adopted for real-time applications. We propose a dynamic buffer management scheme that meets the requirements described above, and can be applied to TCP flow and to data flow for transfer of real-time applications. The proposed scheme consists of a virtual threshold function, an accurate and stable per-flow rate estimation, and a per-flow exponential drop probability. We discuss how this scheme motivates real-time applications to adopt TCP-Friendly Rate Control.

I. INTRODUCTION

TCP is the most widely used transport protocol on the Internet and is appropriate for FTP and Telnet, which both require reliability. However, because it uses an Additive Increase Multiplicative Decrease (AIMD) algorithm and induces coarse timeouts, it can neither ensure smoothly-changing sending rate nor can it be used for real-time applications [16]. Because most current routers use Drop Tail as a buffer management scheme, which does not guarantee fairness or delay bound and delay-jitter bound, there has been no motivation for real-time applications to use end-to-end congestion control mechanisms. For these reasons, real-time applications use robust congestion control schemes than TCP congestion control [11]. Even though Drop Tail is a simple buffer management scheme, it tends to penalize bursty traffic, such as TCP, does not guarantee fairness, and adds unnecessary delay because it doesn't drop any packets before the buffer space is fully exhausted.

Adopting a single FIFO queue, CSFQ (Core-Stateless Fair Queueing) [15] uses per-flow state only in edge routers. Entering the network, packets are marked with an estimate of their sending rate. A core router compares the estimate of each rate with the fair share of that flow and preferentially drops packets if the flow arrives at a higher rate than its fair share. Although CSFQ is much fairer, it requires an extra field in the IP header of every packet and CSFQ must be installed in contiguous fashion on routers.

RED (Random Early Detection) [6] and FRED (Flow Random Early Drop) [9] are the foundation of buffer management schemes because they are practicable and are designed in the consideration of burstiness of TCP flows. RED prevents full exhaustion of buffers and drops packets before congestion becomes severe. However, it does not prevent unresponsive flows from monopolizing buffer space, and TCP-friendly flows attain only a fraction of their fair share [4]. Also, it can not control queue size effectively and can not prevent buffer overflow when there are many flows [3]. To address the problem of unresponsive flows, in [4], authors stressed on the need for end-to-end

congestion control. Furthermore, they insisted that there should be some mechanisms on the network to identify and regulate unresponsive flows. Techniques to identify and punish unresponsive flows have been identified in [10], [5]. While these proposals are simple and feasible schemes that solve the problem of unresponsive flows, they can punish unlucky TCP-friendly flows with non-zero probability. FRED uses a per-flow state to solve the problem of unresponsive flows. Although FRED can not prevent buffer overflow for many flows, it is much fairer than RED and effectively regulates unresponsive flows.

Although RED and its variants can be satisfactory for applications that only require reliability, support for real-time applications requires a router to provide more functions. Moreover, to motivate real-time applications to use TFRC (TCP-Friendly Rate Control) [7], [8] a minimum QoS (Quality of Service) should be guaranteed. A key impediment to deployment of RED and its variants is that RED drops packets *randomly* with the same drop probability. As explained herein, a router should let a real-time application experience *periodic* packet loss when packet loss cannot be avoided. To solve these problems, we propose a new buffer management scheme that ensures better fairness between TCP-friendly flows and unresponsive flows, less delay, less delay-jitter, and smooth sending rates.

The organization of this paper is as follows: We discuss general requirements of buffer management schemes in packet-switched networks in Section 2. In Section 3, details of the algorithm we propose are explained with a discussion of mechanics of operation. In Section 4, we show simulation results obtained using our proposed scheme, RED, and FRED, and analyze the results. Section 5 presents an analysis of various topics relating to our scheme. In the last section, we present a conclusion.

II. REQUIREMENTS OF BUFFER MANAGEMENT SCHEME

RED is a simple and powerful buffer management scheme that drops packets from each flow in proportion to the amount of bandwidth the flows uses on the output link [9], assuming that all flows exhibit the same behavior as TCP flows do in view of packet drop events. However, RED cannot prevent buffer overflow for many flows, cannot regulate unresponsive flows, and is unfair even among TCP flows because it drops packets *randomly* [9], [4], [3], [10]. We suggest the following functions that an intelligent buffer management scheme should support:

1. Regulation of unresponsive flows and fairness
2. Low delay and low delay-jitter values
3. Smooth sending rates for each flow
4. Control of the queue size to prevent overflow and underflow

Fig. 1. Virtual threshold function vs. number of flows

A. How Much Buffer Should a Router Provide?

In ideal situations, routers can provide fairness even with a small buffer. But, TCP, which is the dominant transport protocol, requires a bigger buffer because it uses window-based congestion control that causes frequent coarse timeouts when there is insufficient buffer space. Even TCP-Newreno, which is one of the most widely used TCP variants [14] that is robust to consecutive multiple packet drops [2], wastes much time in fast recovery mode in which its sending rate is relatively low if there is insufficient buffer. This results in short-term unfairness. Although TCP flows requires that at least 4 packets per flow should be buffered in routers to prevent coarse retransmit timeouts [12], most routers provide very small buffers because large buffers cause long delays and long response times without an active buffer management scheme.

While RED maintains average queue size between max_{th} and min_{th} , it fails to eliminate unnecessary delay and does not effectively control per-flow queue sizes when there are only a few active flows. Large variation of per-flow queue sizes indicates a need to provide larger buffer to satisfy minimum per-flow queue sizes. However, with larger buffer size, RED unnecessarily allows more packets to be buffered when there are only a few flows. This feature induces unnecessary delay and delay-jitter. RED also experiences a delay-jitter even greater than Drop Tail [1]. Use of a large buffer is, therefore, not advisable. More efficient buffer management schemes are needed that can actively control per-flow queue sizes and provide minimum per-flow buffer space, eliminating unnecessary delays.

With a per-flow buffer management scheme, the average queue size can be actively controlled and each flow can buffer at least 4 packets because we can control maximum queue sizes of all flows. To minimize unnecessary queueing delay and to allow flow to buffer at least 4 packets, we propose a virtual threshold function, shown in Figure 1. In this figure, we divide router operation into three modes. Each flow can buffer up to $vmax_q/N_{flow}$ bytes. Because each TCP flow does not occupy more than $vmax_q/N_{flow}$ bytes all the time, exploiting the burstiness of TCP, we can maintain an average queue size to a target value, which is shown as $target_q$. In *no congestion* mode, there is sufficient buffer space to allow each flow to buffer 8000 bytes. In this mode, a router can provide highly

satisfactory QoS. In *moderate congestion* mode, there is insufficient buffer space and the queueing delay increases above a satisfactory threshold. In this mode, we allow each flow to buffer a smaller number of packets as the number of active flows increases. In *severe congestion* mode, each flow can buffer only a minimum number of packets and we can not provide low delay and low delay-jitter values. As demands on delay, delay-jitter, and per-flow buffer size can vary, the virtual threshold function can also vary according to these demands.

B. Why Should a Router Drop Packets Periodically?

TCP packet losses are detected in the following two ways: (1) The TCP sender can detect them either when it receives *triple-duplicate* acknowledgements, (four ACK's with the same sequence number), or (2) when *retransmit timeouts* occur [13]. We define the *congestion cycle* CC_i as the i th period between two loss indications. We define α_i as the number of packets including the first packet loss in CC_i . If RED is in steady state, (no recent change in the number of active flows), packets for flow i are dropped with nearly constant drop probability p . Therefore, α_i is distributed geometrically as follows:

$$P[\alpha_i = k] = (1 - p)^{k-1}p, \quad k = 1, 2, \dots \quad (1)$$

As can be seen from this equation, each flow experiences geometrically distributed inter-packet drop times. The mean and standard deviation of α_i are as follows:

$$E[\alpha_i] = \sum_{k=1}^{\infty} (1 - p)^{k-1}pk = \frac{1}{p}, \quad (2)$$

$$S[\alpha_i] = \frac{1}{p}\sqrt{1 - p}. \quad (3)$$

We can determine that $E[\alpha_i] = 10$ and $S[\alpha_i] = 9.5$ with $p = 0.1$, indicating that some flows buffer more than a sufficient number of packets and others buffer fewer than the necessary number of packets. This feature of RED causes unfairness, inefficient buffer usage, and rough sending rates. To avoid these problems, routers should drop packets periodically.

III. BARE ALGORITHM

We propose BARE (Buffer Management based on Rate Estimation) scheme which solves problems discussed in Section 2.

A. A Detailed BARE Algorithm

• Constants:

$\alpha = 1.2$; // increase factor of β
 $w_q = 0.004$; // weight for avg_q calculation
 $\beta_{max} = 10$; // maximum β
 $\beta_{min} = 0.5$; // minimum β
 $\beta_{init} = 7.5$; // initial β
 $K = 0.3sec$; // constant used for rate estimation
 $BW = 1250000 Bps$; // service rate

• Global Variables:

$vmax_q = vth(N_{flow})$; // virtual maximum queue size
 $target_q = vmax_q/2$; // target queue size
 N_{flow} ; // number of active flows
 max_{th} ; // maximum per-flow queue size
 q ; // current queue size
 avg_q ; // average queue size

time; // current real time

• **Per-flow Variables:**

q[*i*]; // queue size
rate[*i*]; // estimated rate
 β [*i*]; // β [*i*]
count[*i*]; // number of bytes processed
// since last β [*i*] update
qtime[*i*]; // last time packet is buffered

• **Functions:**

find_i(*p*); // find the flow number to which *p* belongs
update_q(*mode*) { // update *q* and *avg_q*
 if (*mode* == 1) *value* = *p.size*;
 else *value* = - *p.size*;
 q = *q* + *value*;
 avg_q = (1 - *w_q*) × *avg_q* + *w_q* × *q*;
}
update_beta(*rate_ratio*) { // update β [*i*]
 if (*rate_ratio* > 1 && *avg_q* > *target_q*) {
 β [*i*] = β [*i*] / *rate_ratio*;
 if (β [*i*] < β_{min}) β [*i*] = β_{min} ;
 } else {
 β [*i*] = β [*i*] × α ;
 if (β [*i*] > β_{max}) β [*i*] = β_{max} ;
 }
}
random(); // uniform random number in [0...1]
pow(*a*, *b*); // calculate and return a^b
exp(*c*); // calculate and return e^c

• **For each arriving packet *p*:**

```

1: if ( find_i(p) == false ) {
2:   Nflow = Nflow + 1;
3:   vmaxq = vth(Nflow);
4:   maxth = vmaxq / Nflow;
5:   targetq = vmaxq / 2;
6:   ratefair = BW / Nflow;
7:   q[i] = count[i] = p.size;
8:   rate[i] = 0;
9:    $\beta$ [i] =  $\beta_{init}$ ;
10:  qtime[i] = time;
11:  update_q(1);
12:  return;
13: }
14: q[i] = q[i] + p.size;
15: count[i] = count[i] + p.size;
16: if ( count[i] ≥ 4 × maxth ) update_beta(rate[i] / ratefair);
17: u = random();
18: if ( u < pow(q[i] / maxth,  $\beta$ [i]) ) {
19:   q[i] = q[i] - p.size;
20:   drop(p);
21: } else {
22:   update_q(1);
23:   dt = time - qtime[i];
24:   qtime[i] = time;
25:   rate[i] = (1 - exp(-dt / K)) × p.size / dt
      + exp(-dt / K) × rate[i];
26: }
```

• **For each departing packet *p*:**

```

27: find_i(p);
28: q[i] = q[i] - p.size;
29: if ( count[i] ≥ 4 × maxth ) update_beta(rate[i] / ratefair);
30: update_q(0);
```

BARE determines the per-flow buffer size depending on the number of currently active flows and drops packets based on rate estimation of each flow [15]. As an estimate of per-flow share, either a per-flow average queue size estimation in [6] or a per-flow rate estimation in [15] can be used. In fact, using per-flow average queue size requires replacing “**exp**(-*dt* / *K*)” with a constant “ λ ” and replacing rate estimates with average

queue estimates in code line 25 as follows. (In addition to this replacement, a portion of code should be modified.)

$$rate[i] \leftarrow (1 - e^{-dt/K}) \times \frac{p.size}{dt} + e^{-dt/K} \times rate[i], \quad (4)$$

$$avg_q[i] \leftarrow (1 - \lambda) \times q[i] + \lambda \times avg_q[i]. \quad (5)$$

The per-flow buffer occupancy of flow *i* is proportional to the per-flow output rate of flow *i* with the FIFO discipline [9]. Therefore, we can guess that these two approaches achieve the same performance. However, using per-flow average queue size as an estimate of the per-flow share is not as precise and efficient as using per-flow rate estimation. When per-flow average queue size is used as an estimate of per-flow share, there is no point at which we can achieve both filtering of unnecessary noise and quick responsiveness to rapid rate fluctuations. Assume that end of congestion cycle *CC_i* is caused only by triple-duplicate ACKs, there are only periodic packet losses, and the round trip time is fixed to *RTT*, and *W_i* is defined as the maximum window size in Congestion Cycle *CC_i*. With these assumptions, the inter-packet buffering time of TCP varies from *RTT* / *W_i* to (2 × *RTT*) / *W_i*, so the per-flow share can not be calculated accurately without dependency on *dt*. If there are substantial packet losses caused by timeouts, this discrimination becomes more significant. Therefore, we have chosen to use rate estimation as an estimate of per-flow share. Rate estimation in code line 25 is robust to various packet length distributions and is proven to asymptotically converge to the real rate [15].

Based on per-flow rate estimation and comparison of current average queue size with *target_q*, BARE either increases or decreases β [*i*]. Flow *i* experiences a high drop probability with a small β [*i*] and experiences a low drop probability with a large β [*i*]. Upon decrease, β [*i*] is divided by the *rate_ratio*. Upon increase, β [*i*] is multiplied by the same value of α . In this way, we can effectively regulate flows that are currently using more than their fair share. Flows currently using a fair share do not suffer, although they experience small variations in their drop probability. Using rate estimation in determining the fair share of flow *i* and adjusting the fluctuations of average queue size to nearby *target_q*, we can achieve both of two goals, i.e., per-flow fair rate allocation and per-flow buffer management while other schemes [15], [6], [9] exhibit weaknesses in either of these two areas. Achieving these two goals at the same time is important for real-time applications that require low delay and fairness and so forth simultaneously.

B. Per-flow Exponential Adjustment of the Drop Probability

As shown in code line 18, BARE drops packets for flow *i* with following drop probability:

$$p[i] = \left(\frac{q[i]}{max_{th}} \right)^{\beta[i]}. \quad (6)$$

Because RED drops flow *i*’s packets with a nearly constant drop probability, some flows buffer more than a sufficient number of packets while others flows buffer fewer than the necessary number of packets. This causes several problems (see Section 2). While dropping packets with a constant probability

Fig. 2. Simulation topology

$p = c < 1$ (c is a constant) when $q[i] < max_{th}$ and with a probability $p = 1$ when $q[i] \geq max_{th}$ would be simpler, in many cases, control of the per-flow queue size would be not achieved. With this dropping method, packets of TCP flows are dropped with a probability $p = 1$ (when $q[i]$ is greater than or equal to max_{th}) and not with a probability $p \ll 1$ ($q[i] < max_{th}$) because c should be small to prevent geometrically distributed inter-packet drop times and to prevent packet drops at low per-flow queue sizes. This dropping method can not avoid phase effect and is inefficient in view of packet dropping.

With per-flow exponential adjustment of drop probability, we can achieve a high degree of fairness and smooth sending rates because packets of flow i are dropped nearly periodically. Furthermore, the queue size of each flow is well regulated and each flow is not allowed to buffer more than the necessary number of packets. If the sending rate of flow i does not exceed fair share (when $q[i] \leq max_{th}/2$), packets of flow i are dropped with a negligible probability, i.e., $0.5^{0.5} = 0.0055$. Therefore, this scheme does not drop flow i 's packets when the number of buffered packets for flow i is less than the number of buffered packets for other flows. With this dropping method, we can also effectively control the delay and delay-jitter.

IV. SIMULATION RESULTS

BARE, RED, and FRED are compared based on simulation results. RED is selected as a fundamental scheme and FRED is selected as a comparable scheme with BARE.

A. Simulation Configurations

We simulated the configuration shown in Figure 2. Unless otherwise specified, the following parameters were used. Each output link had a capacity of 10 Mbps, a latency of 2 ms, and a single FIFO buffer of 160 kbytes. For RED and FRED, min_{th} was set to 40 kbytes and max_{th} was set to 160 kbytes. TCP-Newreno was used in all simulations because it is the most widely used TCP variant as shown in [14] for its robustness against consecutive packet drops. The data packet size of TCP flows was set to 1000 bytes and the ACK packet size was set to 40 bytes. All BARE parameters were set to the values indicated in Section 3.1. To avoid the buffer space being fully exhausted, max_q was set to 0.1 for RED and 0.2 for FRED. The FRED min_q value was set to 2000 bytes. All three schemes were im-

Fig. 3. Mean queueing delay vs. number of active flows

Fig. 4. Standard deviation of queueing delay vs. number of active flows

plemented in ns-2 [17]. RED and FRED operated in byte mode, meaning that packets were buffered in bytes and dropped with a probability proportional to their size. To reduce instantaneous noise and to avoid phase effects, each simulation was run for 100 seconds and each flow was started at a random time. We introduce the term G_{adapt_i} as follows.

- For TCP: the G_{adapt_i} of TCP flow i is defined as the number of bytes cumulatively acknowledged by TCP Sink (Receiver), excluding outstanding (not acknowledged) packets, which is divided by simulation time T .
- For CBR(on UDP): the G_{adapt_i} of CBR flow i is defined as the total number of bytes received by CBR Sink which is divided by simulation time T .

B. Queueing Delay and Fairness for TCP flows

We simulated N_{flow} TCP flows and there was no CBR flow. As shown in Figure 3, BARE reduces the unnecessary queueing delay and maintains a much smaller average queue size compared with RED and FRED. In fact, if the average queue size of BARE is set to $target_q$ of the virtual threshold function, the delay is controlled to the corresponding value. Although the delay increases as the number of active flows increases, BARE maintains a much lower delay of nearly half of the values for RED and FRED.

Fig. 5. Standard deviation of $(Goodput_i / Fair Share)$ vs. number of active flows

Fig. 6. Loss events of source 1

As shown in Figure 4, the delay-jitter is greatly reduced because BARE actively and effectively controls the per-flow queue size with a virtual threshold function and a per-flow exponential adjustment of the drop probability. As the number of flows increases, the delay-jitter values of BARE and FRED converges to the same value because, when the number of active flows approaches 40, per-flow buffer sizes are allowed only up to 3 data packets (3000 bytes, because BARE drops flows i 's packets with a probability 1 when $q[i]$ reaches 4000 bytes, according to code line 18), and BARE can not effectively control per-flow queue sizes.

As shown in Figure 5, under the same condition as mentioned above, we measured the standard deviation of the goodput for each flow, which is normalized by the fair share of that flow. The standard deviation S of $(Goodput_i / Fair Share)$ is defined as follows.

$$S = \sqrt{\frac{1}{N_{flow} - 1} \sum_{i=1}^{N_{flow}} \left(\frac{Goodput_i}{BW/N_{flow}} - 1 \right)^2}. \quad (7)$$

BARE achieves extreme fairness that can not be compared with RED. However, as the number of active flows increases, per-flow buffer sizes are allowed only up to 3 data packets and per-flow queue sizes are not well controlled. With 40 flows, BARE achieves nearly the same fairness as FRED.

Packet loss events were observed with 20 TCP flows. In Figure 6, packet loss events of source 1 are shown. RED and FRED drop packets in a random fashion, as expressed in equation 1,

Fig. 7. Average of $(Goodput_i / Fair Share)$

while BARE drops packets nearly periodically. With this *periodic packet drop*, BARE can effectively control per-flow queue sizes and prevent flow i from buffering more than the necessary number of packets. BARE never drops packets when queue size of flow i is small.

C. Fairness for TCP and CBR flows

We simulated TCP and CBR (which uses UDP as a transport protocol) flows. In Figure 7, the total number of flows was set to 20. We changed the fraction of CBR flows. All CBR flows sent data at 1 Mbps (packet size=1000 bytes, inter packet time=8ms), which is twice the fair share value. RED can not protect TCP flows from unresponsive CBR flows, even when there are only 2 CBR flows. With 50% of CBR flows in RED, TCP flows achieve only 29.8% of their fair share. However, we can see that FRED protects TCP flows from CBR flows to a certain extent. While FRED can protect TCP flows with small number of CBR flows, as the number of CBR flows increases, FRED loses the ability to protect TCP flows from CBR flows. Because CBR flows increase per-flow average queue size ($avgq$ in FRED) and FRED often excludes relatively low per-flow queue size of TCP flows from its calculation of $avgq$, the average queue size is greatly increased and CBR flows are not efficiently dropped when there are many CBR flows.

The discrimination between average goodput of TCP and CBR flows is high even with FRED. With 2 CBR flows in FRED, although TCP flows achieve 96.2% of their fair share, CBR flows achieve 134.5% of their fair share. Although TCP flows achieve only slightly less than their fair share, CBR flows achieve much more than their fair share because CBR flows take away a portion of the share from each of the 18 TCP flows. This motivates usage of CBR and discourages usage of TCP-Friendly Rate Control for real-time applications. We do not think that end-to-end congestion control would be used by real-time applications that are currently using CBR (UDP without conforming end-to-end congestion control) unless their share is regulated to be comparable to TCP and TCP-Friendly flows and unless minimum QoS values are supported. With respect to this *encouragement of unresponsive flows*, BARE partially solves this problem. With 2 CBR flows for BARE, TCP flows achieve 98.6% of their fair share and CBR flows achieve 112.8% of their fair share.

Fig. 8. Coefficient of Variation of TFRC flows vs. number of active flows in case of $T_m = 1.0$ seconds

D. Instantaneous Sending Rates of TFRC Flows

We simulated 40% TFRC flows and 60% TCP flows and measured the mean and standard deviation of CoV_i (Coefficient of Variation) of TFRC flows with the measurement time $T_m = 1.0$ seconds [7]. Total number of TFRC and TCP flows was set from 5 to 40 with a step size of 5. The packet size was set to 500 bytes and the other parameters were set to values from [7]. We measured the throughput instead of the goodput. TFRC operates with an equation-based rate control that characterizes TCP sending rates [13] with the following equation:

$$T = \frac{s}{R\sqrt{\frac{2p}{3}} + t_{RT}(3\sqrt{\frac{3p}{8}})p(1 + 32p^2)}. \quad (8)$$

An upper bound on the sending rate T is used, which is a function of the steady-state loss event rate p , data packet size s in bytes, round-trip time R , and the TCP retransmit timeout value t_{RTO} . TFRC estimates the average loss interval, which is a weighted sum of last n loss intervals considering consecutive packet loss events as a single loss event. TFRC uses the average loss interval to calculate the sending rate. We can easily see that TFRC flows should experience periodic packet loss events to estimate p accurately without noisy fluctuation. Results are shown in Figure 8. As can be seen from this figure, TFRC flows in RED experience noisy instantaneous sending rates, in contrast to BARE and FRED. This feature of BARE should encourage adoption of TFRC for real-time applications as a congestion control mechanism. Irregularities in case of $N_{flow} = 35$ and 40 for FRED should be mentioned. Because FRED marks flows that occupy more than 2 times the $avgcq$ value, TFRC flows are sometimes marked more than once ($strike_i$ in [9] is larger than 1) and regulated with a higher drop probability than other flows. If FRED marks flow i , which uses a rate-based congestion control such as TFRC, marking does not expire during the lifetime of flow i because flow i always has at least 1 packet buffered in the router buffer. This causes some TFRC flows to experience a high CoV_i value. For RED, there are severe buffer overflows with $N_{flow} = 40$ because TFRC flows are robust than TCP flows when drop probability is high.

Fig. 9. Total dwell time in full buffer state vs. number of active flows

E. Supporting Many Flows

The number of flows a router can support with a fixed buffer size is an important issue. Although the overall performance suffers degradation with many flows, a buffer management scheme cannot achieve fairness if it can not avoid buffer overflows. If there are significant buffer overflows, a router cannot buffer newly started bursty flows and applications that send data at a relatively low rate requiring only reliability and an immediate response of the the correspondent, such as Web and Telnet applications. Therefore, support for many flows is a crucial responsibility of a router. We define a *full buffer state* as the state when the queue size is greater than or equal to 95% of max_q . In Figure 9, the total dwell time in the state for each scheme is shown. We simulated only TCP flows during a total simulation time set to 100 seconds. BARE supports more flows with the same buffer size, up to 68 flows, without significant buffer overflows.

V. MISCELLANEOUS DETAILS

A. Choosing K

The choice of K involves several tradeoffs. First, while a smaller K value increases the system responsiveness to rapid rate fluctuations, a larger K value better filters noise and avoids potential system instability. Second, K should be large enough to smooth the sending rates of TCP flows because these rates are estimated to be high when flows have large window size just before packet drop events. To control these effects, as a rule of thumb, we recommend that K should be 3 times the maximum queueing delay, which can be calculated based on dividing the buffer size by the link speed.

B. Deleting Per-flow State

Because routers have a limited memory, the per-flow state should be deleted properly, but neither too often nor too seldom. With frequent deletion of the per-flow state, the aggregate queue size fluctuates and significant delay-jitter occurs. With infrequent deletion of the per-flow state, rate estimates are incorrect and some flows suffer unfairly. From code line 25, the rate of flow i is updated according to equation 4. In equation 4,

if $e^{-dt/K}$ is set to $e^{-4} = 0.0183$, dt should be $4K$ and equation 4 becomes:

$$rate[i] \leftarrow 0.98 \times \frac{p.size}{dt} + 0.0183 \times rate[i] \approx \frac{p.size}{dt}. \quad (9)$$

Therefore, routers do not have to maintain the per-flow state of flow i if $dt = 4K$ seconds has elapsed since the last buffering operation of flow i . We recommend that router delete the per-flow state of flow i when there has been no buffering operation of flow i for $4K$ seconds and $q[i] = 0$. Considering long round trip times, TCP retransmit timeouts with multiple links, and queueing delay of the packet that caused the last buffering operation (Because $qtime[i]$ is updated not with a packet departure but with a packet arrival for flow i , this delay should be considered.), we recommend that *Timeout Value* should be set to 1.5 seconds.

VI. CONCLUDING REMARKS

We have proposed a dynamically adjusting per-flow buffer management scheme that can be applied to TCP flows and to flows transferring data of real-time applications. We have simulated various configurations with TCP, CBR and TFRC flows. BARE exhibits better fairness, less delay, less delay-jitter, and better smoothness of sending rates with less complexity than previous schemes. Introduction of a virtual threshold function that divides router operation into three modes allows the average queue size to fluctuate around the $target_q$ value and eliminates unnecessary delay. BARE also produces more efficient buffer usage that helps routers support more flows with better performance than RED and FRED with the same buffer size. The per-flow rate estimation was accurate in view of estimating the per-flow current share, and noisy and rapid fluctuations were filtered. Per-flow exponential adjustment of the drop probability prevents unresponsive flows from achieving an unfairly large share. BARE also controls the per-flow queue size, preventing flows from buffering more than a sufficient number of packets and fewer than the necessary number of packets. BARE

can support real-time applications and can encourage the use of end-to-end congestion control mechanisms, such as TFRC.

REFERENCES

- [1] T. Bonald, M. May, J. Bolot, Analytic Evaluation of RED Performance, in: Proceedings of the IEEE INFOCOM2000, March 2000, pp. 1415–1425.
- [2] K. Fall, S. Floyd, Simulation-based Comparisons of Tahoe, Reno, and SACK TCP, ACM SIGCOMM Comput. Commun. Rev. 26 (3) (1996) 5–21.
- [3] W. Feng, D.D. Kandlur, D. Saha, K.G. Shin, A Self-Configuring RED Gateway, in: Proceedings of the IEEE INFOCOM'99, March 1999, pp. 1320–1328.
- [4] S. Floyd, K. Fall, Promoting the Use of End-to-end Congestion Control in the Internet, IEEE/ACM Trans. Networking 7 (4) (1999) 458–472.
- [5] S. Floyd, K. Fall, Router Mechanisms to Support End-to-End Congestion Control, February 1997. LBL Technical Report.
- [6] S. Floyd, V. Jacobson, Random Early Detection Gateways for Congestion Avoidance, IEEE/ACM Trans. Networking 1 (4) (1993) 397–413.
- [7] S. Floyd, J. Padhye, J. Widmer, Equation-Based Congestion Control for Unicast Applications, in: Proceedings of the ACM SIGCOMM2000, September 2000, pp. 43–56.
- [8] S. Floyd, J. Padhye, J. Widmer, TFRC, Equation-Based Congestion Control for Unicast Applications: Simulation Scripts and Experimental Code, February 2000. URL <http://www.aciri.org/tfrc/>.
- [9] D. Lin, R. Morris, Dynamics of Random Early Detection, in: Proceedings of the ACM SIGCOMM'97, October 1997, pp. 127–137.
- [10] R. Mahajan, S. Floyd, Controlling High-Bandwidth Flows at the Congested Router, November 2000. Work in progress, URL <http://www.aciri.org/red-pd/>.
- [11] A. Mena, J. Heidemann, An Empirical Study of Real Audio Traffic, in: Proceedings of the IEEE INFOCOM2000, March 2000, pp. 101–110.
- [12] R. Morris, TCP Behavior with Many Flows, in: Proceedings of the IEEE ICNP'97, October 1997.
- [13] J. Padhye, V. Firoiu, D.F. Towsley, Modeling TCP Reno Performance: A Simple Model and Its Empirical Validation, IEEE/ACM Trans. Networking 8 (2) (2000) 133–145.
- [14] J. Padhye, S. Floyd, Identifying the TCP Behavior of Web Servers, February 2001. ICSI Technical Report TR-01-002, URL <http://www.aciri.org/tbit/>.
- [15] I. Stoica, S. Shenker, H. Zhang, Core-Stateless Fair Queueing: Achieving Approximately Fair Bandwidth Allocations in High Speed Networks, in: Proceedings of the ACM SIGCOMM'98, September 1998, pp. 118–130.
- [16] D. Tan, A. Zakhor, Real-time Internet Video Using Error Resilient Scalable Compression and TCP-friendly Transport Protocol, IEEE Trans. Multimedia 1 (2) (1999) 172–186.
- [17] UCL/LBNL/VINT Network Simulator - ns (version2), URL <http://www.isi.edu/nsnam/ns/>.