On Reducing the Degree of Self-Similarity in Network Traffic

Biplab Sikdar, Kenneth S. Vastola and S. Kalyanaraman
Dept. of ECSE, Rensselaer Polytechnic Institute
Troy, NY 12180 USA

Abstract— **While it is well known that self-similarity in network traffic leads to larger queueing delays, higher drop rates and extended periods of congestion, reducing the self-similarity of traffic and consequently eliminating its harmful effects has remained an open problem. In this paper we present some techniques to reduce the degree of self-similarity of network traffic, specifically by reducing two of its TCP related causes: (1) timeouts and exponential backoffs (2) burstiness and ACK compression. We show that a simple modification to the RED algorithm, aimed at reducing the timeouts and exponential backoffs in TCP flows, can lead to significant reductions in the traffic self-similarity under a wide range of network conditions, as compared to the currently implemented active and passive buffer management policies. We then show that TCP pacing can result in large reductions in the self-similarity by eliminating the inherent burstiness of TCP flows and the effects of ACK compression. We also show that though our techniques are aimed at TCP related causes of self-similarity, it is also effective in reducing the degree of self-similarity in traffic even when application and user level causes are also present, as long as TCP is used as the underlying transport protocol.**

Keywords—**Self-similarity, traffic modeling, TCP**

I. Introduction

The self-similar nature of network traffic has been established in a variety of network environments and its causes have been traced to various factors. These causes range from source and application level behavior of traffic sources [4], [25] to transport layer [7], [9], [11], [16], [20], [23], [24] as well as human factors [4]. Also, it is well known that the self-similar and multifractal nature of traffic leads to a number of undesirable effects like high buffer overflow rates, large delays and persistent periods of congestion [5], [6], [17], and the severity of these conditions is directly proportional to the degree of self-similarity or the Hurst parameter. However, so far no breakthrough has been made in reducing the degree of self-similarity and thereby limiting these harmful effects. In this paper, we present some techniques which can be used to reduce the degree of self-similarity in TCP traffic.

Supported in parts by DARPA contract F19628-98-C-0057, DoD MURI contract F49620-97-1-0382 and NSF grant ANI 9806660.

Our focus in this paper is to eliminate the causes of self-similarity which arise from the protocol related causes described in [7], [9], [11], [16], [20], [23], [24]. The reason behind this is that while source level behavior [25], file size distributions and human factors like response time are causes of self-similarity [4], they cannot be easily changed. On the other hand, the underlying transport protocols and the queueing disciplines at the routers which affect the performance of these protocols are easily modified and thus are more attractive and practical avenues.

We propose two different techniques in this paper which reduces the degree of self-similarity in TCP traffic. The underlying idea of the first technique is to reduce the incidence of timeouts and exponential backoff in TCP flows which is a major contributor of the self-similarity of TCP traffic as shown in [9], [11], [20]. We do this by looking at existing and also proposing new buffer management policies to reduce correlated losses (and consequently timeouts) in TCP flows. The second technique is to eliminate the inherent burstiness and back-to-back packet transmissions in TCP flows. One of the reasons behind this behavior is the phenomenon of ACK compression in TCP flows [26]. ACK compression has also been suggested as a possible cause of the fractal nature of network traffic in [8]. We look at the impact of reducing the effects of ACK compression and burstiness of TCP sources through TCP pacing [26] on the self-similarity of traffic. Our results show that both these techniques are very successful in reducing traffic self-similarity. These methodologies are successful in eliminating the self-similarity for low and moderate loads while for high loads we have a very large reduction though not complete elimination of self-similarity.

We also note here that the techniques proposed in this paper are effective only with the TCP related causes of network traffic self-similarity. These techniques do not eliminate the contribution of the source, application and user level causes of self-similarity. However, the overall self-similarity of the traffic is from the complex interaction of all these causes and the reduction in one cause can favorably affect the impact of other causes. Also, since most of the traffic in today's networks are transferred using TCP, reducing TCP's contribution itself reduces the overall self-similarity of network traffic considerably. To support this claim, we show using simulations that even with the presence of web-traffic with heavy-tailed file sizes and typical user behavior patterns, the self-similarity of the traffic is eliminated at low loads and significantly reduced at moderate and high loads if we implement our proposed schemes.

The rest of the paper is organized as follows. In Section II we give a brief overview on the contribution of TCP related causes to the self-similarity of network traffic. In Section III we investigate the impact of three buffer management schemes on traffic self-similarity: taildrop, RED and a proposed modified RED algorithm. Section IV we concentrate on reducing the burstiness of TCP flows and also eliminating the harmful effects of ACK compression and their impact on traffic self-similarity. Finally, Section V presents the discussions and concluding remarks.

II. TCP and Self-Similarity

In this section we briefly review the effect of TCP on the self-similarity of network traffic. Protocol related causes of self-similarity have been investigated in [7], [16] where it is shown that closed-loop protocols like TCP lead to greater degree of self-similarity and richer scaling behaviors compared to open loop protocols like UDP. In [23] the authors show that TCP's congestion control mechanism can lead to chaotic behavior and the interaction of a number of such chaotic flows was suggested as the cause of self-similarity in TCP traffic. In [24], it is shown that TCP flows can propagate self-similarity across bottleneck links and the adaptive nature of TCP's transmission rate or $cwnd$ to the available bandwidth is cited as the reason behind it.

In [20] the authors investigated how TCP's timeout and exponential backoff mechanisms contributed to the self-similarity of TCP traffic. The contribution of these causes is most visible in 3-4 time-scales ranging from milliseconds to tens of seconds. Similar conclusion have also been reached independently in [9] and [11] where the scaling over a limited number of time scales has been termed "pseudo self-similarity". These papers showed that the degree of self-similarity in TCP traffic is directly proportional to the degree of losses and larger incidence of timeouts and exponential backoffs implies a greater value of the Hurst parameter.

One of the reasons behind the high percentage of losses being recovered using timeouts in TCP flows are the bursty or correlated losses in the taildrop queues. On the other hand, with the independent losses which might arise in active queue management techniques like Random Early Detection (RED), some wireless and ATM scenarios the degree of timeouts is generally lesser. This can be inferred from the derivations of [12], [19] and [15], [18] where the probability that an arbitrary loss leads to a timeout is evaluated in the independent and correlated loss cases respectively. Thus it seems intuitive that networks with RED queues at the bottleneck lead lead to lesser degrees of self-similarity in TCP traffic passing through them. We verify this in the next section. In the next section, we also propose a change to RED's packet dropping policy which leads to even further reduction in the traffic self-similarity without degrading the throughput and loss rates.

Another reason which contributes to the self-similar nature of traffic is the burstiness of TCP traffic [7], [16]. This inherent burstiness of TCP flows where back to back packets are transmitted also contributes to the bursty nature of the losses. One of the main reasons behind this burstiness is the phenomenon of ACK compression [14], [26]. ACK compression is a phenomenon when a bunch of ACKs are sent over an interval ΔT at the receiver but arrive at the sender over an interval smaller than ΔT. This leads to the source to sending back to back packets thereby making the traffic bursty and with the ACKs being received quicker than they were sent, the sender might be misled into sending more data than the network can accept [14]. This in turn contributes to the phenomenon of multiple packet drops from a window of packets thereby causing timeouts. Also, it has been suggested in [8] that ACK compression in itself can contribute to the fractal nature of network traffic. Thus it is intuitive that eliminating ACK compression or its effects on the packet transmissions in the forward direction would lead to a substantial reduction in the traffic's self-similarity. In Section IV we look at TCP pacing [26] as a solution to the ill effects of ACK compression show that it can indeed lead to significant reductions in the self-similarity of TCP traffic.

III. Buffer Management Policies and Self-Similarity

In this section we look at the effect of different of buffer management policies on the self-similarity of traffic passing through it. We consider both passive and active queue management algorithms as represented by taildrop and RED queues respectively. We also propose a change to the current RED algorithm which results in lower degrees of self-similarity. We first give a brief description of the three policies and evaluate their performance in terms of the packet drop probabilities in the presence of bursty traffic. We also investigate how such a loss pattern will affect the behavior of TCP flows in terms of the number of timeouts before presenting the results on their impact on the self-similarity of network traffic.

We first note here that in [22] the effect of a leaky-bucket based policing mechanism on long-range dependent input traffic. The authors conclude that the long-range dependence cannot be removed by the flow control schemes currently being considered in broadband networks. However, the long-range dependent arrival process considered in [22] is very different from that obtained from TCP flows in the sense that TCP flows react to the losses and are dynamic in nature. In contrast, the arrival process of [22] maintains the same statistical properties irrespective or losses or delays at the queue. Given the adaptive nature of TCP traffic, we expect the buffer management policy to have a strong impact on the self-similarity of the traffic.

A. Taildrop Queues

Taildrop queues are currently the most widely implemented queueing mechanisms in routers in the Internet [15]. The first-in-first-out (FIFO) policy of taildrop queues, coupled with the bursty nature of TCP traffic implies that the packet drops from a taildrop queue become correlated and multiple packets can get dropped from the same window. To get a quantitative feel of the amount of drops resulting in taildrop versus RED and our modified RED queue in the presence of bursty traffic, we use the methodology of [13]. Consider a taildrop queue with a buffer size of K packets. We assume that packets arrive according to a batch Poisson process where bursts of B packets arrive at the queue with an exponentially distributed interarrival with parameter λ. The time to process each packet at the queue is assumed to be exponentially distributed with parameter μ. The offered load to the queue is thus $\rho = B\lambda/\mu$. We note that this arrival process is very simplistic and results obtained with the assumption of Poisson arrivals would be a lower bound on that observed for an input process with long range dependence [13]. The steady state distribution for the queue occupancy can be easily calculated using Markov chains for $M/M/1/K$ queues with batch arrivals. Let $\pi(i)$ denote the stationary probability that there are i packets in the queue. The probability that an arriving packet is dropped is then given by

$$P_{TD} = \pi(K) + \frac{B-1}{B}\pi(K-1) + \cdots + \frac{1}{B}\pi(K-B-1) \tag{1}$$

To model the effect of taildrop queues on the probability of timeouts in TCP flows, we use the correlated loss model used in [15] and [18]. In this model, a packet in a window is lost independently of losses in other rounds. However, losses within a window are correlated and all packets following the first packet to be lost in window are also assumed to be lost. As noted in [15] and [18], this model is quite realistic for taildrop queues with TCP traffic given the bursty nature of TCP sources with back to back packet transmission. The approximation becomes even more more accurate as the load becomes high and the probability $\pi(K)$ becomes large. With correlated losses, the probability that an arbitrary packet loss in a TCP flow with $cwnd = w$ and a loss rate of p leads to a timeout is given by [15], [18]

$$Q(w) = \begin{cases} 1 & \text{for } 1 \le w \le 3 \\ 1 - \frac{p(1-p)^{2w-1}}{1-(1-p)^w} & \text{for } 4 \le w \le 8 \\ 1 - \frac{p(2-p)(1-p)^{2w-2}}{1-(1-p)^w} & \text{for } 9 \le w \le W_{max} \end{cases} \tag{2}$$

where W_{max} is the maximum allowable window size.

B. RED Queues

RED is an active queue management algorithm which randomly drops packets before a queue becomes full, so that end nodes can respond to congestion before buffers overflow and was proposed in [10]. While conflicting claims have been made in literature regarding the effectiveness of RED at decreasing loss rates, preventing global synchronization and providing lower delays or jitter [3], [10], [13], our interest in RED in this paper is from its impact on the traffic self-similarity and not on other performance issues. We, however, also compare the throughput and the loss rates obtained by the three policies.

RED probabilistically drops packets even before the queue is full based on a weighted average of the queue length. By using a weighted average (the weighting factor is denoted by w_q), RED tries to avoid over-reactions to bursts and instead reacts to long term trends. The RED queue maintains two thresholds which determine the rate of packet drops: a lower threshold, min_{th}, and an upper threshold, max_{th}. When a packet arrives at the queue, we have three cases for packet drops depending on whether the weighted average queue 1) is below min_{th}, 2) is between min_{th} and max_{th} and 3) greater than max_{th}. The drop function $d(k)$ as a function of the average queue length k is given by

$$d(k) = \begin{cases} 0 & \text{for } k < min_{th} \\ \frac{k-minth}{max_{th}-min_{th}} max_p & \text{for } min_{th} < k < max_{th} \\ 1 & \text{otherwise} \end{cases} \tag{3}$$

where max_p is a control variable denoting the maximum drop probability. The reader is referred to [10] for the detailed RED algorithm.

To compare the performance of a RED queue against a taildrop queue in the presence of bursty traffic, we again consider a queue with K buffers with batch arrivals. The batches are again of size B with exponentially distributed interarrival and service times with rates λ and μ respectively. As in [13], we now make the assumption that the drop rate $d(k)$ depends on the instantaneous rather than the average queue size. We can again use a Markov chain based on the queue occupancy to calculate the stationary probabilities $\pi(k)$. We note that this distribution is different from that obtained for the taildrop queue and corresponds to a batch $M/M/1/K$ queue with discouraged arrivals. The discouraged arrivals corresponds to the drop function $d(k)$ applied on each arriving batch. The packet loss probability for the RED queue can then be obtained as in [13] and is given by

$$P_{RED} = \pi(K) + d(K-1)\pi(K-1) + \cdots + d(1)\pi(1) \quad (4)$$

The above equation assumes that all the packets that arrive in a burst see the same drop function. In reality, the difference between the drop probabilities experienced by the first and the last packet in the burst can be at most $\delta d(k) = d(k+b-1) - d(k)$. Thus Equation (4) provides a lower bound on P_{RED} and is accurate when min_{th} is small or max_p is large.

For a RED queue the packet drop pattern is closely modeled by an independent loss model as noted in [19] and the references therein. When the average queue length is between min_{th} and max_{th}, as it should be when the control parameters are properly selected, the incoming packets are dropped randomly conforming to the independent loss model. In the independent loss model losses in a window are assumed to be independent of losses in other rounds. Additionally, losses in a given window are also assumed to be independent of each other. From [12] and [19], with the independent loss model the probability that an arbitrary packet drop leads to a timeout in a TCP flow with a window of w packets experiencing a loss rate of p is given by

$$Q(w) = \begin{cases} 1 & \text{for } 1 \leq w \leq 3 \\ 1 - \frac{wp(1-p)^w}{1-(1-p)^w} & \text{for } 4 \leq w \leq 8 \\ 1 - \frac{wp(1-p)^w}{1-(1-p)^w} & \text{for } 9 \leq w \leq W_{max} \\ -\frac{w(w-1)p^2(1-p)^{w+n-2}}{1-(1-p)^w} & \end{cases} \quad (5)$$

Note that with independent losses, the probability of timeouts is much lesser than that predicted by the correlated model. This leads to the intuition that the self-similarity in traffic passing through RED queues will be much lesser than that through taildrop queues. We verify this intuition through simulations later in this section.

Algorithm 1 Modified Dropping algorithm of RED

last_drop_flag $\Leftarrow 0$
for Each Packet Arrival **do**
  if last_drop_flag = 1 **then**
    last_drop_flag = 0;
    goto enqueue;
  else if $min_{th} < avg < max_{th}$ **then**
    with probability $d(k)$, drop the packet
    if packet is dropped **then**
      last_drop_flag = 1;
    end if
  else if $max_{th} < avg$ **then**
    Drop the packet;
    last_drop_flag = 1;
  else
    goto enqueue;
  end if
end for

C. *Modified RED Queues*

The independent loss model for the packet drop in a RED queue is very accurate for the cases when the average queue length stays between max_{th} and min_{th}. However, if the offered load is so high that the average queue length becomes close to max_{th}, RED fails to perform better than taildrop queues and the traffic passing through the RED bottleneck can have higher values of H as compared to taildrop queues. This is due to the fact that when the average queue length becomes greater than max_{th}, RED drops each packet with probability 1. This leads to multiple packet drop from the same window, resulting in timeouts.

To deal with this situation, we propose a small change to RED's dropping policy similar to the one proposed in [2]. The new algorithm for packet dropping is shown in Algorithm 1. The idea is *not* to drop any two consecutive packets which arrive at the queue, unless of course if the queue is full. Since TCP generally sends back to back packets, ensuring that no two consecutive packets are dropped will reduce the probability that multiple packets from the same window are dropped, thereby reducing the occurrence of timeouts. Note that in the algorithm, we do not change the unconditional dropping probability $d(k)$ as calculated for the original RED algorithm.

To calculate the probability that any arbitrary packet arriving when the average queue size is k is dropped, $d'(k)$, we first refer to Figure 1. The three states, denoted by $\{i, j\}$ with $i, j \in 0, 1$ and $i = j \neq 1$, represent the possible conditions the queue can be in depending on whether the current or the previous packet was dropped or not. If

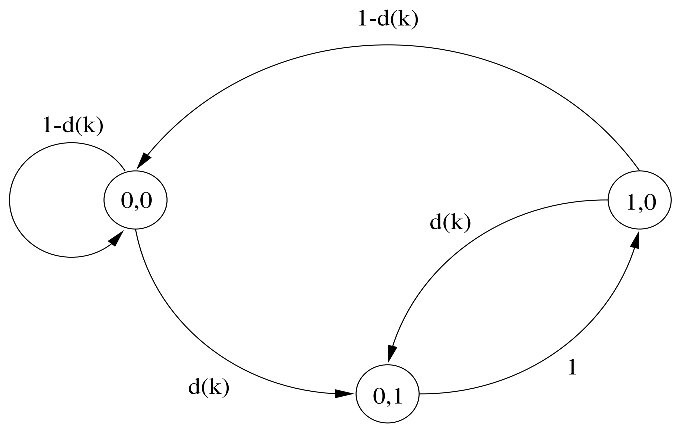

Fig. 1. Packet drop probabilities with the modified RED algorithm

the last packet to arrive in the queue was dropped, the corresponding states have $i = 1$ and $i = 0$ otherwise. Similarly, $j = 1$ implies that the current packet is dropped while $j = 0$ implies that the current packet is not dropped. When the queue is in state $\{0, 0\}$ the next packet to arrive is dropped with probability $d(k)$ and we reach state $\{0, 1\}$. On the other hand, the packet is not dropped with probability $1 - d(k)$ and we stay in state $\{0, 0\}$. When we are in state $\{0, 1\}$, we cannot drop the next packet because we are not allowed to drop two consecutive packets (i.e. the flag last_drop_flag=1) and we move to state $\{1, 0\}$ with probability 1. In state $\{1, 0\}$, the current packet is not dropped implying last_drop_flag=0. Thus the next packet is dropped with probability $d(k)$ and the state becomes $\{0, 1\}$. In case the next packet is not dropped (which occurs with probability $1 - d(k)$) we move on to state $\{0, 0\}$. The global balance equations for the transition probabilities from the three states can be written as

$$\begin{aligned} P_{0,0}d(k) &= P_{1,0}(1 - d(k)) \\ P_{0,1} &= P_{0,0}d(k) + P_{1,0}d(k) \\ P_{1,0} &= P_{0,1} \end{aligned} \quad (6)$$

where $P_{i,j}$ denotes the steady state probability of being in state i, j. These steady-state are then given by

$$\begin{aligned} P_{0,0} &= \frac{1 - d(k)}{1 + d(k)} \\ P_{0,1} &= \frac{d(k)}{1 + d(k)} \\ P_{1,0} &= \frac{d(k)}{1 + d(k)} \end{aligned} \quad (7)$$

The probability that any arbitrary arriving packet is dropped when the average queue length is k, $0 \le k < qlen$, is thus

$$\begin{aligned} d'(k) &= P_{0,0}d(k) + P_{1,0}d(k) \\ &= \frac{d(k)}{1 + d(k)} \end{aligned} \quad (8)$$

The analysis above assumes that successive packets see the same weighted average queue length and thus the same $d(k)$. We note that if max_p is small as is suggested in literature on RED parameter configuring, then this modification does not significantly affect the drop rates while the average queue length is less than max_{th}. However, when the average queue length exceed max_{th} but is less than $qlen$, the packet drop probability becomes 0.5 as compared to 1 in RED. This is a sufficiently high drop rate to force TCP sources to reduce their transmission rates but without inducing a lot of timeouts.

To get a feel of the packet loss probability of the modified RED queue under bursty traffic as compared to taildrop and RED queues, we again use the batch Poisson process as the input process. The packet service times are exponentially distributed with rate μ while the batches arrive with rate λ with size B. Also, as in the RED case, we assume that the drop rate $d(k)$ depends on the instantaneous rather than the average queue size and that all the packets in a burst see the same drop function. We can again use a Markov chain based on the queue occupancy to calculate the stationary queue length distribution $\pi(k)$. Note that this distribution is different from that obtained for the RED queue though it is also modeled as a batch $M/M/1/K$ queue with discouraged arrivals. This is because the function corresponding to the discouraged arrivals is different for the modified RED algorithm and is now additionally dependent on $d'(k)$. The packet loss probability for the modified RED queue can then given by

$$P_{RED} = \pi(K) + d'(K-1)\pi(K-1) + \cdots + d'(1)\pi(1) \quad (9)$$

The probability that a loss in the modified RED queue leads to a timeout can again be calculated from Equation (5) though we note that the probabilities in this case would be lower since the modified RED algorithm reduces the overall drop rates. We also note that since this algorithm does not drop any two consecutive packets, the probability of timeouts would be lesser even at high loads when compared to RED queues.

In Figure 2 we plot the drop probabilities for the three buffer management policies with bursty traffic. While all the queues had a buffer size of $K = 40$, the other parameters of the RED and modified RED queue were $min_{th} = 20$, $max_{th} = 40$ and $max_p = 1.0$. In the figure we use a burst size of 1 and the results for other burst sizes follow a similar pattern. We note that the drop rates of both the RED and the modified RED algorithm are higher than of the taildrop queue. As was noted in [13] for the

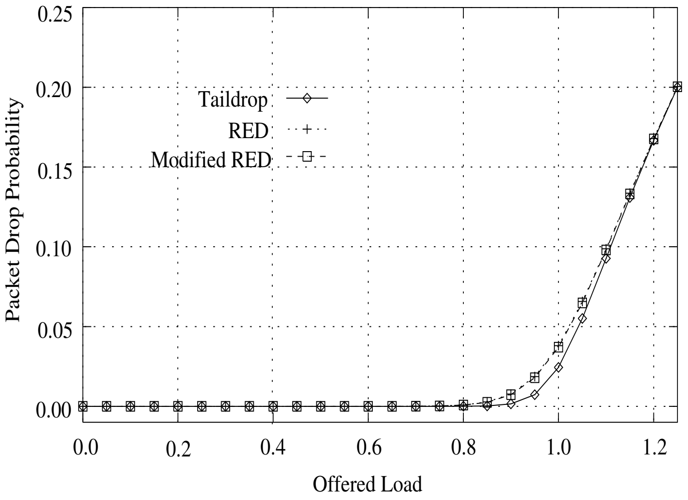

Fig. 2. Comparison of packet drop probabilities for the three buffer management policies.

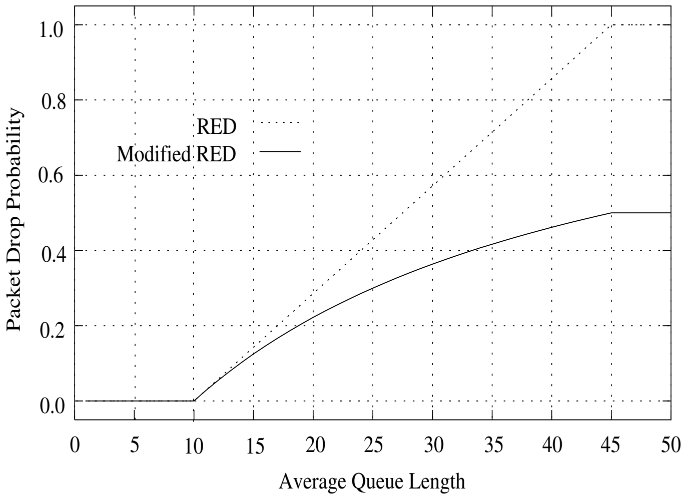

Fig. 3. Comparison of packet drop probabilities in the RED and modified RED buffer management policies.

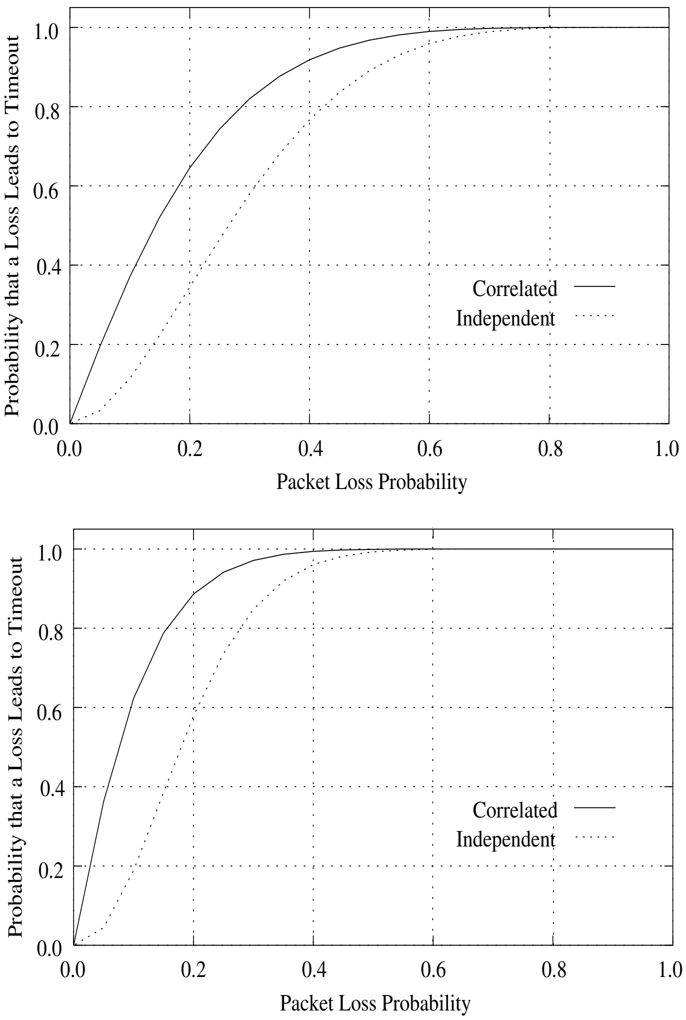

Fig. 4. Comparison of timeout probabilities for TCP flows with windows of 5 (top) and 10 (bottom) with correlated and independent loss models.

RED versus taildrop case, this is a sample path property of the queues. Also, the modified RED queue has lower drop rates compared to RED though it is difficult to see in Figure 2 because of comparatively larger scale for the x-axis. The difference is shown in greater detail in Figure 3 where we used $K = 50$, $min_{th} = 10$, $max_{th} = 45$ and $max_p = 1.0$. Note that when the average queue size becomes greater than max_{th}, the packet loss rate for the modified RED queue is only 0.5 as compared to 1 for RED.

While we note that the sample-path property of the taildrop queues leads to slightly lower drop rates, this does not help much towards reducing timeouts in that case of TCP traffic. In Figure 4 we plot the probability that a loss in a TCP flow with a congestion window of 5 and 10 leads to a timeout in the case of correlated and independent losses. We see that with correlated losses, the probability of timeouts are much higher for the same loss rates. The probability of timeouts saturates to 1 for both cases as the packet drop probability increases. This is because with higher packet drop probabilities, the likelihood of multiple packet drops from the same window becomes very high thereby increasing the probability of timeouts. Also, since the packet drop probabilities for the modified RED queue are smaller that those for a RED queue with the same parameters for a given value of the average queue length, the corresponding probability of timeouts in the modified RED queue would also be smaller.

D. Results

We now compare the self-similarity of the traffic with the three buffer management policies. These results were generated by simulations using the simulator *ns*. While the taildrop and RED queues are already available in the *ns* package, we implemented the modified RED algorithm

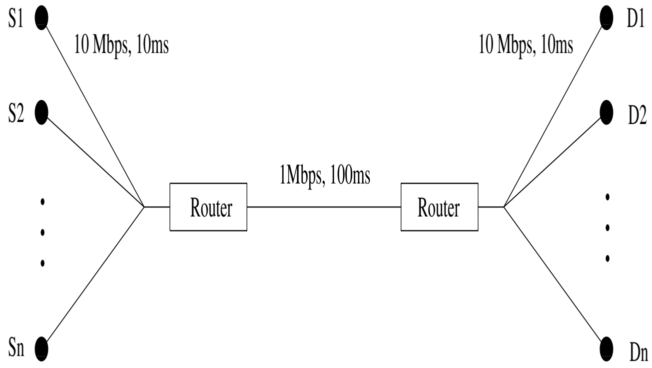

Fig. 5. Topology of the network for the validation tests.

in *ns*. The topology used for the simulations is shown in Figure 5 and is the well known dumbbell topology. The queue management policy at the buffer is changed accordingly for the simulations corresponding to each of the three policies. We present the results for a number of simulation scenarios (which corresponds to a given number of sources and destinations and thus various degrees of network load and multiplexing) for each of the three buffer management algorithm at the bottleneck. The simulations for each queueing policy is again broken in two parts: the presence or absence of web traffic. The web traffic is introduced as background traffic for more realistic wide area networking scenarios. Also, since the variations in the web traffic sessions and the heavy-tailed nature of the file sizes involved in web transfers is one of the causes of self-similarity in network traffic [4], the results from these scenarios allows us to get a feel of the effectiveness of the buffer management techniques in reducing the self-similarity introduced by factors other than TCP.

In the simulation with absence of web traffic, all the sources correspond to very long TCP Reno sources which are active for the entire duration of the simulation. In the simulations with web traffic, a subset of the sources and destinations act as web traffic clients and servers. The column "Configuration" in the tables of this paper enumerate how many how many of the total sources in the topology correspond to the long flows and how many of them are web sessions. The web traffic was generated in the following manner which is derived from [8]. Each TCP connection corresponding to a web traffic session requests a given number of pages and the inter-page request time is assumed to be exponentially distributed with rate 1 while the size of each page is assumed to be constant at 1KB. Each page request triggers the transfer of a number of inlined objects where the inter-object time is exponentially distributed with rate 1 and the object sizes have a heavy-tailed Pareto distribution with average 10KB and shape parameter of 1.2.

For the simulations, the buffer size of the taildrop as well as the RED and modified RED queues was kept at 100 packets. For the RED and modified RED queues, the other parameters were $min_{th} = 30$, $max_{th} = 90$, $max_p = 0.1$ and $w_q = 0.002$. All the simulations were conducted for a "simulated" time of 3600.0 seconds. For the self-similarity plots, we collected the packet arrival statistics at the bottleneck link and represents the aggregate traffic arriving at the bottleneck. For the throughput results, we present the statistics corresponding to each of the long flows in the simulation. For estimating the Hurst parameters, we used three of the widely used methods [21]: the absolute value method, R/S statistics method and the periodogram method. The values of H obtained from each of the three methods are very close and lie within ± 0.03 of each other. We however present the values of H calculated from these plots in a table and do not show the graphs themselves as it is easier to compare the results in the tabular format and because of space limitations.

In Tables I and II we compare the values of H for the three queueing disciplines considered, for simulations with and without web traffic respectively. For the case without web traffic, we note that as expected, the taildrop queue always performs worse that the other two disciplines in terms of the degree of self-similarity. Also, the modified RED queue performs better than the other two queueing disciplines. We note that for the case with 60 flows the RED and the modified queue perform almost the same. The reason behind this is that with such a large number of flows, the average queue length is approximately max_{th}. This leads to *consistent* multiple drops from the same window and thus the flows go into timeouts approximately equally often. The long term benefits of alternate packet drops when the average queue length occasionally crosses max_{th} is lost if the average queue length is almost always at max_{th} and we always have multiple packet drops from a window of packets. However, this particular case corresponds to the situation where the RED and the modified RED queue parameters are not properly tuned. In cases where the average queue length stays between max_{th} and min_{th} (30, 40 and 50 flows) we see that modified RED performs better than simple RED. It is only at extremely high loads that their performance becomes equal. Also note that the modified RED algorithm results in traffic free of self-similarity (as indicated by $H = 0.5$) for low and moderate loads and only at high loads the traffic becomes self-similar for this buffer management policy.

For the results in the simulations with web traffic (Table II) we see a slightly different trend in the results. We see that for lower loads the Hurst parameters corresponding to the taildrop queue are higher than both the RED and

Queues without web traffic

Configuration	Taildrop		RED		Modified RED	
	Throughput	H	Throughput	H	Throughput	H
30 Long	34738.67	0.61	35105.33	0.50	34952.59	0.50
40 Long	26514.67	0.75	26780.50	0.50	26750.33	0.50
50 Long	21545.11	0.80	22086.89	0.55	21819.80	0.50
60 Long	18253.89	0.82	18877.33	0.69	18824.07	0.70

TABLE I
THROUGHPUT (IN BITS/SEC) AND HURST PARAMETERS FOR THE THREE BUFFER MANAGEMENT POLICIES.

Queues with web traffic

Configuration	Taildrop		RED		Modified RED	
	Throughput	H	Throughput	H	Throughput	H
10 Long, 5 web	63988.44	0.58	72982.00	0.50	63055.77	0.50
15 Long, 5 web	42099.70	0.67	44785.19	0.60	47422.37	0.57
10 Long, 10 web	33005.78	0.74	37165.11	0.76	34517.33	0.75
15 Long, 10 web	22318.52	0.77	23449.48	0.84	25204.88	0.77
20 Long, 10 web	17508.89	0.80	19952.00	0.91	21275.88	0.80

TABLE II
THROUGHPUT (IN BITS/SEC) AND HURST PARAMETERS FOR THE THREE BUFFER MANAGEMENT POLICIES.

the modified RED algorithms. As the load on the network increases, the Hurst parameter of the traffic in the RED queue becomes more than the other two scheduling disciplines. However, we note that for all cases, the modified RED algorithm performs better than the others or equals the best performance. The difference in the trend in the traffic self-similarities in this case is due to the introduction of heavy tails through the web traffic. In addition to the contribution of TCP, we now have other factors which influence the traffic through the links. Though the degree of self-similarity increases as the load increases for all the buffer management policies, the rate of increase is not the same for all cases.

The point we would to make here is that the modified RED algorithm performs better that both taildrop and the traditional RED algorithm under a wide range of conditions. While RED leads to lower degrees of self-similarity as compared to taildrop queues when only long flows are present, it does not have the same advantage when web-traffic is present. The modified RED algorithm gives us the best performance or equals the best performance for both web traffic and also when only long TCP flows are present.

In Tables I and II we also compare the throughputs of the long TCP flows in the simulation scenarios. The numbers in the tables represent the average throughput calculated over all the long flows present in the scenario. We see that the performance of all the three policies is almost the same though the RED and the modified RED algorithms perform slightly better than the taildrop queues. The improvement in the throughput with RED queues over taildrop queues is around 1-3% for the cases without web traffic. In the presence of web traffic this improvement increases to between 5-11%. In the absence of web traffic, the throughput of the RED and modified RED are almost identical. However in the presence of web traffic, the throughput of the modified RED generally increases and the increase is around 5%.

E. Caveat

Finally we would like to mention here that that the reduction in the timeouts need not be the only factor which results in the modified RED queue's lower degree of self-similarity. Other external factors like session characteristics and file size distributions also affect it the temporal behavior of queue lengths and thus affect the TCP flows passing through it. It is beyond the scope of this paper to investigate all these interactions in detail and we do not fully understand their impact on the behavior of the buffer management policies. However, the aim of this paper is to explore some ways of reducing the degree of self-similarity

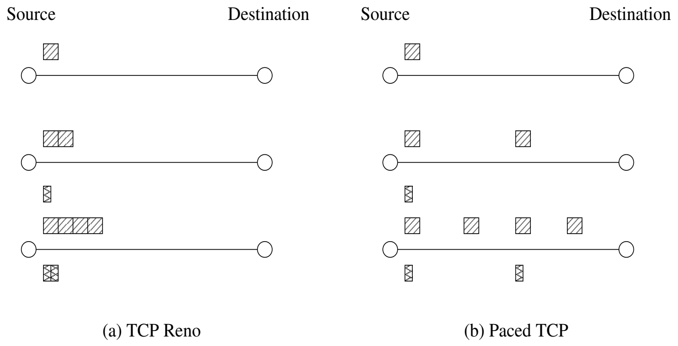

Fig. 6. The packet sending pattern of Reno and paced TCP during slow-start

in network traffic and from that point we have verified that the modified RED queue can indeed perform better than the other two disciplines even in the presence of some of these additional factors.

IV. Reducing Source-Level Burstiness of TCP Flows

TCP traffic is inherently bursty in nature and TCP sources tend to send back to back packets. One of the key reasons behind this behavior is ACK compression as described in [26]. The self-clocking mechanism of TCP depends on the arrival of ACKs at the same spacing with which they were generated by the receiver. However, in the presence of two-way traffic, queueing on the reverse path can alter this spacing and the ACKs arrive closer together than they were sent. The first immediate consequence of this is that the sender becomes bursty and sends more back to back packets. Additionally, with the ACKs being received quicker than they were sent, the sender might be misled into sending more data than the network can accept [14]. This in turn contributes to the losses and timeouts experienced by the TCP flow. It was also conjectured in [8] that the phenomenon of ACK compression might be responsible for the fractal behavior of network traffic. Thus, the prospect of reducing traffic self-similarity by undoing the effects ACK compression on TCP dynamics is very promising and worth of further exploration.

One of the most widely reported mechanisms for smoothing out TCP traffic is through evenly spacing or ”pacing” a window of packets over the round-trip time and was first proposed in [26]. Since then, pacing has been proposed for cases the ACK clocking is lost to avoid slow starts at the beginning of connections, after losses or at the resumption of idle connection. For more details on these and other environments where pacing is used, we refer the reader to [1] and the references therein. TCP pacing aims to evenly spread the transmission of a window of packets over a RTT and can be implemented at either the sender or the receiver side. Pacing is accomplished at the sender (receiver) if instead of transmitting a packet (ACK) everytime an ACK (packet) is received, it is delayed to maintain the proper spacing between two successive packets (ACKs). The delay between two successive packets is given by

$$delay = \frac{RTT}{cwnd} \tag{10}$$

where $cwnd$ is the current value of the congestion window. The concept of TCP pacing and its difference from conventional TCP versions is illustrated in Figure 6. The figure shows that $cwnd$ increase and the packet transmission patterns of TCP Reno and TCP pacing. While, Reno and other versions of TCP send their allowable window of packets as fast as the transmission rate allows resulting in bursty traffic, paced TCP spreads out the transmission of the packets over the RTT.

In [1] a detailed comparison of Reno and Paced TCP is presented in terms of performance issues. In this paper we are concerned only with the effect of pacing on the self-similarity of network traffic though we do report on their performance in terms of the throughput.

In Tables III and IV we compare the self-similarity of the traffic when paced TCP is used instead of TCP Reno. In Table III we consider the case when the bottleneck has

Taildrop queue at bottleneck

Config.	Reno		Paced	
	Throughput	H	Throughput	H
50 Long	21635.24	0.50	22854.49	0.50
60 Long	18280.00	0.70	19305.07	0.50
70 Long	15891.49	0.82	17066.51	0.50
80 Long	14080.67	0.88	15518.37	0.60

TABLE III
Comparison of the throughput (in bits/sec) and Hurst parameter for Reno and Paced TCP flows.

RED queue at bottleneck

Config.	Reno		Paced	
	Throughput	H	Throughput	H
40 Long	26810.06	0.50	27095.44	0.50
75 Long	15665.93	0.55	15609.51	0.50
80 Long	14848.36	0.60	15826.00	0.50
85 Long	14060.00	0.74	15232.61	0.62

TABLE IV
Comparison of the throughput (in bits/sec) and Hurst parameter for Reno and Paced TCP flows.

Queues with web traffic

Configuration	Taildrop		RED	
	Reno	Paced	Reno	Paced
10 Long, 5 web	0.58	0.50	0.50	0.50
15 Long, 5 web	0.67	0.50	0.60	0.50
10 Long, 10 web	0.74	0.50	0.76	0.50
15 Long, 10 web	0.77	0.50	0.84	0.50
20 Long, 10 web	0.80	0.54	0.91	0.58

TABLE V
HURST PARAMETERS FOR RENO AND PACED TCP WITH WEB TRAFFIC FOR TAILDROP AND RED QUEUES.

a taildrop queue while Table IV corresponds to a RED queue. The buffer length for both cases was set at $K = 100$, while the RED queue parameters were $min_{th} = 30$, $max_{th} = 90$, $max_P = 0.10$ and $w_q = 0.002$. We present the results for only the long flows for paced TCP. We note that in both the cases, pacing the TCP packets instead of sending them in bursts leads to significant reductions in the degree of self-similarity. Also, for moderate and medium loads the traffic from paced TCP is not self-similar and only at very high loads does it start showing Hurst parameters greater than 0.50. Even at these very high loads, paced TCP performs much better than TCP Reno. We also note that though we present the results only for TCP Reno, comparisons with other versions of TCP yield similar results since other versions also tend to send back to back packets whenever possible resulting in similarly bursty traffic.

Tables III and IV also show the throughput of paced TCP as compared to TCP Reno. We note that the performance of both are approximately the same though paced TCP has a slight advantage over TCP Reno. However, as noted in [1], we would like to mention that this need not always be the case and there can be situations when the throughput of TCP Reno can be a bit higher than paced TCP. Considering the fact that pacing can significantly reduce the degree of self-similarity in network traffic and that it performs favorably with TCP reno in terms of throughput, it is a very promising candidate when comparing methods for reducing traffic self-similarity.

In Table V we compare the Hurst parameters for the cases when web traffic is also present for both taildrop and RED queues at the bottleneck. We note that pacing is very successful at reducing the degree of self-similarity even with the presence of session and user level causes. Also, we note that pacing leads to larger reductions in H as compared to the modified RED algorithm. This leads us to believe (without rigorous proof) that the inherent burstiness in TCP flows is a greater contributor to traffic self-similarity than timeouts and exponential backoffs. This makes pacing more successful at reducing traffic self-similarity over a large number of scenarios as compared to the modified RED algorithm.

V. SUMMARY

In this paper we explored some methods for reducing the degree of self-similarity in network traffic. The methods were based on two of the causes which contribute to the self-similarity of network and in particular TCP traffic: timeouts and the burstiness of TCP traffic. To address the first issue, we looked at the effect of various buffer management policies on the self-similarity of network traffic. We looked at taildrop and RED queues and also at a proposed variation of the RED queue. Our results show that while the RED queue leads to lower degrees of self-similarity when only long TCP flows are present, in the presence of web-traffic, taildrop queues can lead to lower degrees of self-similarity at high loads. However, the modified RED queue consistently gives the lowest degree of self-similarity for all these scenarios. Also, the modified RED algorithm is able to totally eliminate the self-similarity of traffic at low and moderate loads for the topology and traffic sources considered and only at high loads does the traffic become self-similar.

Another factor contributing to the self-similarity of network traffic is the inherent burstiness of TCP traffic and TCP flows sending back to back packets. We address this issue by considering TCP pacing which spreads out the packet transmissions from a window over the whole RTT thereby mitigating the burstiness and eliminating back to back transmissions. We showed that paced TCP results in significant reductions in the Hurst parameter for both RED and taildrop queues at the bottleneck. TCP pacing is also able to significantly eliminate the self-similarity of network traffic even in the presence of web traffic and is in fact more successful at it than the modified RED algorithm.

While we considered only the causes of self-similarity from the TCP point of view, our solutions are also effective against other causes of self-similarity like session interarrival times and heavy tailed distributions in the file sizes. We carried out simulations where background web traffic generated according to empirical distributions was also present and showed that the modified RED algorithm can reduce the degree of self-similarity in those cases also. However, we note here that though the RED and modified RED algorithms can reduce the degree of self-similarity in the traffic and the reduction of timeouts plays a part in it, this need not be the only reason behind that. Additionally, the presence of other non-TCP contributors to self-similarity can also influence the way in which the buffer management policies affect the TCP flows.

This is a promising area for future exploration and there are numerous possibilities that can be considered. The first is to explore possibilities of addressing the non-TCP related causes of self-similarity. The other important question is how do these causes affect TCP's contribution to traffic self-similarity and characterizing their effect on the three buffer management policies we considered in this paper. Another important direction would be to extend this work towards reducing the multi-fractal scaling of network traffic.

References

[1] A. Aggarwal, S. Savage, and T. Anderson, "Understanding the performance of TCP pacing," *Proceedings of IEEE INFOCOM,* pp. 1157-1165, Tel-Aviv, Israel, March 2000.

[2] K. Chandrayana, "A queuing theoretical approach to configuring Random Early Drop gateways," Master's Thesis, Department of Electrical, Computer and Systems Engineering, Rensselaer Polytechnic Institute, Troy, NY, 2001.

[3] M. Christiansen, K. Jeffay, D. Ott and F. D. Smith, "Tuning RED for web traffic," *Proc. of ACM SIGCOMM,* pp. 139-150, Stockholm, Sweden, September 2000.

[4] M. Crovella and A. Bestavros, "Self-similarity in World Wide Web traffic: Evidence and possible causes," *IEEE/ACM Trans. on Networking,* vol. 5, no. 6, pp. 835-846, Dec 1997.

[5] A. Erramilli, O. Narayan, A. Neidhardt and I. Sainee, "Performance impacts of multi-scaling in wide area TCP/IP traffic," *Proceedings of IEEE INFOCOM,* pp. 352-259, Tel Aviv, Israel, March 2000.

[6] A. Erramilli, O. Narayan and W. Willinger, "Experimental queueing analysis with long-range dependent packet traffic," *IEEE/ACM Transactions on Networking,* vol. 4, no. 2, pp. 209-223, April 1996.

[7] A. Feldmann, A. C. Gilbert and W. Willinger, "Data networks as cascades: Investigating the multifractal nature of Internet WAN traffic," *Computer Communications Review,* vol. 28, no. 4, pp. 42-58, 1998.

[8] A. Feldmann, A. C. Gilbert, P. Huang and W. Willinger, "Dynamics of IP traffic: A study of the role of variability and the impact of control," *Proceeding of ACM SIGCOMM,* Boston, MA, August 1999.

[9] D. R. Figueiredo, B. Liu, V. Misra and D. Towsley, "On the autocorrelation structure of TCP traffic," Technical Report TR 00-55, University of Massachusetts, Computer Science Department, Amherst, MA, 2000.

[10] S. Floyd and V. Jacobson, "Random early detection gateways for TCP congestion avoidance," *IEEE/ACM Transactions on Networking* vol. 1, no. 4, pp. 397-413, August 1993.

[11] L. Guo, M. Crovella and I. Matta, "How does TCP generate pseudo-self-similarity?," *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems,* Cincinnati, OH, August 2001.

[12] A. Kumar, "Comparative Performance Analysis of Versions of TCP in a Local Network with a Lossy Link," *IEEE/ACM Trans. on Networking,* vol. 6, no. 4, pp. 485-498, August 1998.

[13] M. May, T. Bonald and J.-C. Bolot, "Analytic evaluation of RED performance," *Proceedings of IEEE INFOCOM,* pp. 1415-1424, Tel-Aviv, Israel, March 2000.

[14] J. Mogul, "Observing TCP dynamics in real networks," *Proceedings of ACM SIGCOMM,* pp. 305-317, Baltimore, MD, August 1992.

[15] J. Padhye, V. Firoiu, D. Towsley and J. Kurose, "Modeling TCP Reno performance: A simple model and its empirical validation," *IEEE/ACM Trans. on Networking,* vol. 8, no. 2, pp. 133-145, April 2000.

[16] K. Park, G. Kim, and M. Crovella, "On the relationship between file sizes, transport protocols, and self-similar network traffic," *Proceedings of International Conference on Network Protocols,* pp. 171-180, Columbus, OH, October 1996.

[17] V. Paxson and S. Floyd, "Wide area traffic: The failure of Poisson modeling," *IEEE/ACM Trans. on Networking,* vol. 3, no. 3, pp. 226-244, June 1995.

[18] B. Sikdar, S. Kalyanaraman and K. S. Vastola, "An integrated model for the latency and steady state throughput of TCP connections," *Performance Evaluation,* vol. 46, no. 2-3, pp. 139-154, September 2001.

[19] B. Sikdar, S. Kalyanaraman and K. S. Vastola, "TCP Reno with random losses: Latency, throughput and sensitivity analysis," *Proceedings of IEEE IPCCC,* pp. 188-195, Phoenix, AZ, April 2001.

[20] B. Sikdar and K. S. Vastola, "The effect of TCP on the self-similarity of network traffic," *Proceedings of the 35th Conference on Information Sciences and Systems,* Baltimore, MD, March 2001.

[21] M. S. Taqqu and V. Teverovsky, "On estimating long-range dependence in finite and infinite variance series," *A Practical Guide to Heavy Tails: Statistical Techniques and Applications,* R. J. Adler, R. E. Feldman and M. S. Taqqu editors, pp. 177-217, Birkhauser, Boston, 1998.

[22] S. Vamvakos and V. Anantharam, "On the Departure Process of a Leaky Bucket System with Long-Range Dependent Input Traffic," *Queueing Systems: Theory and Applications,* vol. 28, no. 1-3, pp. 191-214, May 1998.

[23] A. Veres and M. Boda, "The chaotic nature of TCP congestion control," *Proceedings of IEEE INFOCOM,* pp. 1715-1723, Tel-Aviv, Israel, 2000.

[24] A. Veres, Z. Kenesi, S. Molnár and G. Vattay, "On the propagation of long-range dependence in the Internet," *Proceedings of ACM SIGCOMM,* pp. 243-254, Stockholm, Sweden, September 2000.

[25] W. Willinger, M. S. Taqqu, R. Sherman and D. V. Wilson, "Self-similarity through high-variability: Statistical analysis of Ethernet LAN traffic at the source level," *IEEE/ACM Trans. on Networking,* vol. 5, no. 1, pp. 71-86, 1997.

[26] L. Zhang, S. Shenker, and D. Clark, "Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic," *Proceedings of ACM SIGCOMM,* pp. 133-147, Zurich, Switzerland, September 1991.