Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000).

On the Burstiness of the TCP Congestion-Control Mechanism in a Distributed Computing System *

Peerapol Tinnakornsrisuphap‡, Wu-chun Feng†§, and Ian Philp†
tinnakor@cae.wisc.edu, feng@lanl.gov, philp@lanl.gov

‡ Department of Electrical & Computer Engineering
University of Wisconsin-Madison
Madison, WI 53706

† Research & Development in Advanced Network Technology (RADIANT)
Computing, Information, and Communications Division
Los Alamos National Laboratory
Los Alamos, NM 87545

§ School of Electrical & Computer Engineering
Purdue University
W. Lafayette, IN 47907

Abstract

Several studies in network traffic characterization have concluded that network traffic is self-similar and therefore not readily amenable to statistical multiplexing in a distributed computing system. This paper examines the effects of the TCP protocol stack on network traffic via an experimental study on the different implementations of TCP. We show that even when aggregate application traffic smooths out as more applications' traffic are multiplexed, TCP introduces burstiness into the aggregate traffic load, reducing network performance when statistical multiplexing is used within the network gateways.

Keywords: *TCP, distributed computing, network traffic characterization, self-similar traffic.*

1. Introduction

High-speed, distributed systems require support for the fluctuating and heterogeneous demands of individual users. The ability to characterize the behavior of the resulting aggregate network traffic can provide insight into how traffic should be scheduled to make efficient use of the network, and yet still deliver expected quality-of-service to end users. These issues are of fundamental importance to the design of the Next-Generation Internet (NGI) [17], and consequently, the resurgence of high-speed, distributed computing environments such as the *Earth System Grid* [3].

Several research efforts in the area of network traffic characterization have concluded that network traffic is self-similar in nature [11, 16]. That is, when traffic is aggregated over varying time scales, the aggregate traffic pattern remains bursty, regardless of the granularity of the time scale. Additional studies have concluded that the heavy-tailed distributions of file size, packet interarrival, and transfer duration fundamentally contribute to the self-similar nature of aggregate network traffic [19].

The problems with the aforementioned research are three-fold. First, the knowledge that self-similar traffic is bursty at coarse-grained time scales contributes little insight into the network's ability to achieve an expected quality of service through the Internet's use of traditional statistical-multiplexing techniques because the effectiveness of such techniques manifests itself at the granularity of milliseconds (particularly in the "small buffer" case), not tens or hundreds of seconds [7]. Second, while current models of network traffic apply to existing file-size distributions and traffic arrival patterns, these models will not generalize as new applications and services are introduced to the NGI [18]. Third, the proofs of the relationship between heavy-tailed

*This work was supported by the U.S. Dept. of Energy through Los Alamos National Laboratory contract W-7405-ENG-36. This paper is LA-UR 00-481.

distributions and self-similar traffic in [9, 19] ignore the involvement of the TCP congestion-control mechanism. Thus, while the heavy-tailed distributions of file size, packet interarrival, and transfer duration may contribute to self-similarity, there are many other factors which have not been investigated thoroughly. Moreover, all the studies thus far have failed to isolate individual aspects of the end-to-end networking path in order to pinpoint the source of self-similarity; instead, various aspects have been intermingled and studied simultaneously.

To address some of the above issues, we present an experimental study on the effect of different versions of TCP on the shape as well as the predictability of traffic generated at the application level. We show that the fluctuations in the TCP congestion window size and dependencies between congestion-control decisions made by multiple TCP streams modulate application traffic to be more bursty and reduce network performance when statistical multiplexing is used within the network gateways. The rest of the paper is organized as follows. Section 2 presents additional background information on the different versions of TCP. Section 3 outlines the simulation model used in our experimental study and presents and analyzes the results of the study. Lastly, Section 4 presents our concluding remarks and future work.

2. Background

TCP is a connection-oriented service which guarantees the reliable, in-order delivery of a stream of bytes, hence freeing the application from having to worry about missing or reordered data. It includes a flow-control mechanism which ensures that a sender does not overrun the buffer capacity of the receiver and a congestion-control mechanism which tries to prevent too much data from being injected into the network, thereby causing packet loss within the network. While the size of the flow-control window is static, the size of the congestion window evolves over time, according to the status of the network.

2.1 TCP Congestion Control

Currently, the most widely-used TCP implementation is TCP Reno [8]. Its congestion-control mechanism consists of two phases: (1) slow start and (2) congestion avoidance. In the slow-start phase, the congestion window grows exponentially (i.e., doubles every time the sender successfully transmits a congestion-window's worth of packets across the network) until a timeout occurs, which implies that a packet has been lost. At this point, a $CongestionThreshold$ value is set to the halved window size; TCP Reno resets the congestion window size to one and re-enters the slow-start phase, increasing the congestion window exponentially up to the $CongestionThreshold$. When the threshold is reached, TCP Reno then enters its congestion-avoidance phase in which the congestion window is increased by "one packet" every time the sender successfully transmits a congestion-window's worth of packets across the network. When a packet is lost during the congestion-avoidance phase, TCP Reno takes the same actions as when a packet is lost during slow start.

To further enhance performance, TCP Reno also implements fast-retransmit and fast-recovery mechanisms for both the slow-start and congestion-avoidance phases. Rather than timing out while waiting for the acknowledgement (ACK) of a lost packet, if the sender receives three duplicate ACKs (indicating that some packet was lost but later packets were received), the sender immediately retransmits the lost packet (fast retransmit). Because later packets were received, the network congestion is assumed to be less severe than if all packets were lost, and the sender only halves its congestion window and re-enters the congestion-avoidance phase (fast recovery) without going through the slow-start phase again.

TCP Vegas [2] introduces a new congestion-control mechanism that tries to prevent congestion rather than react to the congestion after it has occurred. The basic idea is as follows: When the congestion window increases in size, the expected sending rate (ER) increases as well. However, if the actual sending rate (AR) stays roughly the same, this implies that there is *not* enough bandwidth available to send at ER, and therefore, any increase in the size of the congestion window will result in packets filling up the buffer space at the bottleneck gateway. TCP Vegas attempts to detect this phenomenon and avoid congestion at the bottleneck gateway by adjusting the congestion-window size, and hence ER, as necessary to adapt to the available bandwidth.

To adjust the window size appropriately, TCP Vegas defines two threshold values, α and β, for the congestion-avoidance phase, and a third threshold value, γ, for the transition between the slow-start and congestion-avoidance phases. Conceptually, $\alpha = 1$ implies that TCP Vegas tries to keep at least one packet from each stream queued in gateway while $\beta = 3$ keeps at most three packets from each stream queued in the gateway.

If $RateDiff = ER - AR$, then when $RateDiff < \alpha$, Vegas increases the congestion window linearly during the next RTT; when $RateDiff > \beta$, Vegas decreases the congestion window linearly during the next RTT; otherwise, the congestion window remains unchanged. The γ parameter can be viewed as the "initial" β when TCP Vegas enters its congestion-avoidance phase.

To enhance the performance of TCP, Floyd and Jacobson proposed the use of random early detection (RED) gateways [6] to detect incipient congestion. To accomplish this detection, RED gateways maintain an exponentiallyweighted, moving average of the queue length. As long as the average queue length stays below the minimum threshold (min_{th}), all packets are queued, and thus no packets are dropped. When the average queue length exceeds min_{th}, packets are dropped with probability P. And when the average queue length exceeds a maximum threshold (max_{th}), all arriving packets are dropped.

2.2 TCP Probability & Statistics

The Central Limit Theorem states that the summation of a large number of finite-mean, finite-variance, independent variables approaches a Gaussian random variable with less variability (or less "spread" or burstiness) than the original distribution(s). So, if each random variable were to represent traffic generated by a particular communication stream, then the sum of a large number of these streams represents aggregate network traffic with less variability, and thus less variation or spread in the required bandwidth, i.e, network traffic is less bursty or more smooth. Such aggregate traffic behavior enables statistical-multiplexing techniques to be very effective over the Internet. Unfortunately, although application-generated traffic streams may have finite means and variances and may be independent, TCP can modulate these streams in such a way that they are no longer independent, for example. Hence, the thrust of this paper is to examine how TCP modulates application-generated traffic and how it affects the statistical-multiplexing techniques currently being used in the Internet as well as distributed computing systems.

To measure the burstiness of aggregate TCP traffic, we use the *coefficient of variation (c.o.v.)* — the ratio of the standard deviation to the mean of the observed number of packets arriving at a gateway in each round-trip propagation delay. The c.o.v. gives a normalized value for the "spread" of a distribution and allows for the comparison of "spreads" over a varying number of communication streams.

Rather than use the Hurst parameter from self-similar modeling as is done in many studies of network traffic [11, 14, 15, 16, 19], we use c.o.v. because it better reflects the burstiness of the incoming traffic, and consequently, the effectiveness of statistical multiplexing over the Internet [4]. If the c.o.v. is small, the amount of traffic coming into the gateway in each RTT will concentrate mostly around the mean, and therefore will yield better performance via statistical multiplexing.

3. Simulation Study

The goal of this simulation study is to understand the dynamics of how TCP modulates application-generated traffic. While this issue has been largely ignored in the self-similar literature [11, 14, 15, 16, 19], we intend to isolate and understand the TCP modulation so that we may be better able to schedule network resources. Understanding how TCP modulates traffic can have a profound impact on the coefficient of variation (c.o.v.), and hence, throughput and packet loss percentage of network traffic. This, in turn, directly affects the performance of distributed computing systems such as the *Earth System Grid* [3].

3.1 Network Model

To characterize the TCP modulation of traffic, we first generate application traffic according to a known distribution. We then compare the c.o.v. of this distribution to the c.o.v. of the traffic transmitted by TCP. We can then determine whether TCP modulates the traffic, and if it does, how it affects the shape (burstiness) of the traffic, and hence, the performance of the network.

Consider a client-server network with one server and M clients. Each client is linked to a common gateway with a full-duplex link with bandwidth μ_c and delay τ_c. A bottleneck full-duplex link of bandwidth μ_s and delay τ_s connects the gateway to the server. Each client generates Poisson traffic, i.e., single packets are submitted to the TCP stack with exponentially distributed interpacket arrival times with mean $1/\lambda$. All the clients attempt to send the generated packets to the server through the common gateway and bottleneck link. The configuration is shown in Figure 1.

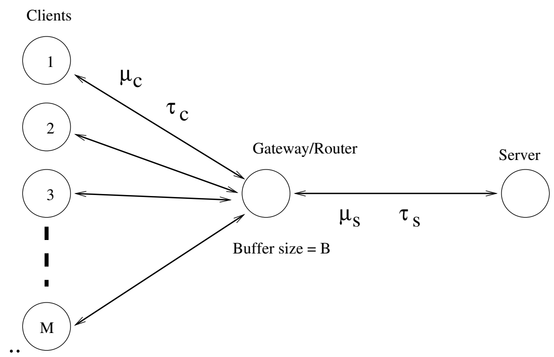

Figure 1. Network Model

In our *ns* [13] simulations, we vary the total traffic load offered by varying the number of clients M. We use UDP, TCP Reno (with delay acknowledgments both on and off), and TCP Vegas as the transport-layer protocols. We also test the effects of two queueing disciplines in the gateway, FIFO (First-In, First-Out) and RED, to see whether the queueing discipline has any effect on the burstiness generated by the TCP protocol stack. We calculate the c.o.v. of the aggregate traffic generated by the clients, based on the known distribution each client uses to generate its traffic, and compare it to the measured c.o.v. of the aggregate TCP modulated traffic as it arrives at the gateway. The parameters used in the simulation are shown in Table 1.

Parameters	Value
client link bandwidth (μ_c)	10 Mbps
client link delay (τ_c)	25 ms
bottleneck link bandwidth (μ_s)	50 Mbps
bottleneck link delay (τ_s)	25 ms
TCP max advertised window	20 packets
gateway buffer size (B)	50 packets
packet size	1500 bytes
average packet intergeneration time ($1/\lambda$)	0.01 s
total test time	200 s
TCP Vegas/α	1
TCP Vegas/β	3
TCP Vegas/γ	1
RED min_{th}	10 packets
RED max_{th}	40 packets

Table 1. Simulation Parameters.

3.2 TCP Reno vs. Vegas

Here we examine how TCP modulates application-generated traffic when all the clients are running the same implementation of TCP.

Since the traffic generated by the application layers is Poisson, the c.o.v. of the number of packets received during one RTT for the unmodulated aggregate traffic is $1/\sqrt{(\lambda\tau)n}$ where n is the number of clients aggregated and $\tau = RTT = 2(\tau_c + \tau_s)$. Thus, the traffic generated from the application layer becomes smoother as the number of sources increases.

Figure 2 shows that UDP does not adversely modulate traffic because the c.o.v. of aggregated UDP traffic is very close to that of the aggregated Poisson process. This result is not surprising since UDP transmits packets received from the application layer to the network without any flow/congestion control. For TCP, we divide the results into three cases.

1. Uncongested: the amount of traffic generated is much lower than the available bandwidth, i.e., the number of clients is less than 10.

2. Moderately congested: the amount of traffic generated causes some, but not severe, congestion, i.e., the number of clients is between 10 and 38.

3. Heavily congested: the amount of traffic generated is higher than what the network can handle, i.e., the number of clients is greater than 38.

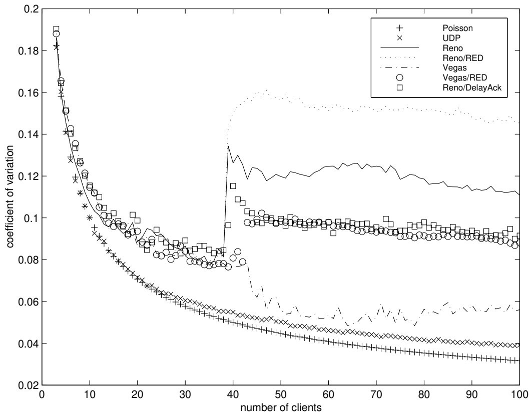

Figure 2. Coefficient of Variation of the Aggregated TCP Traffic.

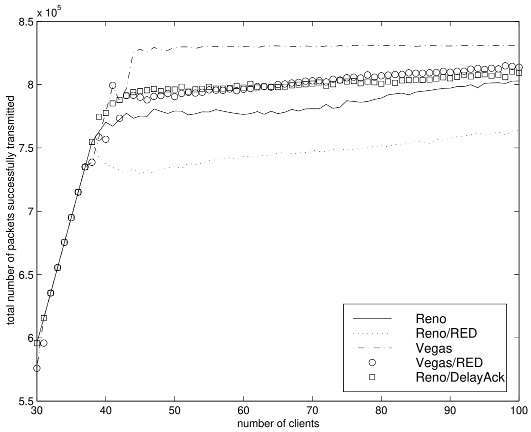

Figure 3. Throughput of the Aggregated TCP Traffic.

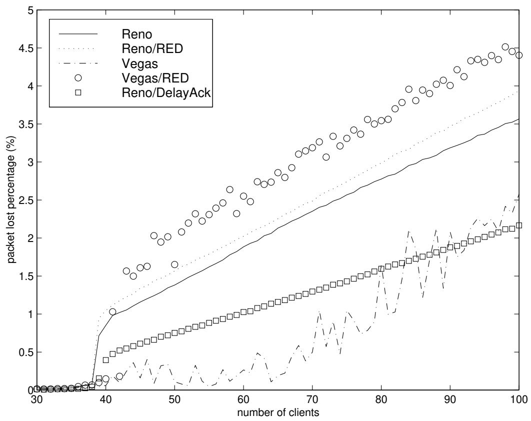

Figure 4. Packet Loss Percentage of the Aggregated TCP Traffic.

In the uncongested case, the traffic entering the gateway is very similar to the traffic that the clients generate. This result is due to the absence of congestion in the network, i.e., the congestion-control mechanism has not activated to control or modulate the application-generated traffic.

When the clients generate a moderate amount of traffic, and hence introduce intermittent congestion, the TCP congestion-control mechanism begins to modulate the application-generated traffic. We can see this effect in Figure 2 as the number of client connections varies from 10 to 38 — the TCP c.o.v. numbers are up to 50% higher than the aggregated Poisson, and hence indicate that the congestion-control mechanisms of TCP noticeably modulate traffic when the network is moderately congested; that is, TCP induces burstiness into the aggregate traffic stream. Because the network only experiences intermittent congestion, this induced burstiness is not strong enough to adversely impact throughput and packet loss, as shown in Figures 3 and 4. (Note: The number of clients starts at 30 for these figures because the different TCP implementations exhibit nearly identical behavior for less than 30 clients.)

Under heavy congestion, the c.o.v. increases sharply for all TCP implementations except TCP Vegas. The TCP Reno and TCP Reno/RED c.o.v. numbers are over 140% and 200% larger than the aggregated Poisson numbers, respectively. This result indicates that TCP Reno and TCP Reno/RED significantly modulate application-generated traffic (Poisson traffic) to be much more bursty. And unfortunately, this modulation is adverse enough to impact the throughput and packet loss percentage of TCP Reno and TCP Reno/RED, as shown in Figures 3 and 4. This leads us to believe that these TCP Reno implementations introduce a high level of dependency between the congestion-control mechanisms of each of the TCP streams.

3.2.1 Analysis of TCP Reno

Figure 5 exhibits a snapshot of TCP Reno's congestion window for three of the 20 client streams (clients 1, 10, and 20) in the uncongested case. Interestingly, nearly all the packet losses occur during slow start. While this phenomenon may initially seem surprising, it can be explained as follows: The application generates traffic *independently* of the congestion window. So, when the congestion window is small, the application layer generates "too much" traffic for the congestion window. Consequently, the TCP send buffers accumulate a lot of data waiting to be transmitted. Since the congestion window grows exponentially during TCP slow start, the likelihood is quite high that a full congestion window's worth of data is transmitted when an acknowledgement is received. For example, having only three of the 20 streams generate bursts of 17 packets each, as implied by Figure 5, can cause packet loss, and thus congestion, since the buffer

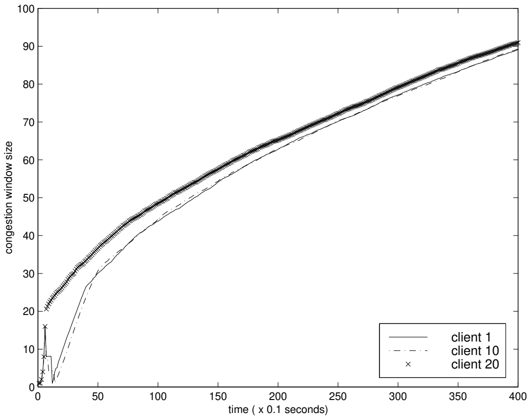

Figure 5. Evolution of TCP Reno's Congestion Window (# clients = 20).

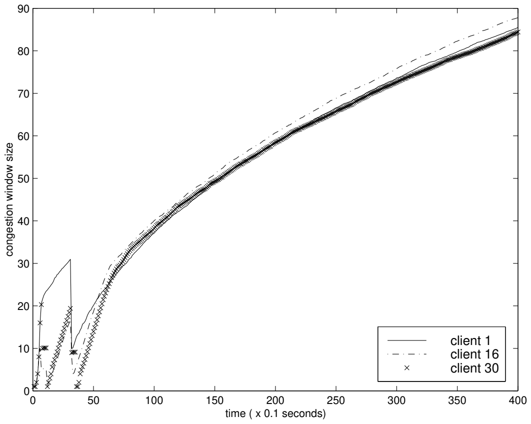

Figure 6. Evolution of TCP Reno's Congestion Window (# clients = 30).

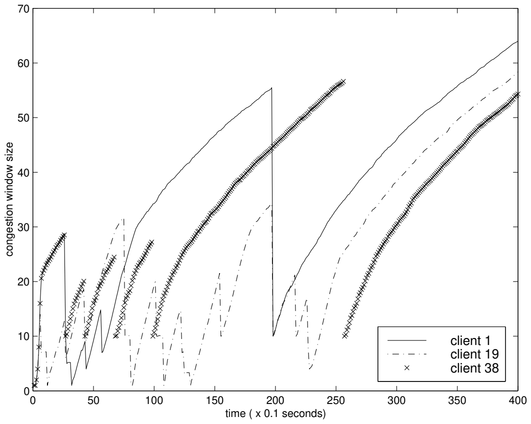

Figure 7. Evolution of TCP Reno's Congestion Window (# clients = 38).

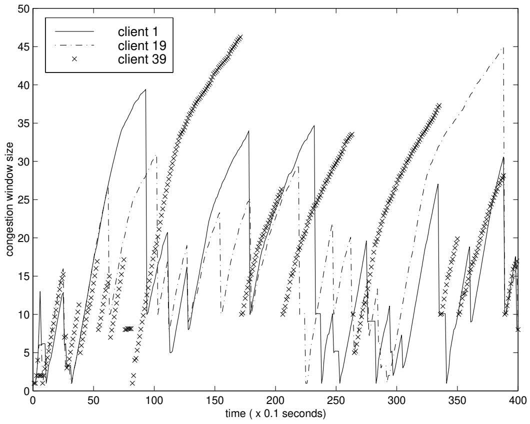

Figure 8. Evolution of TCP Reno's Congestion Window (# clients = 39).

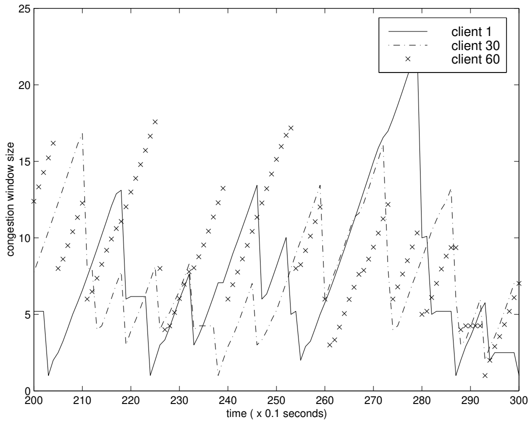

Figure 9. Evolution of TCP Reno's Congestion Window (# clients = 60).

size at the gateway is 50 packets.

As the load increases to 30 clients (see Figure 6), congestion occurs even earlier during slow start due to the same reasoning used in the 20-client test. In addition, congestion also occurs at 40 time units (each time unit = 0.1 second), where the congestion window size decreases *simultaneously* for all three of the client streams shown before stabilizing into a steady linear increase.

The ability to "stabilize" into a steady-state linear increase reaches a crossover point between 38 and 39 clients. As more clients are added (up to and including 38 clients), the longer it takes for the congestion window to reach its steady-state linear increase. In the case of 38 clients, as shown in Figure 7, the congestion window sizes stabilize after 250 time units. However, Figure 8 shows that with 39 clients, the congestion window sizes never stabilize as there are just enough packets being generated to consistently (rather than intermittently) cause congestion in the steady state.

Figures 6 through 8 also indicate that as the traffic load increases so does the likelihood that decreases in the congestion window size are synchronized, either by timeout or fast retransmit. Unfortunately, this synchronization induces wild fluctuations in the aggregate congestion window size, and consequently, queue lengths and packet loss. It is this behavior which causes the c.o.v. to increase sharply at 39 clients, as illustrated in Figure 2. As the number of streams increases to 60, the synchronization is even more pronounced because the network is so congested that most of the TCP streams are making the same congestion-control decisions simultaneously. Thus, the congestion-control mechanism in TCP Reno introduces a high level of dependency between TCP streams, as shown in the zoomed time snapshot of Figure 9.

3.2.2 Analysis of TCP Vegas

TCP Vegas takes a more conservative approach to congestion control than TCP Reno. Rather than allowing every TCP stream to continually try to get as much bandwidth as possible, i.e., a "greedy" linear increase until a packet loss occurs, as is done in TCP Reno, TCP Vegas shares available bandwidth among all TCP connections by performing a linear increase when there is "too little" traffic in the network, i.e., less than α packets are queued per stream, and a linear decrease when there is "too much" traffic in the network, i.e., greater than β packets are queued per stream. Using this approach, each client's TCP Vegas congestion window stays close to its "optimal" value. Therefore, the traffic transmitted from each client is modulated nearly equally each RTT (when the congestion window is near its "optimal" value). The subsequent decrease in the packet-loss percentage also decreases the dependency between the TCP Vegas streams, and therefore contributes to the significantly lower c.o.v.

In addition to the significantly lower c.o.v., Figures 10 through 12 demonstrate that TCP Vegas shares available bandwidth more fairly than TCP Reno. This substantiates the work done by [1, 12]. Moreover, TCP Vegas requires much less buffer space in the gateway to avoid congestion as TCP Vegas tries to keep number of buffered packets from each individual stream between α and β.

3.2.3 Analysis of RED Gateways

While RED gateways were introduced as a way to enhance TCP performance in Reno as well as Vegas, our results show that such gateways increase TCP modulation and actually hurt TCP performance, which consequently affects

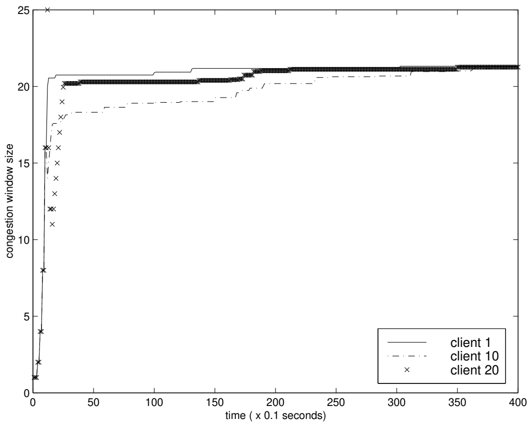

Figure 10. Evolution of TCP Vegas's Congestion Window (# clients = 20).

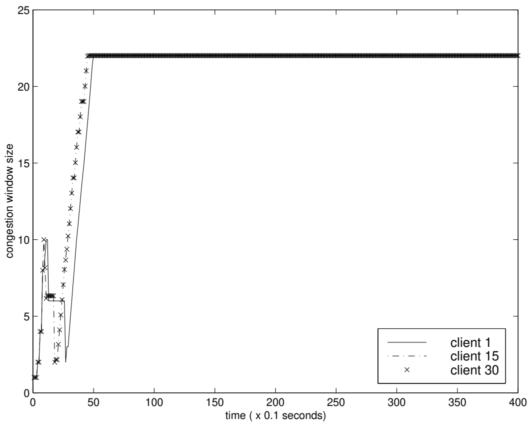

Figure 11. Evolution of TCP Vegas's Congestion Window (# clients = 30).

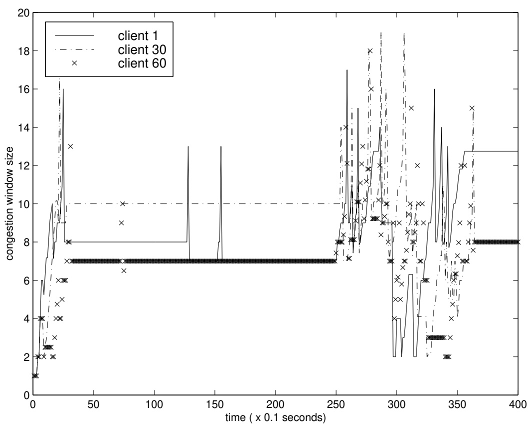

Figure 12. Evolution of TCP Vegas's Congestion Window (# clients = 60).

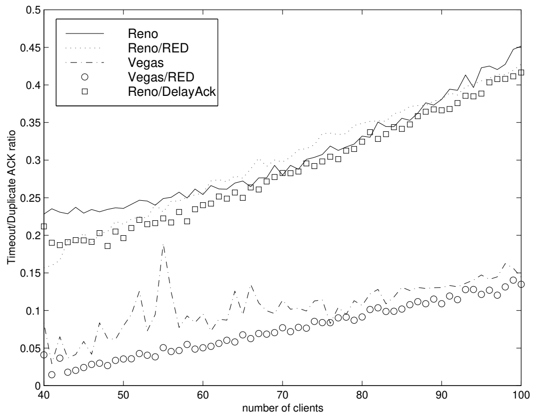

Figure 13. Ratio of Timeouts to Duplicate ACKs.

the network performance in distributed computing systems. This conclusion is also drawn by [5] although for different reasons.

Using the min_{th} and max_{th} parameters, a RED gateway makes the buffer in the gateway appear smaller to the TCP connections. TCP Reno, whose performance varies significantly with respect to the gateway buffer size [10], suffers severely because its buffer requirements can very quickly become large as each stream is attempting to greedily increase its congestion window size. On the other hand, TCP Vegas requires a minimal amount of buffer space per connection and produces smoother traffic than TCP Reno in the presence of a RED gateway, resulting in a better-performing TCP and thus confirming the research findings in [12].

Unfortunately, TCP implementations with RED gateways perform significantly worse than their "plain" counterparts with respect to c.o.v. and throughput (see Figures 2 and 3) because TCP with RED gateways adversely modulate application-generated traffic in such a way as to reduce the effectiveness of statistical multiplexing in the network. Figures 2 and 3 illustrate that Vegas outperforms its Vegas/RED counterpart with respect to c.o.v. and throughput and that Reno outperforms its Reno/RED counterpart.

Interestingly, however, Figure 4 shows that Vegas/RED not only produces higher packet-loss percentage than Vegas, but it also is higher than either Reno implementation. This behavior can be explained by an analysis of the TCP Vegas and Reno algorithms. In our Vegas experiments, the α and β parameters are set to the commonly used values of 1 and 3, respectively, and thus each Vegas stream attempts to keep at least one and at most three packets queued in the gateway. As a result, when the gateway becomes more heavily congested around 40 streams, the aggregate number of packets that Vegas tries to keep queued in the gateway is at least 40 (i.e., one packet per stream) and no more than 120 (i.e., three packets per stream). As a result, with 40 streams, the RED gateway *always* drops packets because the max_{th} of the RED gateway is 40. Of the 40 packets that are actually queued and delivered to the server, duplicate ACKs are generated, causing Vegas to push more data into the network (even though the RED gateway is already "full"), hence causing additional packet loss. Figure 13 supports this explanation by showing that the ratio of timeouts to duplicate ACKs is very low for Vegas.

Although the gateway in TCP Vegas spends more time above max_{th} on average during heavier congestion, hence causing higher packet loss; TCP Reno generates more significant bursts of packets at particular time instances, causing large *sequences* of packet losses, and consequently, a larger number of timeouts — 250-300% higher than TCP Vegas. This greater number of timeouts in TCP Reno more frequently reduces the congestion window size to one. This frequency of drastic window-size adjustment contributes to the higher c.o.v. and reduced throughput in TCP Reno.

4. Conclusion

From our experiments, we have shown that the congestion-control mechanisms of TCP Reno and TCP Vegas modulate the traffic generated by the application layer. The congestion-control mechanism of TCP Reno adversely modulates the traffic to be more bursty, which subsequently affects the performance of statistical multiplexing in the gateway; this modulation occurs for two primary reasons: (1) the rapid fluctuation of the congestion window sizes caused by the continual "additive increase / multiplicative decrease (or re-start slow start)" probing of the network state and (2) the dependency between the congestion-control decisions made by multiple TCP streams which increases as the number of streams increase, i.e., TCP streams tend to recognize congestion in the network at the same time and thus halve their congestion windows at the same time. As a result, TCP Reno traffic does not significantly smooth out even when a large number of streams are aggregated. On the other hand, TCP Vegas, during congestion avoidance, does not modulate the traffic to be as bursty as TCP Reno. This translates to smoother aggregate network traffic, and hence better overall network performance.

References

[1] T. Bonald. Comparison of TCP Reno and TCP Vegas via Fluid Approximation. Technical Report 3563, INRIA, November 1998.

[2] L. Brakmo and L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal of Selected Areas in Communications*, 13(8):1465–1480, October 1995.

[3] W. Feng, I. Foster, S. Hammond, B. Hibbard, C. Kesselman, A. Shoshani, B. Tierney, and D. Williams. Prototyping an Earth System Grid. *http://www.scd.ucar.edu/css/esg*, July 1999.

[4] W. Feng, M. Gardner, I. Philp, and P. Tinnakornsrisuphap. A New Statistical Model for Characterizing Aggregate Network Traffic. *In preparation*, 2000.

[5] W. Feng, D. Kandlur, D. Saha, and K. Shin. A Self-Configuring RED Gateway. In *INFOCOM '99*, March 1999.

[6] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[7] M. Grossglauser and J. Bolot. On the Relevance of Long-Range Dependence in Network Traffic. *Computer Communication Review*, 26(4):15–24, October 1996.

[8] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of the SIGCOMM'88 Symposium*, pages 314–332, August 1998.

[9] T. G. Kurtz. Limit Theorems for Workload Input Models. *Stochastic Networks: Theory and Applications*, pages 339–366, 1996.

[10] T. V. Lakshman and U. Madhow. The Performance of TCP/IP for Networks with High Bandwidth-Delay Products and Random Loss. *IEEE/ACM Transactions on Networking*, 5(3):336–350, June 1997.

[11] W. Leland, M. Taqqu, W. Willinger, and D. Wilson. On the Self-Similar Nature of Ethernet Traffic (Extended Version). *IEEE/ACM Transaction on Networking*, 2(1):1–15, February 1994.

[12] J. Mo, R. J. La, V. Anantharam, and J. Walrand. Analysis and Comparison of TCP Reno and Vegas. In *Proceedings of INFOCOM'99*, March 1999.

[13] ns. UCB/LBNL/VINT Network Simulator. *http://www-mash.cs.berkeley.edu/ns*.

[14] K. Park, G. Kim, and M. Crovella. On the Relationship Between File Sizes, Transport Protocols, and Self-Similar Network Traffic. In *Proceedings of the 4th International Conference on Network Protocols*, October 1996.

[15] K. Park, G. Kim, and M. Crovella. On the Effect of Traffic Self-Similarty on Network Performance. In *Proceedings of the SPIE International Conference on Performance and Control of Network Systems*, 1997.

[16] V. Paxson and S. Floyd. Wide-Area Traffic: The Failure of Poisson Modeling. *IEEE/ACM Transaction on Networking*, 3(3):226–244, June 1995.

[17] S. Shenker. Fundamental Design Issues for the Future Internet. *IEEE Journal of Selected Areas in Communications*, 13(7):1176–1187, 1995.

[18] W. Willinger and V. Paxson. Where Mathematics Meets the Internet. *Notices of the American Mathematical Society*, 45(8):961–970, September 1998.

[19] W. Willinger, V. Paxson, and M. Taqqu. Self-Similarity and Heavy Tails: Structural Modeling of Network Traffic. *A Practical Guide to Heavy Tails: Statistical Techniques and Applications*, pages 27–53, 1998.