TCP Vegas Revisited

U. Hengartner[1], J. Bolliger[1] and Th. Gross[1,2]

[1]Departement Informatik
ETH Zürich
CH 8092 Zürich

[2]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract— **The innovative techniques of TCP Vegas have been the subject of much debate in recent years. Several studies have reported that TCP Vegas provides better performance than TCP Reno. However, the question which of the new techniques are responsible for the impressive performance gains remains unanswered so far. This paper presents a detailed performance evaluation of TCP Vegas. By decomposing TCP Vegas into the various novel mechanisms proposed and assessing the effect of each of these mechanisms on performance, we show that the reported performance gains are achieved primarily by TCP Vegas's new techniques for slow-start and congestion recovery. TCP Vegas's innovative congestion avoidance mechanism is shown to have only a minor influence on throughput. Furthermore, we find that the congestion avoidance mechanism exhibits fairness problems even if all competing connections operate with the same round trip time.**

Keywords— **TCP Vegas, Congestion control, Transport protocols.**

I. INTRODUCTION

TCP Vegas is a new design for TCP that was introduced by Brakmo et al. [6][8]. TCP Vegas includes a modified retransmission strategy (compared to TCP Reno) that is based on fine-grained measurements of the round-trip time (RTT) as well as new mechanisms for congestion detection during slow-start and congestion avoidance. The innovative techniques proposed in [6][8], as well as the impressive performance gains (compared to TCP Reno) reported [6][1], have been the subject of much debate in recent years. This paper takes a fresh look at the design of TCP Vegas and attempts to shed light on the advantages (and disadvantages) of the innovations introduced by TCP Vegas.

TCP Reno's congestion detection and control mechanisms use the loss of segments as a signal that there is congestion in the network [17]. TCP Reno has therefore no mechanism to detect the incipient stages of congestion before losses occur and hence cannot prevent such losses. Thus, TCP Reno is *reactive*, as it needs to create losses to find the available bandwidth of the connection. On the contrary, TCP Vegas's congestion detection mechanism is *proactive*, that is, it tries to sense incipient congestion by observing changes in the throughput rate. Since TCP Vegas infers the congestion window adjustment policy from such throughput measurements, it may be able to reduce the sending rate before the connection experiences losses.

Since TCP Vegas is essentially a combination of several different techniques, each evoking considerable controversy on its own, much of previous work either concentrated on discussing and evaluating a particular mechanism in isolation or tried to characterize the overall behavior of TCP Vegas. The question, however, which of the techniques incorporated in TCP Vegas are responsible for the performance gains reported in [6][8][1], remains unanswered so far. To answer this question, we decompose TCP Vegas into its individual algorithms and assess the effect of each of these algorithms on performance.

The paper is organized as follows: Section II presents the various enhancements of TCP Vegas. Related work is reviewed in Section III. Section IV describes the simulation environment used for our experiments. Section V provides the basis for our detailed evaluation by quantifying the speedup achieved by TCP Vegas (over TCP Reno). The techniques incorporated by TCP Vegas are listed in Section VI, and some problems in their implementation are discussed in Section VII. Section VIII presents the detailed results on how the various algorithms affect overall performance. Finally, Section IX discusses the fairness of TCP Vegas's congestion avoidance mechanism.

II. TCP VEGAS

According to the published papers that describe TCP Vegas [6][8], TCP Vegas differs from TCP Reno as follows:

New retransmission mechanism: TCP Vegas introduces three changes that affect TCP's (fast) retransmission strategy. First, TCP Vegas measures the RTT for every segment sent. The measurements are based on fine-grained clock values. Using the fine-grained RTT measurements, a timeout period for each segment is computed. When a duplicate acknowledgement (ACK) is received, TCP Vegas checks whether the timeout period has expired. If so, the segment is retransmitted[1]. Second, when a non-duplicate ACK that is the first or second after a fast retransmission is received, TCP Vegas again checks for the expiration of the timer and may retransmit another segment. Third, in case of multiple segment loss and more than one fast retransmission, the congestion window is reduced only for the first fast retransmission.

Congestion avoidance mechanism: TCP Vegas does not continually increase the congestion window during congestion avoidance. Instead, it tries to detect incipient congestion by comparing the measured throughput to its notion of expected

U. Hengartner is currently with Carnegie Mellon University (email: uhengart+@cs.cmu.edu).

Effort sponsored in part by the Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0287. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

[1] Since TCP Vegas may trigger a fast retransmission on the first duplicate ACK (TCP Reno waits for three duplicate ACKs), we will—merely for the sake of brevity—refer to this particular algorithm in TCP Vegas as the "more aggressive retransmission strategy".

throughput. The congestion window is increased only if these two values are close, that is, if there is enough network capacity so that the expected throughput can actually be achieved. The congestion window is reduced if the measured throughput is considerably lower than the expected throughput; this condition is taken as a sign for incipient congestion.

Modified slow-start mechanism: A similar congestion detection mechanism is applied during slow-start to decide when to change to the congestion avoidance phase. To have valid comparisons of the expected and the actual throughput, the congestion window is allowed to grow only every other RTT.

In [8], an additional algorithm is presented, which tries to infer available bandwidth during slow-start from ACK spacing. However, this algorithm was marked experimental, and it was not used in the evaluation of TCP Vegas. (Hence, we also excluded it from our evaluation.)

Both [6] and [8] report between 37 and 71% better throughput for TCP Vegas on the Internet, with one-fifth to one-half of the losses. Simulations confirm these measurements; they also show that Vegas does not adversely affect TCP Reno's throughput and that TCP Vegas is not less fair than TCP Reno.

III. Related work

TCP Vegas's new techniques for congestion avoidance, their effect on TCP performance, and TCP Vegas's behavior in the presence of competing TCP Reno connections have been investigated by previous researchers. We now give a short overview of this earlier work.

Ahn et al. [1] performed some live Internet experiments with TCP Vegas. They report 4–20% speedups for transfers to a TCP Reno receiver and 300% speedups for transfers to a TCP Tahoe receiver. For both scenarios, TCP Vegas is found to retransmit fewer segments and to have lower RTT average and variance. Experiments in a WAN emulator with varying degrees of either TCP Reno or TCP Vegas background traffic reveal that TCP Vegas achieves higher throughputs for high congestion, whereas TCP Reno outperforms TCP Vegas in the case of low congestion.

With a fluid model and simulations, Mo et al. [21] show that TCP Vegas, as opposed to TCP Reno, is not biased against connections with long delays, and that TCP Vegas does not receive a fair share of bandwidth in the presence of a TCP Reno connection.

Hasegawa et al. [14] use an analytical model to derive that TCP Vegas's congestion avoidance mechanism is more stable than the one of TCP Reno, that is, the congestion window of a TCP Vegas connection may converge to a fixed value. However, they also find that the mechanism sometimes fails to achieve fairness among several connections with different round-trip times.

With the help of a WAN emulator simulating a satellite link, Zhang et al. [22] study the performance of various TCP versions over long-delay links. TCP Vegas achieves only half the throughput of TCP Tahoe or TCP Reno. However, it retransmits much less than other TCP variants.

Ahn et al. [2] introduce a new technique to speedup simulation of high-speed, wide-area packet networks. The evaluation section presents the results of running a "stripped-down" version of TCP Vegas, which includes only its congestion detection and window adjustment scheme, over a gigabit network. In the experiments, TCP Vegas achieves only half of the throughput of TCP Reno.

Such a restricted version of TCP Vegas is also evaluated by Bolliger et al. [5]; several variants of TCP are implemented as user-level protocols and evaluated in the Internet. TCP Vegas is shown to cause fewer timeouts due to multiple segment loss than TCP Reno. On the other hand, TCP Vegas suffers more "non-trigger" timeouts than TCP Reno. Non-trigger timeouts reflect missed opportunities to enter recovery. In this study, TCP Vegas's throughput is slightly worse than TCP Reno's.

Danzig et al. [10] evaluate a pre-release version of TCP Vegas which did not include Vegas's new congestion avoidance mechanism. Since the authors cannot reproduce the claims made in [6], they conclude that it is indeed TCP Vegas's new congestion avoidance mechanism that is responsible for the performance improvements noted in [6].

The last three reports are contradictory; [2][5] may lead to the conclusion that TCP Vegas's new behavior during congestion avoidance has a negative influence on throughput, whereas [10] suggests that it has a positive influence. Unfortunately, research showing throughput improvements for TCP Vegas has failed to show which of TCP Vegas's new algorithms is responsible to what degree for the reported speedups. This paper tries to address this issue based on simulations. Before turning to a detailed evaluation of the effects of individual mechanisms present in TCP Vegas, the following sections introduce the simulation environment and present an initial performance evaluation of TCP Reno and TCP Vegas which validates the experimental setup.

IV. Simulation environment

This section describes the simulation environment used to investigate the influence of the various new algorithms in TCP Vegas.

A. Simulator

We run our simulation on *x*-sim, a network simulator based on the *x*-kernel [16]. In this environment, actual *x*-kernel protocol implementations run on a simulated network. Our choice of *x*-sim is based on the following two observations: First, the evaluations in the original papers that describe TCP Vegas [6][8] are also performed with *x*-sim. This fact gives us confidence that our results are not biased by using a different implementation of TCP Vegas. Second, we want to evaluate an implementation of TCP Vegas based on production code. This requirement is fulfilled by *x*-sim since its implementation of TCP Reno is directly derived from the BSD implementation of TCP Reno.

We made two changes to the original implementations of TCP Reno and TCP Vegas provided in the *x*-kernel. Both of these changes were proposed in a paper from the inventors of TCP Vegas [7] and have also been applied to the current TCP Reno releases of FreeBSD and NetBSD. The changes include a modification to the algorithm for computing the retransmission timeout value (see discussion below) and a fix of the check to reduce the congestion window upon leaving fast recovery.

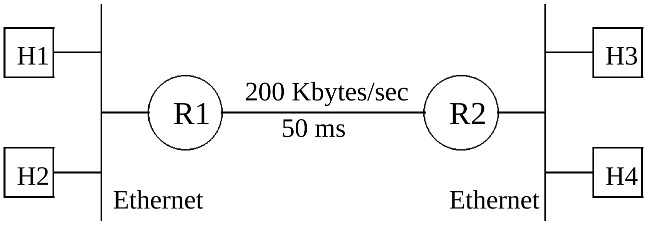

Fig. 1. Network topology for simulations.

B. Topology

For our experiments, we emulate the topology presented in Figure 1. To ensure that the results of our experiments are comparable to those of previous work, we chose exactly the same topology as in the original Vegas paper [6]. For the same reason, the segment size used is 1.4KB, the router queue size is ten segments, and the router queuing discipline is FIFO. However, to prevent any non-congestion-related influences on the congestion window, we chose larger sender and receiver buffer sizes (i.e., 128KB instead of 50KB) for the TCP hosts. The TCP receivers do not employ delayed acknowledgments; since TCP Vegas's congestion detection mechanism reacts to changes in the RTT, delayed acknowledgments could affect the performance severely, as shown in [1].

We validated both the simulator and the network topology by repeating some of the experiments from [6][8]. Our version of TCP Reno performs slightly worse than the original version; this difference is due to the more conservative computation of the timeout value (RTO), that is, the RTO is computed as the smoothed RTT plus four times the RTT variation (as proposed in [17]) instead of only plus two times the RTT variation (as proposed in an earlier version of [17])[2].

V. Performance evaluation

To gain some insight on the performance of TCP Vegas, we simulate a transfer of 1MB of data from host H1 to host H3 for varying degrees and types of background traffic. The background traffic, which flows from host H2 to host H4, is generated by TRAFFIC, an *x*-kernel protocol which simulates Internet traffic and which is based on *tcplib* [9]. Each type of experiment is run fifty times. Tables I and II present the results from these experiments. In the case of low background traffic, TRAFFIC's connection inter-arrival time is 0.1s, for high background traffic, it is 0.03s. With regard to throughput, TCP Vegas outperforms TCP Reno in each of the four scenarios, with improvements ranging from 40% up to 120%. Moreover, TCP Vegas retransmits between 6% and 65% less data than TCP Reno.

These results confirm other research that reports partially impressive improvements and fewer retransmissions for TCP Vegas. Tables I and II serve as starting point for our more detailed evaluation of the performance implications of the individual algorithms incorporated in TCP Vegas.

TABLE I
Average throughput [KB/s] per connection.

	Background traffic			
	low		high	
	Reno	Vegas	Reno	Vegas
Reno	73.4	72.4	16.1	13.3
Vegas	105.5	101.4	35.1	29.2

TABLE II
Average retransmissions [KB] per connection.

	Background traffic			
	low		high	
	Reno	Vegas	Reno	Vegas
Reno	48.6	49.3	122.7	140.5
Vegas	16.8	18.8	113.0	131.9

VI. Algorithms in TCP Vegas

For the evaluation of the algorithms, we take the approach of a 2^k factorial design with replications [19]. This methodology allows us to determine the effect of k factors, each of them having two levels. In the case of TCP Vegas, these factors are the different algorithms, as presented in Section II. The factor levels are "on" and "off"; they indicate whether the TCP Vegas algorithm is used ("on") or whether the algorithm is turned "off", so that the default TCP Reno behavior is used.

A 2^k factorial design requires that each of the factors (algorithms) can be independently turned on or off. Therefore, we first had to modify the TCP Vegas source code to separate the various algorithms from each other and to allow each of the algorithms to be selectable individually. These changes required a close inspection of the source code. This inspection revealed that TCP Vegas contains some more changes above the ones mentioned in [6][8]. The complete list of algorithms that are new in TCP Vegas is presented in the following. Algorithms and changes (A)–(E) are discussed in [6][8] and have already been described in Section II, whereas changes (F)–(J) are not mentioned in [6][8].

A. Congestion detection during slow-start;
B. Congestion detection during congestion avoidance;
C. More aggressive fast retransmit mechanism;
D. Additional retransmissions for non-duplicate ACKs;
E. Prevention of multiple reductions of the congestion window in case of multiple segment loss;
F. Reduction of the congestion window by only 1/4 after a recovery (instead of halving it as in the case of TCP Reno)[3];
G. A congestion window size of two segments at initialization and after a timeout (TCP Reno sets the size of the congestion window to one segment in these situations[4]);

[2] Note that TCP Vegas's fine-grained timeout values are computed with the algorithm proposed in the earlier version of [17]. Also note that TCP Vegas starts the retransmission timer for a segment as soon as it is sent, whereas TCP Reno starts the timer for a segment only when the segment preceding it is acknowledged.

[3] This algorithm has already been identified by Ahn et al. [1] as part of TCP Vegas.

[4] For TCP Reno, an initial congestion window size of two segments has recently been allowed [4].

H. Burst avoidance limits the number of segments that can be sent at once (that is, back-to-back) to three segments;
I. The congestion window is not increased if the sender is not able to keep up, that is, the difference between the size of the congestion window and the amount of outstanding data is larger than two maximum-sized segments;
J. Spike suppression limits the output rate to at most twice the current rate. (This algorithm is turned off by default.)

When separating the algorithms from each other, we kept the necessary code changes to a minimum to avoid any behavioral differences between the original and our implementation of TCP Vegas. We validated our implementation by making sure that our version of TCP Vegas with all algorithms turned off produces the same results as the TCP Reno implementation. Similarly, we checked that our version of TCP Vegas with all algorithms turned on and the original implementation of TCP Vegas achieve identical throughputs.

VII. Deviations from specification

Section VI listed some changes to TCP Reno that were not identified before. In addition, our inspection of the source code of TCP Vegas and its evaluation also revealed some scenarios in which the TCP Vegas implementation does not quite achieve what was intended and/or described by the authors in the original papers [6][8].

A. Timeout behavior

In slow-start and congestion avoidance, TCP Vegas checks once every RTT whether it must modify its strategy for updating the congestion window. In slow-start, it checks whether it must give up the exponential opening of the congestion window and switch to congestion avoidance. In congestion avoidance, it checks whether the congestion window must be increased linearly, must be held constant during the next RTT, or whether it must immediately be reduced by one segment. In case of a timeout during congestion avoidance, the released version of TCP Vegas fails to immediately fall back to exponential opening (as would be appropriate for a slow-start), instead the window is opened only linearly. In the worst case, this conservative opening prevails until all the data sent before the timeout is finally acknowledged, that is, possibly for several RTTs. We altered TCP Vegas to immediately change its strategy for updating the congestion window in case of a timeout.

B. Reset of baseRTT

When executing the check mentioned above, TCP Vegas resets *baseRTT*[5] if only one segment has been transmitted during the last RTT. With the help of this reset, TCP Vegas may be able to cope with routing changes which increase the minimum RTT. Since TCP Vegas employs a minimum size of two segments for the congestion window, this reset is triggered only when the sender is not able to keep up or has no data to send.

In rare cases, this reset can result in setting *baseRTT* to a very small value that is unrelated to the current network conditions. Since there are no routing changes in our simulation and since our sender always has some data to send, we disabled the piece of code resetting *baseRTT* for our evaluations.

TABLE III
Scenario for violated invariants.

Event	Eq.3	Eq.4	*beg_seq*	*snd_nxt*	*snd_una*
			5	10	5
timeout			5	5	5
send 5&6			5	7	5
9 is acked	−2	2	7	10	10
send 10-12			7	13	10
10 is acked	3	6	13	13	11

C. Violation of invariant

In congestion avoidance, TCP Vegas's congestion detection scheme checks every RTT whether network conditions have changed enough to evoke a change in the congestion window adjustment policy. To decide whether and how the size of the congestion window should be adjusted, TCP Vegas compares the expected throughput to the measured actual throughput [6][8]. The expected throughput is computed as

$$expected = \frac{windowSize}{baseRTT}, \tag{1}$$

where *windowSize* is the number of bytes currently in transit. The actual throughput is computed as

$$actual = \frac{rttLen}{rtt}, \tag{2}$$

where *rttLen* reflects the number of bytes transmitted during the last RTT and *rtt* is the average RTT of the segments acknowledged during the last RTT.

In the released TCP Vegas implementation, *windowSize*, the numerator of Eq.(1), is computed as

$$snd\_nxt - snd\_una + min(maxseg - acked, 0), \text{[6]} \tag{3}$$

where *maxseg* is the maximum segment size, and *acked* is the number of bytes acknowledged by the last ACK. *rttLen*, the numerator of Eq.(2), is determined in the following way:

$$snd\_nxt - beg\_seq, \tag{4}$$

where *beg_seq* is the value of *snd_nxt* during the previous computation of *actual* and *expected.* An acknowledgment for *beg_seq* triggers computation of *actual* and *expected.*

According to [6][8], the following invariant must hold:

$$expected \geq actual. \tag{5}$$

Table III shows how this invariant can be violated in case of a timeout due to a single loss. The table gives a time sequence of events that lead to two violations of the invariant. For each event, the values of *snd_nxt*, *snd_una*, and *beg_seq* are displayed after that event has been processed. The ordering of the columns

[5] *baseRTT* is used for computing the expected throughput. According to [6][8], *baseRTT* denotes the RTT of a segment when the connection is not congested. In practice, *baseRTT* reflects the minimum of all measured RTTs.

[6] *snd_nxt* and *snd_una* designate variables from the BSD implementation of TCP and indicate the next segment to be sent resp. to be acknowledged.

(from left to right) is identical to the order in which the variables are updated resp. in which the equations are computed. The two ACKs arriving after the timeout both trigger the recalculation of *actual* and *expected*, and in both cases the invariant is violated, that is, *expected* is smaller than *actual* (assuming $baseRTT \approx rtt$).

The first violation is the consequence of a "large ACK" that acknowledges more than one segment. To remedy this problem, we omitted the last term of Eq.(3) in the TCP Vegas implementation used for our study. The second violation is caused by computing the actual bandwidth over data sent more than one RTT ago. We fixed this problem by resetting *beg_seq* in case of an ACK acknowledging data sent before a timeout. In this way, the computation of the actual bandwidth will not include data sent before the timeout.

D. Discussion

How do these fixes affect the performance of TCP Vegas? First, the fix mentioned in Section VII-A may effect a change to better performance, because it allows the congestion window to open appropriately fast in slow-start, that is, faster than if the sender would erroneously continue to adjust the congestion window size according to the congestion avoidance strategy.

Second, in congestion avoidance, the following condition must hold if the congestion window is to be opened (α is positive and usually set to 1):

$$(expected - actual) * baseRTT < \alpha \qquad (6)$$

If the invariant of TCP Vegas is violated, the difference between *expected* and *actual* is negative, that is, inequation (6) holds and the congestion window size may erroneously be increased. This action may result in a more aggressive window opening than intended. Therefore, by fixing the problem of a violated invariant, we expect TCP Vegas to become less aggressive.

Tables IV and V repeat the results from Tables I and II, and additionally show the results for the TCP Vegas version that incorporates the fixes mentioned (called TCP Vegas'). Note that for the TCP Reno and TCP Vegas' experiments (first and third row of Tables IV and V), we used TCP Vegas' as background traffic. (This is the reason why the results for the TCP Reno experiments differ slightly from those presented in Tables I and II.) For the TCP Vegas experiments (second row of Tables IV and V), the unmodified TCP Vegas was used both as foreground and as background traffic.

The fixes result in slightly lower throughput for all four cases and in slightly more retransmissions in three of the four cases. Overall, TCP Vegas' achieves similar performance when compared to the original version. For the rest of this paper, the term TCP Vegas is used to refer to TCP Vegas'.

VIII. Influence of various algorithms

A. Reduction of complexity

As summarized in Section VI, TCP Vegas employs ten additional algorithms over TCP Reno. A complete $2^k r$ factorial design requires that each possible combination of the $k = 10$ algorithms is chosen and that the experiment described in Section V is run r times for a specific setup [19]. This methodology would allow us to quantify the effect of each individual algorithm and the effects of all possible interactions of the algorithms. For $k = 10$ algorithms, the experiment would result in $2^{10} - 1$ possible effects, most of them being probably rather small[7]. To reduce complexity and increase the "expressiveness" of our experiments, we clustered the algorithms into three groups according to the phase they affect (i.e., slow-start, congestion avoidance, and recovery), and set up a $2^k r$ factorial design with the $k = 3$ phases each representing a factor. The factor levels "on" and "off" mean that either all the algorithms affecting a particular phase are turned on or that all of them are turned off. This design reduces the possible factors and the interaction of factors affecting the performance to $2^3 - 1 = 7$. The algorithms have been clustered as follows:

TABLE IV

Average throughput [KB/s] per connection.

	Background traffic			
	low		high	
	Reno	Vegas(')	Reno	Vegas(')
Reno	73.4	71.8	16.1	13.0
Vegas	105.5	101.4	35.1	29.2
Vegas'	103.7	99.6	34.6	28.4

TABLE V

Average retransmissions [KB] per connection.

	Background traffic			
	low		high	
	Reno	Vegas(')	Reno	Vegas(')
Reno	48.6	49.5	122.7	144.8
Vegas	16.8	18.8	113.0	131.9
Vegas'	16.9	18.5	115.8	139.2

Slow-start: Congestion detection (algorithm (A) presented in Section VI), and congestion window size of two segments (G).
Congestion avoidance: Congestion detection (B).
Congestion recovery: More aggressive fast retransmission strategy (C), retransmission upon ACK for new data (D), reduction of congestion window by $1/4$ (F), and avoidance of multiple reductions of congestion window (E).
(The algorithms "burst avoidance" (H), "no congestion window increases" (I), and "spike suppression" (J) are always turned off.)

Each of the 2^3 experiments is repeated $r = 50$ times. We determined the effect of the algorithms in the three phases on the throughput of TCP Vegas and on the number of retransmissions by applying the methodology described in [19].

B. Results for throughput

The 2^3 factorial design allows us to compute the throughput y for a specific combination of algorithms in the following way [19]:

$$y = q_{mean} + q_{ss} \cdot x_{ss} + q_{ca} \cdot x_{ca} + q_{rec} \cdot x_{rec} +$$

[7] We conducted such an experiment and found that the influence of most combinations was indeed smaller than 1%.

TABLE VI

THROUGHPUT [KB/S] (LOW BACKGROUND TRAFFIC).

	TCP Reno		TCP Vegas	
	Effect	Percentage	Effect	Percentage
	q	of variation	q	of variation
mean	86.64		83.90	
ss	7.14	27.71	5.57	19.38
ca	2.06	2.30	2.25	3.17
rec	6.64	23.98	6.68	27.91
ss_ca	0.17[a]	0.02	0.46[a]	0.13
ss_rec	1.09	0.65	0.95	0.56
ca_rec	0.64	0.22	0.40[a]	0.10
ss_ca_rec	−0.68	0.25	−0.61	0.24
error		44.88		48.50
90%	0.59		0.57	

[a] Not significant.

TABLE VII

THROUGHPUT [KB/S] (HIGH BACKGROUND TRAFFIC).

	TCP Reno		TCP Vegas	
	Effect	Percentage	Effect	Percentage
	q	of variation	q	of variation
mean	26.31		20.97	
ss	0.02[a]	0.00	0.07[a]	0.00
ca	−0.77	0.41	−0.69	0.42
rec	9.65	65.22	7.96	55.78
ss_ca	0.52	0.19	0.90	0.72
ss_rec	−0.96	0.64	−0.79	0.55
ca_rec	−0.53	0.19	−0.34[a]	0.10
ss_ca_rec	0.33[a]	0.07	0.37[a]	0.12
error		33.27		42.30
90%	0.45		0.45	

[a] Not significant.

$$q_{ss\_ca} \cdot x_{ss} \cdot x_{ca} + \ldots +$$
$$q_{ss\_ca\_rec} \cdot x_{ss} \cdot x_{ca} \cdot x_{rec},$$

where x_i is 1 if all the algorithms in phase i are turned on and -1 if they are turned off (*ss*: slow-start, *ca*: congestion avoidance, *rec*: recovery), q_i is the effect of the algorithms in phase i, q_{i_j} indicates the effect of the interactions between the algorithms in phases i and j (similar for $q_{i\_j\_k}$), and q_{mean} is the mean throughput of all experiments.

Table VI presents the results for low TCP Reno background traffic and for low TCP Vegas background traffic. The table reports the mean throughput for all $2^k r = 400$ experiments and the effects q of all factors and their interactions on the (mean) throughput. We can compute the average throughput y, for example, for the configuration where all the algorithms in slow-start and congestion avoidance are turned on and all the algorithms in recovery are turned off, and TCP Reno is used for background traffic as follows:

$$\begin{aligned} y &= 86.64 + 7.14 \cdot 1 + 2.06 \cdot 1 + 6.64 \cdot -1 + \\ &\quad 0.17 \cdot 1 \cdot 1 + 1.09 \cdot 1 \cdot -1 + 0.64 \cdot 1 \cdot -1 + \\ &\quad -0.68 \cdot 1 \cdot 1 \cdot -1 \\ &= 88.32 [KB/s] \end{aligned}$$

The columns “percentage of variation” in Table VI indicates how much of the variation of the throughput y can be explained by effect q and is therefore a measure for the “importance” of a factor. Since the measurements are repeated $r = 50$ times, the percentage of the total variation that can be attributed to experimental errors can be determined. The row “error” reports this variation. Moreover, the value given in the “90%” row allows computation of the 90% confidence intervals for the mean throughput and each effect (e.g., in the case of TCP Reno background traffic, the 90% confidence interval for q_{ss} is 7.14 ± 0.59). Confidence intervals that include 0 indicate that the particular factor (or factor combination) is not statistically significant.

From Table VI, we conclude that for low TCP Reno background traffic, TCP Vegas’s new algorithms in slow-start have the largest effect on throughput, followed by the changes during recovery. TCP Vegas’s congestion detection mechanism during congestion avoidance is responsible for only 2% of the variation. The interactions between the phases have only a small effect on throughput or are not statistically significant. 45% of the variation in throughput is due to experimental errors.

For low TCP Vegas background traffic, 28% of the variation can be explained with the modified algorithms during recovery, followed by the changes during slow-start. 3% of the variation can be explained with the changes during congestion avoidance. The interactions between the different phases are again small or not statistically significant. Nearly half of the variation is due to experimental error.

The data for the high background traffic scenarios is given in Table VII. In the case of TCP Reno background traffic, the dominant effect is the changed behavior during recovery. All other effects have only small influence and/or are statistically not significant. Note that TCP Vegas’s new congestion avoidance mechanism has a (small) negative effect on performance. Experimental errors account for $1/3$ of the total variation.

The results for the high TCP Vegas background traffic scenario look similar to those for the TCP Reno scenario, that is, the changes during recovery explain most of the variation seen in the experiments. Again, the effect of the modified behavior during congestion avoidance is negative.

C. Results for retransmissions

Table VIII presents the influence of the three phases on the amount of retransmitted data for low background traffic. Both for TCP Reno and for TCP Vegas background traffic, the changes in slow-start dominate, followed by the changes in congestion avoidance. Note that the modifications in recovery and the interactions between the modifications in slow-start and in congestion avoidance increase the amount of retransmitted data.

In the case of high background traffic, as shown in Table IX, the experimental error explains about 90% of the variation both for TCP Reno and for TCP Vegas background traffic. Compared to the experimental error, the effects of the individual phases on

TABLE VIII

Retransmissions [KB] (low background traffic).

	TCP Reno		TCP Vegas	
	Effect *q*	Percentage of variation	Effect *q*	Percentage of variation
mean	29.02		32.75	
ss	−10.74	43.12	−12.43	54.91
ca	−8.65	27.97	−6.38	14.48
rec	3.08	3.56	3.05	3.31
ss_ca	5.42	10.96	2.41	2.06
ss_rec	0.32[a]	0.04	0.40[a]	0.06
ca_rec	−1.98	1.47	−1.53	0.83
ss_ca_rec	0.43	0.07	0.25[a]	0.02
error		12.82		24.34
90%	0.38		0.54	

[a] Not significant.

TABLE IX

Retransmissions [KB] (high background traffic).

	TCP Reno		TCP Vegas	
	Effect *q*	Percentage of variation	Effect *q*	Percentage of variation
mean	120.84		144.02	
ss	2.55	1.09	5.58	3.45
ca	−2.21	0.81	−1.84[a]	0.37
rec	−4.79	3.83	−6.31	4.41
ss_ca	1.41[a]	0.33	1.45[a]	0.23
ss_rec	−3.24	1.75	−3.22	1.14
ca_rec	0.25[a]	0.01	−0.24[a]	0.01
ss_ca_rec	0.97[a]	0.16	−0.26[a]	0.01
error		92.02		90.39
90%	1.52		1.85	

[a] Not significant.

the number of retransmissions are negligible. This is not surprising as TCP Vegas (that is, the conglomerate of all algorithms) does not seem to be not particularly successful in reducing the number of retransmissions (compared to TCP Reno) in the case of high background traffic in the first place (Table V).

D. Conclusions

D.1 Slow-start

For the low background traffic scenarios, the changes in slow-start are important, especially if the background traffic is TCP Reno. An inspection of the packet traces reveals that TCP Vegas's congestion-sensitive window update strategy is successful in avoiding timeouts in the initial slow-start. TCP Reno's faster and unresponsive exponential opening of the congestion window in this phase may result in overshooting the available bandwidth and loosing multiple segments. Such damage can then be overcome only with a timeout. Since background traffic is low, transfers are short (on the order of a few seconds). Therefore, a timeout affects throughput severely. By sensing the incipient (self-induced) congestion in slow-start, TCP Vegas can avoid such timeouts and thus perform considerably better than TCP Reno. The evaluation of a more detailed 2^7 experiment, where each of algorithms (A)-(G) represents a factor, shows that congestion detection in slow-start in fact has the largest positive effect on throughput of all the algorithms (it explains about 25% of the variation), whereas the influence of the second change in slow-start (initial window of two segments (G)) is negligible. The problem of overshooting the available bandwidth in the initial slow-start has also been recognized by other researchers, and since the release of TCP Vegas, a number of papers addressing this problem have been published [15][3]. By reducing the likelihood of timeouts in slow-start, congestion detection also succeeds to reduce the number of retransmissions. The 2^7 experiment shows that nearly 50% of the variation can be explained with it. Interestingly, algorithm (G), which is responsible for 3% of the variation, increases the number of retransmissions. Therefore, initializing the size of the congestion window to two segments may be too aggressive.

TABLE X

Average throughput [KB/s] (WAN scenario).

	Background traffic	
	Reno	Vegas
Reno	15.2	14.7
Vegas	18.6	16.4

In the high background traffic scenario, the changes in slow-start have virtually no effect. The discrepancy between the effectiveness of these changes for the low and the high background traffic scenarios is surprising and warrants a closer inspection. For this purpose, we repeated our simulations for a WAN scenario. The topology for the WAN scenario is identical to the one described in Section IV-B; the delay of the bottleneck link is 400ms, its bandwidth is 1.5Mbit/s, and the size of the router queues is 50 segments. We simulate high background traffic (with a connection inter-arrival time of 0.03s). Table X shows the throughput achieved by TCP Reno and TCP Vegas in the WAN scenario. We note that the performance improvements by TCP Vegas are less pronounced (10–20%) than for the original topology. Table XI lists the influence of the three phases and their interactions on TCP Vegas's throughput. It is interesting to note that the changes in slow-start negatively affect throughput. This observation implies that in cases with high background traffic, TCP Vegas's sensing of incipient congestion in slow-start and switching to congestion avoidance is too conservative and that the performance improvements (compared to TCP Reno) must all be attributed to the changes in recovery (see below). When examining the amount of retransmitted data for the WAN scenario, we find that the slow-start changes do not help to decrease the number of retransmissions, either.

D.2 Recovery

The changes in recovery have the largest effect on throughput (except for the case of low TCP Reno background traffic, where the slow-start changes are slightly more effective). One may suspect that TCP Vegas's more aggressive fast retransmission

TABLE XI

THROUGHPUT [KB/S] (WAN SCENARIO).

	TCP Reno		TCP Vegas	
	Effect *q*	Percentage of variation	Effect *q*	Percentage of variation
mean	17.96		16.26	
ss	−2.87	12.31	−1.54	4.61
ca	−0.19[a]	0.05	−0.63	0.78
rec	4.80	34.29	3.25	20.53
ss_ca	0.39	0.22	0.05[a]	0.00
ss_rec	−1.74	4.52	−0.90	1.59
ca_rec	0.30[a]	0.13	0.16[a]	0.05
ss_ca_rec	0.01[a]	0.00	−0.20[a]	0.08
error		48.48		72.36
90%	0.37		0.40	

[a] Not significant.

policy (C) is mainly responsible for the gain in performance. However, the evaluation of the 2^7 experiment reveals that in the high background traffic case, reducing the congestion window by only 1/4 (F) has the largest effect (about 28% for TCP Reno background traffic and about 7% for Vegas background traffic), followed by the retransmissions triggered by ACKs for new data (D; Reno: 9%, Vegas: 2%). The influence of the more aggressive fast retransmission policy (C) is even smaller (Reno: 3%, Vegas: 2%). Algorithm (E), which avoids multiple reductions of the congestion window in recovery, has no effect on throughput.

There are two reasons why algorithm (F) improves TCP Vegas's performance: First, shrinking the congestion window by only 1/4 (instead of halving it) results in a larger congestion window after recovery. Second, algorithm (F) alters the recovery behavior[8]. By halving the congestion window, TCP Reno must wait for about half an RTT until enough duplicate ACKs have arrived to let the congestion window become larger than the amount of data currently outstanding. On the other hand, TCP Vegas must wait only for about 1/4 of an RTT. Even though algorithm (F) results in considerable throughput improvements, it may do so at the cost of other connections. For example, we note that (F) explains 28% of the variation in case of high TCP Reno background, but only 7% in case of high TCP Vegas background. This result seems to indicate that algorithm (F) allows Vegas to grab a larger share of the bottleneck bandwidth. This reasoning is supported by the observation that TCP Reno generally achieves lower throughput when running on TCP Vegas background traffic (Table I). We find that TCP Reno indeed suffers when competing with TCP Vegas background traffic and that TCP Vegas is able to "steal" bandwidth from TCP Reno. Since the changes in congestion recovery (and among those, algorithm (F)) have the largest effect, they are mainly responsible for this asymmetry (or unfairness).

Algorithm (D) helps to avoid timeouts due to multiple segment loss. As shown in [5], the majority of the timeouts in TCP Reno are caused by multiple segment loss, therefore, changes which help to reduce the number of such timeouts prove very helpful. The results for TCP Vegas's fast retransmission policy (C) support Jacobson's argumentation [18] who claimed that the new policy most likely results in only a negligible performance gain. The fact that algorithm (E) has virtually no effect on throughput indicates that the multiple segment loss situations that cannot be remedied by algorithm (D) can hardly be survived without incurring a timeout, most likely because the congestion window is simply too small to allow for further fast retransmissions to be triggered (in scenarios with high background traffic).

In summary, TCP Vegas's techniques for congestion recovery (in particular algorithm (D) that addresses problems related to multiple segment loss) prove to be very effective and mainly responsible for the impressive performance gains over TCP Reno observed. Although effective, algorithm (F) may be problematic in terms of fairness. We note that TCP Reno's deficiencies in dealing with multiple segment loss have also been pointed out by other researchers (e.g., [12][15]), and in recent years a number of solutions have been proposed that enhance TCP Reno's data and congestion recovery mechanisms to remedy these problems (e.g., SACK TCP [20], or "NewReno" TCP [13], etc.).

D.3 Congestion avoidance

Our experiments show that the probably most innovative feature of TCP Vegas, i.e., its congestion detection mechanism during congestion avoidance, actually has the least influence. Its influence is even negative in the high TCP Reno background traffic scenario. Table XII summarizes these findings. It repeats the results from Table IV and in addition shows the throughput achieved by a version of TCP Vegas (Vegas w/o *ca*) that excludes the congestion detection mechanisms in congestion avoidance[9]. These numbers suggest that the congestion avoidance mechanism of TCP Vegas is only rather moderately effective. Furthermore, considering a version of TCP Vegas which *merely* includes the novel congestion avoidance mechanisms (Vegas only *ca*), we see only a minor improvement over TCP Reno for low background traffic loads and even lower throughputs in situations with high background load.

One would expect TCP Vegas to perform better (than TCP Reno) because of its congestion avoidance mechanism for the following reason: TCP Vegas can proactively reduce the congestion window by small amounts (a single segment at a time) to avoid packet loss. A packet loss would result in a congestion window reduction by a large amount. So, the expectation is that (a number of) small rate reductions affects throughput less than the rate-halvings after packet loss. However, we observe in this paper (and other research supports these results [2] [5]) that this hope is in vain. Thus, the empirical evidence indicates that TCP Vegas's congestion avoidance mechanism is too conservative.

As far as the impact on the amount of retransmitted data is concerned, we find that TCP Vegas's congestion detection mechanism is quite successful. However, we also note that both for low and high background traffic, other factors contribute more to the reduction than this particular mechanism.

[8] After the fast retransmission, the congestion window is set to half of its previous size plus three segments. It is increased by one segment for every duplicate acknowledgment. New data can be sent as soon as the size of the congestion window is larger than the amount of outstanding data.

[9] Recall that TCP Vegas refers to the corrected version described in Section VII.

TABLE XII

AVERAGE THROUGHPUT [KB/S] PER CONNECTION.

	Background traffic			
	low		high	
	Reno	Vegas	Reno	Vegas
Reno	73.4	71.8	16.1	13.0
Vegas	105.5	101.4	35.1	29.2
Vegas w/o *ca*	99.3	94.6	35.5	28.0
Vegas only *ca*	74.5	73.4	15.3	11.3

IX. PROBLEMS OF CONGESTION AVOIDANCE

As shown in Section VIII, the influence of TCP Vegas's novel congestion avoidance mechanisms on throughput is at best moderate. This section shows that this mechanism may even exhibit fairness problems.

A. Unfair treatment of "old" connections

In congestion avoidance, TCP Vegas begins to decrease the congestion window when the following condition is met (β is positive and usually set to 3; see Section VII-C for a definition of terms):

$$\left(\frac{windowSize}{baseRTT} - \frac{rttLen}{rtt}\right) * baseRTT > \beta.$$

Assuming that *windowSize* is equal to *rttLen*, that is, the number of segments in transit corresponds to the number of segments sent during the last RTT (this assumption is valid when TCP Vegas reaches equilibrium), the above condition is true for

$$windowSize > \frac{\beta}{1 - \frac{baseRTT}{rtt}}.$$

Consider the scenario when a TCP Vegas connection is initiated in an uncongested network. This connection's $baseRTT_1$ is thus fairly close to the minimal RTT possible. If the network becomes congested later on, the measured RTT (rtt_1) increases and thus the term $baseRTT_1/rtt_1$ decreases. (For our simulation topology, we observed factors smaller than 0.5.) Now, assume that a second connection is started. Since the network is congested, the second connection's estimate of $baseRTT_2$ is bigger than $baseRTT_1$, so $baseRTT_2/rtt_2$ is also bigger than $baseRTT_1/rtt_1$ (assuming that $rtt_1 \approx rtt_2$). This implies that the critical value of *windowSize*, which would trigger a reduction of the congestion window size, is bigger for the second connection than for the first connection. Therefore, the second connection can achieve higher bandwidths than the first ("older") connection.

Figure 2 illustrates this fairness problem of TCP Vegas's congestion avoidance mechanism. The graph shows the congestion window sizes of five staggered connections. The connections share a bottleneck and are started in one second intervals[10]. The first connection reacts the most to the congestion caused by the other connections, and in equilibrium, its congestion window is the smallest. On the other hand, the connection started last achieves the largest congestion window and thus gets the largest share of the bottleneck bandwidth. (The size of the congestion window of the second connection is identical to the one of the first connection in equilibrium.) Note that the size of the congestion window of the first connection at $t = 3s$ corresponds to the size of the congestion window of the last connection during equilibrium. However, whereas the first connection is forced to reduce its congestion window at $t = 3s$, later, the last connection does not have to adjust its window size.

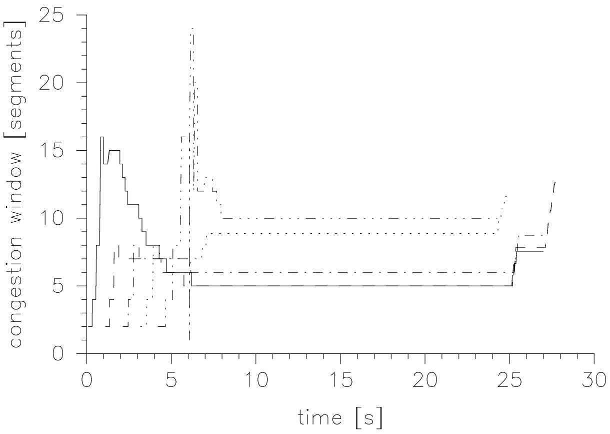

Fig. 2. Connections sharing bottleneck.

The algorithm that triggers the increase of the congestion window suffers from a similar problem. The congestion window is increased when the following condition holds (α is usually set to 1):

$$windowSize < \frac{\alpha}{1 - \frac{baseRTT}{rtt}}.$$

Since the term on the right side is bigger for a connection A, initiated in an congested network, than for a connection B, initiated while the network was uncongested, the condition is more likely to be fulfilled for type A connections than for type B connections. Again, this means that connections of type A can obtain an unfair share of the link bandwidth.

Note that TCP Reno's congestion avoidance strategy may not guarantee fairness, either. However, since every connection suffers some losses from time to time (due to self-induced packet loss), there is at least a chance for other connections to catch up. This may not be the case for TCP Vegas, since TCP Vegas specifically tries to prevent the self-induced losses.

B. Persistent congestion

In addition to the problem discussed in Section IX-A, there are concerns about TCP Vegas behavior in situations with persistent congestion [11]. In such a situation, a TCP Vegas connection overestimates *baseRTT* and although it believes that it has between α and β segments in transit, in fact, it has many more segments in transit. A detailed description of the problem is given in [21].

C. Discussion

There have been suggestions to overcome the problem described in Sections IX-B by setting *baseRTT* to a bigger value

[10] The connections run over the topology shown in Section IV-B; on each side, three hosts are added; the size of the router queues are set to 25 segments.

in case of persistent congestion (e.g., [21]). The problem in Section IX-A may be overcome in a similar way. Although resetting *baseRTT* may be an adequate measure in case of routing changes, where the minimum RTT indeed may become larger, it is not an adequate work-around for the two problem cases described, since *baseRTT* is by definition "the RTT of a segment when the connection is not congested" [6][8]. So, setting *baseRTT* to a bigger value than the minimum measured RTT violates this definition and compromises the congestion avoidance mechanism's theoretical foundation.

X. Conclusions

Our evaluation of TCP Vegas confirmed the results reported by previous work [6][1] which showed that TCP Vegas can achieve significantly higher throughputs than TCP Reno. In addition to previous work, our in-depth analysis of TCP Vegas allows us to determine the effect of the various algorithms and mechanisms proposed by the inventors of TCP Vegas on performance.

Our experiment shows that TCP Vegas's techniques for slow-start and congestion recovery have the most influence on throughput as they are able to avoid timeouts due to multiple segment loss. Therefore, TCP Vegas seems to be quite successful in overcoming a well-known problem of TCP Reno. However, TCP Vegas's most innovative feature, that is, its congestion detection mechanism during congestion avoidance, has only minor or even negative effect on throughput. Moreover, we found that the congestion avoidance mechanism may exhibit problems related to fairness among competing connections. As a consequence, for a conclusive comparative evaluation of TCP Vegas and TCP Reno, one must compare the effectiveness of Vegas's techniques for slow-start and congestion recovery to the effectiveness of similar enhancements to TCP Reno. We limited the discussion to a scenario that has been investigated by the developers of TCP Vegas; different scenarios may still deserve further investigations.

Transport protocols such as TCP incorporate a number of complex algorithms (e.g., for congestion control, data recovery, etc.) whose effects on performance and whose interactions are often not clearly understood. Our approach of a factor analysis allowed us to shed some light on the effectiveness of various algorithms of such a protocol and on the interactions of these algorithms. We want to encourage future protocol developers to decompose design and implementation early on to allow such performance analysis and experimentation.

References

[1] J. S. Ahn, P. Danzig, Z. Liu, and L. Yan. Evaluation of TCP Vegas: Emulation and Experiment. In *Proceedings of ACM SIGCOMM '95*, pages 185–195, August 1995.

[2] J.S. Ahn and P.B. Danzig. Packet Network Simulation: Speedup and Accuracy Versus Timing Granularity. *IEEE Transactions on Networking*, 4(5):743–757, October 1996.

[3] M. Allman and V. Paxson. On Estimating End-to-End Network Path Properties. In *Proceedings of ACM SIGCOMM '99*, pages 263–274, August 1999.

[4] M. Allman, V. Paxson, and W. Stevens. RFC 2581: TCP Congestion Control, April 1999.

[5] J. Bolliger, U. Hengartner, and T. Gross. The Effectiveness of End-to-End Congestion Control Mechanisms. Technical Report 313, ETH Zürich, February 1999.

[6] L. S. Brakmo, S. W. O'Malley, and L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proc. of ACM SIGCOMM '94*, pages 24–35, London, October 1994.

[7] L.S. Brakmo and L.L. Peterson. Performance Problems in BSD4.4 TCP. *Computer Communication Review*, 25(5):69–86, Oct 1995.

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

[9] P.B. Danzig and S. Jamin. A Library of TCP Internetwork traffic Characteristics. Technical Report 91-495, Computer Science Department, USC, 1991.

[10] P.B. Danzig, Z Liu, and L. Yan. An Evaluation of TCP Vegas by Live Emulation. Technical Report 94-588, Computer Science Department, USC, 1994.

[11] S. Floyd. Re: TCP Vegas, April 1994. end2end-tf mailinglist.

[12] S. Floyd. TCP and Successive Fast Retransmits, February 1995. ftp://ftp.ee.lbl.gov/papers/fastretrans.ps.

[13] S. Floyd and T. Henderson. RFC 2582: The NewReno Modification to TCP's Fast Recovery Algorithm, April 1999.

[14] G. Hasegawa, M. Murata, and H. Miyahara. Fairness and Stability of Congestion Control Mechanisms of TCP. In *Proceedings of INFOCOM '99*, pages 1329–1336, March 1999.

[15] J. C. Hoe. Improving the Start-Up Behavior of a Congestion Control Scheme for TCP. In *Proceedings of ACM SIGCOMM '96*, pages 270–280, August 1996.

[16] N.C. Hutchinson and L.L. Peterson. The *x*-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[17] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of ACM SIGCOMM 88*, pages 314–329, August 1988.

[18] V. Jacobson. problems with Arizona's vegas, March 1994. end2end-tf mailinglist.

[19] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., 1991.

[20] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. RFC 2018: TCP Selective Acknowledgment Options, October 1996.

[21] J. Mo, R.J. La, V. Anantharam, and J. Walrand. Analysis and Comparison of TCP Reno and Vegas. In *Proceedings of INFOCOM '99*, pages 1556–1563, March 1999.

[22] Y. Zhang, E. Yan, and S.K. Dao. A Measurement of TCP over Long-Delay Network. In *Proceedings of 6th Intl. Conf. on Telecommunication Systems*, pages 498–504, March 1998.