Analysis and Improvement of Fairness between TCP Reno and Vegas for Deployment of TCP Vegas to the Internet

Go Hasegawa, Kenji Kurata and Masayuki Murata
Graduate School of Engineering Science, Osaka University
1-3, Machikaneyama, Toyonaka, Osaka 560-8531, Japan
Phone: +81-6-850-6616, Fax: +81-6-850-6589
{hasegawa,k-kurata, murata}@ics.es.osaka-u.ac.jp

Abstract

According to the past researches, a TCP Vegas version is able to achieve higher throughput than TCP Tahoe and Reno versions, which are widely used in the current Internet. However, we need to consider a migration path for TCP Vegas to be deployed in the Internet. In this paper, by focusing on the situation where TCP Reno and Vegas connections share the bottleneck link, we investigate the fairness between two versions. From the analysis and the simulation results, we find that the performance of TCP Vegas is much smaller than that of TCP Reno as opposed to an expectation on TCP Vegas. The RED algorithm improves the fairness to some degree, but there still be an inevitable trade-off between fairness and throughput. Accordingly, we next consider two approaches to improve the fairness. The first one is to modify the congestion control algorithm of TCP Vegas, and the other is to modify the RED algorithm to detect misbehaved connections and drop more packets from those connections. We use both of analysis and simulation experiment for evaluating the fairness, and validate the effectiveness of the proposed mechanisms.

1 Introduction

TCP (Transmission Control Protocol) is widely used by many Internet services including HTTP (and World Wide Web) and FTP (File Transfer Protocol). Even if the network infrastructure may change in the future, it is very likely that TCP and its applications would be continuously used. However, TCP Tahoe and Reno versions (and their variants), which are widely used in the current Internet, are not perfect in terms of throughput and fairness among connections, as having been shown in the past literatures. Therefore, active researches on TCP have been made, and many improvement mechanisms have been proposed (see, for example, [1, 2, 3] and the references therein). Among them, a TCP Vegas version [4] is one of the most promising mechanisms by its high performance. TCP Vegas enhances the congestion avoidance algorithm of TCP Reno. In essence, TCP Vegas dynamically increases/decreases its sending window size according to observed RTTs (Round Trip Times) of sending packets, whereas TCP Tahoe/Reno only continues increasing its window size until packet loss is detected. The authors in [4] concludes through simulation and implementation experiments that TCP Vegas can obtain even 40% higher throughput than TCP Reno.

However, we need to consider a migration path when a new protocol is deployed in the operating network, i.e., the Internet. It is important to investigate the effect of existing TCP versions (Tahoe and Reno) on TCP Vegas in the situation where those different versions of TCP co-exist in the network. The authors in [5] have pointed out that when connections of TCP Reno and Vegas share the bottleneck link, the Vegas connection may suffer from significant unfairness. However, the authors have assumed that only a single TCP Reno connection shares the link with another TCP Vegas connection.

In this paper, we focus on the situation where multiple TCP Reno and Vegas connections share the bottleneck link, and investigate the fairness between two versions of TCP to seek the possibility of a future deployment of TCP Vegas. One important point we should take into account is the underlying network assumed by TCP Vegas. When the original TCP Vegas was proposed in [4], the authors did not consider the RED (Random Early Detection) mechanism [6], which is now being introduced in the operating network. TCP Vegas may or may not be effective even when the router is equipped with the RED mechanism. We therefore consider two packet scheduling mechanisms, the RED router as well as the conventional drop-tail router, in our study. One of the contributions in this paper is to derive analysis results of the throughput of TCP Reno and Vegas in such a situation to explain why TCP Vegas cannot obtain the good throughput when sharing the link with TCP Reno. The accuracy of our analysis is validated by comparing with the simulation results. Through the analysis and simulation results, we will show the fairness between TCP Reno and Vegas as follows. TCP Vegas receives significant low and unfair throughput compared with TCP Reno, when the router employs the drop-tail router. With the RED algorithm, on the other hand, the fairness can be improved to some degree, but there still exists an inevitable trade-off between fairness and throughput. That is, if the packet dropping probability of RED is set to be large, the throughput of TCP Vegas can be improved, but the total throughput is degraded.

We believe that the subject treated in this paper is a good example for considering the protocol migration from the existing immature one. It is true that TCP Vegas solely can obtain higher performance than TCP Reno, and it has a good feature of having a backward compatibility with the older versions of TCP. Nevertheless, it is unlikely that a current version

of TCP Vegas penetrates in the Internet as our results clearly indicate. Accordingly, we next suggest two possible ways to improve the fairness between TCP Reno and Vegas. The one is to modify the congestion control algorithm of TCP Vegas. The key idea is that if the network congestion is not caused by TCP Vegas connections, the TCP Vegas connection is permitted to increase its sending window size and to send more packets into the network, in order to compete equally with TCP Reno connections. Another way is to modify the RED algorithm for detecting packets from TCP Reno connections. For this purpose, we can utilize the algorithm proposed in [7] to detect *mis-behaving flows*, which is TCP Reno connections in the current content. Then throughput of TCP Reno connections can be decreased by intentionally dropping packets of TCP Reno at the router. We will present the above two approaches in Sections 5 and 6.

The rest of this paper is organized as follows. Section 2 briefly introduces congestion control mechanisms of TCP Reno and TCP Vegas. We next describe the network model used in our analysis and simulation experiments in Section 3. Section 4 shows the analysis results of fairness between two versions of TCP, which are validated by the simulation results. We next show two proposed mechanisms to improve the fairness. We introduce TCP Vegas+ approach which enhances the congestion control algorithm of TCP Vegas in Section 5, and a ZL-RED algorithm which enhances the RED algorithm in Section 6. Finally, we conclude our paper and present some future works in Section 7.

2 Congestion Control Mechanisms of TCP

In this section, we summarize the congestion control mechanisms of two versions of TCP; TCP Reno and Vegas. For detailed explanation, refer to [8] for TCP Reno and [4] for TCP Vegas. An essence of the congestion avoidance mechanism of TCP is to dynamically control the window size according to the congestion level of the network. Here, we denote the current window size of the sender host at time t as $cwnd(t)$.

2.1 TCP Reno

In TCP Reno, the window size is cyclically changed in a typical situation. The window size continues to be increased until packet loss occurs. TCP Reno has two phases in increasing its window size; slow start phase and congestion avoidance phase. When an ACK packet is received by TCP at the sender side at time $t+t_A$ [sec], the current window size $cwnd(t+t_A)$ is updated from $cwnd(t)$ as follows (see, e.g., [8]);

$$cwnd(t+t_A) = \begin{cases} \text{slow start phase :} & \\ \quad cwnd(t)+1, & \text{if } cwnd(t) < ssth(t); \\ \text{congestion avoidance phase :} & \\ \quad cwnd(t)+\frac{1}{cwnd(t)}, & \text{if } cwnd(t) \geq ssth(t); \end{cases} \tag{1}$$

where $ssth(t)$ [packets] is a threshold value at which TCP changes its phase from slow start phase to congestion avoidance phase. When packet loss is detected by retransmission timeout expiration, $cwnd(t)$ and $ssth(t)$ are updated as [8];

$$cwnd(t) = 1; \qquad ssth(t) = cwnd(t)/2 \tag{2}$$

On the other hand, when TCP detects packet loss by a fast retransmit algorithm [8], it changes $cwnd(t)$ and $ssth(t)$ as;

$$ssth(t) = cwnd(t)/2; \qquad cwnd(t) = ssth(t) \tag{3}$$

2.2 TCP Vegas

TCP Vegas controls its window size by observing RTTs (Round Trip Time) of packets that the sender host has sent before [4]. If observed RTTs become large, TCP Vegas recognizes that the network begins to be congested, and throttles the window size. If RTTs become small, on the other hand, the sender host of TCP Vegas determines that the network is relieved from the congestion, and increases the window size again. Hence, the window size in an ideal situation is expected to be converged to an appropriate value. That is,

$$\begin{aligned} cwnd(t+t_A) &= \\ &\begin{cases} cwnd(t)+1, & \text{if } diff < \frac{\alpha}{base_rtt} \\ cwnd(t), & \text{if } \frac{\alpha}{base_rtt} \leq diff \leq \frac{\beta}{base_rtt} \\ cwnd(t)-1, & \text{if } \frac{\beta}{base_rtt} < diff \end{cases} \\ diff &= cwnd(t)/base_rtt - cwnd(t)/rtt \end{aligned} \tag{4}$$

where rtt [sec] is an observed round trip time, $base_rtt$ [sec] is the smallest value of observed RTTs, and α and β are some constant values.

3 Network Model

Figure 1 shows the network model used in this paper. It consists of N_r sender hosts using TCP Reno (SR$_1$, ... SR$_{N_r}$), N_v sender hosts using TCP Vegas (SV$_1$, ... SV$_{N_v}$), a receiver host, an intermediate router, and links connecting the router and the sender/receiver hosts. The bandwidth of each link between the sender hosts and the router is bw [Mbps]. The bandwidth of the bottleneck link between the router and the receiver host is BW [Mbps] = μ [packets/sec]. The size of the buffer at the router is B [packets]. The propagation delay between the sender hosts and the router and that between the router and the receiver host are represented by τ_{sx} [sec] and τ_{xd}[sec], respectively. We denote the total propagation delay between the sender hosts and the receiver host by τ, being equal to τ_{sx} + τ_{xd}. As the scheduling discipline at the router buffer, we consider drop-tail and RED algorithms.

4 Fairness Comparison

4.1 Analysis

In what follows, we use the network model depicted in Figure 1, and derive the average throughput of each TCP connection through a mathematical analysis. In the analysis, we assume that the throughput of each connection becomes proportional to buffer occupancy at the drop-tail router. It is also

appropriate for the RED router as we will explain in the below. Note that the validation of our approximate analysis will be given in Subsection 4.2.

4.1.1 Case of Drop-Tail Router

In Figure 2, we illustrate a typical change of the total number of packets queued at the router buffer when the drop-tail algorithm is utilized. Here, we assume that all TCP Reno connections behave identically. Since TCP Reno connections continue to increase their window sizes until packet loss occurs at the buffer, the change of the window size also has cycles triggered by packet losses , even when the TCP Reno connections share the link with TCP Vegas connections. By assuming that all packet losses can be detected by the fast retransmit algorithm, it takes one RTT [sec] for the sender side TCP to detect the packet loss after the packet loss actually occurs at the route buffer. It corresponds to the flat part of buffer occupancy shown in Figure 2.

TCP Vegas connections, on the other hand, control their window sizes according to the observed RTTs of sending packets. Each of those tries to keep the number of queued packets in the router buffer between α and β [packets] [3]. As RTTs becomes large, TCP Vegas connections continue to decrease their window sizes. On the other hand, TCP Reno connections continue to increase their window sizes regardless of the increased RTT, which results in that the window sizes of the TCP Vegas connections are decreased until those reach within the range from α to β [packets]. See Eq. (4). From the above observation, the total of window sizes of N_v TCP Vegas connections, W_v [packets], is obtained as;

$$N_v\alpha < W_v < N_v\beta. \tag{5}$$

We determine $\overline{W_v}$ [packets], the average value of W_v, from Eq. (5) as follows;

$$\overline{W_v} = N_v\frac{\alpha+\beta}{2}, \tag{6}$$

which is a reasonable assumption from its behavior.

TCP Reno connections continue to increase their window sizes until the router buffer becomes full and eventually some packets are lost. Accordingly, W_r [packets], the total of the window sizes of TCP Reno connections when packet loss occurs at the router buffer, can be obtained as;

$$W_r = 2\tau\mu + B - W_v. \tag{7}$$

The number of lost packets during buffer overflow duration becomes N_r [packets], since from Eq. (1), the window sizes of TCP Reno connections are increased by 1 [packet/RTT] in the congestion avoidance phase as having been explained in Section 2. By assuming that a packet loss probability for each connection is proportional to its window size, we can obtain L_r [packets] and Lv [packets], the numbers of packet losses of TCP Reno and Vegas connections during buffer overflow duration, respectively, as;

$$L_r = N_r\frac{W_r}{W_r+W_v}; \qquad L_v = N_r\frac{W_v}{W_r+W_v} \tag{8}$$

Each of TCP Reno connections detecting the packet loss halves its window size according to the fast retransmit algorithm. Therefore, W'_r [packets], the total window size of the TCP Reno connections just after the buffer overflow, can be determined by Eqs. (1) and (8) as;

$$\begin{aligned} W'_r &= \frac{1}{2}\cdot\frac{W_r}{N_r}\cdot L_r + \frac{W_r}{N_r}\cdot(N_r - L_r) \\ &= \frac{W_r+2W_v}{2(W_r+W_v)}\cdot W_r \end{aligned} \tag{9}$$

From Eq. (1) (and Figure 2), the following equation holds for $\overline{W_r}$ [packets], the average value of the total window size of TCP Reno connections;

$$\overline{W_r} = \frac{\frac{1}{2}(W_r+W'_r)\frac{W_r-W'_r}{N_r} + W_r}{\frac{W_r-W'_r}{N_r}+1} \tag{10}$$

Accordingly, we obtain $\overline{B_r}$ [packets] and $\overline{B_v}$ [packets], the average number of packets at the router buffer for TCP Reno and Vegas, respectively;

$$\overline{B_r} = \overline{W_r}\cdot\frac{B}{2\tau\mu+B}; \qquad \overline{B_v} = \overline{W_v}\cdot\frac{B}{2\tau\mu+B} \tag{11}$$

We finally have ρ_r [packets/sec] and ρ_v [packets/sec], the average throughput of the connections of two versions of TCP as;

$$\rho_r = \mu\cdot\frac{B_r}{B_r+B_v}; \qquad \rho_v = \mu\cdot\frac{B_v}{B_r+B_v}, \tag{12}$$

since we have assumed that they become proportional to the buffer occupancy at the router.

4.1.2 Case of RED Router

The RED algorithm drops incoming packets at the preset probability when the number of packets in the buffer exceeds a certain threshold value [6]. For simplicity of the following analysis, it is assumed that all packet losses occur with probability p by the RED algorithm, and no buffer overflow takes place.

Even with the RED algorithm, TCP Reno connections continue to increase their window sizes until packet loss occurs. Therefore, as in the drop-tail case, the TCP Vegas connections cannot open their window sizes and keep them ranging from α to β. Therefore, the following equations yield for W_v and $\overline{W_v}$;

$$N_v\cdot\alpha < W_v < N_v\cdot\beta; \qquad \overline{W_v} = N_v\frac{\alpha+\beta}{2} \tag{13}$$

Each of TCP Reno connections, on the other hand, changes its window size cyclically triggered by packet losses as in the drop-tail router case. Since all arriving packets are dropped with probability p by our assumption, the connection can send $1/p$ packets in one cycle (between two events of packet losses) on average. We define the number of packets transmitted during one cycle as N_p, and is given by

$$N_p = 1/p \tag{14}$$

Different from the drop-tail router case, we focus on a certain TCP Reno connection because we assume that all TCP Reno connections behave identically under the stochastic packet dropping algorithm employed by RED.

Although the RED algorithm can eliminate the bursty packet losses, retransmission timeout expiration cannot be perfectly avoided [9]. Even if timeout expiration rarely happens, the effect of timeout expiration on throughput is not negligible. Therefore, we must take into account throughput degradation caused by timeout expiration. We denote the probability of occurring timeout expiration within the window by p_{to}. By using $\overline{w_r}$, the average value of the window size of a certain TCP Reno connection when packet loss is detected, we determine p_{to} by a following simple equation;

$$p_{to} = \sum_{i=2}^{\overline{w_r}} \binom{\overline{w_r}}{i} \cdot p^i \cdot (1-p)^{\overline{w_r}+1-i} \qquad (15)$$

In what follows, we distinguish two cases of detecting packet loss; retransmission timeout expiration (*TO case*) and the fast retransmit (*FR case*), because in each of two cases, a different algorithm of changing the window size is used.

In the *TO case*, that is, if packet loss is detected by retransmission timeout expiration, the window size is reset to 1 [packet]. It is then updated according to the slow start phase (Eq. (1)) until it reaches $\overline{w_r}/2$ [packets]. From Eq. (1), we can determine $T_{to,1}$ [sec], the time duration of the slow start phase, and $A_{to,1}$ [packets], the number of packets transmitted in the slow start phase, by the following equations.

$$T_{to,1} = rtt \cdot \log_2(\overline{w_r}/2); \qquad A_{to,1} = (\overline{w_r}/2) - 1 \qquad (16)$$

where rtt [sec] is the mean value of RTTs of sending packets. Furthermore, we can easily obtain $T_{to,2}$ [sec] and $A_{to,2}$ [packets], which are the time duration and the number of transmitted packets in the following congestion avoidance phase, respectively, from Eq. (1) as;

$$T_{to,2} = rtt \cdot (\overline{w_r} - \overline{w_r}/2) \qquad (17)$$

$$A_{to,2} = \frac{1}{2}(\overline{w_r} + \overline{w_r}/2)(\overline{w_r} - \overline{w_r}/2) \qquad (18)$$

These equations hold due to the fact that the window size is increased by 1 [packet] per RTT [sec] in the congestion avoidance phase (Eq. (1)).

On the other hand, if the TCP Reno connection detects the packet loss by the fast retransmit algorithm (*FR case*), the window size is halved to $\overline{w_r}/2$, and the congestion avoidance phase starts again. That is, time duration and the number of transmitted packets during the slow start phase (denoted as $T_{fr,1}$ and $A_{fr,1}$, respectively) are zeros, i.e.,

$$T_{fr,1} = 0; \qquad A_{fr,1} = 0 \qquad (19)$$

Similarly, time duration and the number of transmitted packets in the congestion avoidance phase ($T_{fr,2}$ and $A_{fr,2}$) are represented as

$$T_{fr,2} = rtt\,(\overline{w_r} - \overline{w_r}/2) \qquad (20)$$

$$A_{fr,2} = \frac{1}{2}(\overline{w_r} + \overline{w_r}/2)(\overline{w_r} - \overline{w_r}/2) \qquad (21)$$

Consequently, the following equations are satisfied for the number of transmitted packets and the average window size during one cycle from Eqs. (16)-(21);

$$N_p = p_{to}(A_{to,1} + A_{to,2}) + (1-p_{to})(A_{fr,1} + A_{fr,2}) \qquad (22)$$

$$\overline{w_r} = rtt \cdot p_{to}\left(\frac{A_{to,1} + A_{to,2}}{T_{to,1} + T_{to,2} + rto}\right) + rtt \cdot (1-p_{to})\left(\frac{A_{fr,1} + A_{fr,2}}{T_{fr,1} + T_{fr,2}}\right) \qquad (23)$$

where rto [sec] is the retransmission timeout value of the connection. Since we can obtain p_{to} and $\overline{w_r}$ by solving Eqs. (22) and (23), the average value of the total window size of all TCP Reno connections, $\overline{W_r}$, can be easily obtained as follows;

$$\overline{W_r} = N_r \overline{w_r} \qquad (24)$$

Finally, ρ_r and ρ_v in the RED case can be determined similarly to the drop-tail router case, from Eqs. (11)–(12), (13), (13) and (24).

4.2 Numerical Examples and Discussions

In this Subsection, we show some numerical examples by using analysis results presented in the previous Subsection, which are aimed at discussing the fairness between two versions of TCP. Simulation results are also provided to assess the accuracy of our analysis. In what follows, we set τ_{sx} = 0.0015 [sec], $\tau_{xd} = 0.005$ [sec], $bw = 10$ [Mbps] and $BW = 1.5$ [Mbps] as network parameters. For the RED router, we set the threshold values, $th_{min} = 5$ [packets] and $th_{max} = 0.6 \times B$ [packets].

4.2.1 Case of Drop-Tail Router

Figure 4 shows the average throughput of TCP Reno and TCP Vegas connections as a function of the buffer size B [packets] of the drop-tail router. We consider three cases for the number of connections of TCP Reno and Vegas (N_r and N_v); $N_r = 5$, $N_v = 5$ for Figure 4(a), $N_r = 5$, $N_v = 10$ for Figure 4(b), and $N_r = 10$, $N_v = 5$ for Figure 4(c). In these figures, we show both of the analysis and simulation results for validating our analysis presented in Subsection 4.1. We can see in these figures that our analysis gives appropriate estimations of throughput, regardless of the number of connections of two versions of TCP. However, especially when the router buffer size is very small (< 20 [packets]), however, our analysis under-estimates the throughput of TCP Reno connections, and over-estimates that of TCP Vegas connections. It is because the assumption that the window sizes of TCP Vegas connections are fixed at $\overline{W_v} = (\alpha + \beta)/2$ does not hold for too small buffer size, while such a very small buffer size is not realistic.

An important observation obtained from Figure 4 is that TCP Vegas connections suffer from significantly low throughput, compared with TCP Reno connections. It is due to the difference of buffer occupancy at the router. TCP Reno connections can increase their window sizes until the buffer becomes full and packet loss occurs. On the other hand, TCP

Vegas connections does not inflate the window size larger than β, as have been described in Subsection 4.1. This observation can be confirmed by our analysis in the previous subsection. From Eqs. (7) and (10), the average window size of TCP Reno connections becomes large as the router buffer size B [packets] is increased. Since the increase of the window size of each TCP Reno connection can directly lead to the throughput improvement, as can be seen from Eqs. (11) through (12). On the other hand, the window size of TCP Vegas connections remain unchanged regardless of the router buffer size (see Eq. (6)). Therefore, buffer occupancy of TCP Vegas connections is decreased as the router buffer size is set to be large. That is, the larger the router buffer size becomes, the worse the fairness between TCP Reno and TCP Vegas connections becomes.

In this subsection, we have considered the drop-tail router. The mechanism of the RED router can inhibit the bursty losses of packets from the same connection to improve the fairness among connections. Such a mechanism is also useful in our case, which will be examined in the next subsection.

4.2.2 Case of RED Router

We next show the case of the RED router in Figure 5. In this case, the packet dropping probability, p, is set to be 1/30. Analysis results in the figure are not affected by the router buffer size. It is because we have assumed that in our analysis, the packet dropping probability is constant, and that all packet drops are caused by stochastic dropping of the RED algorithm, not by the buffer overflow of packets. The differences between analysis and simulation results become apparent when the buffer size is small because in that region, throughput degradation caused by buffer overflow cannot be negligible. However, such a small buffer size is not realistic in the operating network and our analysis results can well illustrate how different the throughput performance of two versions of TCP are.

We can observe from Figure 5 that the fairness between two versions of TCP is greatly improved when compared with the case of drop-tail router, while the total throughputs of all connections are almost identical for the large buffer size. It can be explained as follows. With the RED algorithm, TCP Reno connections does not inflate their window sizes until the router buffer becomes fully–utilized, since packet loss occurs before the buffer becomes full due to an essential nature of the RED algorithm. It results in the decrease of buffer occupancy of TCP Reno connections, leading to throughput degradation of TCP Reno connections. It also contributes the throughput improvement of TCP Vegas connections. The observation can be confirmed by our analysis. In contrast with the drop-tail router case, the window size of TCP Reno is independent on the router buffer size, since the total number of packets transmitted between two events of packet losses is only dependent on the packet dropping probability of the RED algorithm p as shown in Eq. (14). Therefore, throughput values of two versions are not changed even when the router buffer size becomes large.

From the above discussion, one may expect that if the packet dropping probability is further increased, the fairness can be improved because the average window sizes of TCP Reno connections gets smaller. This observation can be partly confirmed by Figure 6, where we increase the packet dropping probability to 1/10 (from 1/30 in the previous case). We can see the fairness enhancement by comparing with the previous results in Figure 5. It can be verified by Eq. (14), i.e., the number of packets that the sender host can transmit in one cycle is decreased as p becomes large. It causes the decrease of the average window size of TCP Reno connections, because it inflates its window size until the packet loss is detected. Then, buffer occupancy of TCP Reno connections is decreased, and that of TCP Vegas connections is increased, since the window size of TCP Vegas is not affected by p. Hence, the fairness between the two versions of TCP can be improved.

As one can naturally imagine, however, we cannot avoid the degradation of the total throughput if the packet dropping probability of RED algorithm is set too high for further fairness improvement. Figure 3 shows simulation results for the throughput of TCP Reno and Vegas connections and the total throughput, by changing p (the packet dropping probability of the RED algorithm). In obtaining this figure, we fix the other parameters; $N_r = 5$, $N_v = 5$, and $B = 100$ [packets]. We can see from the figure that when the packet dropping probability becomes large (> 0.01), the fairness between two versions of TCP can be much improved, but the total throughput degrades. In other words, there exists an inevitable trade-off between fairness and throughput in the RED algorithm. Furthermore, it would be difficult to choose an appropriate value of p in the operating network since it must be affected by the active numbers of connections of two TCP versions.

In this paper, we have considered two versions of TCP. The one is TCP Reno; an existing and widely used protocol. The other is TCP Vegas; the newly proposed protocol which gives higher throughput than TCP Reno as having been demonstrated in the original paper of TCP Vegas [4]. TCP Vegas also has an excellent feature of the backward compatibility to the older versions of TCP including TCP Reno. However, when two versions of TCP share the bottleneck link, the performance of TCP Vegas is much degraded, which was not originally expected. For the new protocol to be deployed in the operating network, its migration path should be taken into account. In this sense, TCP Vegas does not seem to be successful.

However, there are several approaches to overcome the above problem. One possible solution is to improve the congestion control algorithm of TCP Vegas itself to be able to compete equally with TCP Reno. For this, the window of TCP Vegas should be increased more aggressively as TCP Reno does. Another approach is to modify the RED algorithm at the router so that the router can detect *mis-behaving connections*, which correspond to TCP Reno connections in the current context. Then the router eliminates the unfairness by intentionally dropping more packets from the mis-behaving connections than well-behaving connections. In next two sections, we investigate those approaches in turn.

5 Modification to TCP Vegas

As described in the previous section, one reason of the unfairness between TCP Reno and Vegas is due to the difference of their congestion control algorithms. An *aggressive* increase of window sizes in TCP Reno much affects the performance of TCP Vegas controlling their window sizes *moderately*. Therefore, we modify TCP Vegas so that it has an ability to compete the link at least equally with TCP Reno connections, while preserving the merit of TCP Vegas of the stability of the window size. In this section, we propose an approach, called *TCP Vegas+*, and show some simulation results to verify its effectiveness.

5.1 Algorithm

In TCP Vegas+, we only change the updating algorithm for the window size in the original TCP Vegas, and remains unchanged for other functions, which include the detection algorithm of packet loss, and the *slow* slow start mechanism (Section 2). TCP Vegas+ normally behaves identically with TCP Vegas, but it enters the other mode to increase its window size more aggressively when it perceives to have competing connections of TCP Reno. More specifically, TCP Vegas+ has two modes for updating its window size;

Moderate Mode: In the moderate mode, the TCP Vegas+ sender behaves identically to the original TCP Vegas, i.e., the window size is updated according to Eq. (4).

Aggressive Mode: In the aggressive mode, the TCP Vegas+ sender host behaves identically to TCP Reno. That is, it updates the window size according to Eq. (1). This mode is for TCP Vegas+ connections to keep fair throughput against TCP Reno connections.

Most important is to switch between the above two modes. For this purpose, we introduce new variables $count$ and $count_{max}$. First, $count$ is updated according to the following algorithm.

1. On every receipt of an ACK packet, the sender observes its window size and the RTT value. If RTT is larger than the previous value while the window size is not increased, the sender increments $count$ by 1.
2. On the other hand, if RTT becomes smaller, the sender decrements $count$ by 1.
3. If packet loss is detected by a fast retransmit algorithm, $count$ is halved.
4. If packet loss is detected by a retransmission timeout expiration, $count$ is reset to 0.

TCP Vegas+ then changes its mode according to the $count$ value;

Moderate Mode → Aggressive Mode: If $count$ reaches a certain threshold value $count_{max}$, the sender changes its mode from the moderate mode to the aggressive mode.

Aggressive Mode → Moderate Mode: If $count$ becomes 0, it goes back to the moderate mode.

A rational behind the above algorithm is as follows; if the RTT value becomes larger whereas the window size is unchanged, it can be considered that the increase of RTT is not caused by the TCP Vegas+ connection itself, but by other TCP Reno connections, which increases its window size more aggressively than the TCP Vegas+ connection. Then the TCP Vegas+ connection should increase its window size more aggressively to compete equally with the other connections. When packet loss is detected, on the other hand, the TCP Vegas+ should change its mode from the aggressive mode to the moderate mode. It is because the packet loss indicates the network congestion occurrence, and the congestion may be caused by the aggressive increase of the window size of itself.

5.2 Simulation Results and Discussions

We first show the time–dependent behaviors of the window size in Figures 7(a) and 7(b) where TCP Vegas and Vegas+ are applied, respectively. We used the network model depicted in Figure 1, and set $N_r = 5$, $N_v = 1$, and $B = 100$ [packets]. The five TCP Reno connections sends data from 250 [sec] to 500 [sec], and from 750 [sec] to 1000 [sec] of the simulation time. The drop-tail router is assumed in this experiment. For the parameter of TCP Vegas+, we set $count_{max} = 8$. Instantaneous throughput value of two cases are shown in Figure 8.

As can be observed from Figure 7(a), the window size of the original TCP Vegas remains almost unchanged even when the TCP Reno connections starts packet transmission. Therefore, the throughput is degraded when TCP Reno connections exists because buffer occupancy at the router becomes very low. See Figure 8(a). In the case of TCP Vegas+, on the other hand, the TCP Vegas+ connection can increase its window size up to almost equal values with the TCP Reno connections when the TCP Reno connections join the network (Figure 7(b)). Furthermore, when the TCP Reno connections do not exist (from 0 [sec] to 250 [sec] and from 500 [sec] to 750 [sec] of the simulation time), the window size is stable as in the case of TCP Vegas. This is just a behavior of TCP Vegas+ that we want to realize. Consequently, rather good fairness can be achieved in terms of throughput as shown in Figure 8(b).

We next present several results on fairness corresponding to the results presented in Section 4. We set $\tau_{sx} = 0.0015$ [sec], $\tau_{xd} = 0.005$ [sec], $bw = 10$ [Mbps] and $BW = 1.5$ [Mbps]. $Count_{max}$ for the TCP Vegas+ parameter is unchanged to be 8. Figure 9 shows simulation results; Figure 9(a) for $N_r = 5$ and $N_v = 5$, Figure 9(b) for $N_r = 5$ and $N_v = 10$, and Figure 9(c) for $N_r = 10$ and $N_v = 5$. In these figures, we also show results of the original TCP Vegas case. Those are same as Figure 4 in Section 4. These figures show that the fairness between two versions of TCP is significantly improved especially when the buffer size is comparatively large. It can also be observed that fairness improvement can be achieved regardless of the number of connections of TCP Reno and Vegas+.

6 Modification to the Scheduling Algorithms at the Router

In this section, we consider another way to improve fairness between TCP Reno and Vegas, by modifying the RED algorithm at the router. Our proposed algorithm, called *ZL-RED* (Zombie Listed RED), is based on the mechanism proposed in [7], and a function of dropping incoming packets is added to improve the fairness.

6.1 ZL-RED Algorithm

In Section 4, we have shown that the original RED can improve the fairness to some degree, but there is an inevitable trade-off between fairness and throughput. The main reason was that RED drops incoming packets from different connections with same probability, regardless of the characteristics of the connections. Then, as we set the packet dropping probability to a higher value, throughput values of the TCP Reno and Vegas connection become lower, and the total throughput gets smaller while we can obtain better fairness. Therefore, for further fairness improvement without throughput degradation, we need two mechanisms; the one is how to detect TCP Reno connections, and the other is how to drop more packets from TCP Reno connections. Our mechanism is inspired by SRED proposed in [7], where the way to find a *mis-behaving* flow are described. In our context, it corresponds to TCP Reno connections, and we need additional mechanisms as will be described below.

6.1.1 How to Detect Mis-behaving Connections

Several methods have already been proposed to identify mis-behaving connections, and to provide fair service at the router [10, 11, 12]. However, most of them use *per-flow information* to determine the mis-behaving connections, and such methods have an essential problem; inscalability against the number of accommodated connections. Accordingly the authors in [7] have introduced an algorithm which does not use any per-flow information for detecting mis-behaving connections. Since our method is based on SRED, we briefly summarize the SRED algorithm first.

Instead of per-flow information, SRED maintains a fixed-size table called a *zombie list*. Each entry of the zombie list contains information on incoming packets (i.e., source/destination addresses and possibly port numbers), a timestamp and a counter. The zombie list is initialized to be empty. When the packet arrives at the router, the router adds a new entry for the packet. If the zombie list is full on packet arrival, the router randomly selects one entry from the zombie list. If information of the selected entry is identical to that of the arriving packet (the authors in [7] call it *hit*.), the router increments the counter of the entry by 1. Otherwise, the router replaces the selected entry by the information of the arrived packet with a certain probability. By this algorithm, mis-behaving connections can be detected as follows. Suppose that a certain connection is mis-behaving, that is, a connection sends more packets to the router than other connections. Then, the zombie list tends to contain more entries of the mis-behaving connections. Therefore, when the packets from the mis-behaving connection arrives at the router, the packet *hits* in the zombie list more frequently, because the probability that the randomly selected entry coincides with the arriving packet becomes larger. Furthermore, the counter value of the entry of the mis-behaving connections also becomes larger. By these two mechanisms, the router can identify the mis-behaving connections.

6.1.2 How to Drop More Packets from Mis-behaving Connections

To keep fairness between mis-behaving connections and well-behaving connections, the router should intentionally drop packets from the mis-behaving connections with higher probability. The authors in [7] have proposed the mechanism to detect mis-behaving connections using the zombie list, but have not shown any method to improve fairness between mis-behaving connections and well-behaving connections by using the zombie list. In this subsection, we propose one possible way, ZL-RED (Zombie Listed RED), to realize it,

In ZL-RED, packet dropping is performed in two steps. At the first step, an incoming packet is dropped with probability p_1. Its determination follows the algorithm presented in [7];

$$p' = \begin{cases} 0 & \text{if } q_{len} < th_{min} \\ p_{min} & \text{if } th_{min} \le q_{len} < th_{max} \\ p_{max} & \text{if } th_{max} \le q_{len} \end{cases} \quad (25)$$

$$p_1 = p'\left(1 + Hit(t)/P(t)\right), \quad (26)$$

where q_{len} is the number of packets in the router buffer when the packet arrives at the router, and p_{min}, p_{max}, th_{min} [packets] and th_{max} [packets] are some constant values. $P(t)$ is the probability with which the incoming packet *hit*s. According to [7], $1/P(t)$ becomes the average number of active connections at the router, if the packet arriving rates of the all connections are equal. Further, $Hit(t)$ is defined as;

$$Hit(t) = \begin{cases} 1, & \text{if the incoming packet } hit\text{s} \\ 0, & \text{otherwise} \end{cases} \quad (27)$$

In the second step, the router drops the packet with probability p_2 when the packet is not dropped in the first step. The probability p_2 is determined by the following equation.

$$p_2 = \begin{cases} \frac{hit_{drop}}{1+P(t)}, & \text{if } Hit(t) = 1 \text{ and } q_{len}P(t) \ge 2.0 \\ 0, & \text{otherwise} \end{cases} \quad (28)$$

where hit_{drop} is a control parameter of the ZL-RED algorithm. It affects the packet dropping probability of mis-behaving connections; if we set hit_{drop} to a larger value, packets from mis-behaving connections are more frequently dropped than those from well-behaving connections.

6.2 Simulation Results and Discussions

In this subsection, we present an effectiveness of our ZL-RED algorithm through several simulation results. Figure 10 shows

throughput and packet loss rate of TCP Reno and Vegas connections as a function of the buffer size at the router. We set the ZL-RED parameters as follows; $p_{min} = 1/50$, $p_{max} = 1/10$, $th_{min} = 5$ [packets], and $th_{max} = 0.6\,B$ [packets] where B [packets] is the buffer size at the router. The size of the zombie list is set to be 1000 [entries]. Further, in obtaining Figure 10, we set hit_{drop} to 0.8. The results are shown in Figure 10. We can see from this figure that the fairness between TCP Reno and Vegas becomes better than case of RED (Figure 5 and 6). It means that our ZL-RED can effectively drop packets from mis-behaving connections, i.e., TCP Reno connections here. It is because in the first step of ZL-RED, the packet dropping probability is increased if an incoming packet often *hits* as shown in Eq. (26). That is, the packet loss rate of TCP Reno connections becomes higher than that of TCP Vegas connections due to its aggressive window size increase. It is shown in Figures 10(b), 10(d), and 10(f). Note that since the original RED algorithm drops incoming packets with a constant probability, the packet loss rate of TCP Reno and Vegas connections becomes identical.

However, the ZL-RED algorithm proposed in this section cannot eliminate the unfairness between TCP Reno and TCP Vegas perfectly as shown in Figure 10 while the fairness is much better than the original RED algorithm. Further fairness improvement using ZL-RED should be a future research topic.

7 Conclusion

In this paper, we have investigated the fairness between TCP Reno and Vegas in the case where the TCP connections of the two versions share the bottleneck link. We have observed the following results through the mathematical analysis and the simulation experiments; TCP Vegas suffers from serious performance degradation with drop-tail routers, because of the difference of buffer occupancy at the router. RED routers can improve the fairness to some degree, but there exists an inevitable trade-off between fairness and throughput.

We have then proposed two approaches to improve the fairness. The first one is TCP Vegas+, which enhances the congestion control algorithm of the original TCP Vegas so that TCP Vegas can compete equally with TCP Reno. The second proposal is the ZL-RED algorithm, which detects the mis-behaving connections at the router buffer, and tries to intentionally drops more arriving packets from the mis-behaving (i.e. TCP Reno) connections than that from well-behaving (TCP Vegas) connections. We have confirmed the effectiveness of the proposed mechanisms through the simulation experiments. Then, which is better? ZL-RED is a natural extension of the existing algorithm eliminating the mis-behaving flows. However, since our motivation is to achieve fairness between TCP Vegas and Reno, an extension to TCP Vegas seems to be more adequate. The problem is that since both approaches require the appropriate parameter choice, we cannot decide it at this moment, and we need more researches on the subject.

Acknowledgement

This work was partly supported by Research for the Future Program of JSPS under the Project "Integrated Network Architecture for Advanced Multimedia Application Systems," Special Coordination Funds for promoting Science and Technology of the Science and Technology Agency of the Japanese Government, Telecommunication Advancement Organization of Japan under the Project "Global Experimental Networks for Information Society Project," a Grant-in-Aid for Scientific Research (A) (2) 11305030 from The Ministry of Education, Science, Sports and Culture of Japan, and financial support on "Research on transport-layer protocol for the future high-speed network," from the Telecommunications Advancement Foundation.

References

[1] Michel Perloff and Kurt Reiss, "Improvements to TCP performance," *Communications of ACM*, vol. 38, no. 2, pp. 90–100, February 1995.

[2] Matthew Mathis and Jamshid Mahdavi, "Forward acknowledgment: Refining TCP congestion control," *ACM SIGCOMM Computer Communication Review*, vol. 26, no. 4, pp. 281–291, October 1996.

[3] Go Hasegawa, Masayuki Murata, and Hideo Miyahara, "Fairness and stability of the congestion control mechanism of TCP," in *Proceedings of IEEE INFOCOM'99*, March 1999, pp. 1329–1336.

[4] Lawrence S. Brakmo, Sean W.O'Malley, and Larry L. Peterson, "TCP Vegas: New techniques for congestion detection and avoidance," in *Proceedings of ACM SIGCOMM'94*, October 1994, pp. 24–35.

[5] Jeonghoon Mo, Richard J. La, Venkat Anantharam, and Jean Walrand, "Analysis and comparison of TCP reno and vegas," in *Proceedings of IEEE INFOCOM'99*, March 1999.

[6] Sally Floyd and Van Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, August 1993.

[7] Teunis J. Ott, T. V. Lakshman, and Larry Wong, "SRED: Stabilized RED," in *Proceedings of IEEE INFOCOM'99*, March 1999.

[8] W. Richard Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley, Reading, Massachusetts, 1994.

[9] K. Fall and S. Floyd, "Simulation-based comparisons of Tahoe, Reno, and SACK TCP," *ACM SIGCOMM Computer Communication Review*, vol. 26, no. 2, pp. 5–21, July 1996.

[10] D. Lin and R. Morris, "Dynamics of random early detection," in *Proceedings of ACM SIGCOMM'97*, October 1997, pp. 127–137.

[11] I. Stoica, S. Schenker, and H. Zhang, "Core-stateless fair queueing: Achieving approximately bandwidth allocations in high speed networks," in *Proceedings of ACM SIGCOMM'98*, September 1998, pp. 118–130.

[12] M. Shreedhar and George Varghese, "Efficient fair queuing using deficit round robin," *IEEE/ACM Transactions on Networking*, vol. 4, no. 3, pp. 375–385, June 1996.

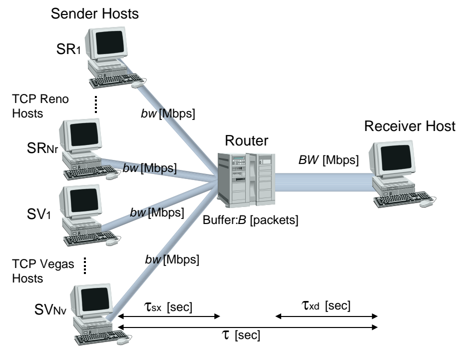


Figure 1: Network Model

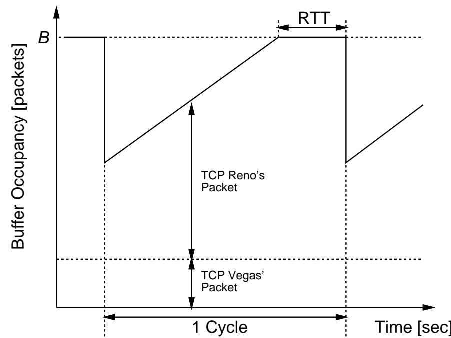


Figure 2: A Typical Change of Buffer Occupancy at Drop-tail Router

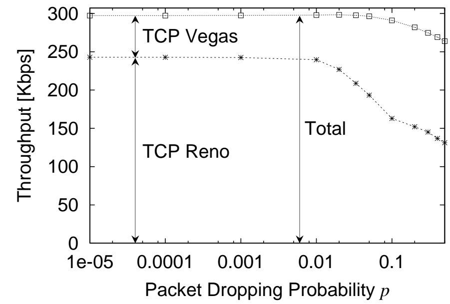
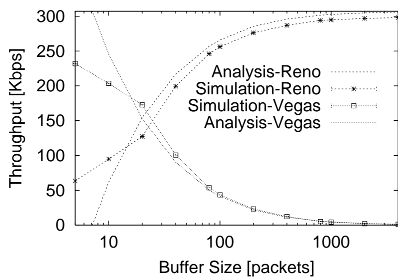
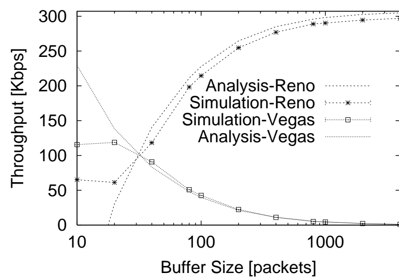


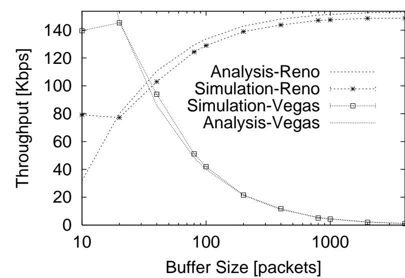
Figure 3: Throughput vs. Packet Dropping Probability of RED Algorithm



(a) $N_r = 5$, $N_v = 5$.

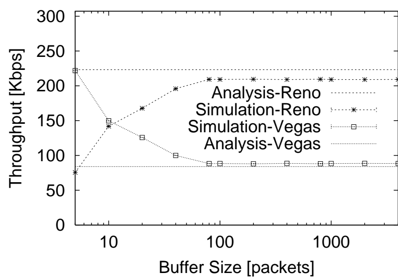


(b) $N_r = 5$, $N_v = 10$.

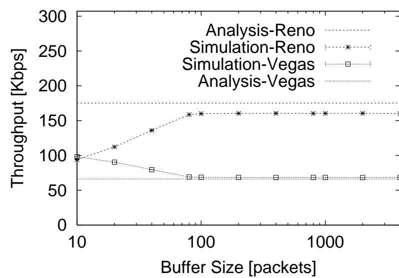


(c) $N_r = 10$, $N_v = 5$.

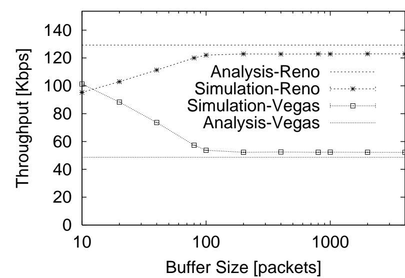
Figure 4: Case of Drop-Tail Router



(a) $N_r = 5$,$N_v = 5$.

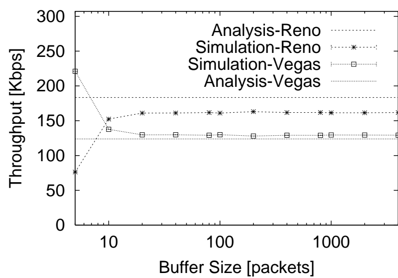


(b) $N_r = 5$, $N_v = 10$.

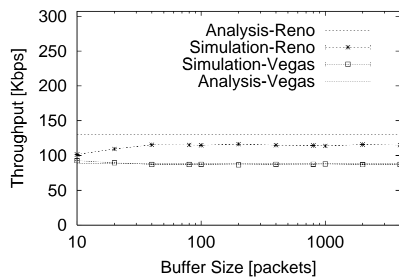


(c) $N_r = 10$, $N_v = 5$.

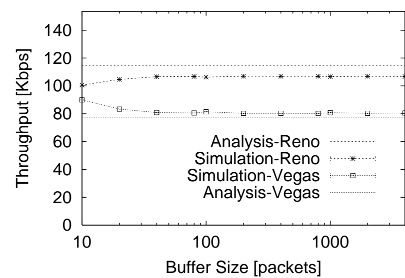
Figure 5: Case of RED Router: $p = 1/30$



(a) $N_r = 5$, $N_v = 5$.

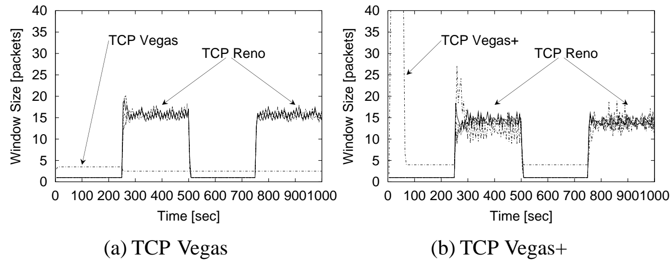


(b) $N_r = 5$, $N_v = 10$.



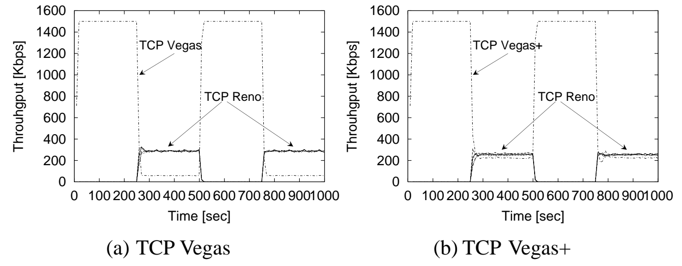
(c) $N_r = 10$, $N_v = 5$.

Figure 6: Case of RED Router: $p = 1/10$



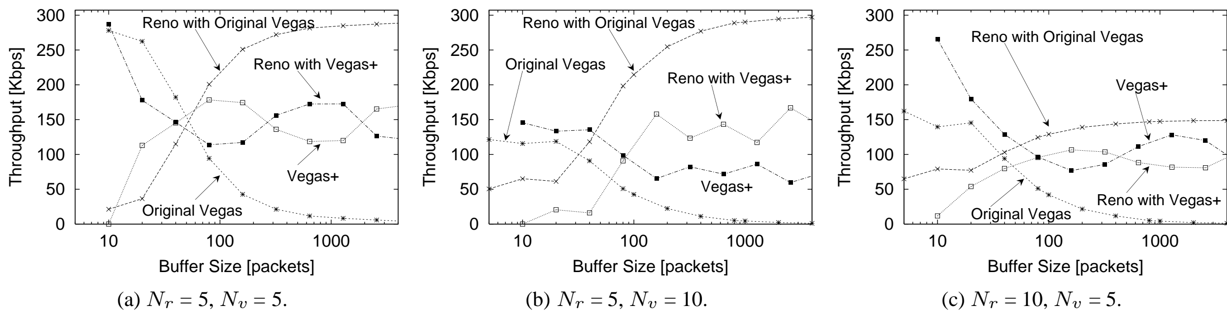
(a) TCP Vegas (b) TCP Vegas+

Figure 7: Changes of Window Size



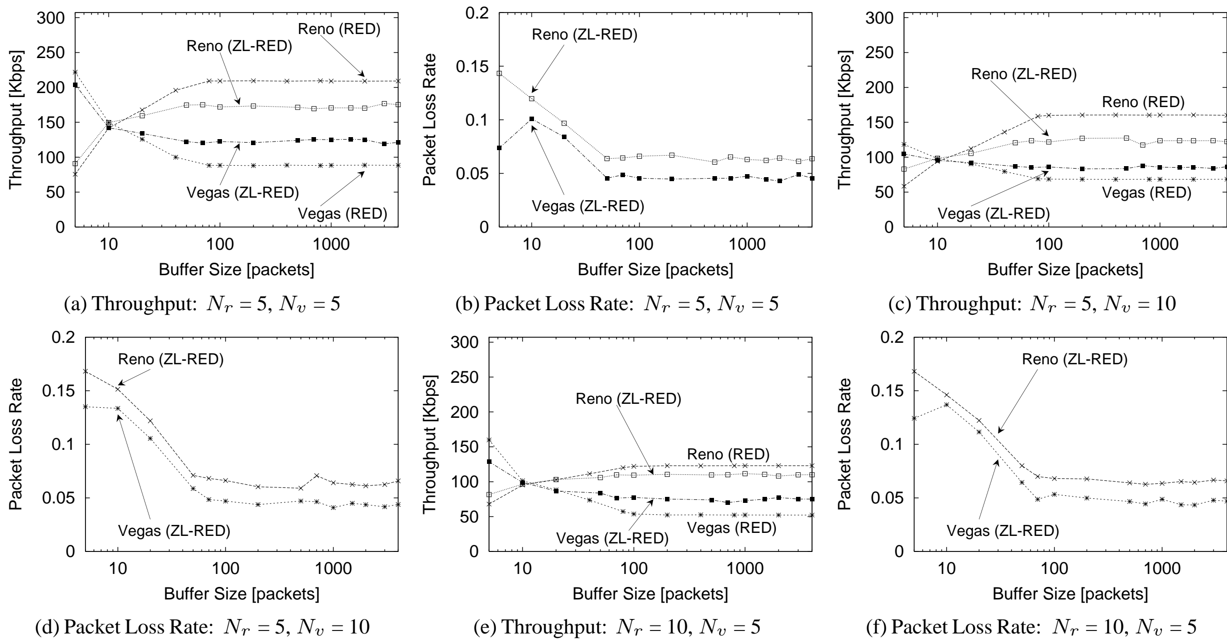
(a) TCP Vegas (b) TCP Vegas+

Figure 8: Changes of Instantaneous Throughput



(a) $N_r = 5$, $N_v = 5$. (b) $N_r = 5$, $N_v = 10$. (c) $N_r = 10$, $N_v = 5$.

Figure 9: Fairness Evaluation between TCP Reno and Vegas+: $count_{max} = 8$



(a) Throughput: $N_r = 5$, $N_v = 5$ (b) Packet Loss Rate: $N_r = 5$, $N_v = 5$ (c) Throughput: $N_r = 5$, $N_v = 10$

(d) Packet Loss Rate: $N_r = 5$, $N_v = 10$ (e) Throughput: $N_r = 10$, $N_v = 5$ (f) Packet Loss Rate: $N_r = 10$, $N_v = 5$

Figure 10: Fairness Evaluation under ZL-RED Algorithm: $hit_{drop} = 0.8$