Computer Communications 33 (2010) 1013–1029

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

CODE TCP: A competitive delay-based TCP

Yi-Cheng Chan [a,*], Chia-Liang Lin [a], Chia-Tai Chan [b], Cheng-Yuan Ho [c]

[a] Department of Computer Science and Information Engineering, National Changhua University of Education, No. 2, Shi-Da Road, Changhua City 500, Taiwan
[b] Institute of Biomedical Engineering, National Yang-Ming University, No. 155, Sec. 2, Linong Street, Taipei City 112, Taiwan
[c] Department of Computer Science, National Chiao Tung University, No. 1001 Ta Hsueh Road, Hsinchu City 300, Taiwan

ARTICLE INFO

Article history:
Received 19 September 2008
Received in revised form 9 January 2010
Accepted 11 January 2010
Available online 18 January 2010

Keywords:
Congestion control
TCP Reno
TCP Vegas
Fairness
Transport layer protocol

ABSTRACT

TCP Vegas is a well-known delay-based congestion control mechanism. Studies have indicated that TCP Vegas outperforms TCP Reno in many aspects. However, Reno currently remains the most widely deployed TCP variant in the Internet. This is mainly because of the incompatibility of Vegas with Reno. The performance of Vegas is generally mediocre in environments where it coexists with Reno. Hence, there exists no incentive for operating systems to adopt Vegas as the default transport layer protocol. In this study, we propose a new variant of Vegas called COmpetitive DElay-based TCP (CODE TCP). This variant is compatible with Reno and it can obtain a fair share of network resources. CODE is a sender-sided modification and hence it can be implemented solely at the end host. Simulations and experiments confirm that CODE has better fairness characteristics in network environments in which it coexists with Reno while retaining the good features of Vegas.

1. Introduction

Transmission Control Protocol (TCP) is a connection-oriented, end-to-end, and reliable protocol. Nowadays, a majority of Internet traffic is carried by TCP. Therefore, the behavior of TCP is tightly coupled with the overall Internet performance. To improve network efficiency, many TCP variants have been proposed. Two of these variants are noteworthy. One is Reno [1], which has been widely deployed in the Internet; the other is Vegas [2], which claims to have a throughput that is 37–71% greater than that of Reno.

TCP Vegas is a delay-based congestion control mechanism. Unlike TCP Reno which uses a binary congestion signal, packet loss, to adjust its window size, Vegas adopts a more fine-grained signal, queuing delay, to avoid congestion. Vegas can detect network congestion in the early stage and successfully prevent periodic packet loss that usually occurs in Reno. Delay-based congestion control schemes have attracted considerable attention because of their innovative control policy [3–10].

As compared to Reno, Vegas generally performs better with respect to overall network utilization [2], stability [11,12], fairness [11,12], throughput, packet loss [2], and burstiness [13] in homogeneous environments. Vegas also outperforms TCP Newreno [14]. However, studies have shown that when Reno and Vegas connections coexist in the same network, Reno obtains a greater amount of bandwidth as compared to Vegas [12,15,16]. Consequently, although Vegas has been available for a few years, it has not been adopted widely because of its perceived incompatibility with Reno.

To deal with the fairness problem, we propose a new mechanism called COmpetitive DElay-based TCP (CODE TCP), which is a variant of Vegas. Most of the operations in CODE TCP are similar to those in Vegas except that the two thresholds α and β are adaptive to the state of the network. When CODE senses the occurrence of network congestion and it does not consider itself to be responsible for the congestion, it begins to increase α and β instead of reducing its own rate. This makes CODE behave more similarly to Reno. Therefore, if the competing source is Reno, CODE reacts against its aggressiveness and its performance does not decrease. Conversely, if the competing source is Vegas, after a transition period, CODE recovers to a stable status as a Vegas source by reducing α and β. Changing the values of α and β instead of directly altering the value of its congestion window (*CWND*) should allow CODE to preserve the properties of Vegas in reaching an operating point.

The remainder of this paper is organized as follows. We describe the related work in Section 2. Section 3 presents a detailed description of the proposed mechanism, CODE TCP. Section 4 presents and discusses the results of both NS-2 simulations and experiments on a Linux platform. Finally, the conclusions are presented in Section 5.

2. Related work

To ensure network efficiency, TCP controls its sending rate based on feedback from the network. In order to control the

* Corresponding author. Tel.: +886 4 7232105x7044; fax: +886 4 7211284.
E-mail addresses: ycchan@cc.ncue.edu.tw (Y.-C. Chan), 94612005@mail.ncue.edu.tw (C.-L. Lin), ctchan@bme.ym.edu.tw (C.-T. Chan), cyho@csie.nctu.edu.tw (C.-Y. Ho).

0140-3664/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.comcom.2010.01.007

sending rate, TCP estimates the available network bandwidth via a bandwidth estimation scheme. In TCP Reno, packet losses are used to detect network congestion while in TCP Vegas, the queuing delay is used to estimate the network condition. In this section, we present a summary of these two congestion control mechanisms and explain why they are incompatible. Then two algorithms, NewVegas [17] and Vegas-A [18], that attempt to solve the fairness problem between Vegas and Reno are described.

2.1. TCP reno

TCP Reno uses a congestion window (*CWND*) to control the amount of data transmitted in a round-trip Time (*RTT*) and a maximum window (*MWND*) that is set by the receiver to limit the maximum value of *CWND*. The congestion control scheme of Reno can be divided into three phases: slow-start, congestion avoidance, and fast retransmission and fast recovery. In the interest of conciseness, we only describe the congestion avoidance phase since it is most closely related to our work. The descriptions of the other two phases can be found in [1].

Since the window size in the slow-start phase expands exponentially, packets sent at this increasing speed would quickly lead to network congestion. To avoid this, the congestion avoidance phase begins when *CWND* exceeds a preset slow-start threshold (*ssthresh*). In this phase, *CWND* is incremented by $1/CWND$ packet every time an *ACK* is received in order to make *CWND* grow linearly. This process continues until a packet loss is detected; subsequently the scheme switches to the fast retransmission and fast recovery phase.

2.2. TCP vegas

TCP Vegas adopts a more sophisticated bandwidth estimation scheme that attempts to avoid congestion rather than react to it. It uses the measured *RTT* to accurately calculate the number of data packets that a source can send. Vegas features three improvements as compared to TCP Reno: (1) a modified slow-start mechanism, (2) an improved congestion avoidance mechanism, and (3) a new retransmission mechanism. Its window adjustment algorithm also consists of three phases. The *CWND* is updated based on the currently executing phase. Fig. 1 shows the state transition diagram of TCP Vegas. A connection begins with the slow-start phase. The window-adjustment phase transition is attributable to specific events, as depicted along the edges.

2.2.1. Slow-start

During the slow-start phase, Vegas allows a connection to quickly ramp up to the available bandwidth. However, in order to detect and avoid congestion during this phase, Vegas doubles its *CWND* only every other *RTT*. In between, the *CWND* remains fixed so that a valid comparison of the *Expected* and *Actual* sending rates can be made. Vegas estimates a suitable amount of extra data to be maintained in the network pipe and controls the *CWND* accordingly. It records *RTTs* and sets the *BaseRTT* to the minimum *RTT* value measured. The amount of extra data (Δ) is estimated as follows:

$$\Delta = (Expected - Actual) \times BaseRTT, \tag{1}$$

where *Expected* rate is the current *CWND* size divided by the *BaseRTT*, and *Actual* rate is the *CWND* divided by the newly measured smoothed *RTT*.

This detection mechanism is applied during the slow-start phase to decide when to switch the phase. If the estimated amount of extra data is greater than the threshold γ, Vegas reduces its *CWND* by one-eighth and transitions from the slow-start phase to the congestion avoidance phase.

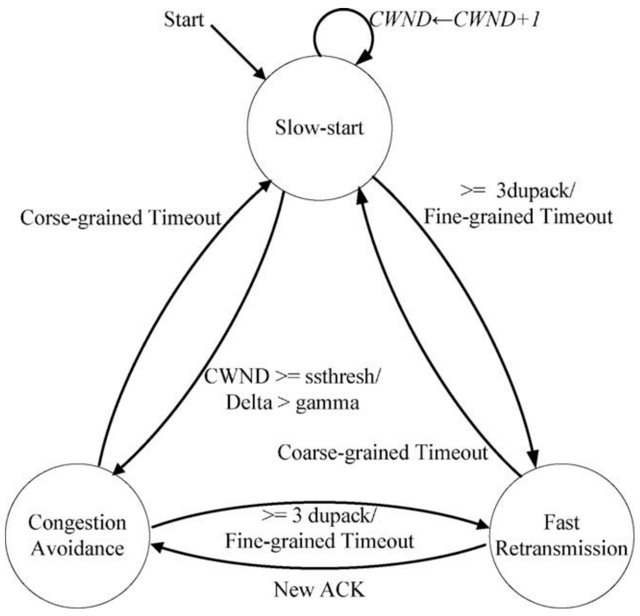


Fig. 1. State transition diagram of Vegas.

2.2.2. Congestion avoidance

During the congestion avoidance phase, Vegas does not continually increase the *CWND*. Instead, it attempts to detect incipient congestion by comparing the *Actual* rate with the *Expected* rate. The *CWND* is kept constant when Δ lies between two thresholds α and β. If Δ is greater than β, it is considered to indicate incipient congestion and thus the *CWND* is reduced. On the other hand, if Δ is lesser than α, the connection may be underutilizing the available bandwidth and thus the *CWND* is increased. *CWND* is updated on a per-*RTT* basis. The rule for updating *CWND* can be expressed as follows:

$$CWND = \begin{cases} CWND + 1, & \text{if } \Delta < \alpha \\ CWND - 1, & \text{if } \Delta > \beta \\ CWND, & \text{if } \alpha \leqslant \Delta \leqslant \beta \end{cases}. \tag{2}$$

2.2.3. Fast retransmission and fast recovery

TCP Vegas measures the *RTT* for every packet sent based on fine-grained timer values. By using fine-grained *RTT* measurements, a timeout period is computed for each packet. When a duplicate *ACK* is received, Vegas checks whether or not the timeout period of the oldest unacknowledged packet has expired. If it has, the packet is retransmitted. This modification leads to packet retransmission after just one or two duplicate *ACKs*. When a non-duplicate *ACK* that is the first or second *ACK* after a fast retransmission is received, Vegas again checks for the expiration of the oldest unacknowledged packet following which it may retransmit another packet.

After a packet retransmission is triggered by a duplicate *ACK* and the *ACK* of the lost packet is received, the *CWND* will be reduced to alleviate the network congestion. Vegas sets the *CWND* in two cases. If a lost packet has been transmitted just once, the *CWND* will be three-fourth of the previous window size. Otherwise, it is considered to indicate more serious congestion, and *CWND* is set to one half of the previous window size. It should be noted that if multiple packet losses occur during one round-trip time and trigger more than one fast retransmission, the *CWND* will be reduced only for the first retransmission.

If a loss episode is sufficiently severe for no ACKs to be received, these triggering the fast retransmission algorithm, the losses will be eventually identified by a Reno-style coarse-grained timeout. When this occurs, *ssthresh* is set to one half of *CWND*, following which *CWND* is reset to two; finally, the connection restarts from the slow-start phase.

2.3. Incompatibility of TCP vegas with TCP reno

Some previous works have demonstrated that TCP Vegas outperforms other implementations of TCP in many cases [2]. However, when a Vegas connection competes with other connections that use TCP Reno, it does not receive a fair share of bandwidth due to its proactive congestion avoidance mechanism [12,20,21].

Reno continues to increase the window size until a packet is lost. Packet losses mainly occur due to buffer overflows. This bandwidth estimation mechanism results in a periodic oscillation of window size and buffer-filling behavior. Thus, while Vegas attempts to maintain a smaller queue length, Reno keeps many more packets in the buffer on average, thus stealing greater bandwidth.

The Reno congestion avoidance scheme is aggressive in that it leaves little room in the buffer for other connections, while Vegas is conservative and attempts to occupy little buffer space. When a Vegas connection shares a link with a Reno connection, the Reno connection uses most of the buffer space; the Vegas connection interprets this as a sign of network congestion and slows down. This is the main reason why Vegas is not deployed widely despite having many desirable properties.

Several schemes have been proposed to deal with this incompatibility problem. These can be divided into two categories. The first type sets α and β to optimized values based on an analysis and on characteristics of the network environment such as buffer size at the router and number of connections passing through the router [15,16,22–24]. However, such schemes could not be deployed widely in the Internet due to certain problems. First, a connection may pass through a number of routers and thus it cannot determine which one is the real bottleneck. Second, if no router provides useful information, such schemes cannot function properly. The second type attempt to solve the incompatibility problem by dynamically adjusting α and β based on feedback from the network [17,18]. Such schemes attempt to detect whether or not Reno connections exist in the network. If Reno connections exist, they will compete with Reno in a more aggressive manner. When Reno connections leave the network, they will behave in a manner similar to Vegas. Such schemes are easier to deploy because they do not require information from the router.

2.4. TCP NewVegas

The disadvantage of TCP Vegas in heterogeneous network scenarios is that when the available bandwidth is fully utilized, it does not increase its own *CWND* while TCP Reno does, as shown in Fig. 2. NewVegas [17] was originally designed to overcome the drawbacks of Vegas. The changes introduced in NewVegas are confined to the congestion avoidance mechanism. Initially, NewVegas sets the thresholds α and β to default values α_0 and β_0 (set to 1 and 3, respectively). Whenever the target *ACK* arrives at the source, NewVegas updates the thresholds as given by the following pseudocode, where *RTT* is the round-trip delay of the target packet and RTT_{old} is the previous one. *W* is the current *CWND* and W_{old} is the previous one.

```
if (receive_dupack)
    if (loss_event_is_true)
        fast_ retransmission_ and_ recovery
        $\alpha = \alpha_0$
        $\beta = \beta_0$
else
    if ($\Delta > \beta$)
        W=W-1
    elseif ($\Delta < \alpha$)
        W=W+1
    else
        W=W
    if ($RTT > RTT_{old}$ and $W \leqslant W_{old}$)
        $\alpha = \alpha + 1$
        $\beta = \beta + 1$
    if ($RTT \leqslant RTT_{old}$ and $\alpha > \alpha_0$)
        $\alpha = \alpha - 1$
        $\beta = \beta - 1$
```

The two thresholds are updated once every *RTT*. In the first condition, if the congestion is increasing (last *RTT* greater than the previous one) but NewVegas has not increased its *CWND*, it does not consider itself to be responsible for the network congestion. Then, the source is allowed to increase the number of packets it can keep in the bottleneck buffer by increasing the thresholds α and β. The thresholds α and β are both increased by 1 in order to follow the behavior of Reno in the congestion avoidance phase and thus insert an extra packet into the network.

On the other hand, in the second condition, if the *RTT* begins decreasing, it implies that either the other sources are also Vegas or some source has been switched off. Therefore, NewVegas can reduce its two thresholds.

When timeout expiration occurs or three duplicate *ACK*s are received, the two thresholds are set to the initial values α_0 and β_0 and NewVegas restarts from the congestion avoidance phase.

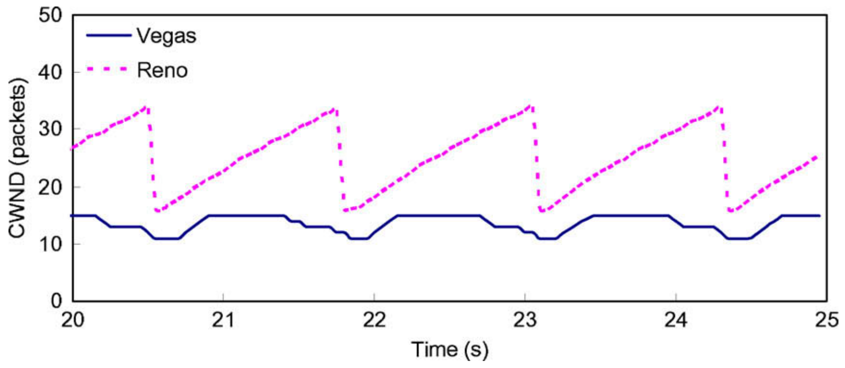


Fig. 2. Changes in *CWND* of Reno and Vegas for the same bottleneck.

2.5. TCP Vegas-A

TCP Vegas uses fixed values of 1 and 3 for thresholds α and β, respectively. The strategy of Vegas is to adjust the *CWND* to keep a small number of packets buffered at the router. The average number of packets buffered at the router is to be maintained within α and β. The main idea behind Vegas-A is that rather than fixing α and β, they can be made dynamically adjustable and more adaptive. At the start of a connection, α and β are set to 1 and 3, respectively. These values are then changed dynamically depending on the network conditions in an attempt to probe the network capacity. The detailed operation and pseudocode of Vegas-A is given below.

```
if ($\beta > \Delta > \alpha$)
    if ($Th_t > Th_{t-rtt}$)
        $W = W + 1$
        $\alpha = \alpha + 1$
        $\beta = \beta + 1$
    else
        $W = W$
elseif ($\Delta < \alpha$)
    if ($\alpha > 1$ and $Th_t > Th_{t-rtt}$)
        $W = W + 1$
    elseif ($\alpha > 1$ and $Th_t < Th_{t-rtt}$)
        $W = W - 1$
        $\alpha = \alpha - 1$
        $\beta = \beta - 1$
    elseif ($\alpha = 1$)
        $W = W + 1$
elseif ($\Delta > \beta$)
    if ($\alpha > 1$)
        $\alpha = \alpha - 1$
        $\beta = \beta - 1$
    $W = W - 1$
else
    $W = W$
```

The notation Th_t is the actual throughput rate at time t and Th_{t-rtt} is the actual throughput rate measured one *RTT* before t. The operation of Vegas-A appears reasonable and it can dynamically adjust its thresholds α and β. However, when Vegas-A and Reno share the same bottleneck, this may not always be the case. Consider the following case when one Vegas-A connection and one Reno connection share the same bottleneck. Assume that Δ of Vegas-A lies between α and β. If Reno keeps increasing its *CWND* every *RTT*, the throughput probed by Vegas-A decreases and thus it will not increase its thresholds α and β. Therefore, the performance of Vegas-A is limited by its α and β update algorithm. The performance of Vegas-A will be further examined in Section 4.

3. CODE TCP

The proposed algorithm, CODE TCP, is inspired by the idea of NewVegas. In this section, we first discuss the drawbacks of NewVegas and then describe the operation of CODE TCP. The parameter analysis is described in the last subsection.

3.1. Drawbacks of NewVegas

TCP NewVegas appears to be compatible with TCP Reno. However, the simulation results shown in Fig. 3 indicate otherwise. It is observed that NewVegas sends less packets as compared to Reno in a repetition cycle (period between two packet loss episodes), and therefore, the throughput of NewVegas is lower than that of Reno in this case. There are two drawbacks that lead to NewVegas having a lower throughput.

First, at the *i*th second, both NewVegas and Reno detected a packet loss. After a packet loss, NewVegas sets α and β to the default values, that is, α_0 and β_0. Consider the following case: since NewVegas will adjust its thresholds α and β, assume that the maximum values of α and β of NewVegas are 15 and 17 before a packet loss and α_0 and β_0 are 1 and 3, respectively. In this case, Δ will range from 15 to 17; assume that Δ is 16. After the packet loss, α and β were immediately set to the default values 1 and 3, respectively. This will lead to *CWND* being decreased by 1 packet every *RTT* until Δ lies between α and β while Reno will increase its *CWND* 1 packet every *RTT*. This situation occurs during the period between the *i*th and *j*th seconds in Fig. 3. This first drawback prevents NewVegas from competing with TCP Reno.

The second drawback of NewVegas is that every time it increases the thresholds α and β, it requires another *RTT* to increase its *CWND*. It is impossible to increase *CWND* and the two thresholds α and β in the same *RTT*. As shown in Fig. 4., which shows a flowchart of the operation of NewVegas, if α and β are increased by one in the current *RTT*, *CWND* can be increased by one in the next *RTT*. Thus, $W \leqslant W_{old}$ would not hold at the next *RTT*, and α and β will not be increased. Therefore, NewVegas cannot increase its *CWND* by 1 packet every *RTT* while Reno can.

For example, consider the following case shown in Table 1. Assume Δ is 5 while α and β are 5 and 7, respectively, in RTT_j. If α and β are increased by one in RTT_j, they would be 6 and 8 in RTT_{j+1}. At RTT_{j+1}, since W remains the same before the updating algorithm, and assuming that the variation of *RTT* is relatively small, Δ would remain 5. After the updating algorithm in the congestion avoidance phase at RTT_{j+1}, W would be increased by one so that Δ could be kept between α and β. From Fig. 4., $W \leqslant W_{old}$ will not hold at RTT_{j+1} because W is increased by one and α and β will remain the same. Thus, W will not change at RTT_{j+2}. This is the second drawback of NewVegas that prevents it from compet-

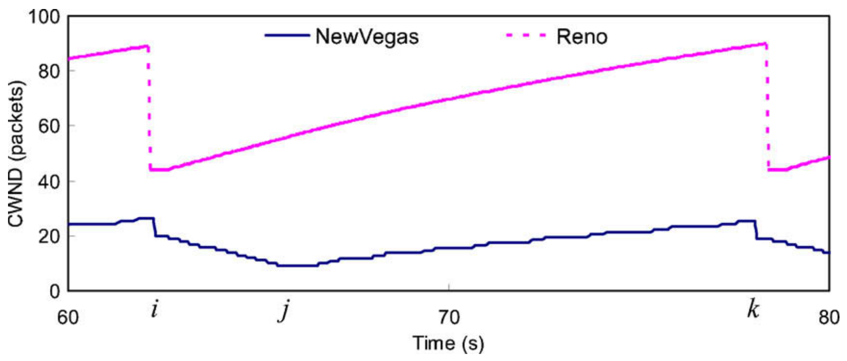


Fig. 3. Changes in *CWND* of NewVegas and Reno for the same bottleneck.

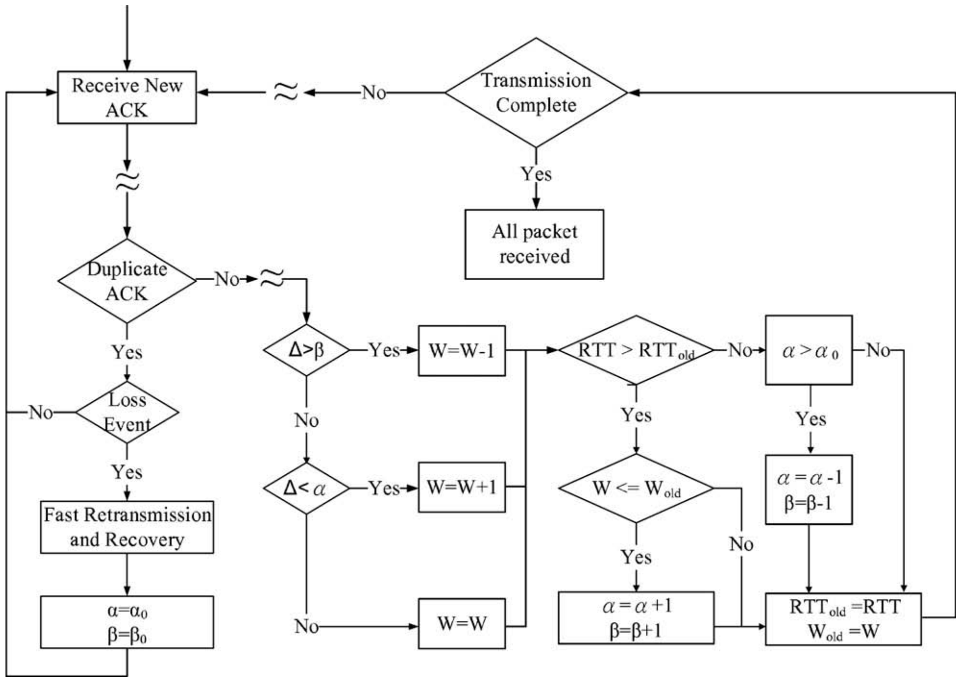


Fig. 4. Flowchart of NewVegas operation.

Table 1
Evolution of the parameters of NewVegas.

RTT	RTTj	RTTj+1	RTTj+2	RTTj+3	RTTj+4	RTTj+5
Wold	20	20	21	21	22	22
$\boldsymbol{\alpha}$	5 + 1	6	6 + 1	7	7 + 1	8
$\boldsymbol{\beta}$	7 + 1	8	8 + 1	9	9 + 1	10
$\boldsymbol{\Delta}$	5	5	6	6	7	7
W	20	21	21	22	22	23

ing with Reno. This situation occurs during the time interval between the *j*th and *k*th seconds in Fig. 3. It is observed that the *CWND* of Reno increases at a greater rate than that of NewVegas during this period.

3.2. Operation of CODE TCP

Since we know the drawbacks of NewVegas when competing with Reno, we aim to revise NewVegas such that it can compete with Reno while maintaining the good characteristics of Vegas. This improved variant is called COmpetitive DElay-based TCP, or CODE TCP. The main difference between CODE and NewVegas is that when a packet loss is detected by three duplicate ACKs, the two thresholds α and β will be multiplied by the factor ϵ, while NewVegas directly sets α and β to the default values. ϵ is set to 0.68 based on a numerical analysis shown in the next subsection. This modification is used to deal with the first drawback of NewVegas. The second modification is that when CODE finds that a Reno connection coexists in the network, α and β are increased by two to mitigate the effect of the second drawback of NewVegas, and thus, CODE may increase 2 packets every three *RTTs*.

These two modifications will make CODE more compatible with Reno and enable a greater performance improvement as compared to NewVegas, as shown in Fig. 5. The operation of CODE is given by the following pseudocode.

```
if (receive_dupack)
    if (loss_event_is_true)
        fast_retransmission_and_recovery
        $\alpha = \epsilon \times \alpha$
        $\beta = \epsilon \times \beta$
else
    if ($\Delta > \beta$)
        W=W-1
    elseif ($\Delta < \alpha$)
        W=W + 1
    else
        W=W
    if ($RTT > RTT_{old}$ and $W \leqslant W_{old}$)
        $\alpha = \alpha + 2$
        $\beta = \beta + 2$
    if ($RTT \leqslant RTT_{old}$ and $\alpha > \alpha_0$)
        $\alpha = \alpha - 1$
        $\beta = \beta - 1$
```

3.3. Numerical analysis of ϵ

In this section, we present a numerical analysis of the parameter ϵ. Throughout our analysis, we adopt a dumbbell network topology. We assume a fluid model and that the sources always have packets to transmit. There is no congestion in the *ACK* path, and therefore, the effect of *ACK* compression is negligible. One CODE connection and one Reno connection share the bottleneck. Both connections have the same round-trip delay.

We try to model CODE's and Reno's *CWND* with respect to a "round," or equivalently, a "window transmission." A round begins with the transmission of *W* packets (back-to-back), where *W* is the size of the congestion window in that round. A round ends when the source receives the *ACK* of the first packet in that round, and then, the source starts sending a new packet of the next round. For the sake of simplicity, we do not consider the fast retransmission and recovery phase, as shown in Fig. 6. In addition, we also assume that Δ is a function of *CWND* [15] so that it would be affected by the value of thresholds α and β.

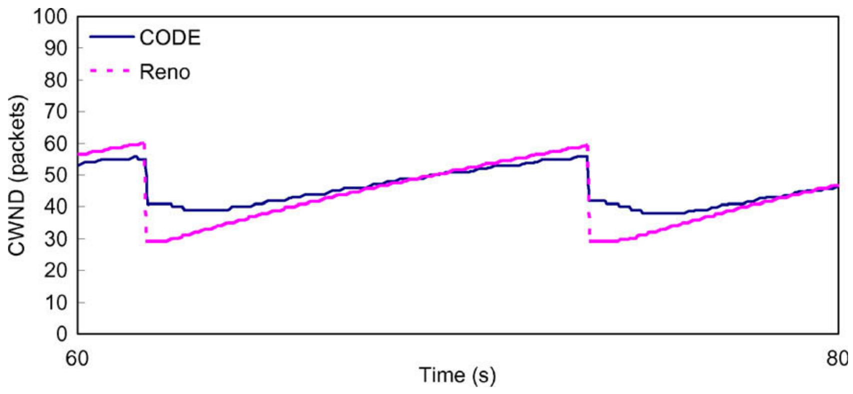


Fig. 5. Changes in *CWND* of CODE and Reno for the same bottleneck.

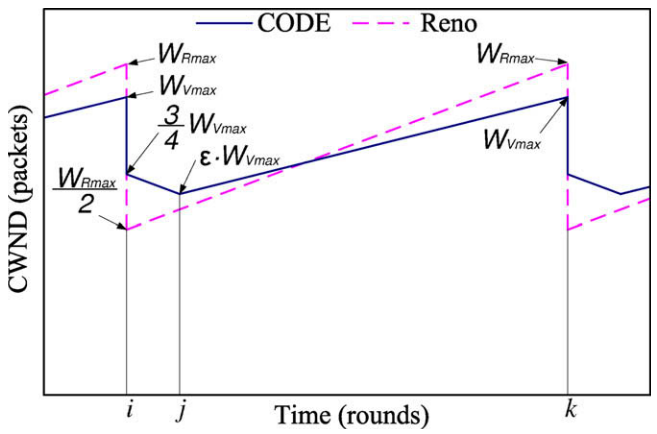


Fig. 6. Analysis of *CWND* of CODE and Reno for the same bottleneck.

From Fig. 6, we know that the ratio of CODE's throughput (λ_V) and Reno's throughput (λ_R) can be expressed as the ratio of the packets sent during a cycle. If we want CODE to get a fair share with Reno, we can simply set the ratio to 1, implying that CODE and Reno send the same number of packets during a cycle and have the same throughput, that is

$$\frac{\lambda_V}{\lambda_R} = \frac{\sum_{n=i}^{k} W_{Vn}}{\sum_{n=i}^{k} W_{Rn}} = 1, \tag{3}$$

where W_{Vn} and W_{Rn} are CODE and Reno's window size in the *n*th round. As shown in Fig. 6, if we want CODE to get a fair share with Reno, the area of CODE and Reno during a cycle (*i*th round to *k*th round) should be the same. For the sack of simplicity, we focus on how to set the parameter ϵ so that CODE and Reno can have the same area during a cycle. After a packet loss in NewVegas, the thresholds α and β are set to the default values; however in CODE, the thresholds are multiplied by the parameter ϵ, having a value greater than 0 and lesser than 1.

For CODE, the *CWND* after a packet loss will be reduced to three-forth of the largest window size before it, that is

$$\frac{3}{4} \cdot W_{Vmax}, \tag{4}$$

where W_{Vmax} is the maximum value of *CWND* before packet loss during a cycle. Assume that after *t* rounds, *CWND* reaches the minimum value, which is

$$\epsilon \cdot W_{Vmax} \tag{5}$$

at *j*th round.

Since the area of CODE during a cycle can be divided into two parts (*i*th round to *j*th round and *j*th round to *k*th round), we calculate the areas separately.

Assuming that there are *t* rounds between *i*th and *j*th rounds, the average *CWND* through *i* to *j* is

$$\frac{\frac{3}{4} \cdot W_{Vmax} + \epsilon \cdot W_{Vmax}}{2} = \left(\frac{3+4\epsilon}{8}\right) W_{Vmax}, \tag{6}$$

and therefore, the area between *i* and *j* is

$$\left(\frac{3+4\epsilon}{8}\right) W_{Vmax} \cdot t. \tag{7}$$

For each cycle, CODE and Reno will experience the same number of rounds. Since Reno adds 1 packet to its *CWND* every round, the number of rounds through *i* to *k* can be expressed as $\frac{W_{Rmax}}{2}$. W_{Rmax} is the maximum value of *CWND* before the packet loss during a cycle. For the remaining area of CODE, the average *CWND* through *j* to *k* is

$$\frac{\epsilon \cdot W_{Vmax} + W_{Vmax}}{2} = \left(\frac{\epsilon+1}{2}\right) W_{Vmax}. \tag{8}$$

The area between *j* and *k* is

$$\left(\frac{\epsilon+1}{2}\right) W_{Vmax} \cdot \left(\frac{W_{Rmax}}{2} - t\right). \tag{9}$$

Therefore, the packets sent by CODE in a cycle can be expressed as the area between *i* and *k* in Fig. 6. The total packets sent by CODE through *i*–*k* is

$$\left(\frac{3+4\epsilon}{8}\right) W_{Vmax} \cdot t + \left(\frac{\epsilon+1}{2}\right) W_{Vmax} \cdot \left(\frac{W_{Rmax}}{2} - t\right). \tag{10}$$

With regard to Reno, the case is much simpler. The rounds between *i* and *k* is $\frac{W_{Rmax}}{2}$ and the average *CWND* through *i*–*k* is

$$\frac{\frac{1}{2} W_{Rmax} + W_{Rmax}}{2} = \frac{3}{4} W_{Rmax}. \tag{11}$$

The total packets sent by Reno in a cycle is equal to the area through *i*–*k*, that is,

$$\frac{3}{4} W_{Rmax} \cdot \frac{W_{Rmax}}{2} = \frac{3}{8} W_{Rmax}^2. \tag{12}$$

Since we know that CODE will decrease its *CWND* by 1 in each round between *i* and *j*,

$$\frac{3}{4} \cdot W_{Vmax} = \epsilon \cdot W_{Vmax} + t. \tag{13}$$

Solving (13) for *t*, we obtain

$$t = \left(\frac{3}{4} - \epsilon\right) W_{Vmax}. \tag{14}$$

Using the second modification of CODE, *CWND* can increase by two every three *RTTs*, while in NewVegas, it can increase by one every other *RTT*; the slope of CODE between j and k is around two-thirds of that of Reno, and therefore, we assume that CODE will increase its *CWND* by two every three *RTTs* between j and k. Substituting (14) into (10), the total packets sent by CODE in a cycle is

$$\left(\frac{3+4\epsilon}{8}\right)W_{Vmax}\cdot\left(\frac{3}{4}-\epsilon\right)W_{Vmax}+\left(\frac{\epsilon+1}{2}\right)W_{Vmax}\cdot\frac{3}{2}(1-\epsilon)W_{Vmax}. \tag{15}$$

Using (3) and (12), the relation between CODE and Reno is

$$\left(\frac{3+4\epsilon}{8}\right)W_{Vmax}\cdot\left(\frac{3}{4}-\epsilon\right)W_{Vmax}+\left(\frac{\epsilon+1}{2}\right)W_{Vmax}\cdot\frac{3}{2}(1-\epsilon)W_{Vmax}=\frac{3}{8}W^2_{Rmax}. \tag{16}$$

Since we know the rounds of CODE and Reno during a cycle are the same,

$$\left(\frac{3}{4}-\epsilon\right)W_{Vmax}+\frac{3}{2}(1-\epsilon)W_{Vmax}=\frac{1}{2}W_{Rmax} \tag{17}$$

Substituting (17) into (16), we obtain

$$34\epsilon^2-54\epsilon+21=0. \tag{18}$$

Solving (18), we finally obtain

$$\epsilon\approx 0.68 \quad\text{or}\quad 0.91. \tag{19}$$

The value of 0.91 is unsuitable because it will make (14) unreasonable; the value of t cannot be negative. Since we assume that *CWND* is a function of Δ, when a packet loss occurs in CODE, it will not set thresholds α and β to the default values α_0 and β_0 directly but multiply them by 0.68 so that *CWND* can decrease to a suitable value and then begin increasing.

4. Performance evaluation

In this section, we present the simulation results obtained using Network Simulator (NS-2) [25] and the Linux platform to verify the effectiveness of CODE TCP. The network topologies used in the simulations are shown in Figs. 7 and 8. A set of real network experiments are also described at the end of this section.

In all of the following simulations, the TCP packet size is set to 1000 bytes, and the TCP congestion window is assumed to be not limited by the receiver window. Unless stated otherwise, we assume that the sources always have data to send (FTP source model).

4.1. One reno vs. one vegas/vegas-a/newvegas/CODE

First, we consider the network configuration shown in Fig. 7, in which one Reno connection competes with either one Vegas,

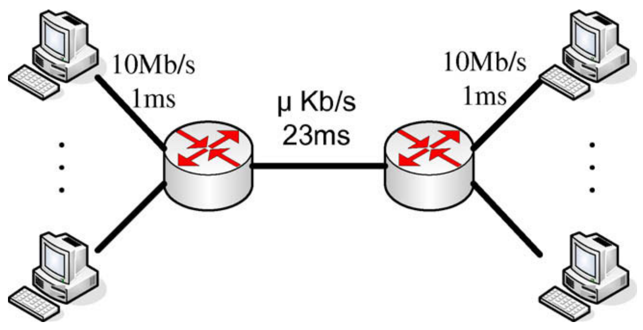


Fig. 7. Dumbbell network topology used in simulations.

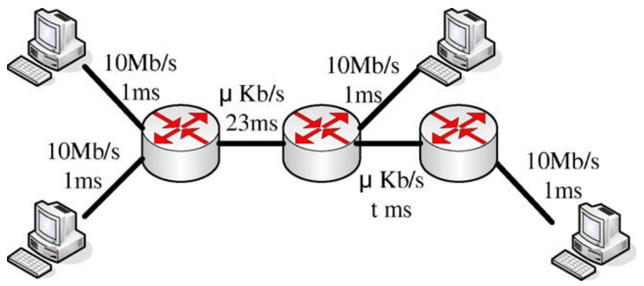


Fig. 8. Network topology used for connections with different *RTTs*.

Vegas-A, NewVegas, or CODE connection. Each simulation scenario is repeated 20 times by randomly changing the start time of the connections. The connection start time ranges from 0 to 25 s. Since the start times of two competing connections are probably different, we calculate the average throughput of each connection from second 50 to 200. In Fig. 9, we show the results of two connections competing in a single bottleneck by varying the bandwidth of the bottleneck link and setting the buffer size to 100 packets.

Since the two thresholds (α and β) of Vegas are fixed at 1 and 3, respectively, Vegas will keep a maximum of three packets at the bottleneck, while Reno can use the rest of the buffer size. Because of this, Vegas has a considerably small throughput as compared to that of Reno, as shown in Fig. 9(a).

The results of one Vegas-A connection competing with one Reno connection are shown in Fig. 9(b). Although Vegas-A can alter its two thresholds α and β, the throughput that it can achieve is still considerably smaller than that of Reno.

Fig. 9(c) shows the results for a NewVegas connection that is affected by the two drawbacks described in Section 3; again, the throughput is found to be smaller than that of Reno. However, since NewVegas adjusts its thresholds dynamically, it achieves a noticeable improvement in throughput as compared to Vegas. In the simulation results shown in Fig. 9(d), we observe that the throughputs of CODE and Reno are close to each other.

To further study the simulation results, we also calculate the confidence intervals of the throughput ratio of each set of two competing connections. The corresponding 95% confidence intervals (CI) can be found in each sub-figure. If the throughput ratio is 1, then it implies that the two competing connections share the bottleneck link fairly. From the 95% CI of each sub-figure, we can find that the throughput ratio of CODE/Reno is closer to 1 than that of the other combinations.

4.2. Connections with different RTTs

In this subsection, we describe simulations performed for connections with different *RTTs*. The network configuration is shown in Fig. 8. We fix the minimal round-trip delay of one connection and vary the minimal round-trip delay of the other connection to determine the behavior of connections with different *RTT*s. Each simulation scenario is also repeated 20 times by randomly changing the start time of connections. The average throughputs of each competing connection between 50 and 200 s are shown in Figs. 10–13.

Since Vegas is restricted to its fixed thresholds α and β, it can only queue a maximum of β packets at the bottleneck, thus inducing inefficiency when competing with Reno. Based on the results shown in Figs. 10(a) and (b), irrespective of whether the *RTT* of Vegas is larger or smaller than that of Reno, Vegas always has a considerably smaller throughput than Reno.

Fig. 11(a) and (b) show the results of Vegas-A competing with Reno with different *RTT*s. It is obvious that Vegas-A cannot obtain a fair share of the bottleneck bandwidth.

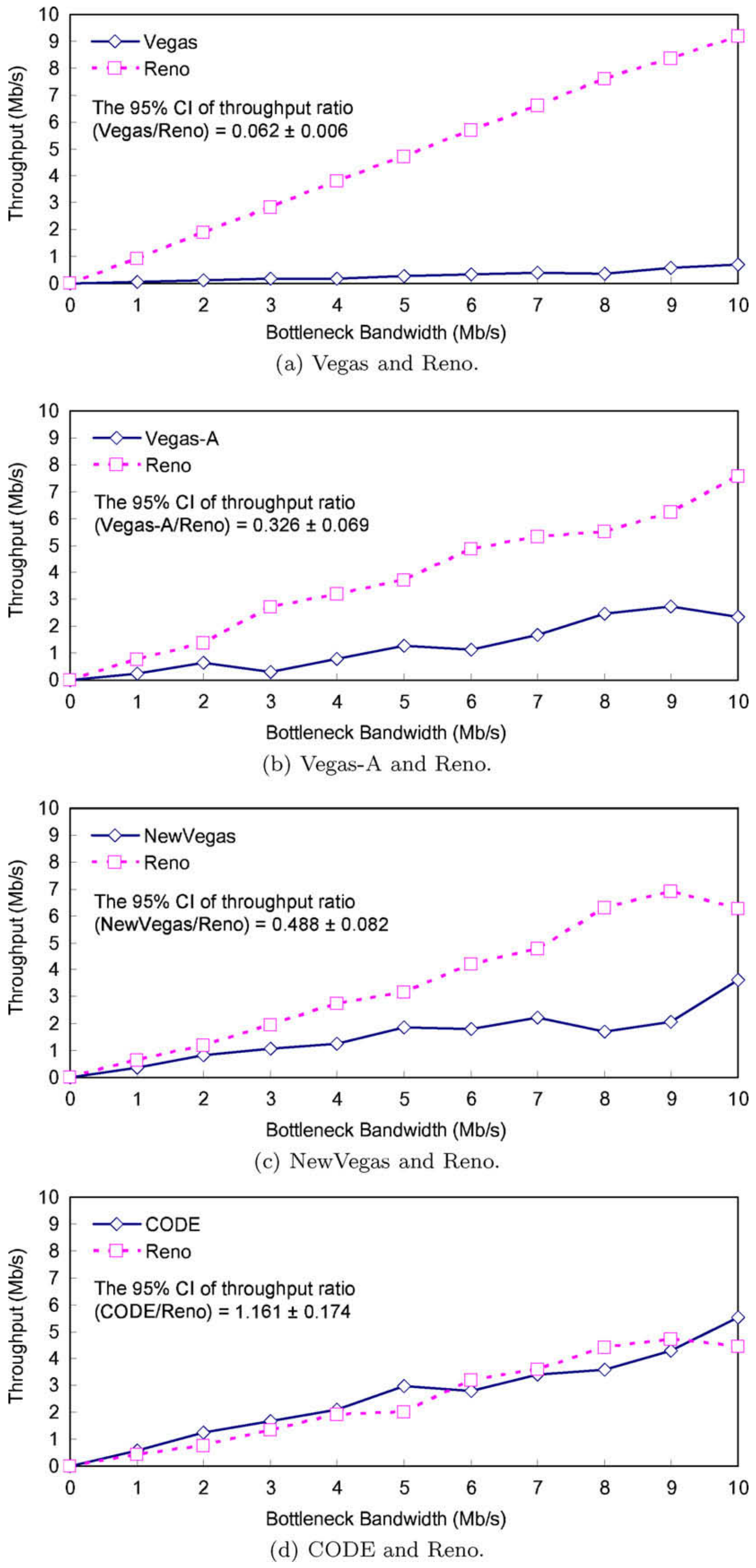
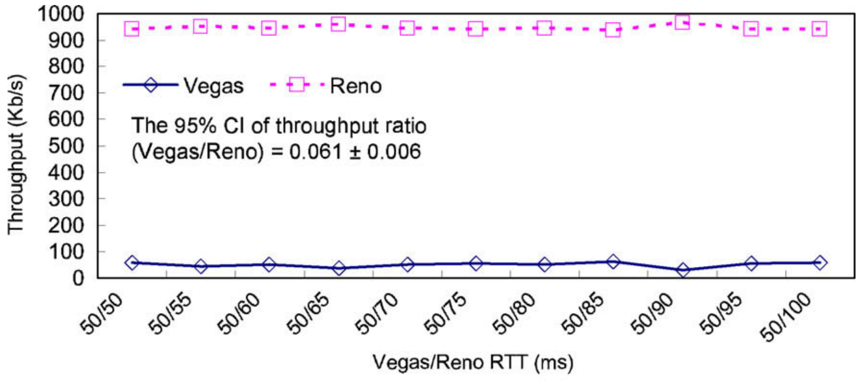
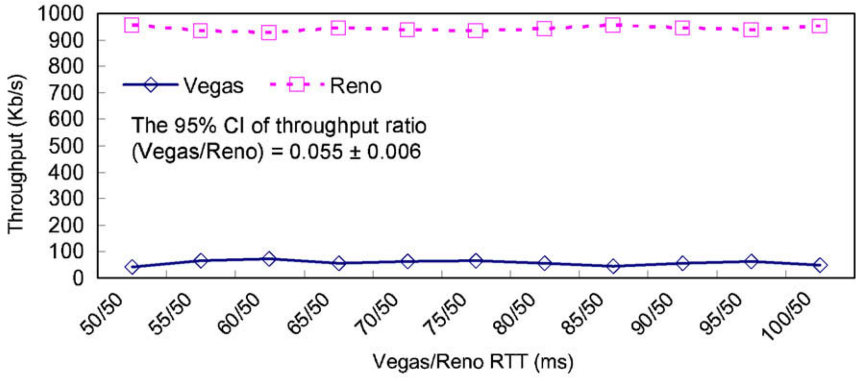


Fig. 9. Variation in the achieved throughput of two competing connections with the bottleneck bandwidth.

The results of NewVegas and Reno with different *RTT*s are shown in Fig. 12(a) and (b). Since the throughput of NewVegas remains restricted because of its drawbacks, the performance improvement is limited. Again, irrespective of whether the *RTT* of NewVegas is larger or smaller than that of Reno, NewVegas has a smaller throughput than Reno. It is notable that some fluctuations

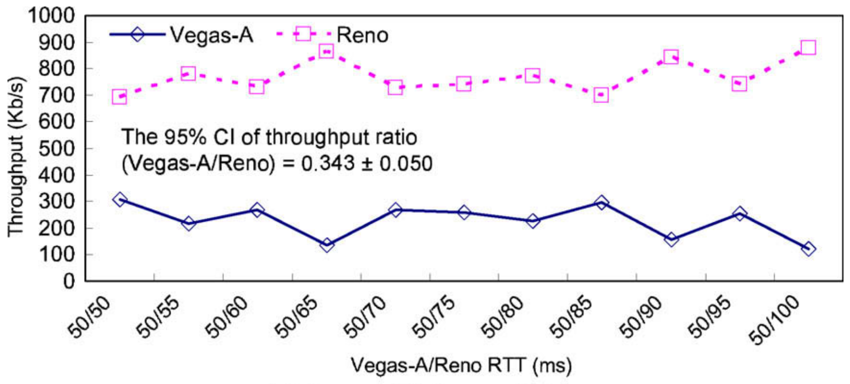


(a) Reno with larger RTT.

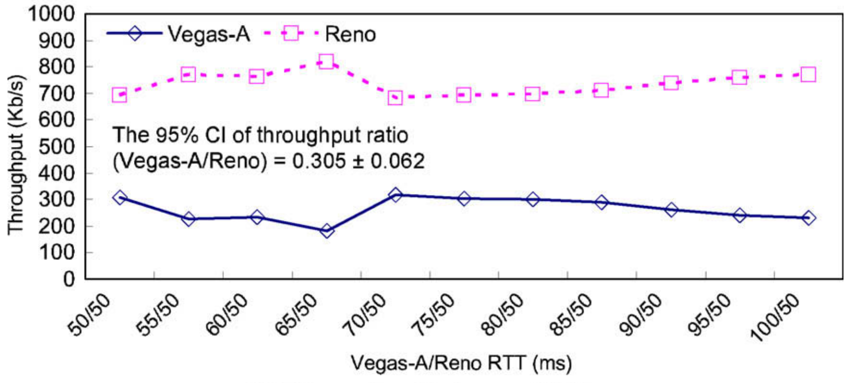


(b) Vegas with larger RTT.

Fig. 10. Achieved throughput of two competing connections, Vegas and Reno.

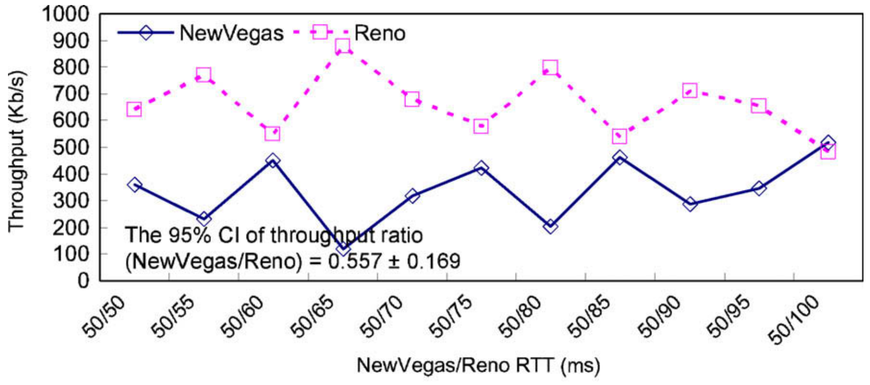


(a) Reno with larger RTT.

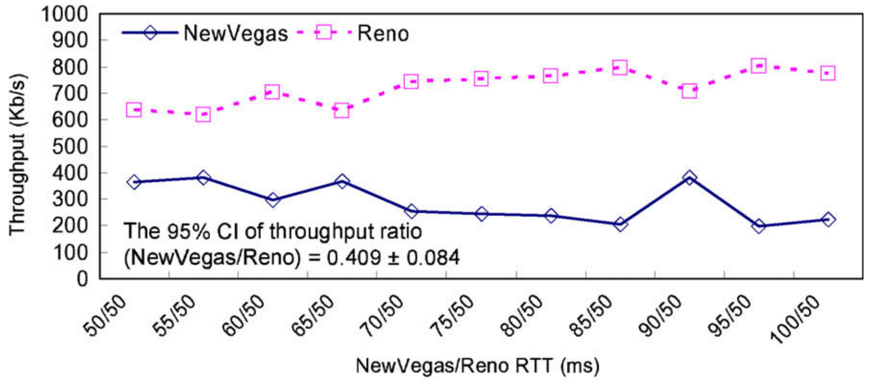


(b) Vegas-A with larger RTT.

Fig. 11. Achieved throughput of two competing connections, Vegas-A and Reno.

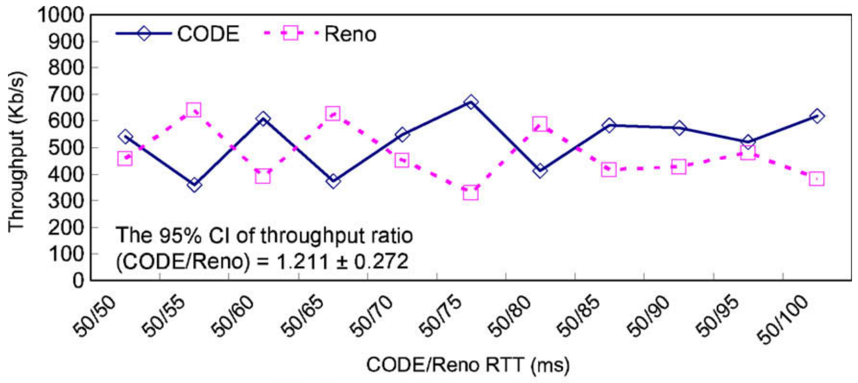


(a) Reno with larger RTT.

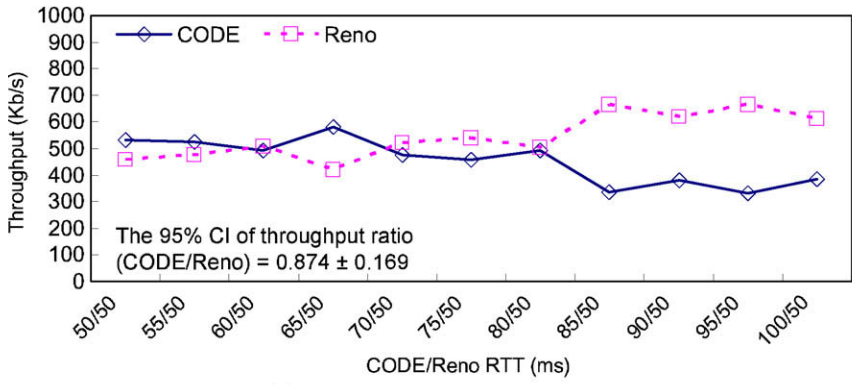


(b) NewVegas with larger RTT.

Fig. 12. Achieved throughput of two competing connections, NewVegas and Reno.

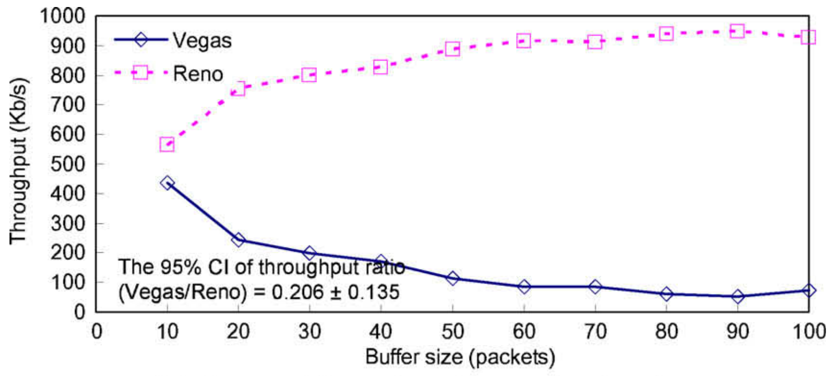


(a) Reno with larger RTT.

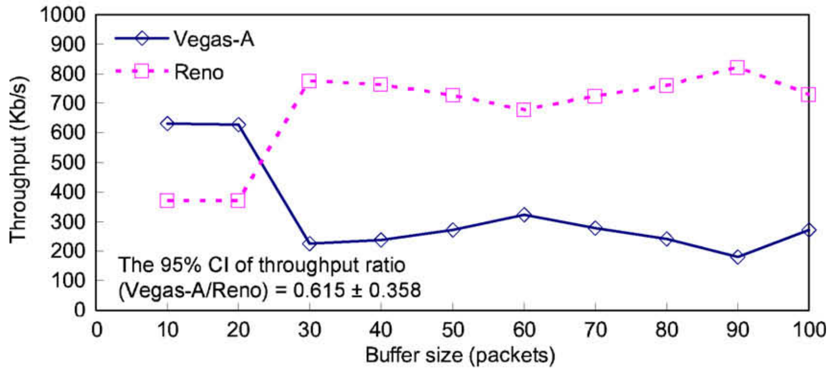


(b) CODE with larger RTT.

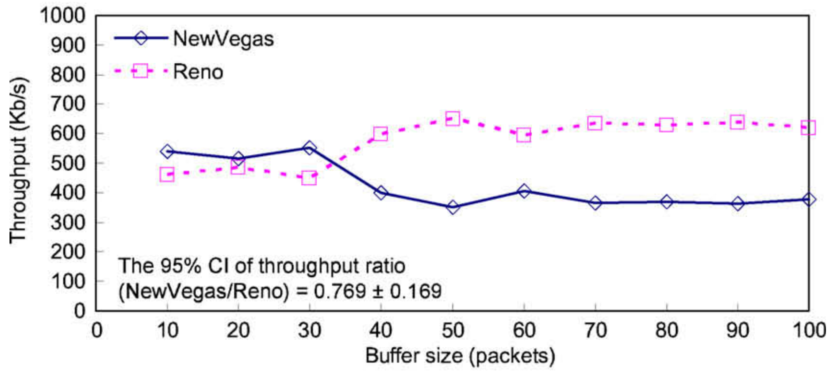
Fig. 13. Achieved throughput of two competing connections, CODE and Reno.



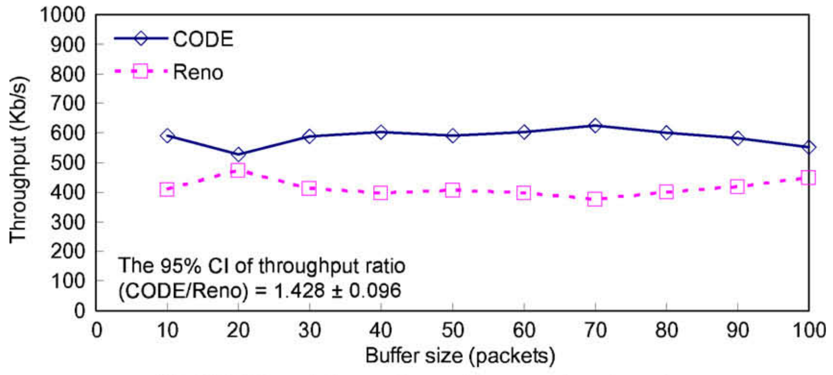
(a) Vegas and Reno share the same bottleneck.



(b) Vegas-A and Reno share the same bottleneck.



(c) NewVegas and Reno share the same bottleneck.



(d) CODE and Reno share the same bottleneck.

Fig. 14. Variation in the achieved throughput of two competing connections with different buffer sizes.

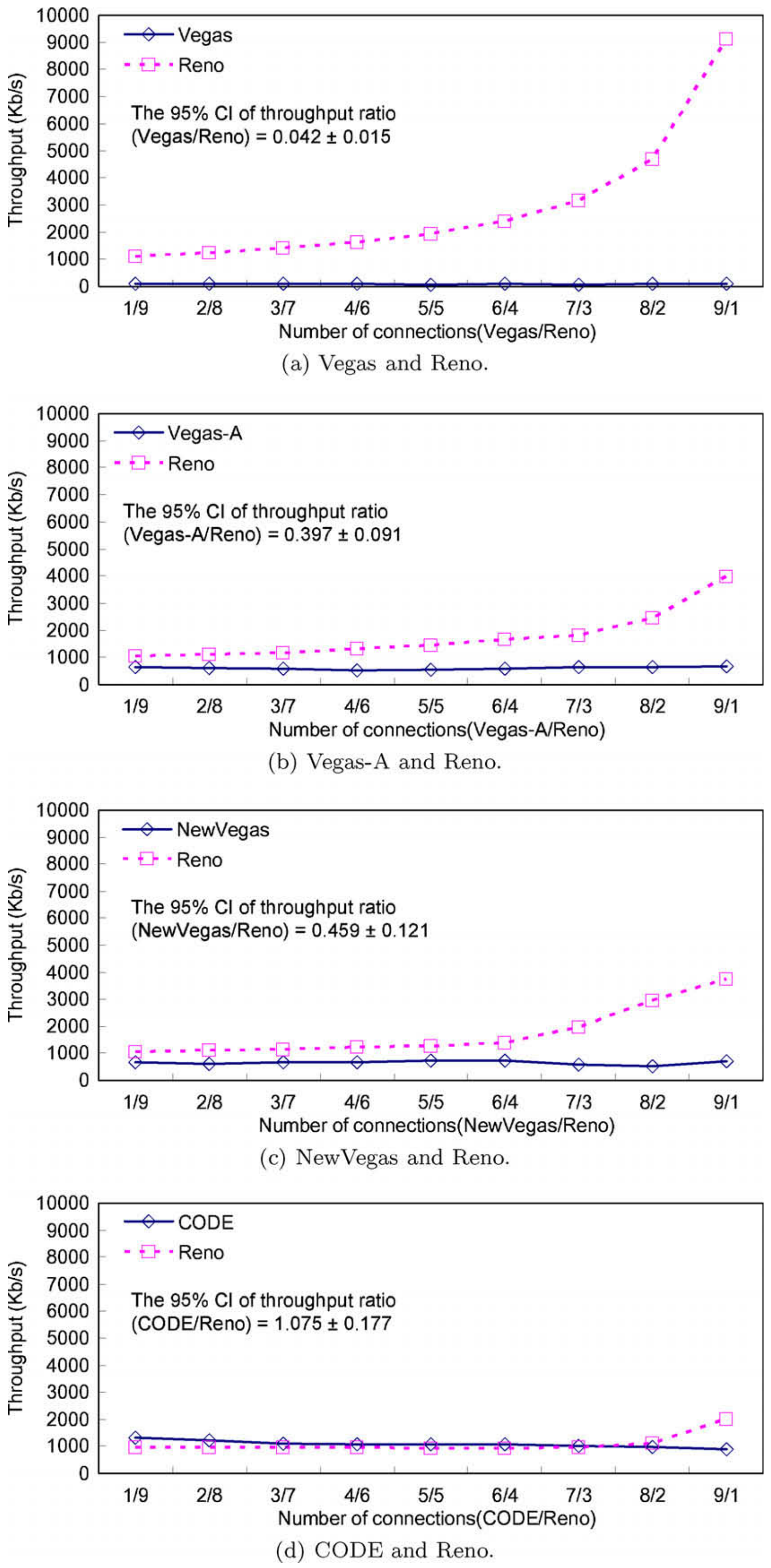


Fig. 15. Variation in the average throughput with the ratio of connection numbers of two TCP variants.

exist in the measured throughputs. Although each simulation scenario is repeated 20 times, the random start times of competing connections induce uncertainty.

The results of CODE and Reno competing with the same bottleneck with different *RTT*s are shown in Fig. 13(a) and (b). It is well known that when Reno connections compete at the same

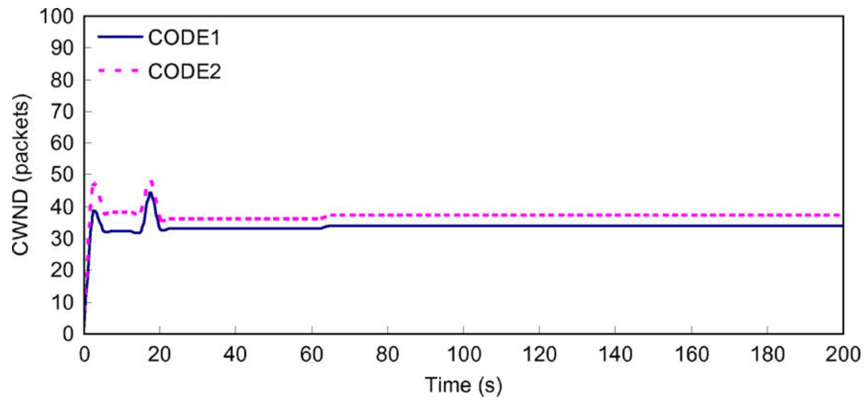


Fig. 16. Changes in the CWND of two CODE connections for the same bottleneck.

bottleneck, a connection with a small *RTT* would be more favorable than one with a large *RTT*. Fig. 13(a) indicates that CODE is preferable. However, the result shows that CODE does not steal too much bandwidth from Reno and only takes its fair share. In Fig. 13(b), the *RTT* of Reno is smaller than that of CODE. Although Reno's *RTT* is smaller and favorable, CODE can maintain a relative fair share with Reno as compared to the others.

The corresponding 95% confidence intervals of throughput ratio for the results in Figs. 10–13 are calculated and shown in each sub-figure. From the 95% CI of each sub-figure, we also find that CODE is more compatible with Reno as compared to the other Vegas variants.

4.3. Bottleneck with different buffer sizes

In this subsection, we test the fairness of two connections that share the same bottleneck under different buffer size settings. The network configuration is shown in Fig. 7. Each simulation scenario is repeated 20 times by randomly changing the start time of the connections. The average throughputs of each competing connection between 50 and 200 s are shown in Fig. 14.

Vegas uses fixed values of α and β, and therefore, it always attempts to keep the number of packets between α and β at the bottleneck buffer. The values of α and β are set at 1 and 3, respectively, while Reno uses the remainder of the bottleneck buffer. From the results shown in Fig. 14(a), it can be concluded that when the buffer size is small, Vegas may maintain a tolerable throughput. However, when the buffer size is increased, the throughput of Vegas worsens.

Vegas-A can alter the values of its two thresholds α and β. When the buffer size is small, Vegas-A outperforms Reno. However, Vegas-A cannot compete well with Reno when the buffer size is large. The results are shown in Fig. 14(b). It appears that the buffer size is a key factor influencing the performance of Vegas-A. It is also worth noting that when the buffer size is 60 packets, the throughputs of two competing connections are closer than in other circumstances. We infer that this may be caused by the random connection start times. The connection start time affects Vegas-A's performance when it competes with Reno.

From the results of NewVegas shown in Fig. 14(c), we can conclude that NewVegas outperforms Reno when the buffer size is small. However, as the buffer size increases, Reno is favored, and hence, it takes advantage of NewVegas. Therefore, Reno obtains more bandwidth from NewVegas. In Fig. 14(d), we observe that CODE achieves a higher throughput than Vegas-A or NewVegas.

The 95% confidence intervals of the throughput ratio for the results in Fig. 14 are calculated and shown in each sub-figure. It appears that none of the delay-based TCPs can fairly share the bottleneck with Reno under different buffer size settings. However, CODE maintains a more stable behavior and always achieves a higher throughput than Reno. Although this feature may encourage network users to adopt CODE TCP, the reasons of leading to this feature need to be further studied.

4.4. Varied ratio of connection numbers

In this subsection, we describe simulations of 10 connections sharing the same bottleneck. Some of the sources are Reno and the others are either Vegas, Vegas-A, NewVegas, or CODE. The network configuration is shown in Fig. 7. In Fig. 15, we show the average throughput of connections with a varying number of Reno connections to test the behavior of the other four Vegas variants. Each simulation scenario is also repeated 20 times by randomly changing the start time of the connections.

Vegas connections control their window sizes according to the observed *RTT*s and calculate the number of packets buffered at the bottleneck. Each connection attempts to keep the number of queued packets in the router buffer between α and β. As *RTT* increases because of the increasing queuing delay, Vegas connections continue to decrease their window sizes. On the other hand, Reno connections continue to increase their window sizes irrespective of the increasing *RTT*; because of this, the window sizes of the Vegas connections continue to decrease until a packet loss occurs.

Vegas-A also cannot compete well with Reno in these scenarios. Although NewVegas can dynamically adjust its thresholds α and β, as the queuing delay increases, its algorithm may not function instantaneously. Vegas, Vegas-A, and NewVegas cannot obtain their fair share even if the number of Reno connections is relatively small as compared to the number of Vegas/Vegas-A/NewVegas connections. The respective results are shown in Fig. 15(a)–(c). On the other hand, the result shown in Fig. 15(d) indicates that CODE achieves a reasonable fairness irrespective of the number of Reno connections. The same inference can also be obtained from the 95% confidence intervals of throughput ratio those are presented in each sub-figure.

4.5. Fairness feature in homogeneous scenarios

The simulations presented in this subsection are intended to verify whether or not CODE preserves a good fairness feature in a manner similar to NewVegas and Vegas when only the same variant sources exist in the network.

Fig. 16 shows the changes in the *CWND* of two CODE connections that share the same bottleneck. It is found that CODE preserves the good characteristics of Vegas and it can be stable in a homogeneous network environment. When only CODE sources

Table 2
Variation of fairness index value of 10 competing connections with the bottleneck bandwidth.

Bandwidth (Mb/s)	Code	NewVegas	Vegas
5	0.99	0.99	0.97
10	0.97	0.96	0.97
15	0.98	0.98	0.95
20	0.99	0.98	0.95
25	0.99	0.98	0.97
30	0.99	0.96	0.97
35	0.99	0.98	0.99
40	0.99	0.99	0.99
45	0.99	0.99	0.99
50	0.99	0.97	0.99

exist in the network, they behave like Vegas. Fig. 16 shows that a peak exists around second 20. This is because CODE uses the minimum of the *RTT*s measured within the last window of transmitted packets to update the thresholds and it is thus sensitive to network variations.

To evaluate the fairness among connections, we use the fairness index proposed in [26]. Given a set of throughputs $(x_1, x_2, \ldots, x_n)$, the fairness index of the set is defined as:

$$F = \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n \cdot \sum_{i=1}^{n} x_i^2}. \quad (20)$$

The value of the fairness index lies between 0 and 1. If the throughputs of all connections are the same, the index will be 1.

In this simulation, we adopt the network topology shown in Fig. 7, and the network environment is similar to that described in the last subsection except that the bandwidth is variable. Ten connections of the same TCP variant start at the same time and share the same bottleneck. From the simulation results shown in Table 2, we observe that CODE maintains the fairness feature and behaves in a similar manner to NewVegas and Vegas in a homogeneous scenario.

Fig. 17 presents the changes in the *CWND* of two TCP connections that share the same bottleneck (5 Mb/s). The new connection arrives at the bottleneck link that is occupied by the previously started connection in equilibrium. Both Vegas and CODE can achieve equilibrium and fairly share the bottleneck after a transient period. The transient period of CODE is longer than that of Vegas because CODE TCP needs to detect whether or not the new competition comes from Reno. The bottleneck queue status shown in Fig. 18 also reveals the same information. After a transient period, both Vegas and CODE maintain a stable bottleneck queue. It is notable that CODE attains a buffer utilization that is as low as that of Vegas.

4.6. Experiments on the Linux platform

In this subsection, we adopt Fedora 6 with kernel 2.6.18.8 to implement CODE TCP and TCP NewVegas. Since Linux kernel 2.6.18.8 already has a built-in Vegas module, the implementation is easier. In addition, FreeBSD uses ipfw as a packet filter and thus we can use it to emulate the router and set the bandwidth, delay, and queue length at the bottleneck. We also adopt

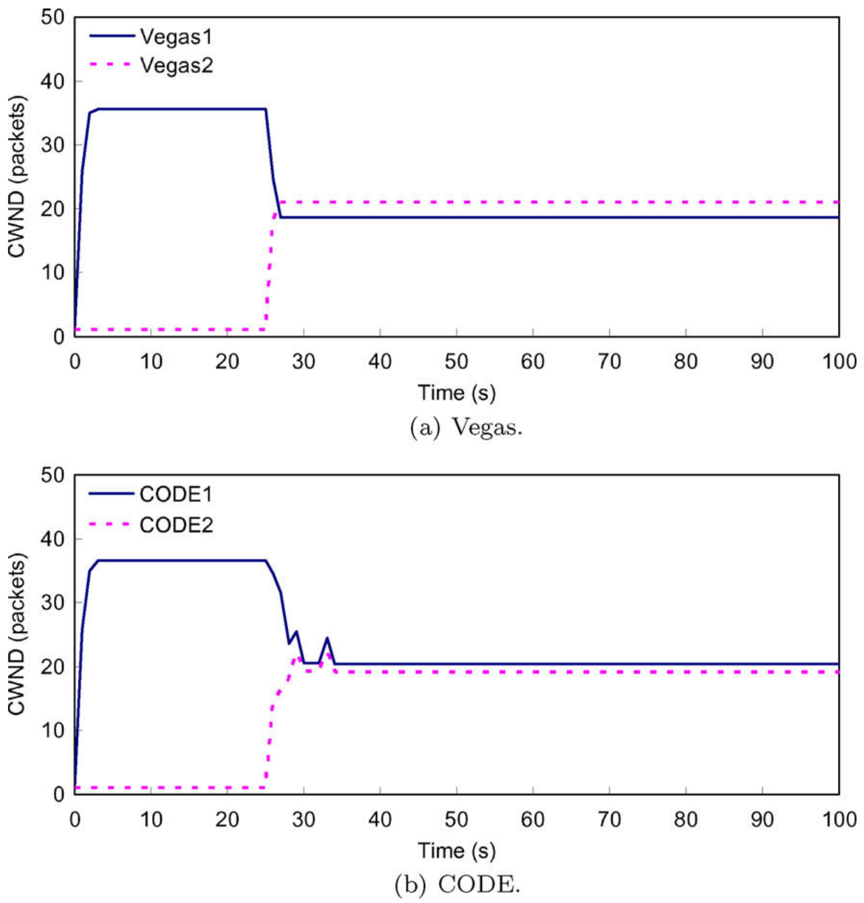


Fig. 17. Changes in the *CWND* of two TCP connections for the same bottleneck. A new connection arrives at the bottleneck link that is occupied by the previously started connection in equilibrium.

Iperf, a bandwidth measurement tool, to run the simulations. The network configuration for the experiments is shown in Fig. 7. One computer is set to be a server (sender) and the other is a client (receiver). A server and a client represent a traffic pair. Two traffic pairs share the same bottleneck and start at the same time.

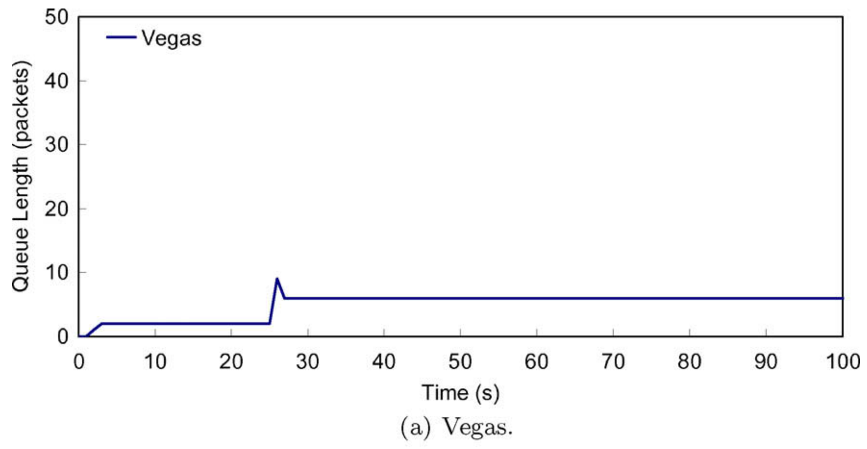
The experiment results are obtained by varying the bandwidth of the bottleneck link with a queue length of 100 packets and a link delay of 23 ms. Fig. 19(a)–(c) show the results of one Reno connection competing with a Vegas, NewVegas, and CODE connection, respectively.

The experimental results shown in Fig. 19 are similar to those of the NS-2 simulation in Section 4.1. In all the experiments, the throughput of CODE almost equals that of Reno, confirming the result of the numerical analysis and the good agreement with the NS-2 simulations. NewVegas was affected by the two drawbacks described in Section 3, and therefore, its throughput is small as compared to that of Reno and it is difficult to obtain a fair share with Reno. However, NewVegas can adjust it thresholds dynamically, and therefore, its throughput can be higher than that of Vegas. The thresholds of Vegas are fixed and therefore it will keep at most three packets at the bottleneck while Reno can use the rest of the buffer size. Therefore, Vegas has a much lower throughput as compared to that of Reno.

4.7. Results on the Internet

Since we have implemented CODE TCP on a Linux platform, the measurements of Vegas/CODE competing with Reno can be performed over the Internet. Specifically, two FTP clients upload files having sizes of 53,612 kB each to an FTP server. One FTP client uses Vegas or CODE as its transport protocol, and the other uses Reno. The two FTP clients start at the same time and thus compete for resources of the same network path over the Internet. The FTP server is at National Changhua University of Education (NCUE), and the FTP clients are at National Chiao Tung University (NCTU), as shown in Fig. 20. The two universities are approximately 100 km apart.

We conduct 10 successive rounds of measurement. Each round consisted of two independent experiments. In one, Vegas competes with Reno, while in the other, CODE competes with Reno. Table 3 summaries the results. The throughput of each sample is calculated dividing the file size (53,612 kB) by the file transfer time. When competing with Vegas, by using its more aggressive congestion avoidance scheme, Reno has a considerably higher throughput than Vegas. It should be noted that although the average throughput of Reno is 1.59 times that of Vegas, it does not seem to agree with the simulation results, such as shown in Fig. 9(a). In our real network experiments, FTP clients with different TCP variants start at the same time and send files having equal sizes to the FTP server. The FTP client with Reno completes its transmission first. The FTP client with Vegas then utilizes the network resources released by Reno. Therefore, the throughput of Vegas is not very poor. From Table 3, we conclude that the average throughputs of Reno and CODE are almost equal. Furthermore, from the confidence intervals under different confidence levels, we find that the competing results of CODE and Reno are quite stable. These results confirm the effectiveness of CODE.



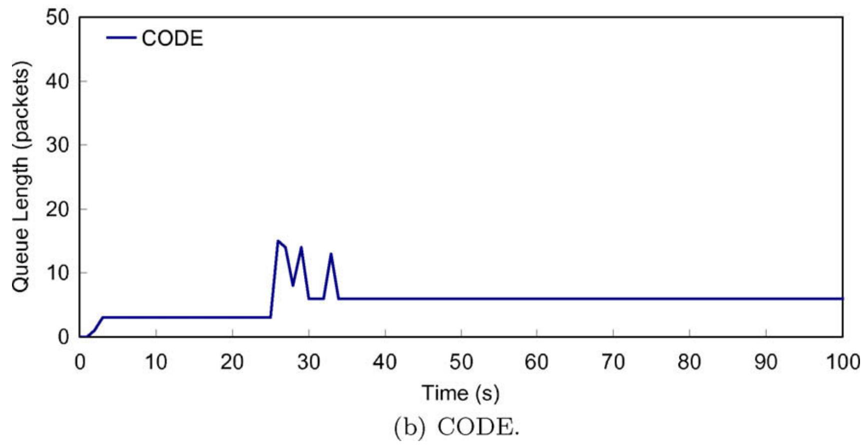


Fig. 18. Status of the bottleneck queue. A new connection arrives at the bottleneck link that is occupied by the previously started connection in equilibrium.

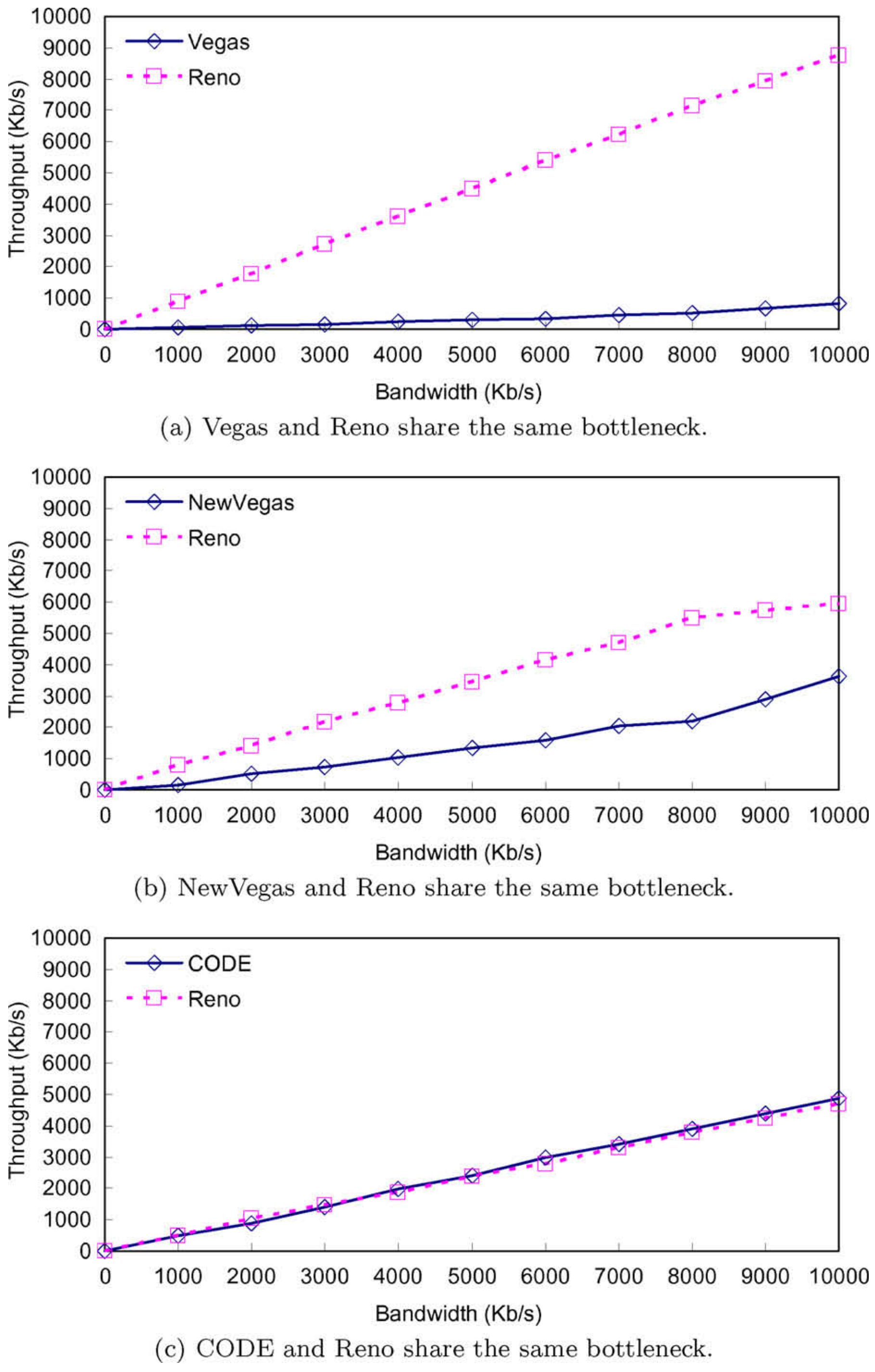


Fig. 19. Variation of achieved throughput of two competing connections with the bottleneck bandwidth.

5. Conclusion

In this study, we propose a new TCP variant, COmpetitive DE-lay-based TCP (CODE TCP), that is more compatible with Reno and may get its fair share of network resources. CODE is a sender-sided modification and hence it can be easily implemented at the end host. Accordingly, CODE can be easily deployed in the Internet. It provides rather flexible control and also retains the good features of the original Vegas. Simulations and experiments confirm that CODE can dynamically adjust Vegas thresholds to overcome the unfairness problem when coexisting with Reno.

An issue that has not been addressed in the present work is the behavior of CODE in networks with non-persistent traffic sources. The slow-start mechanism of CODE is the same as that in Vegas. It appears that Reno may gain an advantage because of its more aggressive start-up. Implementing Reno's slow-start in CODE would reduce this disadvantage. However, this may affect some features of CODE, such as its stability and fairness. In addition, transmission in high bandwidth-delay product networks has become increasingly popular. Quickly adjusting the transmission rate to the bandwidth on the bottleneck link is an important issue for any new TCP variant. Therefore, we intend to design a new

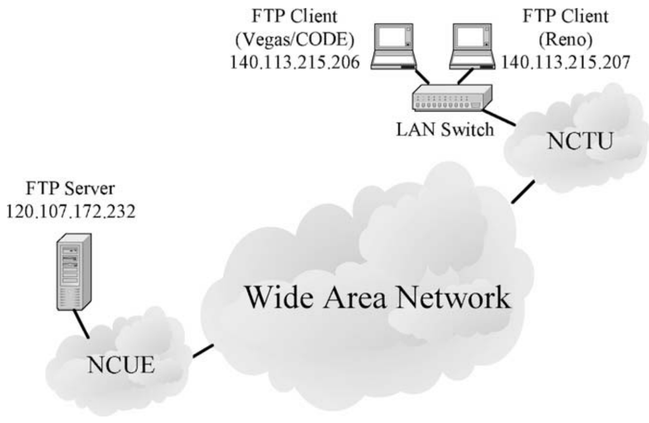


Fig. 20. Real test-bed network.

Table 3
Throughput (KB/s) of two competing connections over the internet.

Round	Reno vs. Vegas	Reno vs. CODE
1	547.23 315.69	546.66 542.51
2	553.59 339.91	548.96 551.91
3	552.21 381.16	535.13 540.28
4	552.14 358.31	544.94 548.57
5	555.34 275.88	546.05 551.51
6	552.82 368.05	546.25 542.45
7	553.80 343.03	545.71 548.23
8	552.98 315.91	546.04 549.97
9	551.96 390.85	546.92 543.66
10	552.39 380.89	546.41 540.12
Avg.	552.45 346.97	545.31 545.92
99% CI	552.45 ± 1.71 346.97 ± 29.59	545.31 ± 3.03 545.92 ± 3.74
95% CI	552.45 ± 1.30 346.97 ± 22.52	545.31 ± 2.31 545.92 ± 2.85
90% CI	552.45 ± 1.09 346.97 ± 18.90	545.31 ± 1.94 545.92 ± 2.39

slow-start mechanism and adapt CODE to high-speed networks in the future.

References

[1] W. Stevens, TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, in RFC 2001, January 1997.
[2] L.S. Brakmo, L.L. Peterson, TCP Vegas: end to end congestion avoidance on a global internet, in: IEEE Journal on Selected Areas in Communications, vol. 13, issue 8, August 1995, pp. 1465–1480.
[3] D. Kim, H. Bae, C. Toh, Improving TCP-Vegas performance over MANET routing protocols, in: IEEE Transactions on Vehicular Technology, vol. 56, issue 1, January 2007, pp. 372–377.
[4] C. Yuan, L. Tan, L. Andrew, W. Zhang, M. Zukerman, A generalized FAST TCP scheme, in Computer Communications, vol. 31, issue 14, September 2008, pp. 3242–3249.
[5] D.X. Wei, C. Jin, S.H. Low, S. Hegde, FAST TCP: motivation, architecture, algorithms, performance, in: IEEE/ACM Transactions on Networking, vol. 14, issue 6, December 2006, pp. 1246–1259.
[6] N. Bigdeli, M. Haeri, AQM controller design for networks supporting TCP vegas: a control theoretical approach, in: ISA Transactions, vol. 47, issue 1, January 2008, pp. 143–155.
[7] R. Awdeh, Compatibility of TCP Reno and TCP Vegas in wireless ad hoc networks, in: IET Communications, vol. 1, issue 6, Dec. 2007, pp. 1187–1194.
[8] L. Ding, X. Wang, Y. Xu, W. Zhang, Improve throughput of TCP-Vegas in multihop ad hoc networks, in: Computer Communications, vol. 31, issue 10, June 2008, pp. 2581–2588.
[9] S. Herrena-Alonso, M. Rodriguez-Perez, A. Suarez-Gonzalez, M. Fernandez-Veiga, C. Lopez-Garcia, Improving TCP Vegas fairness in presence of backward traffic, in: IEEE Communications Letters, vol. 11, issue 3, March 2007, pp. 273–275.
[10] S. Liu, T. Basar, R. Srikant, TCP-Illinois: A loss- and delay-based congestion control algorithm for high-speed networks, in: Performance Evaluation, vol. 65, issues 6–7, June 2008, pp. 417–440.
[11] G. Hasegawa, H. Miyahara, Fairness and stability of congestion control mechanisms of TCP, in: Proceedings of IEEE INFOCOM'99, vol. 3, 1999, pp. 1329–1336.
[12] J. Mo, R.L.V. Anantharam, J. Walrand, Analysis and comparison of TCP Reno and Vegas, in: Proc. of IEEE INFOCOM'99, vol. 3, March 1999, pp. 1556–1563.
[13] W. Feng, P. Tinnakornsrisuphap, The failure of TCP in high-performance computational grids, in: Proc. of Supercomputing'00, pp. 37–37, November 2000.
[14] Y. Chan, C. Lin, C. Ho, Quick Vegas: improving performance of TCP Vegas for high bandwidth-delay product networks, in: IEICE Transactions on Communications, vol. E91-B, issue 4, April 2008, pp. 987–997.
[15] W. Feng, S. Vanichpun, Enabling compatibility between TCP Reno and TCP Vegas, in: Proceedings of the IEEE Symposium on Applications and the Internet, January 2003, pp. 301–308,.
[16] Y.C. Lai, C.L. Yao, Performance comparison between TCP Reno and TCP Vegas, in: Computer Communications, vol. 25, issue 18, December 2002, pp. 1765–1773.
[17] A. DeVendictis, A. Baiocchi, M. Bonacci, Analysis and enhancement of TCP Vegas congestion control in a mixed TCP Vegas and TCP Reno network scenario, in: Performance Evaluation, vol. 5, issue 3–4, August 2003, pp. 225–253.
[18] K. Srijith, L. Jacob, A. Ananda, TCP Vegas-A: improving the performance of TCP Vegas, in: Computer Communications, vol. 28, issue 4, March 2005, pp. 429–440.
[20] Y.C. Lai, C.L. Yao, The performance comparison between TCP Reno and TCP Vegas, in: Seventh International Conference on Parallel and Distributed Systems'00, July 2000, pp. 61–66.
[21] K. Takagaki, H. Ohsaki, M. Murata, Analysis of a window-based flow control mechanism based on TCP Vegas in heterogeneous network environment, in: Proceedings of the IEEE ICC'01, vol. 10, June 2001, pp. 3224–3228.
[22] G. Hasegawa, K. Kurata, M. Murata, Analysis and improvement of fairness between TCP Reno and Vegas for deployment of TCP Vegas to the internet, in: Proceedings of the ICNP, pp. 177–186, November 2000.
[23] Y.C. Lai, Improving the performance of TCP Vegas in a heterogeneous environment, in: Proceedings of the IEEE ICPADS, June 2001, pp. 581–587.
[24] E. Weigle, W. Feng, A case for TCP Vegas in high-performance computational grids, in: High Performance Distributed Computing, August 2001, pp. 158–167.
[25] Network Simulator 2 (NS-2), Available from <http://www.isi.edu/nsnam/ns>.
[26] R. Jain, A. Durresi, G. Babic, Throughput fairness index: an explanation, Available from <http://www.cis.ohio-state.edu/~jain/atmf/a99-0045.htm>, February 1999.