在不同擁塞控制的方法下之 TCP 效能分析與公平性改進

TCP Performance Analysis and Fairness Improvement in Different Congestion Control Methods

研 究 生：張均瑋　　Student：Chun-Wei Chang

指導教授：陳耀宗　博士　　Advisor：Prof. Yaw-Chung Chen

國 立 交 通 大 學

資 訊 工程 系

碩 士 論 文

A Thesis

Submitted to Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science and Information Engineering

June 2004

Hsinchu, Taiwan, Republic of China

中華民國九十三年六月

在不同擁塞控制的方法下之TCP效能分析與公平性改進

研究生 ： 張均瑋　　　指導教授 ：陳耀宗　博士

國立交通大學資訊工程學系

中文摘要

在不同的擁塞控制的方法下，TCP Vegas 明顯的優於現行最常被使用的 TCP Reno，然而；當 TCP Vegas 的使用者在和 TCP Reno 的使用者競爭可用頻寬時，TCP Vegas 很明顯的遠輸給 TCP Reno，這也就是為什麼在網際網路的使用者裡，很少人採用 TCP Vegas 的最主要原因。

在這篇論文中，我們提出了一種方法，藉由我們估計出在路由器裡的連線數，動態的去調整 RED 裡的最低下限值和最高上限值，達到偵測出 TCP Reno 的連線封包，根據我們審慎的分析，TCP Vegas 和 TCP Reno 是如何利用在 RED 路由器裡的緩衝空間，我們可以得知假使在 RED 路由器裡的緩衝空間能被適當的設定，那麼 TCP Vegas 將能夠有能力和 TCP Reno 競爭，我們用了分析和模擬實驗來評估公平性的問題，而且我們也證實了我們方法的可行性。

關鍵字： TCP Reno；TCP Vegas；RED 路由器；擁塞控制；公平性

TCP Performance Analysis and Fairness Improvement in Different Congestion Control Methods

Student：Chun-Wei Chang　　Advisor：Dr. Yaw-Chung Chen

Department of Computer Science and Information Engineering

National Chiao Tung University

Abstract

TCP Vegas outperforms the more widespread TCP Reno in congestion control. However, TCP Vegas sources show high weakness in taking the available bandwidth when competing with other TCP Reno sources. This is a major reason that hinders the spread of TCP Vegas among Internet users. In this thesis, we propose a method to modify the RED algorithm for detecting packets from TCP Reno connections, just simply dynamically adjusting the value of *MinThresh* and *MaxThresh* in RED Routers according to the number of connections we predicted. With a careful analysis of how Vegas and Reno use buffer space in RED routers, we will show that Reno and Vegas can be compatible with one another if the buffer in RED Router is configured properly. We use both of analysis and simulation experiment for evaluating the fairness, and validate the effectiveness of the proposed mechanisms.

Keyword: TCP Reno; TCP Vegas; RED Router; Congestion Control; Fairness

Acknowledgement

First, I would like to express my deep gratitude to my advisor, Prof. Yaw-Chung Chen, for his enthusiastic guidance and continual encouragements in my graduate life. Also, I would deeply appreciate to the new Ph.D. from our lab, Dr. Yi-Cheng Chan. He gave me many beneficial suggestions and valuable comments in my research and thesis writing.

Moreover, I would offer my heartfelt thanks to my lab-mate Iori, Kent, and Kyho in the Multimedia Communications Laboratory and my friends in DSNS laboratory. They made our laboratory such an enjoyable place to work and we had a great time in my semester.

Finally, I would express my indebtedness to my family for their endless love, inspiration, and support. I am really grateful to my lovely ex-girlfriend, I-Yi Liu. She made my college life colorful and gave me many unforgettable pleasant memories. Especially, I would thanks to Wan-Ting Huang. With her consideration and inspiration, I have the power to complete this thesis.

Chun-Wei Chang

National Chiao Tung University in Hsinchu, Taiwan.

June 2004

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Evolution of Transmission Control Protocol (TCP)

With the fast growth of Internet traffic, TCP has become the most widely used end-to-end transport protocol on the Internet for congestion control. TCP uses window-based flow control to pace the transmission of packets. Each source maintains a "window size" variable that limits the maximum number of packets that can be outstanding: transmitted but not yet acknowledged. When a window's worth of data is outstanding the source must wait for an acknowledgment before sending a new packet. Two features of this general strategy are important. First, the algorithm is "self-clocking" meaning that TCP automatically slows down the source when the network becomes congested and acknowledgments are delayed. The second is that the window size variable determines the source rate: roughly one window's worth of packets is sent every roundtrip time. This second feature is exploited in Jacobson's paper [1] in which he proposed an Additive-Increase-Multiplicative-Decrease algorithm to adapt the window size to network congestion.

Transmission Control Protocol (TCP) has several implementation versions which intend to improve network utilization. The first version of TCP, standardized in RFC793, defined the basic structure of TCP, i.e. the window-based flow control scheme and a coarse-grain timeout timer. The second version, TCP Tahoe, added the congestion avoidance scheme and fast retransmission proposed by Van Jacobson [1]. The third version, TCP Reno, extended the congestion control scheme by including fast recovery scheme [2]. Today, TCP Reno is the most popular version and most TCP congestion controls are based on this approach. The reason of using such a window-based adjustment approach is the simplicity in implementation.

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. TCP Reno congestion control reacts to the network congestion only after detecting a loss, that is, after the network congestion has already occurred. This avoids congestion collapse, since the transmission rate is reduced as soon as a packet loss occurs; but this generates an intrinsic instability of the congestion control, whose evidence is the periodical oscillation of the source transmission rate.

TCP Vegas, proposed by Brakmo and Peterson in [3], represents a valid alternative to the congestion control performed by the currently standard and most widespread version of TCP, called TCP Reno. Although it introduces new techniques into all the main mechanisms of TCP, it is fully compatible with all the standard versions of TCP, because the changes only concern the TCP sending side. TCP Vegas, by modifying the congestion avoidance scheme of TCP Reno, can achieve a 37–71% higher throughput than TCP Reno did in a homogeneous environment. In addition to a higher throughput, TCP Vegas has a fairer and more stable bandwidth share than TCP Reno does.

TCP Vegas enhances the congestion avoidance algorithm of TCP Reno. The key idea is that 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 a packet loss is detected.

1.2 Motivation and Contribution

Motivation Several simulation works have verified that TCP Vegas is better than TCP Reno in terms of throughput (between 37 and 71% better than Reno), fairness, stability, packet loss rate, end-to-end delay and ability in avoiding network congestion in a very large number of network environments (see [3–6]). For this reason TCP Vegas has been the subject of a number of recent studies (e.g., [7-8]).

The merits of TCP Vegas would appear to suggest that it should replace TCP Reno soon. However, an unfairness problem occurs when TCP Reno and Vegas connections coexist [4,9]. TCP Reno is an aggressive control scheme in which each connection captures more bandwidth until the transmitted packets are lost. Meanwhile, TCP Vegas is a conservative scheme in which each connection obtains a proper bandwidth. Thus, the TCP Reno connections take bandwidth from the TCP Vegas connections when they coexist. The problem is that TCP Reno is intrinsically much more aggressive than TCP Vegas, because it reduces its transmission rate only after packet loss detection. This represents the main reason why TCP Vegas cannot be widely proposed to the users as a reliable transport protocol.

TCP Vegas performs poorly in a situation in which two versions coexist. Users prefer not to adopt TCP Vegas despite its superior performance to that of TCP Reno in a single version environment. However, TCP Vegas is actually a very good congestion control and worth adopting. Resolving this unfairness is very important in the transition from TCP Reno to Vegas.

Therefore, given that TCP Vegas basic approach is a sound one, there is a need for some modifications of its basic congestion control algorithm to stimulate its use among users.

Contribution In this thesis, we propose a method to modify the RED algorithm for detecting packets from TCP Reno connections, just simply dynamically adjusting the value of *MinThresh* and *MaxThresh* in RED Routers according to the number of connections we predicted [10]. For this purpose, we can utilize the algorithm proposed in [11] to detect mis-behaving flows, which are TCP Reno connections in the current content. Then the throughput of TCP Reno connections can be decreased by intentionally dropping packets of TCP Reno at the router.

With a careful analysis of how Reno and Vegas use buffer space in RED routers, we will show that Reno and Vegas can be compatible with one another if the buffer in RED Router is configured properly. Further, overall network performance actually improves with the addition of properly configured Vegas flows competing head-to-head with Reno flows, thus encouraging the incremental adoption of Vegas.

1.3 Thesis Outline

The rest of this thesis is organized as follows. In Chapter 2, we first introduce congestion control mechanisms of TCP Reno and TCP Vegas, and then elucidate several TCP Vegas variants to solve the fairness problem. We next describe the RED Router we proposed in this thesis and the way to set the value of *MinThresh* and *MaxThresh* in RED Routers properly in Chapter 3. In Chapter 4, we present the network model used in our simulation experiments and show the analysis results of fairness between two versions of TCP, which are validated by the simulation results. Finally, we conclude the thesis and present some future works in Chapter 5.

CHAPTER 2

Related Work

In this Chapter, we first summarize the congestion control mechanisms of two versions of TCP; TCP Reno and TCP Vegas. For detailed explanation, refer to [12] for TCP Reno and [3] for TCP Vegas. Then, we describe several methods to solve the fairness problem between TCP Reno and TCP Vegas presented in [13][14][15][16]

2.1 TCP Congestion Control Mechanisms

Congestion control is a distributed algorithm to share network resources among competing sources. It consists of a source algorithm (e.g., Reno, Vegas, etc.), that dynamically adjust source rates based on congestion in end-to-end paths, and a link algorithm (e.g., Drop Tail, RED, REM, etc.), that updates, implicitly or explicitly, a certain congestion measure at each link and feeds it back, implicitly or explicitly, to sources that use this link. Different protocols use different metrics as congestion measure. For example, Reno uses loss probability, and as it turns out Vegas uses queuing delay. Congestion control which is a distributed algorithm carried out by sources and links over the network to solve a global optimization problem, and that different protocols (Reno, Vegas, Drop Tail, RED, REM, etc.) are different ways to solve the same prototypical problem with different objective functions.

A TCP connection contains three entities: sender, router, and receiver. A sender uses some algorithms to estimate and control the amount of data to be retransmitted to the receiver through some routers. A router in a TCP connection only forwards data packets. A receiver acknowledges the received packets to the sender by sending acknowledgement (ACK) packets. If any out-of-order packet is received, the receiver sends a duplicate ACK back to the sender.

To ensure efficient use of network bandwidth, TCP controls its sending rate based on the feedback from the network. In order to control the sending rate, TCP estimates the available bandwidth in the network via a bandwidth-estimation scheme [4]. In Tahoe and Reno, the bandwidth-estimation scheme uses packet losses (as an indication of network congestion) to estimate available bandwidth while Vegas uses the difference in the expected and actual sending rates.

TCP Reno and TCP Vegas use slow start at the beginning of the connection, also whenever a packet loss is detected via timeout. When the congestion window reaches a threshold value, called slow start threshold, TCP Reno and TCP Vegas leave slow start and enter congestion avoidance.

Both protocols can detect packet losses by means of two mechanisms. If the timeout (set when the packet is sent) expires, they reduce their congestion window to one packet size, and then they start again in slow start mode. Otherwise, if three duplicated acknowledgements (ack's) come back to the sender before the timeout, the protocols perform fast retransmit and fast recovery.

During the connection, the TCP receiver can limit the sender congestion window by advertising the receiver window value.

TCP Vegas differs from TCP Reno in the way slow start, congestion avoidance and fast retransmit are implemented.

TCP Reno and TCP Vegas [3] adopt an end-to-end closed-loop adaptive window congestion control. It is based on five fundamental mechanisms: slow start, congestion avoidance, retransmission timeout, fast retransmit and fast recovery.

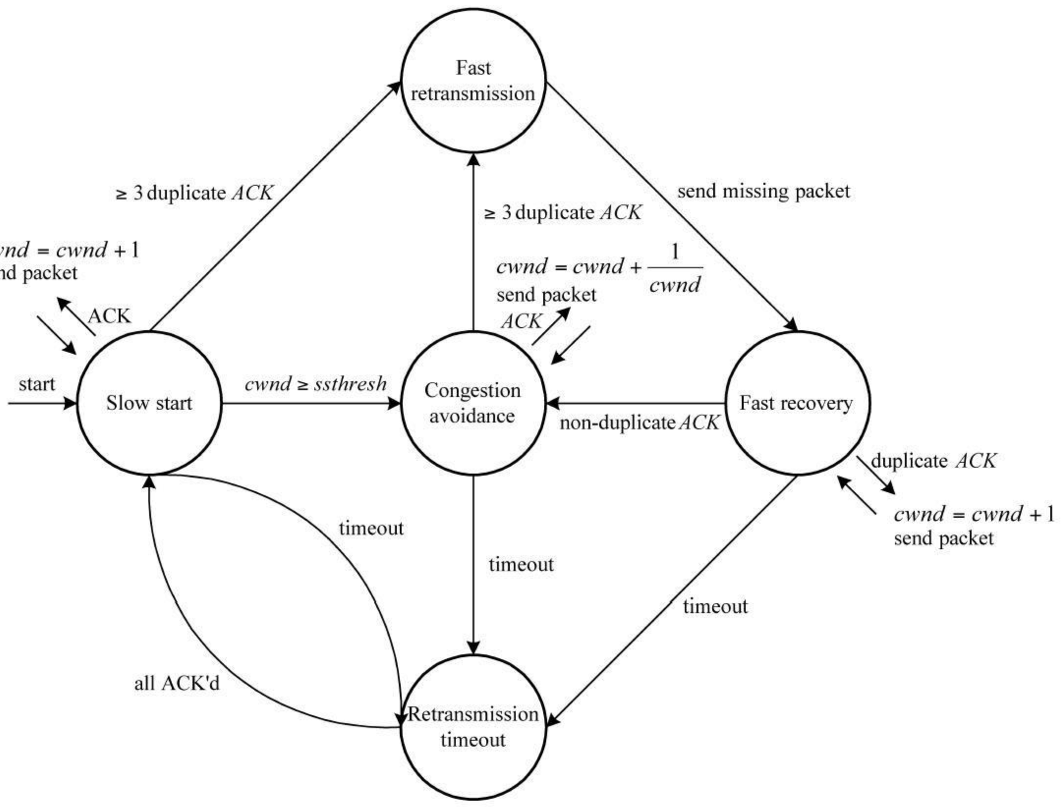


Figure 2.1 Congestion control diagram of TCP Reno.

TCP typically uses the acknowledgements to estimate the available bandwidth. Reno and Vegas differ mainly in how to estimate bandwidth. Reno treats the packet loss as an indicator of network congestion. The source is aware that it overuses the bandwidth when the transmitted packets are lost. Vegas calculates the expected and actual rates based on the round-trip time of each packet and controls the amount of transmitted packet by using the difference between the expected and actual rates. This section describes the control scheme of Reno [12] and then presents TCP Vegas [3].

2.1.1 TCP Reno

In TCP Reno, the windows size is cyclically changed in a typical situation. The window size continues to increase until packet loss occurs. TCP Reno has two phases in increasing its window size; slow start phase and congestion avoidance phase.

While there are no packet losses, Reno continues to increase its window size, and hence the sending rate by one packet each round-trip time, thus allowing congestion to eventually occur. Reno then detects congestion via packet loss and recovers from it by halving the size of the sender window (i.e., halving the sending rate).

Reno uses a congestion window (*cwnd*) to control the amount of transmitted data in one round trip time (RTT) and a maximum window (*mwnd*) to limit the maximum value of *cwnd.* The control scheme of Reno can be divided into five parts, which are interpreted as follows. **Figure 2.1** schematically depicts the TCP Reno version specified with these parts.

1. **Slow-start.** As a connection starts or a timeout occurs, the slow-start state begins. The initial value of *cwnd* is set to one packet in the beginning of this state. The sender increases *cwnd* exponentially by adding one packet each time it receives an ACK. Slow-start controls the window size until *cwnd* achieves a preset threshold, slow-start threshold (*ssthresh*). When *cwnd* reaches *ssthresh*, the congestion avoidance' state begins.
2. **Congestion avoidance.** Since the window size in the slow start state expands exponentially, the packets sent at this increasing speed would quickly lead to network congestion. To avoid this, the 'congestion avoidance' state begins when *cwnd* exceeds *ssthresh*. In this state, *cwnd* is added by 1/cwnd packet every receiving an ACK to make the window size grow linearly.

3. **Fast retransmission.** The duplicate ACK is caused by an out-of-order packet received in the receiver. The sender treats it as a signal of a packet loss or a packet delay. If three or more duplicate ACKs are received in a row, packet loss is likely. The sender performs retransmission of what appears to be the missing packet, without waiting for a coarse-grain timer to expire.
4. **Fast recovery.** When fast retransmission is performed, *ssthresh* is set to half of *cwnd* and then *cwnd* is set to *ssthresh* plus three packet sizes. *Cwnd* is added by one packet every receiving a duplicate ACK. When the ACK of the retransmitted packet is received, *cwnd* is set to *ssthresh* and the sender re-enters the congestion avoidance. *Cwnd* is reset to half of the old value of *cwnd* after fast recovery.
5. **Timeout retransmission.** For each packet sent, the sender maintains its corresponding timer, which is used to check for timeout of non-received ACK of the packet. If a timeout occurs, the sender resets the *cwnd* to one and restarts slow-start. The default value of clock used for the round-trip ticks is 500 ms, i.e. the sender checks for a timeout every 500 ms.

When an ACK 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;

Slow start phase:

$cwnd(t + t_A) = cwnd(t) + 1,$ if $cwnd(t) < ssth(t)$;

Congestion avoidance phase:

$cwnd(t + t_A) = cwnd(t) + 1/cwnd(t),$ if $cwnd(t) \geqq ssth(t)$;

Timeout expiration:

$cwnd(t) = 1$; $ssth(t) = cwnd(t)/2$

Fast retransmit:

$ssth(t) = cwnd(t)/2$; $cwnd(t) = ssth(t)$

2.1.2 TCP Vegas

TCP Vegas [3] was introduced in 1994 as an alternative to TCP Reno [BrakmoandPeterson1995]. It improves upon each of the three mechanisms of TCP Reno. The first enhancement is a more prudent way to grow the window size during the initial use of slow-start and leads to fewer losses. The second enhancement is an improved retransmission mechanism where timeout is checked on receiving the first duplicate acknowledgment, rather than waiting for the third duplicate acknowledgment (as Reno would), and leads to a more timely detection of loss. The third enhancement is a new congestion avoidance mechanism that corrects the oscillatory behavior of Reno. In contrast to the Reno algorithm, which induces congestion to learn the available network capacity, a Vegas source anticipates the onset of congestion by monitoring the difference between the rate it is expecting to see and the rate it is actually realizing. Vegas' strategy is to adjust the source's sending rate (window size) in an attempt to keep a small number of packets buffered in the routers along the path.

Vegas enhances Reno by adopting a bandwidth-estimation scheme that tries to avoid rather than react to congestion. Vegas uses the measured RTT to accurately calculate the amount of data packets that the sender can send to avoid packet losses. In Vegas, the sender must record the RTT and the sending time of each packet. The minimum round trip time, baseRTT, must also be kept. Herein, three modifications are proposed based on the measured RTT [3].

(1) New congestion avoidance. Specifically, Vegas uses the difference in the expected and actual flow rates to estimate the available bandwidth in the network. When the network is not congested, the actual flow rate is close to the expected flow rate; otherwise, the actual rate is smaller than the expected rate, indicating that buffer space in the network is filling up and that the network is approaching a congested state. The difference in flow rates can be translated into the difference between the window size and the number of acknowledged packets during the RTT, respectively.

When receiving an ACK, the sender calculates the difference of the expected and the actual throughputs as follows:

diff = (expected – actual) * baseRTT,

expected = cwnd / baseRTT,

actual = cwnd / average measured RTT.

Expected throughput represents the available bandwidth for this connection without network congestion, and actual throughput represents the bandwidth currently used by the connection. To adjust the size of the congestion window (cwnd) appropriately, Vegas defines two thresholds (α, β) (whose default values are 1 and 3, respectively) as a tolerance that allows the source to control the difference between expected and actual throughputs in one RTT. Cwnd is increased by one packet if diff $< \alpha$ and decreased by one packet if diff $> \beta$. That is

$cwnd(t + t_A) = cwnd(t) + 1$, if diff $< \alpha$

$cwnd(t + t_A) = cwnd(t) - 1$, if diff $> \beta$

$cwnd(t + t_A) = cwnd(t)$, if otherwise ($\alpha \leq$ diff $\leq \beta$)

Conceptually, Vegas tries to keep at least α packets but no more than β packets queued in the network. Thus, with only one Vegas connection, the window size of Vegas converges to a point that lies between *window* + α and *window* + β where *window* is the maximum window size that does not cause any queuing.

Selecting α and β holds an implicit tradeoff between network utilization, goodput, and fairness. By using the default settings for these parameters, i.e., $\alpha = 1$, $\beta = 3$, prior research inadvertently favored Reno over Vegas [4].

(2) Earlier packet loss detection. Upon receiving a duplicate ACK, the sender checks whether the difference between the current time and the sending time plus baseRTT of the relevant packet is greater than the timeout value. If it is, Vegas retransmits the packet without waiting for three duplicate ACKs. This modification can avert a situation in which a sender never receives three duplicate ACKs and, therefore, must rely on the coarse-grain timeout.

Aiming to multiple-packet-loss problem, Vegas pays special attention to the first two partial ACKs after a retransmission. The sender determines whether this is a multiple-packet-loss by checking the timeout of unacknowledged packets. If any timeout occurs, the sender immediately retransmits the packet without waiting for any duplicate ACK. Also to avoid the over-reduction of window size due to the loss occurred at the previous window size, Vegas compares the timestamp of the retransmitted packet and the timestamp of the last window decrease. When the retransmitted packet is sent before the last decrease, Vegas will not decrease cwnd on receiving duplicate ACKs for this packet because this packet loss occurred due to the previous window size. For Vegas, it is very important that designing such a mechanism to avoid unnecessary reduction of cwnd because of its quick response of packet loss.

(3) Modified slow-start mechanism. To detect and avoid congestion during slow-start, Vegas doubles it window size every other RTT, instead of every RTT.

When a TCP Vegas source receives three duplicate Ack’s, it performs fast retransmit and fast recovery as TCP Reno does. Actually, TCP Vegas develops a more refined fast retransmit mechanism based on a fine-grain clock, whose details are described in [3]. After fast retransmit TCP Vegas sets the congestion window to 3/4 of the current congestion window and performs again the congestion avoidance algorithm.

2.2 Source-Based Congestion Avoidance Mechanisms

In this section, we describe several methods to solve the fairness problem between TCP Reno and TCP Vegas presented in another paper.

2.2.1 Enlarge α and β

Lai [13] proposes a simple approach, parameter adjustment for Vegas, to solve the fairness problem between TCP Reno and TCP Vegas.

Vegas relates the difference between the actual and expected throughput to the α and β thresholds, which values can thus be thought in terms of the number of extra buffers the connection occupies in the network. Thus, the Vegas connections with larger α and β allow more packets queuing in the buffer and result in higher throughput. We adjust α and β to observe the effects of these parameters.

A simulation is conducted by varying α and β. **Figure 2.2** is the simulation topology and **Figure 2.3** shows that Vegas with larger α and β behaves more aggressively. Thus, Lai [13] finally suggests choosing a large α and β.

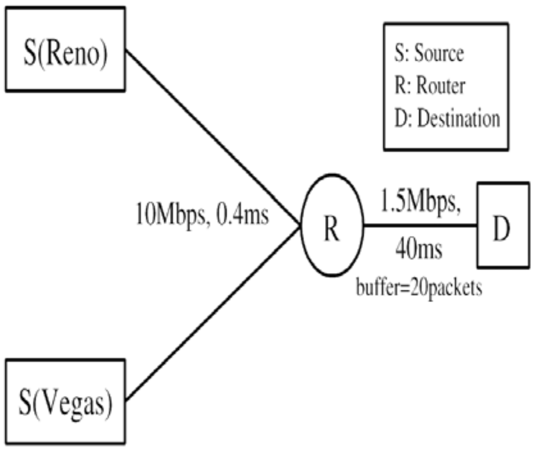


Figure 2.2 Network topology.

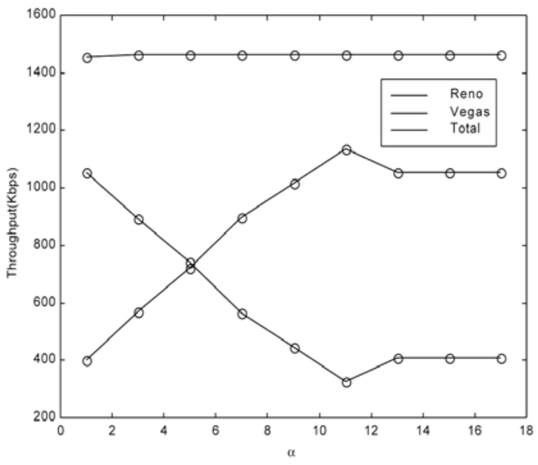


Figure 2.3 Throughput (β=α+2).

2.2.2 TCP NewVegas

Andrea De Vendictis [14] describe a new adaptive mechanism for TCP Vegas, called TCP NewVegas, designed in order to improve its performance even in heterogeneous network scenarios.

TCP NewVegas is able to overcome this weakness of TCP Vegas, by hindering it to perform the last two phases when it competes with TCP Reno sources. The changes introduced in TCP NewVegas are confined within the congestion avoidance mechanism. Initially, TCP NewVegas sets the thresholds α and β to default values α0 and β0 (in this work, respectively, set to 1 and 3). Whenever an ack arrives at the source, TCP NewVegas updates the thresholds as follows:

```
if (( $RTT > RTT_{old}$ ) and ( $W <= W_{old}$ ))
{ $\alpha = \alpha + 1$;   $\beta = \beta + 1$; }
if (( $RTT <= RTT_{old}$ ) and ( $\alpha > \alpha 0$ ))
{ $\alpha = \alpha - 1$;   $\beta = \beta - 1$; }
```

where RTT is the round trip delay of the target packet, RTT*old* the previous one, W the current congestion window size and W*old* the previous one.

Threshold updates are allowed once every round trip time. When a timeout expiration or a triple duplicate ack's occur, the two thresholds are set again to the initial values α0 and β0. The key concept, expressed by the first condition, is that if the congestion is increasing (last RTT larger than the previous one) but TCP NewVegas has not increased its congestion window, it does not consider itself responsible for the network congestion. Then, the source is allowed to increase the number of packets it can put into the network buffers. We increase α and β by 1 in order to follow the exact behavior of TCP Reno in congestion avoidance, given that it put into the network an extra packet every round trip time.

2.2.3 TCP Vegas-A

K.N. Srijith [15] introduced a modified version of TCP Vegas, TCP Vegas-A which is able to obtain a fairer share of the network bandwidth when competing with other TCP flows. The main idea in Vegas-A is that fixing α and β be made adaptive.

TCP Vegas-A

```
Th(t) = actual throughput at time t; Th(t-rtt) = actual throughput at previous time rtt;
if β > diff > α {
    if Th(t) > Th(t-rtt)    { cwnd = cwnd – 1; α = α - 1; β = β - 1; }
    /* The reasoning is that, even though diff > α, the throughput has been
    increasing.  This indicates that the network is not fully utilized and that
    network bandwidth is still available.  Hence, the sending rate can be increased,
    to probe the network. */
    /* Since throughput is increasing over time, diff is decreasing.  The existing
    small values of α and β prevent the connection from making use of the available
    bandwidth.  Hence we increase α and β to help congestion window grow. */
    else if Th(t) <= Th(t-rtt)    {no update of cwnd, α, β}
}
else if diff < α {
    if α > 1 and Th(t) > Th(t-rtt)    { cwnd = cwnd – 1; }
    else if α > 1 and Th(t) < Th(t-rtt)    { cwnd = cwnd – 1; α = α - 1; β = β - 1; }
    else if α = 1    { cwnd = cwnd + 1; }
}
else if diff > β    { cwnd = cwnd – 1; α = α - 1; β = β - 1; }
else    {no update of cwnd, α, β}
```

2.2.4 TCP Vegas+

Go Hasegawa [16] 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.

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.

Aggressive Mode: In the moderate mode, The TCP Vegas+ sender host behaves identically to the original TCP Reno.

To switch between the above two modes, we introduce new variables *count* and *countmax*. First, count is updated according to the following algorithm.

1. 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, sender decrements count by 1.
3. If packet loss is detected by fast retransmit algorithm, count is halved.
4. If packet loss is detected by retransmission timeout, 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 *countmax*, 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.

The proposed solutions present the following benefits:

1. The changes are confined to the TCP sender side, so modifications are not required at the receiver side nor within the network; this makes these modified versions of TCP Vegas easy to implement with respect to other solutions proposed in [16] that require changes in the network.
2. The algorithm does not introduce further thresholds generally hard to set, since it is completely adaptive to the status of the network; in this prospect TCP NewVegas appears to be more efficient than another algorithm proposed in [16], in which TCP Vegas should react to the aggressiveness of TCP Reno when a parameter exceeds a threshold, whose value is not easy to choose.
3. These modified versions of TCP Vegas behavior are not much different from that of the original TCP Vegas in presence of other TCP Vegas sources; so it is able to preserve all the nice features of the original TCP Vegas: good throughput performance, stability, fairness, ability in network congestion avoidance.

The reason why we didn't choose end-to-end TCP Vegas algorithm to solve the fairness problem between TCP Reno and TCP Vegas is that it always needs more buffer size in routers, because those methods enable TCP Vegas connections to compete with TCP Reno connections by increasing its congestion window when it detects TCP Reno connections coexist (whether it is true or not). The fundamental concept of these methods is based on the in formation of the measured RTT and current congestion window to predict if TCP Reno connections exist. Mis-prediction of those methods will cause network more congestion, while our proposed RED Router algorithm is another way to restrict the mis-behave connections by early packet drop. The RED Router algorithm we proposed is present in Chap 3 and the performance evaluation is discussed in Chap 4.

CHAPTER 3

Proposed Approach based on RED Routers

In this Chapter, we describe the reason why the fairness between TCP Reno and TCP Vegas occurs at first, then the RED router in detail. Through a simple network topology, we will show the fairness can be improved by RED router and discuss how to set the parameter *MaxThresh* and *MinThresh* properly to utilize network bandwidth.

3.1 Problem Description

TCP Vegas is a more refined congestion control mechanisms based on the estimation of round trip delays, it outperforms the more widespread TCP Reno congestion control, which is based only on the packet loss detection, in various environments. However, these mechanisms make TCP Vegas less aggressive with respect to TCP Reno; thereby TCP Vegas sources show serious weakness in taking the available bandwidth when competing with other TCP Reno sources. This is a major reason that hinders the spread of TCP Vegas among Internet users.

In a homogeneous case, TCP Vegas outperforms TCP Reno owing to its higher throughput and stability. However, TCP Vegas is in a squally situation when TCP Reno and Vegas coexist. Because two TCP versions share the same buffer in the routers, the aggressive behavior of TCP Reno and the conservative behavior of TCP

Vegas cause unfairness when they are used simultaneously. TCP Reno obtains a significant amount of bandwidth, which originally belonged to TCP Vegas. Also, as more connections change their version, TCP Reno takes more bandwidth. Thus, users delay, even protest, to use TCP Vegas despite its better performance than TCP Reno in a homogeneous environment. This accounts for why TCP Vegas is still unpopular.

Hasegawa et al. [16] show that with the RED routers in place of the tail-drop routers, the fairness between Vegas and Reno can be improved to some degree. But there 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.

Replacing drop-tail policy with RED policy, a low average queue length is maintained. Using the drop-tail policy, Reno will unlimitedly increase its window size until buffer is full. However, using the RED, it will encounter packet loss earlier and more frequently. Thus Reno will release some bandwidth and allow Vegas to improve its performance.

Lai [13] points out that when packet losses occur, the Reno window is halved and the Vegas window is decreased by one. Thus the aggressive behavior of Reno is constrained. Hence, *MaxThresh* and *MinthThresh* should be set small enough to favor Vegas. How ever, if *MaxthThresh* is too low, too many packets are lost and the total throughput falls accordingly. He suggests *MinThresh* should be set low enough and *MaxThresh* should not be too large.

This gives us a good idea to use RED router to solve the fairness problem between TCP Reno and TCP Vegas. However, Lai [13] doesn't address the proper value to RED Router, and we should set the two thresholds dynamically to adapt the network environment, so we use the prediction of the number of connections [10] as a key to set two thresholds dynamically, and the fairness problem can be improved.

3.2 Characteristics of Random Early Detection (RED)

Random Early Detection (RED) is a queuing discipline for routers in which, when congestion is anticipated, packets are randomly dropped to alert the senders to slow down.

To understand the basic idea, consider a simple FIFO queue. Rather than wait for the queue to become completely full and then be forced to drop each arriving packet (the tail drop policy), we could decide to drop each arriving packet with some *drop probability* whenever the queue length exceeds certain *drop level*. This idea is called *early random drop*. The RED algorithm defines the details of how to monitor the queue length and when to drop a packet.

First, RED computes an average queue length using a weighted running average similar to the one used in the original TCP timeout computation. That is, **AvgLen** is computed as

AvgLen = (1 - Weight) * AvgLen + Weight * SampleLen

Where 0 < **Weight** < 1 (usually 0.002) and **SampleLen** is queue length each time a packet arrives. In most software implementations, the queue length is measured every time a new packet arrives at the gateway. In hardware, it might be calculated at some fixed sampling interval.

The reason for using an average queue length rather than an instantaneous one is that it more accurately captures the notion of congestion. Because of the busty nature of Internet traffic, queues can be filled up very quickly and then become empty again. If a queue is most of the time empty, then it's not appropriate to conclude that the router is congested and to tell the hosts to slow down. Thus, the weighted running average calculation tries to detect long-lived congestion by filtering out short-term changes in the queue length. We can think of the running average as a low-pass filter, where **Weight** determines the time constant of the filter.

Second, RED has two queue length thresholds that trigger certain activity: **MaxThreshold and MinThreshold.** When a packet arrives at the router, RED compares the current **AvgLen** with these two thresholds according to the following rules:

```
if AvgLen <= MinThreshold then
        → enqueue the packet
if MinThreshold < AvgLen < MaxThreshold then
        → calculate probability P
        → drop arriving packet with probability P
if ManThreshold <= AvgLen then
        → drop arriving packet
```

That is, if the average queue length is smaller than the lower threshold, no action is taken, while if the average queue length is larger than the upper threshold, the packet will always be dropped. If the average queue length is between the two thresholds, then the newly arriving packet will be dropped with probability **P**. This situation is depicted in **Figure 3.1**. The approximate relationship between **P** and **AvgLen** is shown in **Figure 3.2**. Note that the probability of drop increases slowly when **AvgLen** is between the two thresholds, reaching **MaxP** at the upper threshold, at which point it jumps to unity. The rationale behind this is that if **AvgLen** reaches the upper threshold, then the gentle approach (dropping few packets) is not working and a smoother transition from random dropping to complete dropping rather than the discontinuous approach shown here, may be appropriate.

Although **Figure 3.2** shows the probability of dropping as a function of only **AvgLen**, the situation is actually a little more complicated. In Fact, **P** is a function of both **AvgLen** and how long it has been since the last packet was dropped. Specifically, it is computed as follows:

TempP= MaxP * (AvgLen - MinThreshold)/(MaxThreshold - MinThreshold)

P = TempP / (1 - count * TempP)

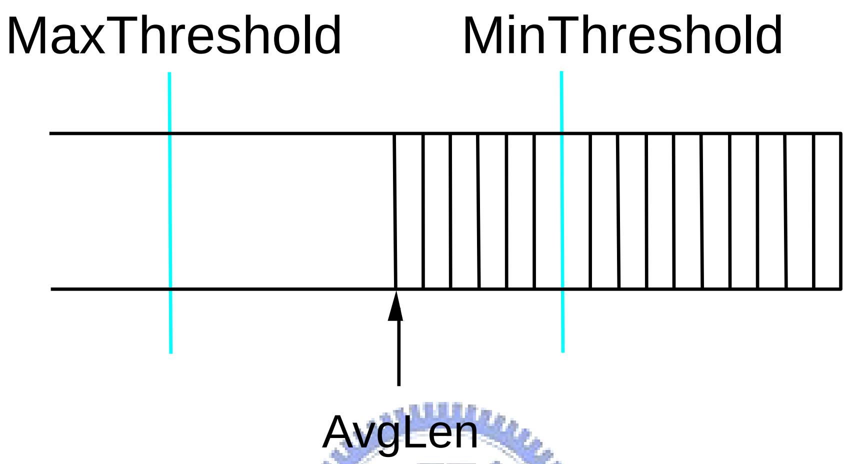


Figure 3.1 RED thresholds on a FIFO queue.

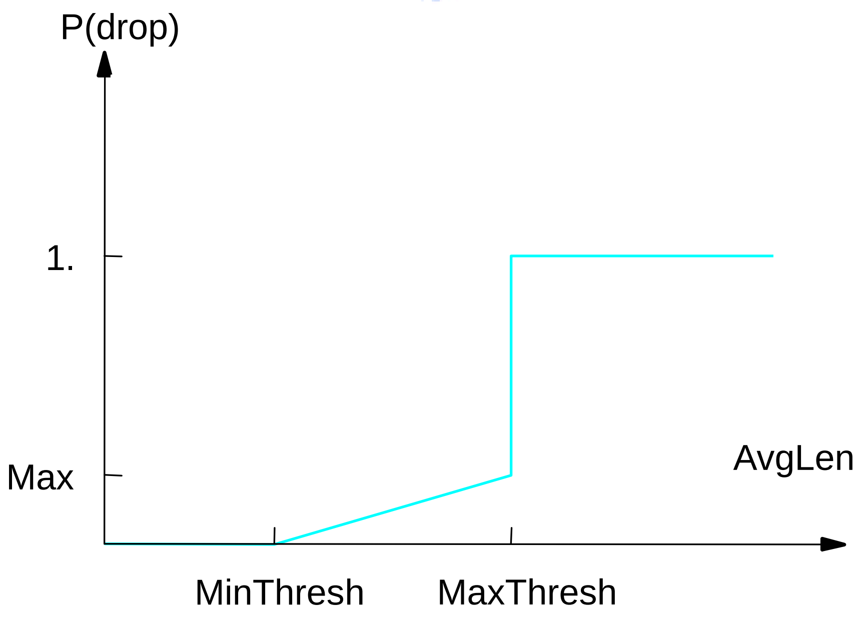


Figure 3.2 Drop probability function for RED.

TempP is the variable plotted on the *y*-axis in **Figure 3.2**. **count** keeps track of how many newly arrived packets have been queued (not dropped) while **AvgLen** has been between the two thresholds. **P** increases slowly as **count** increases, thereby making a drop more likely as the time increases since the last drop. This makes closely spaced drops relatively less likely than widely spaced drops. The extra step in calculating **P** was introduced by the inventors of RED when they observed. Without it, the packet drops were not well distributed over time, but instead tended to occur in clusters. Because packet arrivals from a certain connection are likely to arrive in bursts, this clustering of drops is likely to cause multiple drops in a single connection. This is not desirable became only one drop per round-trip time is needed to cause a connection to reduce its window size, whereas multiple drops might cause it back into slow start.

Hopefully, if RED drops a small percentage of packets when **AvgLen** exceeds **MinThreshold**, the effect will be TCP connections reducing their window sizes, this in turn will reduce packets arriving rate at the router. All going well, **AvgLen** will then decrease and congestion can be avoided. The queue length can be kept short, while throughput remains high since few packets are dropped.

Because RED is operating based on a queue length averaged over time, it is possible for the instantaneous queue length to be much longer than **AvgLen.** In this case, if a packet arrives and there is nowhere to store it, it will be dropped. When this happens, Red is operating in tail drop mode. One of the goals of RED is to prevent tail drop behavior if possible.

The random nature of RED confers an interesting property on the algorithm. Because RED drops packets randomly, the probability that RED decides to drop a particular flow's packets is roughly proportional to the share of the bandwidth that flow is currently getting at that router. This is because a flow that is sending a relatively large number of packets is providing more candidates for random dropping. Thus there is some sense of fair resource allocation built in RED, although it is by no means precise.

Consider the setting of the two thresholds, **MaxThreshold and MinThreshold.** If the traffic is fairly busty, then **MinThreshold** should be sufficiently large to allow the link utilization to be maintained at an acceptabe high level. Also, the difference between the two thresholds should be larger than the typical increase in the calculated average queue length in one RTT. Setting **MaxThreshold** to twice **MinThreshold** seems to be a reasonable rule of thumb given the traffic mix on today's Internet. In addition, since we expect the average queue length to hover between the two thresholds during periods of high load, there should be enough free buffer space above **MaxThreshold** to absorb the natural bursts that occur in Internet traffic without forcing the router to enter tail drop mode.

Note that a fair amount of analysis has gone into setting the various RED parameters-for example, **MaxThreshold**, **MinThreshold**, **MaxP**, and **Weight**- all in the name of optimizing the power function (throughput-to-delay ratio). The performance of these parameters has also been confirmed through simulation, and the algorithm has been shown not to be overly sensitive to them. It is important to keep in mind, however, that all of this analysis and simulation hinges on a particular characterization of the network workload. The real contribution of RED is a mechanism by which the router can more accurately manage its queue length. Defining precisely what constitutes an optimal queue length depends on the traffic mix and is still a subject of research with real information now being gathered from operational deployment of RED in the Internet.

3.3 Effect with or without RED Router

Before presenting our RED algorithm, we would like to demonstrate a simple scenario to show the effect between two connections, TCP Reno and TCP Vegas, if the RED router is used.

The network simulator (ns) [17] developed by the Lawrence Berkeley Laboratory is used to run our simulations. This simulator is often used in TCP-related studies [4, 9]. According to **Figure 3.3**, a two-router configuration is used as the network topology, where S, D, and R denote 'Source', 'Destination', and 'Router', respectively. The bottleneck link between two neighboring routers is 40 ms in delay and 1.5 Mbps capacity. The access links between sources and neighboring routers or receivers and neighboring routers are 0.4 ms in delay and 10 Mbps capacity. Each packet is fixed at a length of 1000 bytes. The buffer size in the bottleneck link router is set to 20 packets. The Minimum threshold and the Maximum threshold we used in RED routers are 4 (packets) and 6 (packets). The simulation time lasts for 100 seconds.

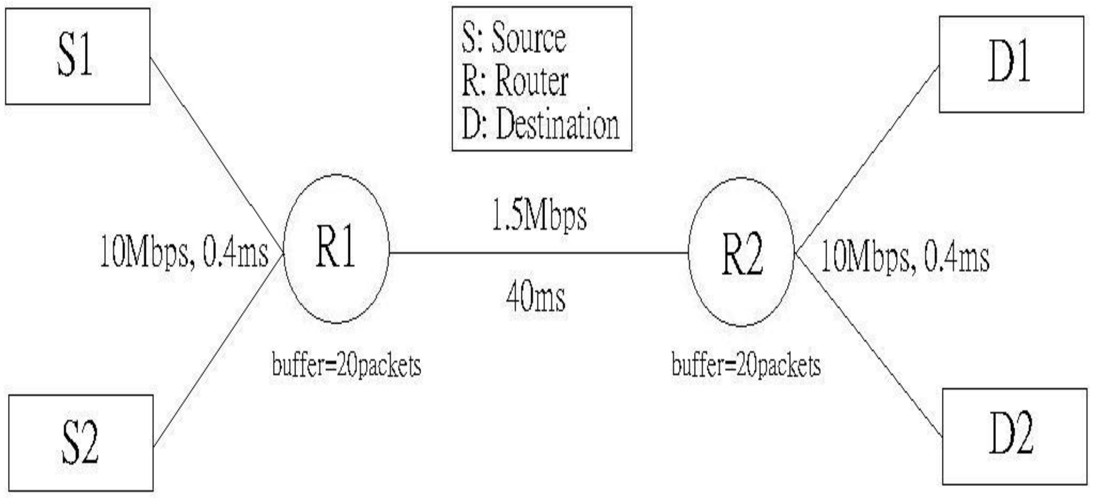


Figure 3.3 Network topology.

As it is shown in **Table 3.1**, two versions of TCP, where V and R stand for Vegas and Reno respectively, are used in connection1 (S1-R1) and connection2 (S2-R2) to show the effect between Drop-tail Router and RED Router. The Table summarizes the total *Arrive* and *Drop* in packets, and the throughput of *Source 1*, *Source 2* and *total* in Kbps. The *Fairness* between TCP Vegas and TCP Reno is calculated by which the throughput in Vegas is divided by the throughput in Reno. The ideal throughput is about 1500 Kbps, so the network *Utilization* is count by 1500 divides *Total* throughput.

Undoubtedly, we can see the TCP Vegas works well in Homogeneous network. There is no packet drop both in Drop-tail routers and RED routers and the bottleneck link can be full utilized (where 1496 rather than 1500, it is because for slow-start phase and we have to wait the network stable).

However, the throughput in TCP Vegas is dramatically decreased when competing with TCP Reno. The reason is that TCP Reno keep increasing its congestion window until packet loss occurs while TCP Vegas only keep amount of data packets in bottleneck link routers.

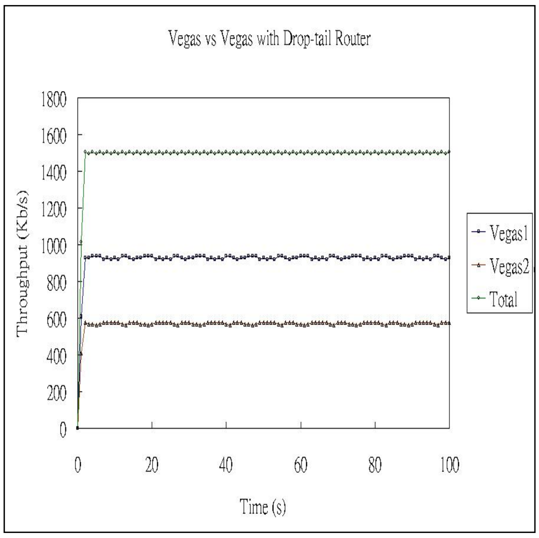
	Drop-tail Router			RED Router		
	V vs V	V vs R	R vs R	V vs V	V vs R	R vs R
Arrive	18711	18139	17918	18711	18291	17333
Drop	0	62	140	0	151	211
S1	926	323	711	926	646	725
S2	570	1120	708	570	801	636
Total	1496	1443	1419	1496	1448	1362
Fairness	X	0.28886	X	X	0.80604	X
Utilization	0.99781	0.96256	0.94613	0.99781	0.96544	0.90826

Table 3.1 Comparison between two connections with/without RED Router.

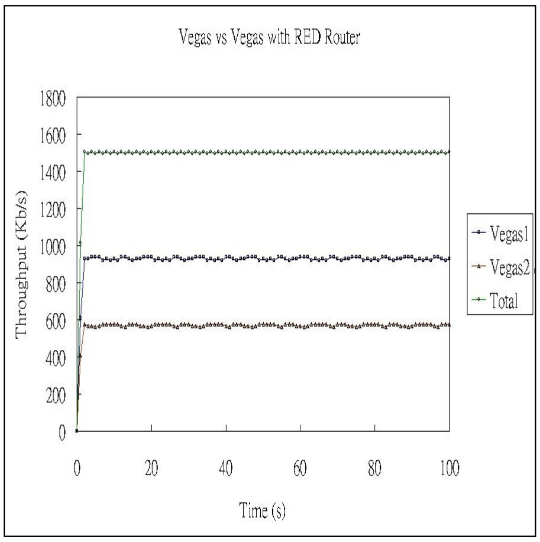
It can be seen obviously that whether the Drop-tail routers and RED routers are used or not, if the TCP Reno exists, the packet loss occurs inevitably. The more TCP Reno connections exist, the more packets loss in bottleneck link routers. The link utility gradually decreases when the number of TCP Reno connections increase and it will lead to more packet drops. The Minimum threshold and the Maximum threshold we used in RED routers are set to 4 (packets) and 6 (packets). This is because among several experiments, we observe a better result based on both the Total Throughput and the Fairness between TCP Vegas and TCP Reno. It is sure that if we set these two thresholds smaller, TCP Vegas gets more throughputs. When two connections competes the buffers in bottleneck link router, the one get more buffer size, it can increase its congestion window size, and get more bandwidth. So, lower threshold can prevent TCP Reno from inflating its window unnecessarily, and release more bandwidth shared with TCP Vegas.

However, there is a tradeoff between network utility and fairness. The lower threshold means higher probability of packet drop and lower performance. It leaves us a good question, how to set the RED router parameters properly to fully utilize the network and give TCP Vegas connections higher bandwidth, even higher than TCP Reno connections.

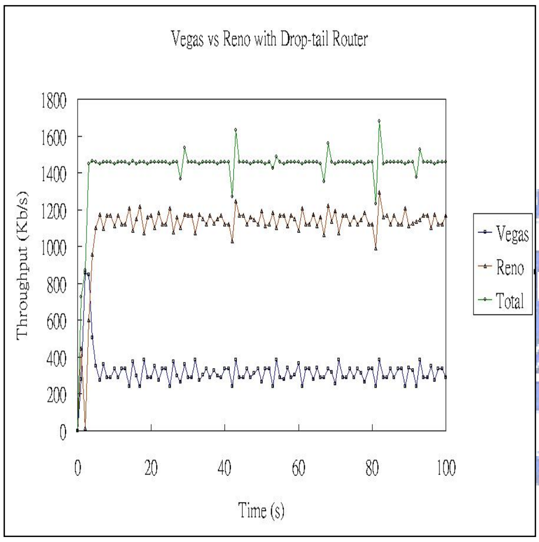
Figure 3.4 shows the throughput comparison between TCP Vegas and TCP Reno with Drop-tail router or RED router. There is no difference between **Fig. 3.4(a)** and **Fig. 3.4(b)** when two connections are TCP Vegas. The throughput difference in TCP Vegas and TCP Reno gets closer when a Drop-tail router in Fig. 3.4(c) is replaced by RED router in **Fig. 3.4(d)**. When there are only TCP Reno connections exist, the throughput in **Fig. 3.4(e)** is more stable than in **Fig. 3.4(f)**. The fact is that TCP Reno keeps increasing its window size until its buffer in bottleneck link router is filled up, so the more bottleneck link buffer in router, the longer TCP Reno Saw-tooth cycle.



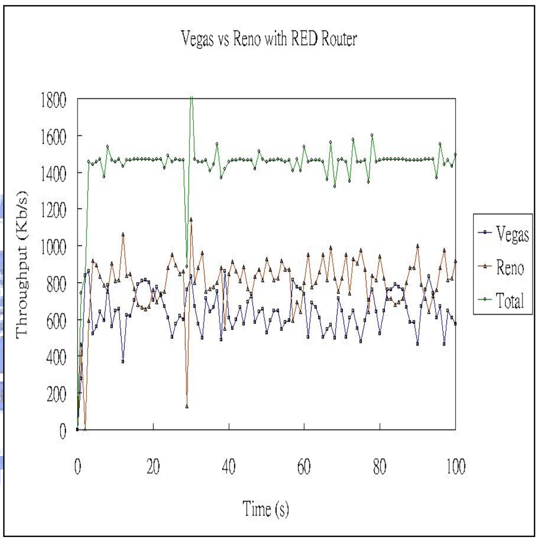
(a)



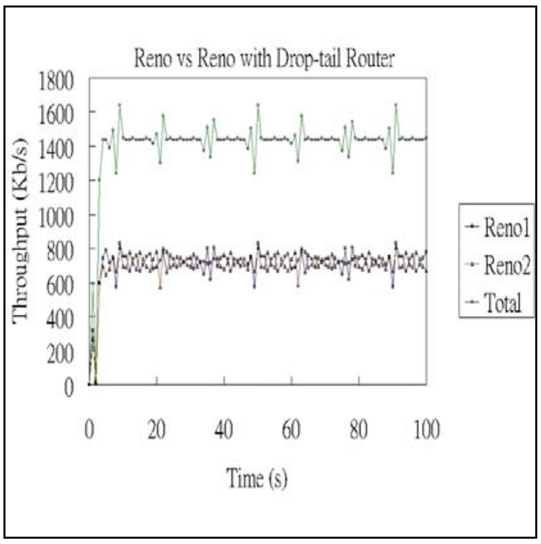
(b)



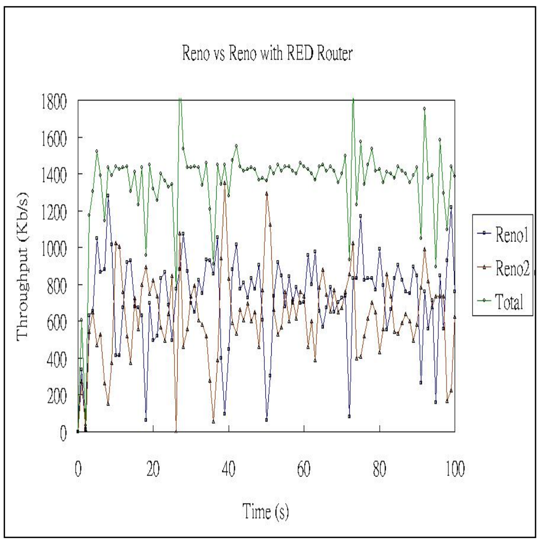
(c)



(d)



(e)



(f)

Figure 3.4 Throughput comparison between Drop-tail Routers and RED Routers.

3.4 Proposed RED Router based Algorithm

On the previous section, we have seen that the RED router will favor TCP Vegas connections, but how to set these two thresholds (*MinThresh* and *MaxThresh*) properly still needs investigation. Before introducing our RED Router based Algorithm, I would like to show some phenomena to describe the algorithm.

The network topology is the same as **Fig. 3.3** in Sec. 3.3. **Figure 3.5** shows the throughput varying depending on the buffer size B. It is clear that the fairness problem results from the buffer size when two version of TCP competing the same bottleneck link. So, we have to prevent the buffer from mis-behaved connections by RED Router.

Figure 3.6 is the same experiment in Sec. 3.3. The queue size in bottleneck link routers tell us that 5 packets in buffer is enough for 2 TCP Vegas connections used, while the rest of buffer will give TCP Reno a chance to get more bandwidth.

Figure 3.7 is the same experiment as that in Sec. 3.3. The *CWND* in **Fig. 3.7(d)** shows that RED router will enable TCP Vegas to compete with TCP Reno.

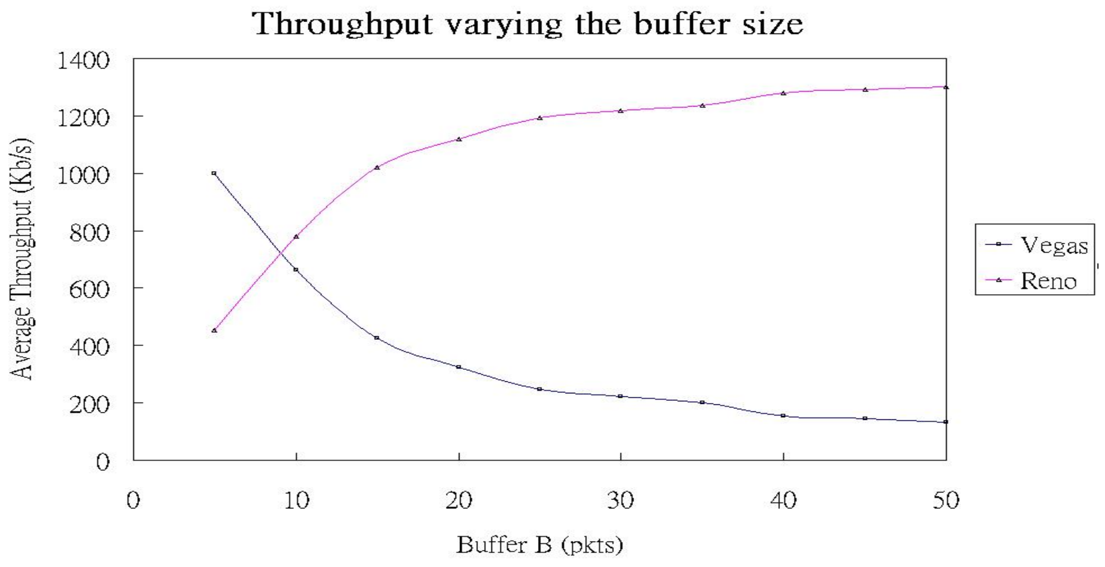
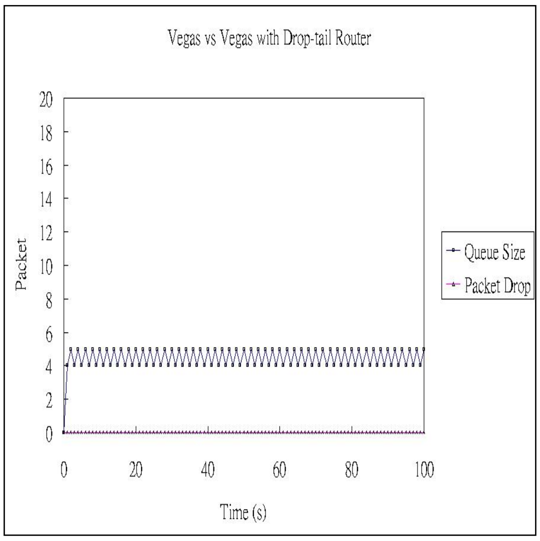
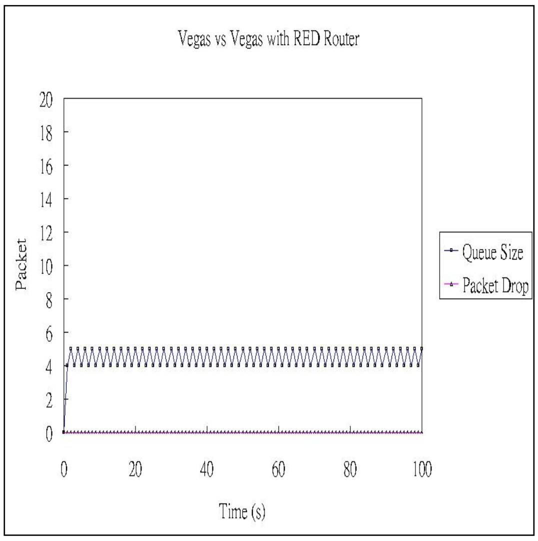
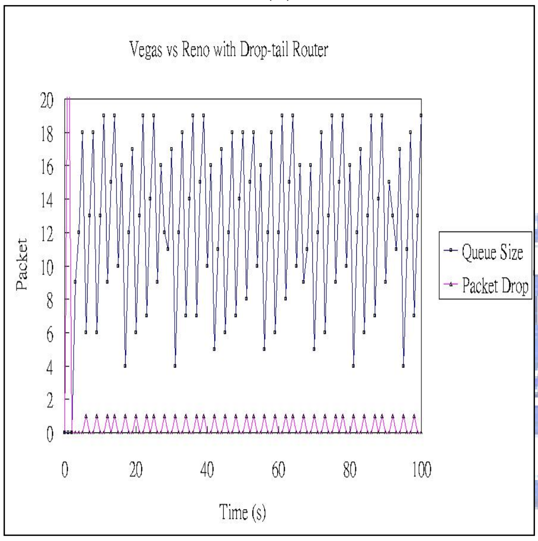


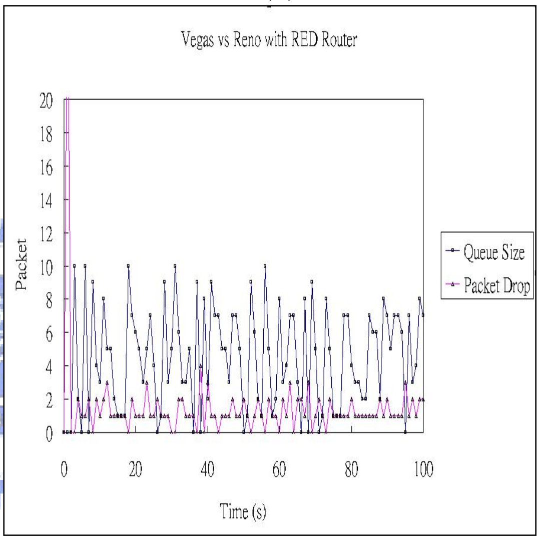
Figure 3.5 Throughput varying the buffer size B (bandwidth = 1500 Kb/s).



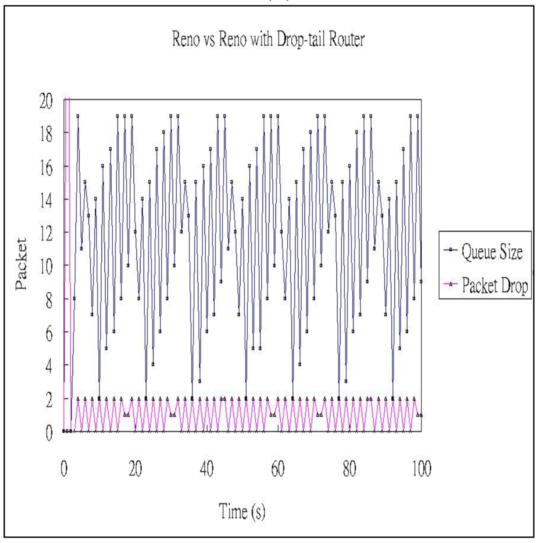


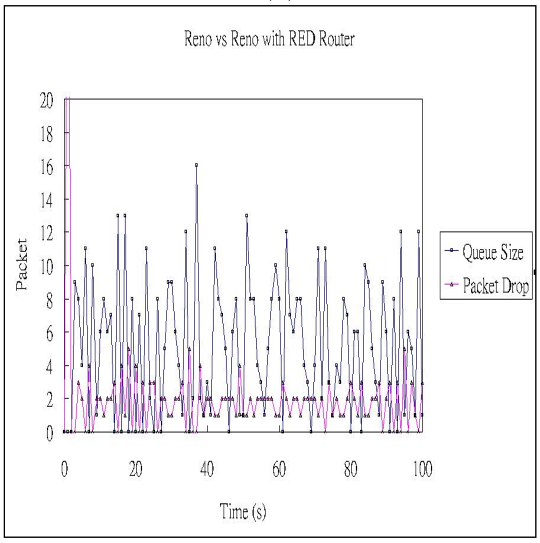
(a) (b)





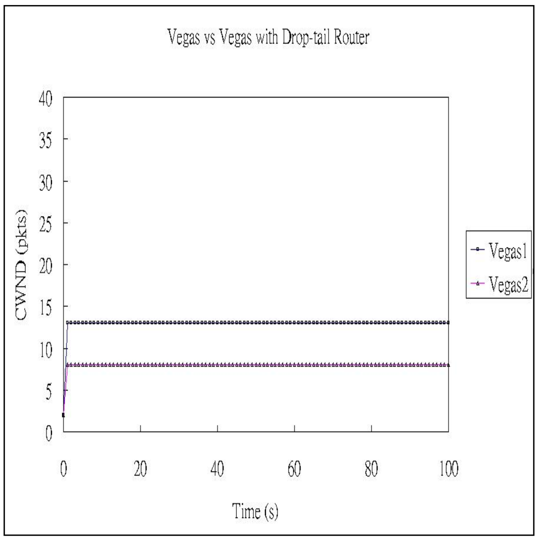
(c) (d)



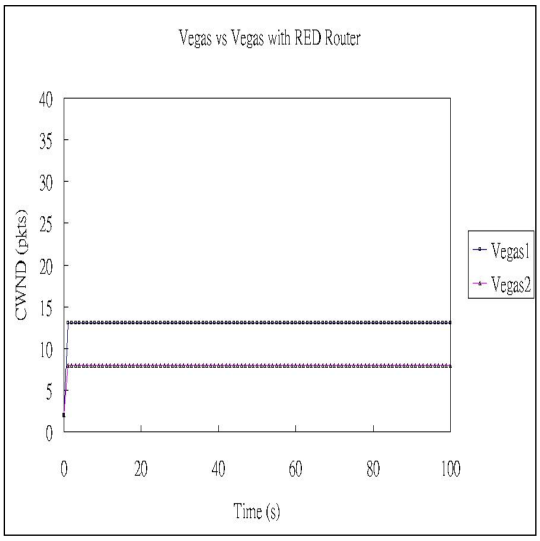


(e) (f)

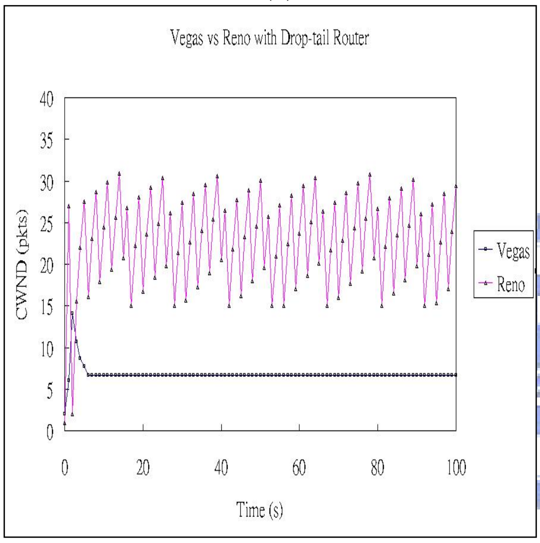
Figure 3.6 Queue occupancy comparison between Drop-tail Routers and RED Routers.



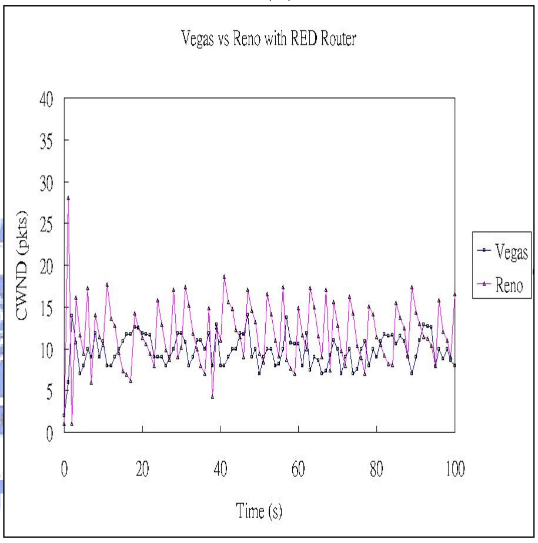
(a)



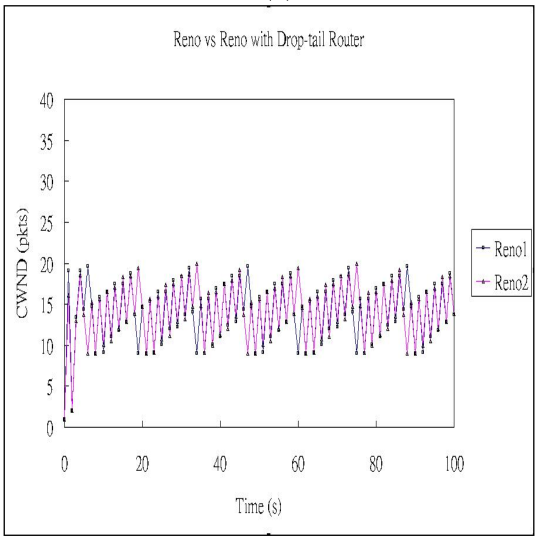
(b)



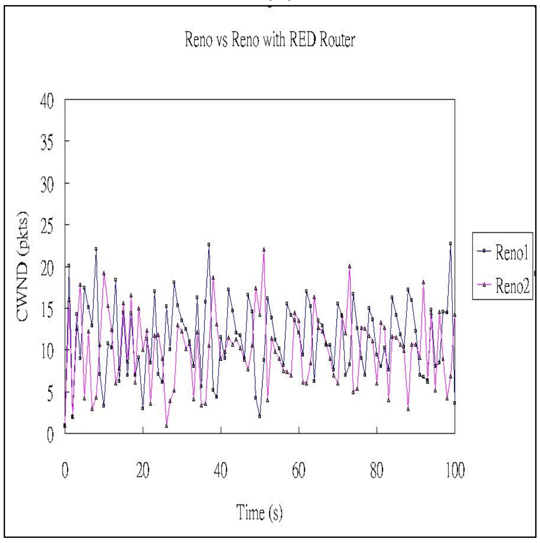
(c)



(d)



(e)



(f)

Figure 3.7 Congestion Window comparison between Drop-tail Router and RED Router.

Modification to RED Router

In order to realize our algorithm, we need some modification to the parameters of RED routers. These parameters are show as follows:

n : number of connections in router we predict by [10]

B : Buffer size in bottleneck link router

β : TCP Vegas parameter

MinThresh : The lower bound in RED Router for queue length threshold

MaxThresh : The upper bound in RED Router for queue length threshold

Time Interval for RED Router to compute the number of connections:

Every time a new packet arrives at the router.

```
If ( B > βn ) MinThresh = βn;
    else MinThresh = B;
If ( B > (β+1)n ) MaxThresh = (β+1)n;
    else MaxThresh = B;
```

The reason we use βn and (β+1)n for *MinThresh* and *MaxThresh* is described as follows.

1. Since the buffer size in bottleneck link routers will be mis-used by TCP Reno when two versions of TCP compete with each other, after examining the packets queuing in the bottleneck router, βn is enough for TCP Vegas connections used.
2. The value of α and β in TCP Vegas means the number of packets that will be queued in the bottleneck router. The default value of α and β in TCP Vegas is 1 and 3. So, we choose the max value 3 for *MinThresh*, and 4 for *MaxThresh*.
3. The number of connections is predicted by the mechanism in [10]. There maybe some inaccurate predictions, like overestimation or underestimation, and we have to eliminate the problem because underestimation leads to packet losses.

4. The trade-off between the fairness ratio and the throughput is still an open issue, because lower threshold in RED Router will result in smaller throughput.

5. In Lai's paper [13], he states that estimate the prerequisite for TCP Vegas connections to achieve stability. When the formula holds, TCP Vegas can achieve a fair and stable share of bandwidth. If it is not held, instability possibly occurs. So, we have to choose *MinThresh* a little larger than β to avoid instability caused by lack of buffer.

(buffer size in packets / number of Vegas connections) $\geqq \beta$

We will see more simulations to evaluate our proposed approach based on RED routers in Chapter 4.

CHAPTER 4

Performance Evaluation

In this Chapter, we compare the performance between TCP Vegas and TCP Reno by using network simulator ns-2.27. We first show that the performance result before/after our RED router-based algorithm is used, then inaccurate prediction in connections is discussed and finally we will show how to deploy our RED router based algorithm in the current Internet gradually.

4.1 Fairness Improvement

In order to validate the analytical model, we carried out simulations under ns [17], the simulation tool is widely used in the networking research community.

Figure 4.1 shows the network used in this section. It consists of *n* sender hosts using TCP Reno and TCP Vegas, n receiver hosts, two intermediate routers, and links connecting the router and the sender/receiver hosts. The bandwidth of each link between the sender/receiver hosts and the router is 50 Mbps. The bandwidth of the bottleneck link between routers is *w* Mbps. The size of buffer at the router is 100 packets. The propagation delay between the sender/receiver hosts and the router and that between the routers is represented by 1ms and 40 ms, respectively. As the scheduling discipline at the router buffer, we consider drop-tail and RED algorithm.

In our first experiment, we set bottleneck link bandwidth *w* = 1.5 Mbps and there are 5 connections, so we set the Minimum threshold and the Maximum threshold used in RED routers 15 (packets) and 20 (packets), respective, according to our RED router algorithm. Each packet is fixed at a length of 1000 bytes. All sources start at time 0 and the simulation lasts 100 s.

As shown in **Table 4.1**, two versions of TCP, V and R, standing for Vegas and Reno respectively, are used in 5 connections (Sx-Rx) to show the effect between Drop-tail Router and RED Router. The Table summarizes the total *Arrival* and *Drop* in packets, and the throughput of *Source 1*, *Source 2, Source 3*, *Source 4, Source 5* and *total* in Kbps. The average throughput of Vegas and Reno is computed in Kbps. The *Fairness* between TCP Vegas and TCP Reno is calculated by which the mean throughput in Vegas is divided by the average throughput in Reno. The throughput is 1500 Kbps, so the *Utilization* is calculated as *Total* throughput divided by 1500.

Undoubtedly, TCP Vegas works well in Homogeneous network. There is no packet drop both in Drop-tail routers and RED routers, and the bottleneck link can be fully utilized (where 1499 rather than 1500, it is because of slow-start phase and we have to wait the network until it is stable).

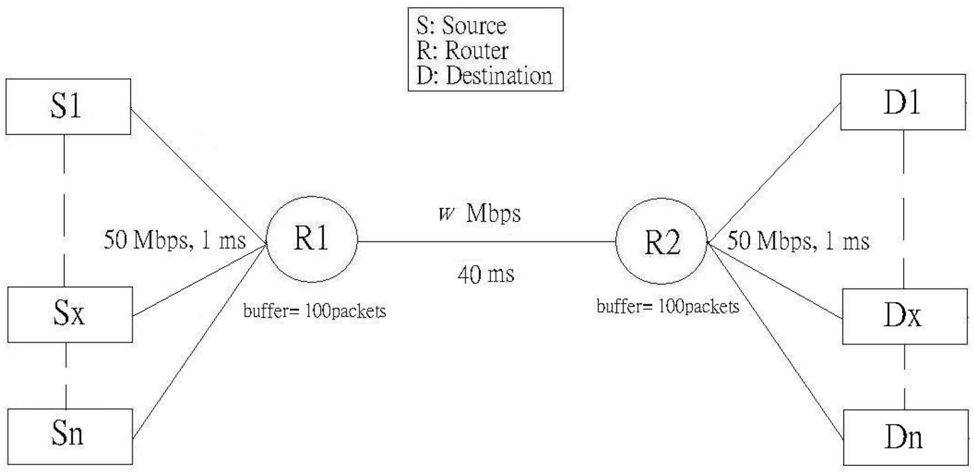


Figure 4.1 Network topology.

However, the throughput in TCP Vegas is dramatically decreased when a TCP Reno connection exists. The reason is that TCP Reno keeps increasing its congestion window until packet loss occurs, while TCP Vegas only keeps a amount of data packets in the bottleneck link router.

We can see whether we use the Drop-tail routers or RED routers, if the TCP Reno exists, the packet loss occurs inevitable. The more TCP Reno connections exist, the more packets loss in the bottleneck link routers. The link utilization gradually decreases when the number of TCP Reno connections increase because of more packet drops.

It can be seen obviously that TCP Vegas is fairly treated when RED router is used. The performance improvement is about 230~259% (which is counted based on dividing TCP Vegas throughput in Drop-tail Router by that in RED router).

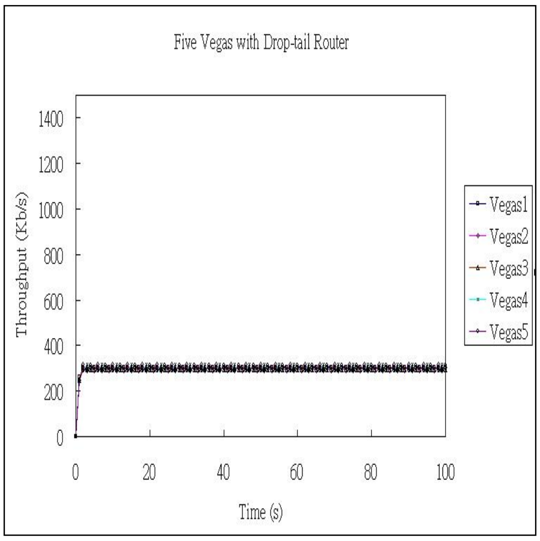
	Drop-tail Router			RED Router		
	5V	4V1R	1V4R	5V	4V1R	1V4R
Arrival	18750	18320	18159	18750	18370	18464
Drop	0	87	152	0	166	498
S1	300	80	81	300	210	187
S2	300	81	346	300	206	306
S3	299	79	338	299	191	273
S4	299	79	340	299	222	317
S5	299	1131	326	299	620	337
Vegas	299	80	81	299	207	187
Reno	X	1131	338	X	620	308
Fairness	X	0.07073	0.23964	X	0.33387	0.60714
Total	1499	1451	1431	1499	1450	1421
Utilization	0.99973	0.9672	0.95424	0.99973	0.96661	0.94704

Table 4.1 Comparison between five connections with/without RED Router.

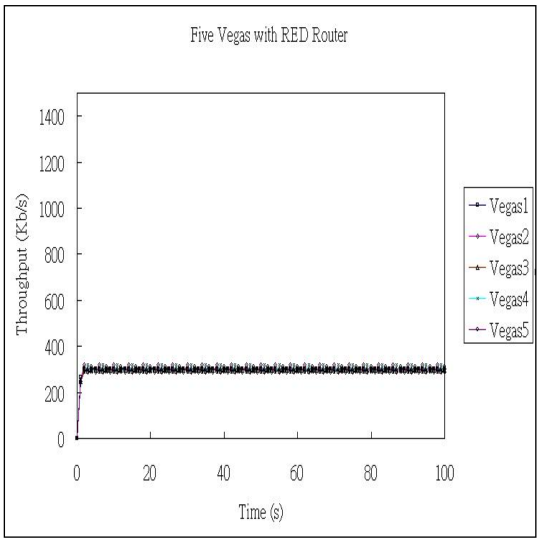
Figure 4.2 shows the throughput comparison between TCP Vegas and TCP Reno with Drop-tail router and RED router. There is no difference between **Fig. 4.2(a)** and **Fig. 4.2(b)** when five connections are all TCP Vegas. This is because the queue size in the bottleneck link router is always around 13 packets as shown in **Fig.4.4(a)** and **Fig.4.4(b),** and two thresholds ,*MinThresh* and *MaxThresh* set by our RED algorithm is 15 and 20 respectively. Therefore, there won't be any packet loss occur, and there is no side effect when we use RED router. We can see that TCP Reno takes almost all bandwidth in Drop-tail routers and leaves only small bandwidth to share with TCP Vegas when there is only one TCP Reno connection in **Fig. 4.2(c)**. The throughput difference in TCP Vegas and TCP Reno gets closer when Drop-tail routers in **Fig. 4.2(c)** are replaced by RED routers in **Fig. 4.2(d)**. The TCP Vegas connection is still in low throughput while the other TCP Reno connections share the most bandwidth in **Fig. 4.2(e).** The fairness will be improved with RED router in **Fig. 4.2(f)**.

Figure 4.3 demonstrates the queuing size in the bottleneck link router and the time that packet drop occurs. TCP Vegas behaves well in **Fig. 4.3(a)** and **Fig. 4.3(b)** because there is no packet drop and the queue length keeps at a low level. The saw-tooth cycle in **Fig. 4.3(c)** and **Fig. 4.3(e)** occurs when there TCP Reno connections exist. This is because that TCP Reno increases its congestion size until the buffer in bottleneck link router used up. TCP Reno will have early packet loss with RED routers in **Fig. 4.3(d)** and **Fig. 4.3(f)**.

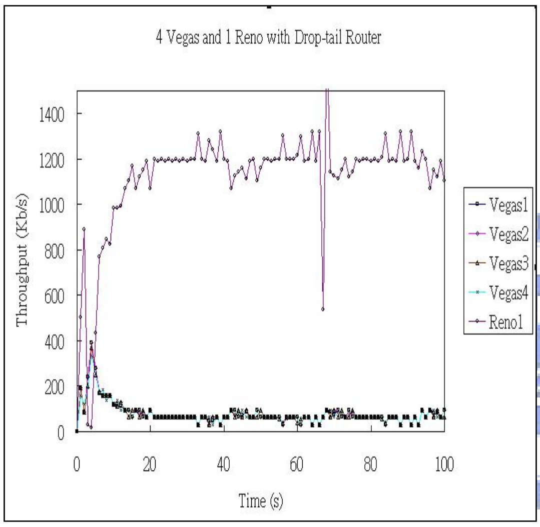
Figure 4.4 shows the CWND variation in TCP Vegas and TCP Reno with Drop-tail router and RED router. Again, there is no difference between **Fig. 4.4(a)** and **Fig. 4.4(b)**. In **Fig. 4.4(c)** and **Fig. 4.4(e)**, there are *n* TCP Reno connections, so *n* saw-tooth cycles are shown, while all TCP Vegas' CWND are always kept at a low level. It will be improved in **Fig. 4.4(d)** and **Fig. 4.4(f)** when RED router is used.



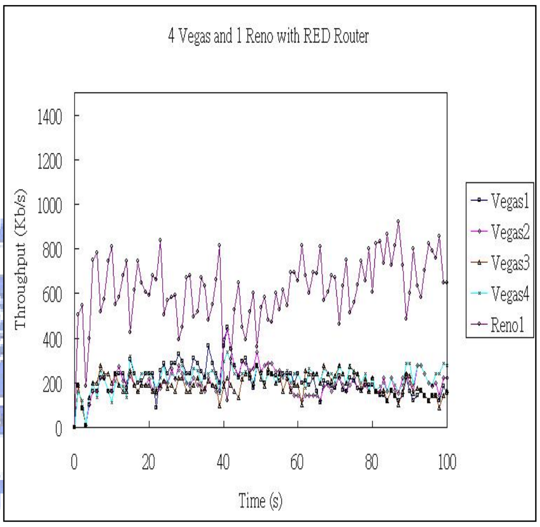
(a)



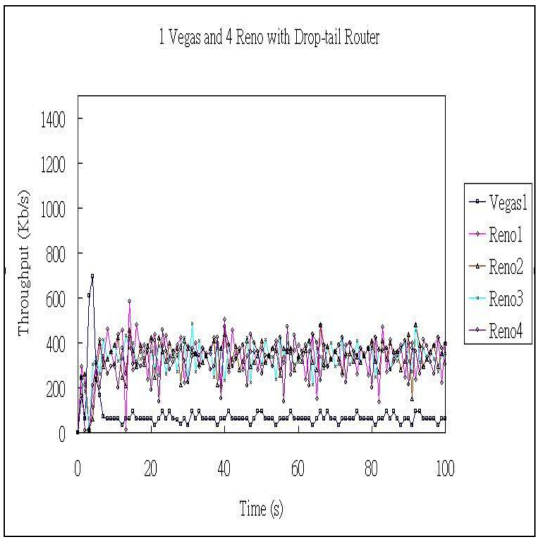
(b)



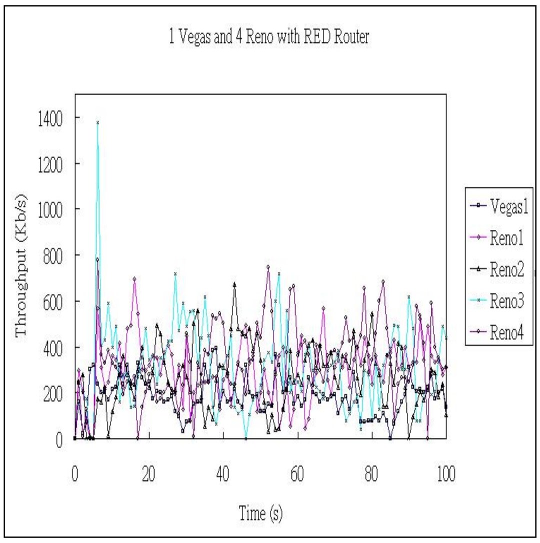
(c)



(d)

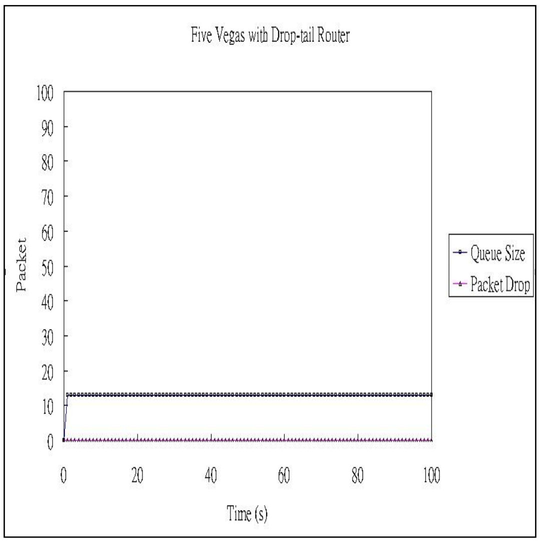


(e)

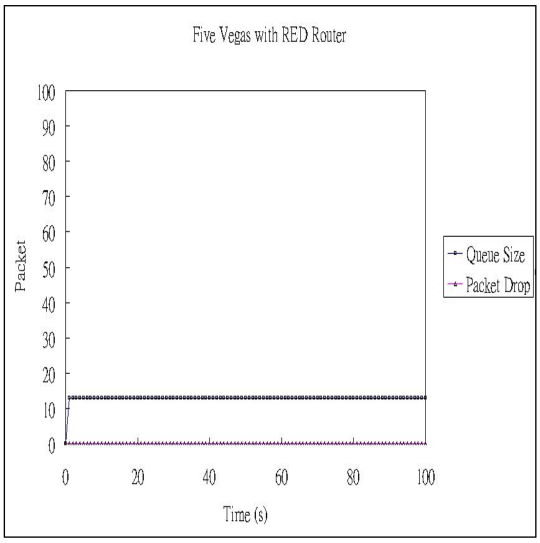


(f)

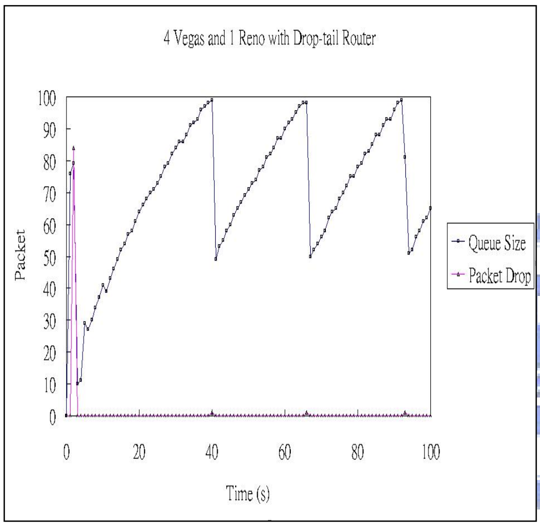
Figure 4.2 Throughput comparison between Drop-tail Routers and RED Routers.



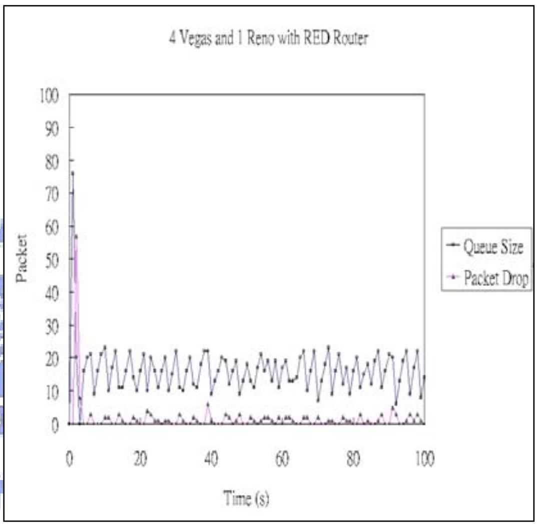
(a)



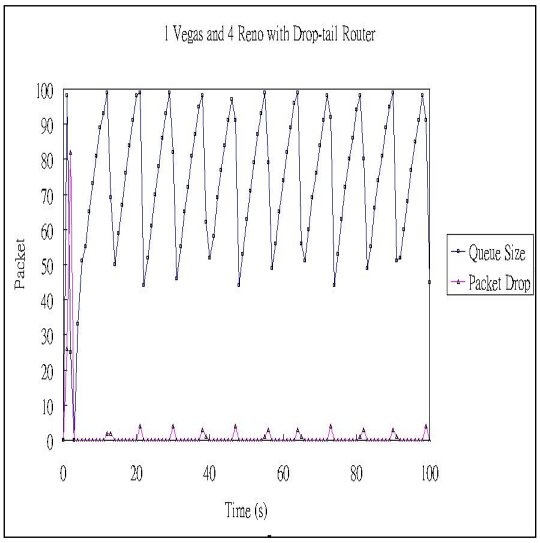
(b)



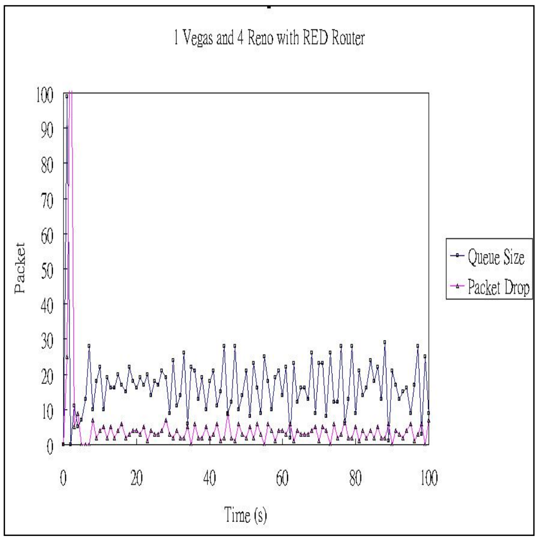
(c)



(d)

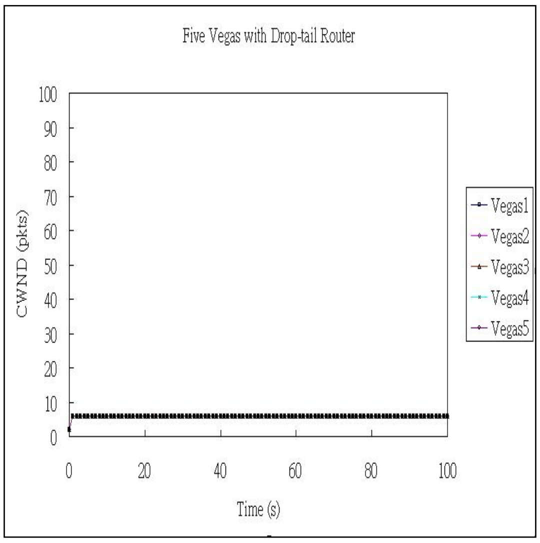


(e)

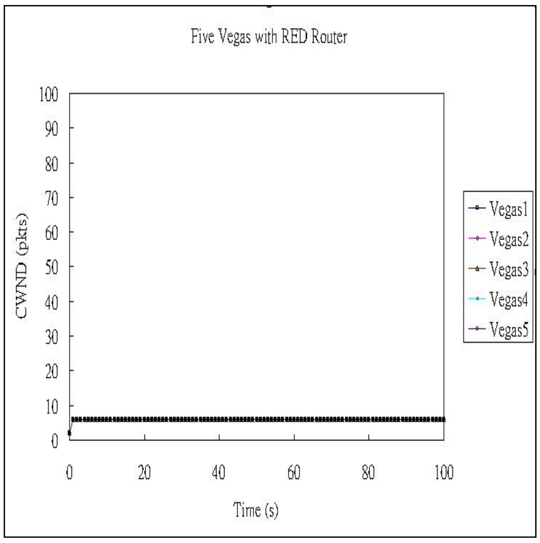


(f)

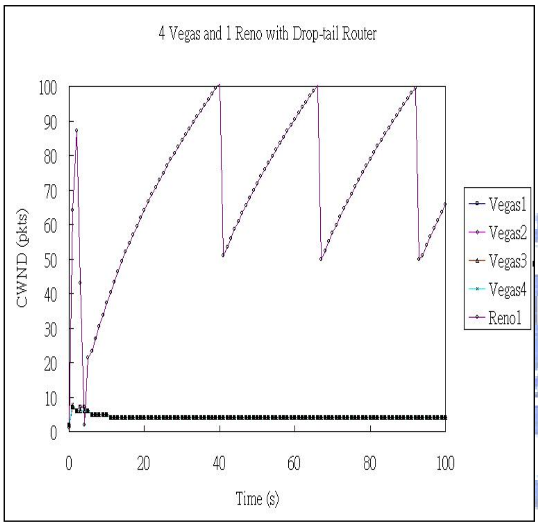
Figure 4.3 Queue occupancy comparison between Drop-tail Routers and RED Routers.



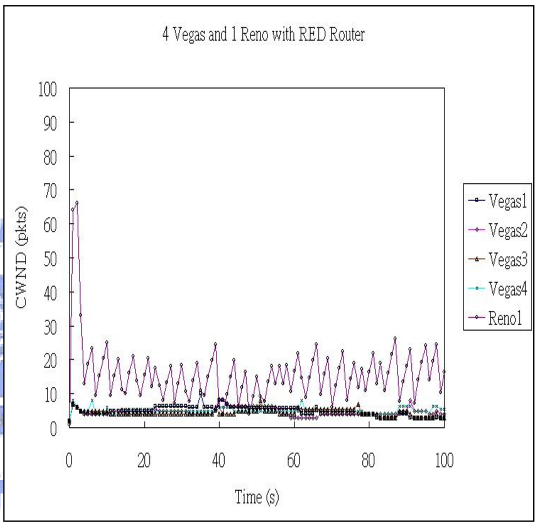
(a)



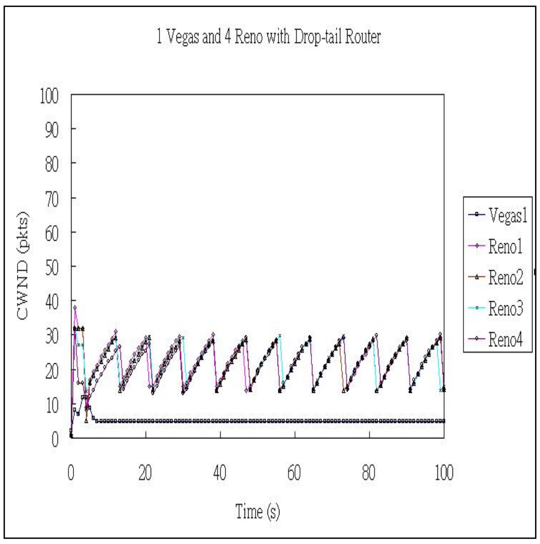
(b)



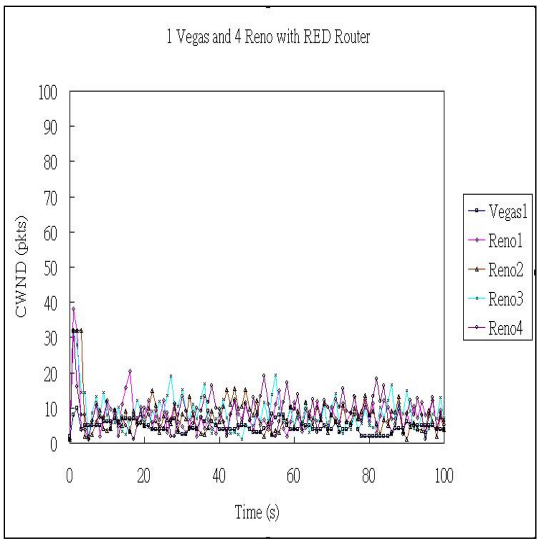
(c)



(d)



(e)



(f)

Figure 4.4 Congestion Window comparison between Drop-tail Router and RED Router.

In our second experiment, we set bottleneck link bandwidth *w* = 2 Mbps. There are 10 connections, so we set the Minimum threshold and the Maximum threshold used in RED routers 30 (packets) and 40 (packets), respectively, according to our RED router algorithm. All sources start at time 0 and the simulation lasts 100 s.

As shown in **Table 4.2,** V and R stand for Vegas and Reno respectively. The table summarizes the *Arrival* and *Drop* of packets, and the individual and total throughput in Kbps. The average throughput of Vegas and Reno is computed in Kbps. The *Fairness* between TCP Vegas and TCP Reno is calculated based on dividing the mean throughput in Vegas by the average throughput in Reno. The ideal throughput is 2000 Kbps, so the *Utilization* is calculated based on dividing the *Total* throughput by 2000.

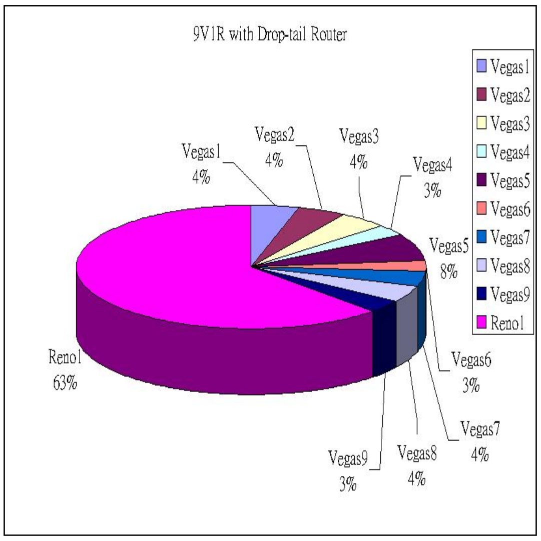
	Drop-tail Router			RED Router		
	9V1R	5V5R	1V9R	9V1R	5V5R	1V9R
Arrival	24531	24469	24354	24792	24828	24557
Drop	66	166	337	143	482	707
S1	87	86	91	128	108	95
S2	87	86	197	150	96	194
S3	87	86	183	135	107	210
S4	60	89	210	93	152	194
S5	131	89	210	172	135	196
S6	57	300	196	130	282	214
S7	74	300	222	181	258	191
S8	87	300	205	130	258	193
S9	64	299	203	80	250	199
S10	1213	300	194	766	284	215
Vegas	82	87	91	133	120	95
Reno	1213	300	202	766	266	200
Fairness	0.0676	0.29067	0.4505	0.17392	0.45113	0.475
Total	1951	1937	1910	1965	1930	1892
Utilization	0.97564	0.96828	0.95524	0.98228	0.96504	0.94616

Table 4.2 Comparison between ten connections with/without RED Router.

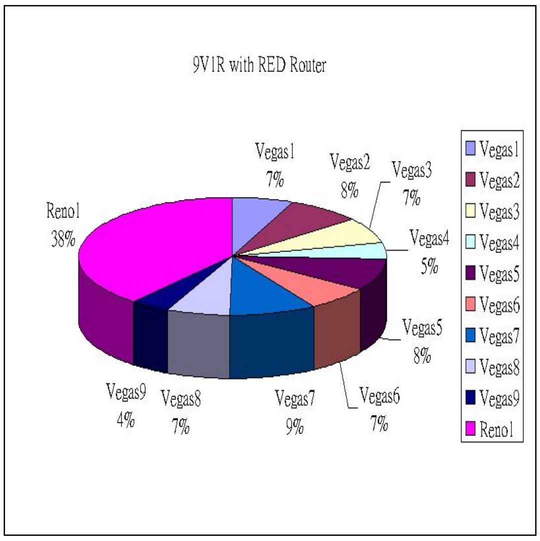
It is the same result that the TCP Reno takes most bandwidth from TCP Vegas, but the situation will be improved when RED Router is used. There are more total packet arrivals and more packet drops in RED Router, it is because the RED Router early drop these mis-behaved flow packet (like TCP Reno) to reduce its aggressive increasing window size. This will release some bandwidth for TCP Vegas, so the improvement in this experiment is about 6% to 157% based on our network topology.

Figure 4.5 demonstrates the throughput comparison between TCP Vegas and TCP Reno with Drop-tail router or RED router. **Fig. 4.5(a)** tells us that a TCP Reno connection has more than half of bandwidth, while other TCP Vegas connections share the rest of bandwidth. We can see that in **Fig. 4.5(b)**, the TCP Reno connection will be limited by RED router, and let the other TCP Vegas connections get more bandwidth. There are five TCP Vegas and five TCP Reno connections in **Fig. 4.5(c)** and **Fig. 4.5(d)**, and one TCP Vegas and ten TCP Reno connections in **Fig. 4.5(e)** and **Fig. 4.5(f)**. We will find that when TCP Reno connections increase, the bandwidth of TCP Vegas connections decrease gradually in RED router, but increase slightly in drop-tail router. This is because when more TCP Reno connections in drop-tail router, they will compete with each other, and leave more bandwidth for TCP Vegas connections, while RED router drops TCP Reno connections packets early.

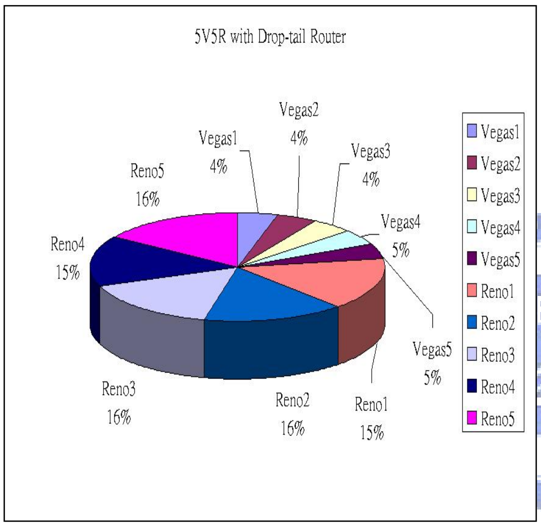
In the above two experiments, we will see that if we can control the *MinThresh* and *MaxThresh* parameter properly, the fairness problem in TCP Vegas and in TCP Reno will be solved. **Fig. 4.5(a)** and **Fig. 4.5(b)** show that five TCP connections will keep about 13 packets in Router whether drop-tail or RED queuing discipline is used, while ten TCP connections will keep about 24 packets in our second experiment. That is the reason why we choose 3n and 4n for the *MinThresh* and *MaxThresh* parameter, respectively. This will prevent RED router from miss-dropping TCP Vegas packets while there only TCP Vegas connections exist.



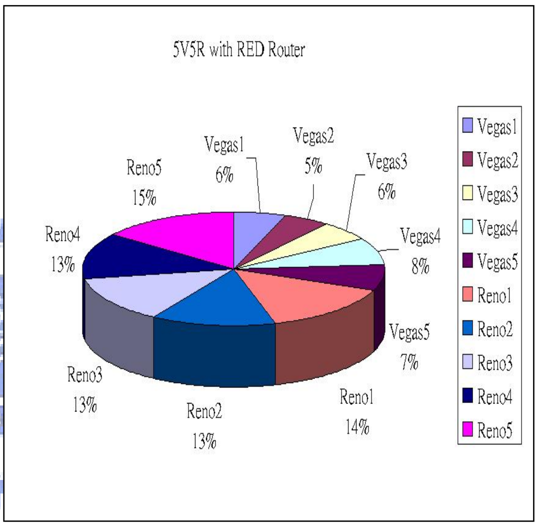
(a)



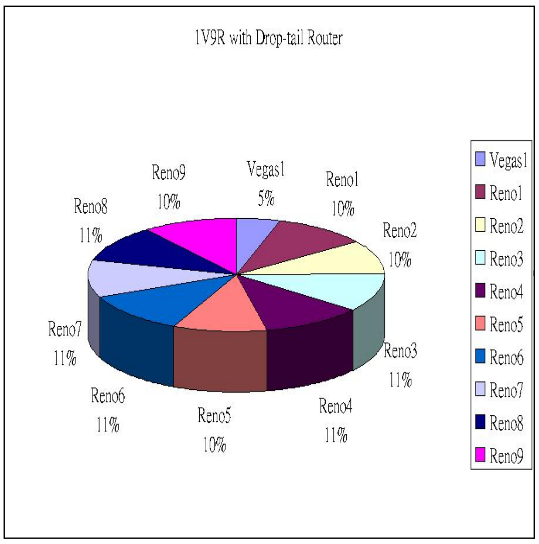
(b)



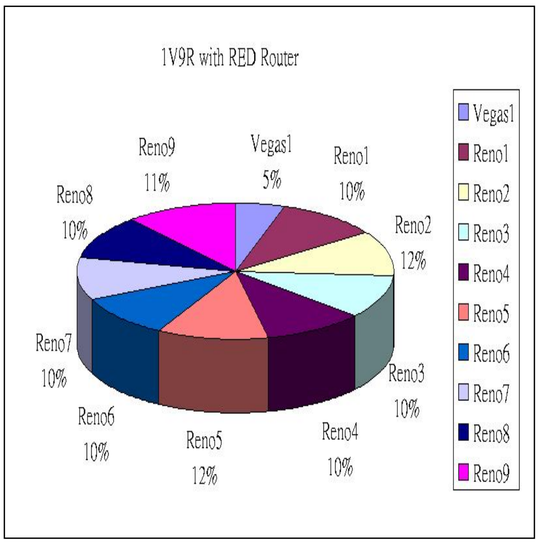
(c)



(d)



(e)



(f)

Figure 4.5 Throughput Ratio with Drop-tail Router or RED Router

4.2 Evaluation of Inaccurate Prediction

Lee [10] proposes an approach which is able to estimate the number of flows passing through routers. However, the prediction maybe not so precise to estimate the number of connections, because of the prediction time interval we choose and the average queue length RED router calculate. In this section, we focus on what may happen if we mis-predict the number of connections.

The network topology we take in our experiment is same as that in **Fig 4.1**. It consists of 2 sender hosts using TCP Reno and 8 sender hosts using TCP Vegas, 10 receiver hosts, two intermediate routers, and links connecting the router and the sender/receiver hosts. The bandwidth of each link between the sender/receiver hosts and the router is 50 Mbps. The bandwidth of the bottleneck link between routers is 2 Mbps. The size of buffer at the router is 100 packets. The propagation delay between the sender/receiver hosts and the router and that between the routers is represented by 1ms and 40 ms, respectively.

Every 10 seconds, connections of Vegas from Vegas1 to Vegas8 successively enter the network, and connections of Reno enter the network at 40 second. The Minimum threshold and the Maximum threshold we used in RED routers are dynamically increased by 3 and 4, respectively, for each connection entering into our network according to our RED router algorithm. So the *MinThresh* and *MaxThresh* will be 30 and 40 after 40 seconds of simulation time. The time we calculate our average throughput for each connection is from 70 sec to 100 sec. This is because when a new connection enters the network, the bandwidth will be recalculated by every TCP source, so we have to wait for the network until it becomes stable. Each packet size is fixed at 1000 bytes. The simulation lasts for 101 seconds.

Table 4.3 summarizes the comparison of 8 TCP Vegas and 2 TCP Reno connections with inaccurate RED Router. According to our RED router algorithm, the *MinThresh* and the *MaxThresh* will be 30 and 40. The accuracy range we choose is from +20% to -20%, so the *MinThresh* will vary from 24 to 36 and *MaxThresh* will vary from 32 to 48 at the end of simulation time.

The throughput of TCP Vegas in Drop-tail router is always less than that in RED router. Considering the error prediction, when the estimation difference is positive, the TCP Reno connections will get more bandwidth because the *MinThresh* and the *MaxThresh* is overestimated. When the estimation difference is negative, TCP Reno connections will suffer packet drop earlier and experience lower bandwidth.

	Drop-tail	RED				
	8V2R	+20%	+10%	8V2R	-10%	-20%
Arrival	24740	24834	24803	24854	24982	24988
Drop	79	159	169	148	222	229
Vegas1	80	64	128	80	104	112
Vegas2	80	120	72	120	96	96
Vegas3	80	64	88	152	112	96
Vegas4	80	144	112	96	80	120
Vegas5	80	144	176	184	184	128
Vegas6	80	104	152	152	168	328
Vegas7	80	112	136	80	232	216
Vegas8	80	168	72	192	256	192
Reno1	664	504	488	432	352	304
Reno2	672	512	512	456	360	336
Vegas	80	103	117	132	154	154
Reno	668	508	500	444	356	320
Fairness	0.11976	0.2028	0.234	0.2973	0.4326	0.48125
Total	1976	1936	1936	1944	1944	1928
Utilization	0.988	0.968	0.968	0.972	0.972	0.964

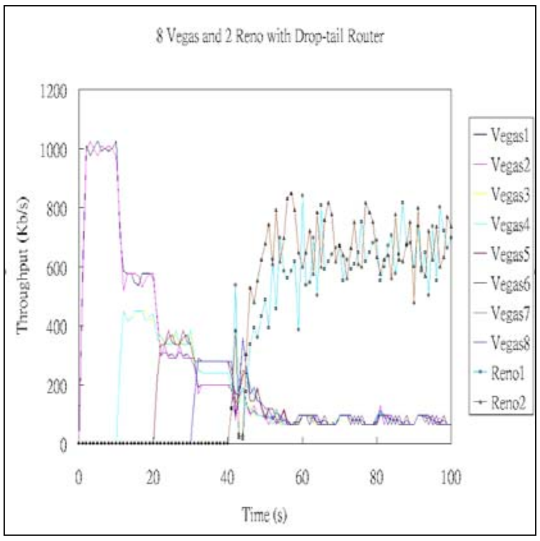
Table 4.3 Comparison of ten connections with Inaccurate RED Router

It can be easily seen that if the number of connection is underestimated, the fairness will be further improved. But the reason why we choose 3n and 4n for the *MinThresh* and the *MaxThresh* is that if we underestimate the number of connections, we will have enough buffer for those connections. Taking 2n and 3n for the *MinThresh* and the *MaxThresh* in our experiment as an example, the value of the *MinThresh* and the *MaxThresh* will be 20 and 30. If we underestimate the connection by 20%, the value of the *MinThresh* and the *MaxThresh* will be 16 and 24, respectively. This will cause all connections unstable, even in homogeneous networks; TCP Vegas connections will behave worse than TCP Reno connections and lead to more packet drop. So after observing the queue size in bottleneck link router, we suggest that the value of the *MinThresh* and the *MaxThresh* should be 3n and 4n, respectively.

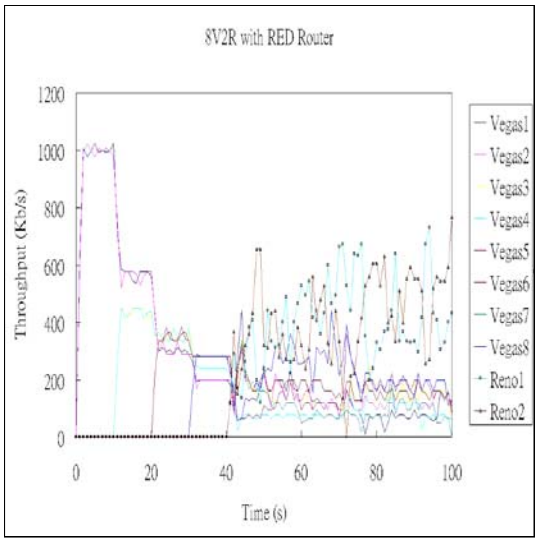
The fairness improvement will be 69% to 302% when taking error estimate into consideration.

Figure 4.6 demonstrates the throughput, queue size, and congestion window between TCP Vegas and TCP Reno with Drop-tail router or RED router. The throughput of Vegas1 and Vegas2 steps down when new connections enter into network in **Fig. 4.6(a)** and Fig. **4.6(b)**. After 40 seconds, two connections of TCP Reno join, so network become unstable and TCP Vegas connections suffer low bandwidth. This will be improved by RED router. We can see that queue size in **Fig. 4.6(d)** is kept lower than 40 packets while we have to wait buffer full and packet loss in **Fig. 4.6(c)**. This is because the TCP Reno keeps increasing its congestion window size in **Fig. 4.6(e)** and let the TCP Vegas starve to death. The RED router used in **Fig. 4.6(f)** will solve such difference issue between TCP Reno and TCP Vegas.

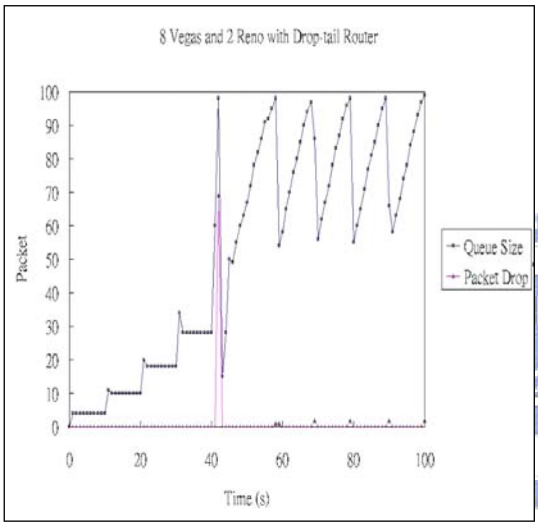
Figure 4.7 illustrates the throughput of 8 TCP Vegas and 2 TCP Reno connections with RED Router with inaccuracy. We can see that TCP Reno connections favor overestimate while TCP Vegas connections favor underestimation.



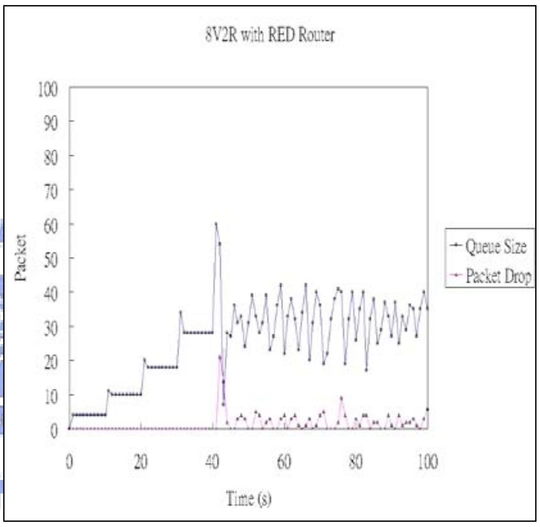
(a)



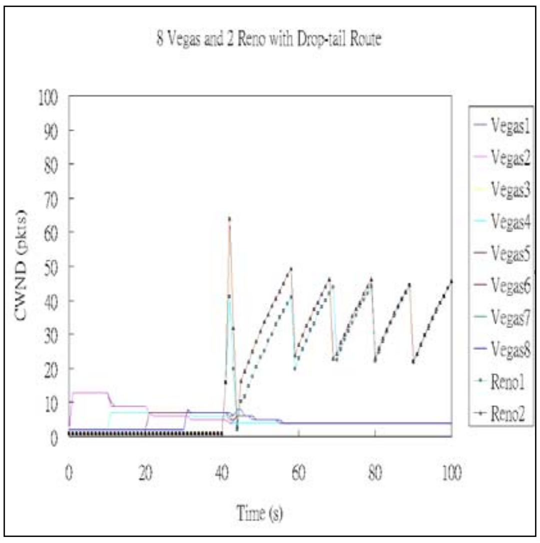
(b)



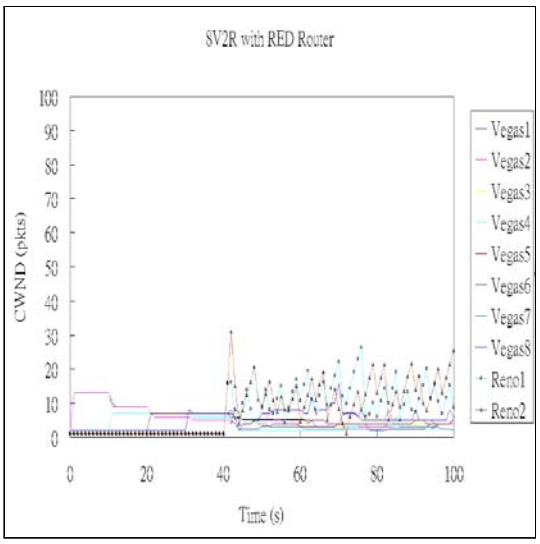
(c)



(d)

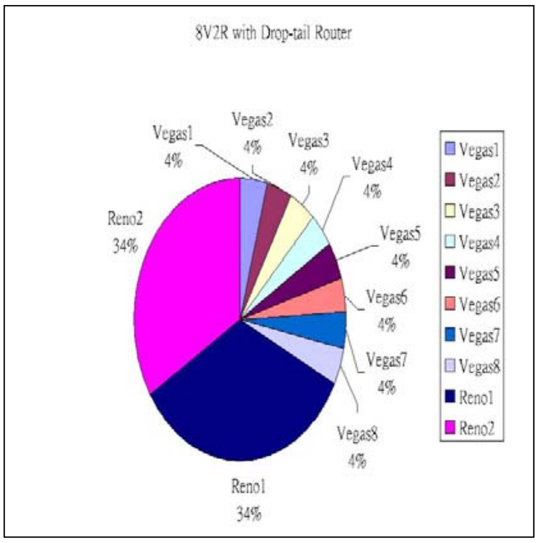


(e)

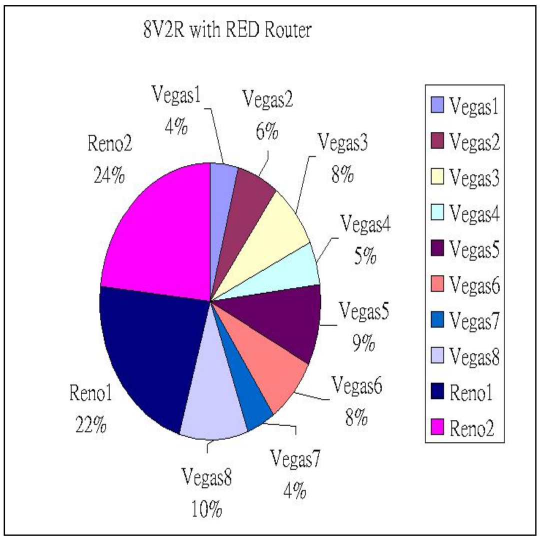


(f)

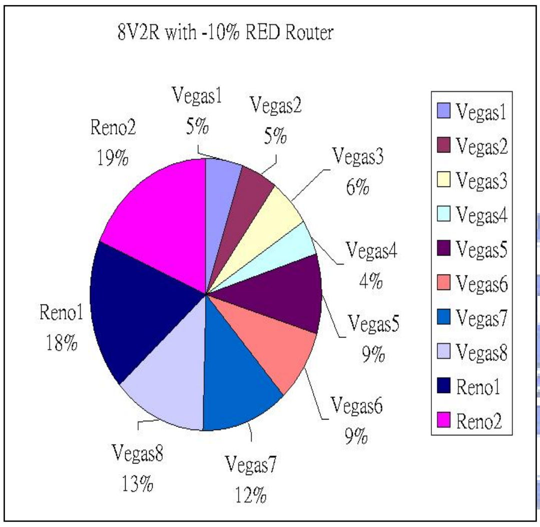
Figure 4.6 8V2R comparison between Drop-tail Routers and RED Routers.



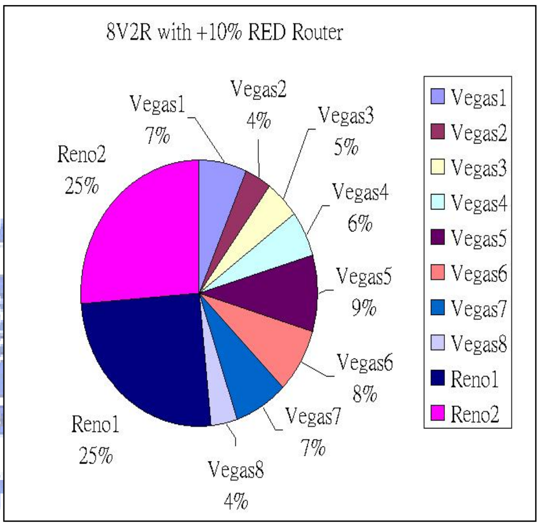
(a)



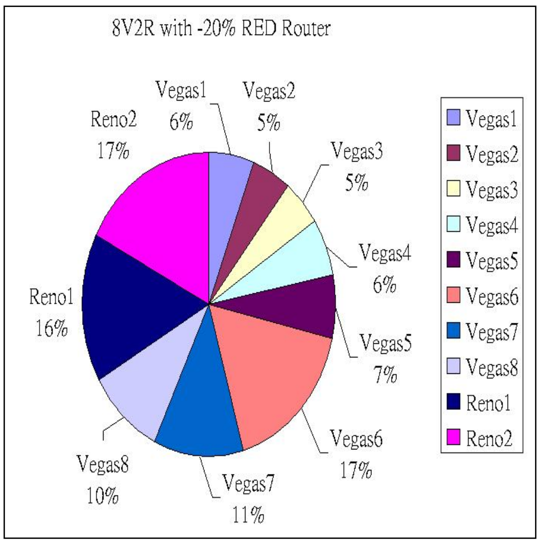
(b)



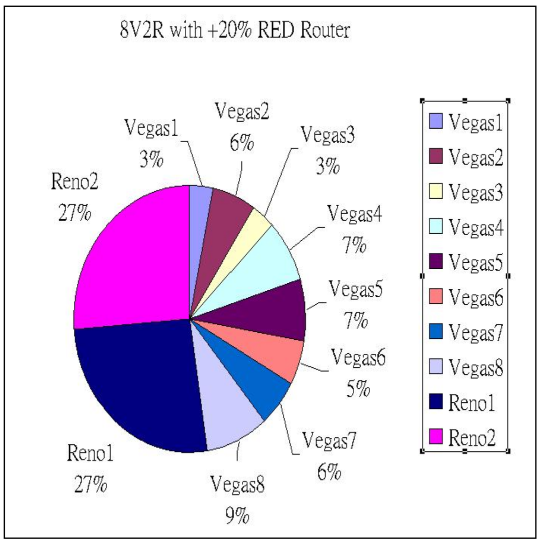
(c)



(d)



(e)



(f)

Figure 4.7 Throughput Ratio with Drop-tail Routers or Inaccurate RED Routers.

4.3 Gradual Deployment

In the real networks, it can not expect that all routers are RED routers and all RED routers are able to calculate the number of connections passing through it, while our RED router algorithm is based on such assumption. So, we have to test what if only one RED router is used while the others are all drop-tail routers in the networks. Comparing with all drop-tail routers and all RED routers used in our simulation, fortunately, we found that even there is only one our RED router used, the fairness improvement between TCP Reno and TCP Vegas connections will be significantly improved. In this section, we give a simple experiment to demonstrate and to show our RED router algorithm is usable to gradual deployment for the current networks.

Figure 4.8 shows the network model used in this section. It consists of 1 sender hosts (S1) using TCP Reno and 3 sender hosts (S2, S3, S4) using TCP Vegas, 4 receiver hosts, 6 intermediate routers, and links connecting the router and the sender/receiver hosts. The bandwidth of each link between the sender/receiver hosts and the router is 10 Mbps. The bandwidth of each link between the routers is 1.5 Mbps. The bandwidth of the bottleneck link (R1-R2) between routers is 2 Mbps.

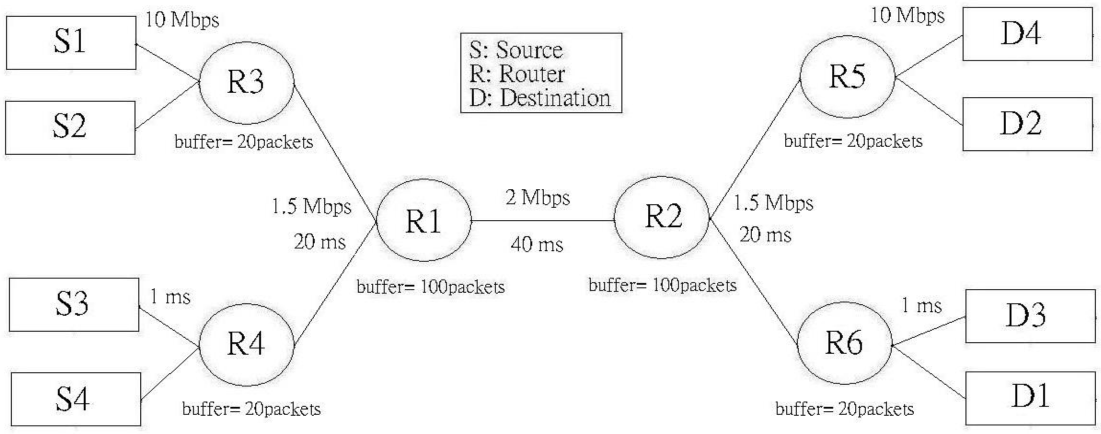


Figure 4.8 Network topology for gradual deployment.

The buffer size at the intermediate routers (R3, R4, R5, and R6) is 20 packets and the buffer size at the bottleneck link routers (R1, R2) is 100 packets. The propagation delay between the sender/receiver hosts and the intermediate routers and that between the bottleneck routers are represented by 1ms and 40 ms, respectively. The propagation delay between the intermediate routers and the bottleneck routers is 20 ms. Each packet is fixed at a length of 1000 bytes. As the scheduling discipline at the router buffer, we consider drop-tail and RED algorithm.

S2 (Vegas1) and S3 (Vegas2) start at 0 sec, while S1 (Reno) and S4 (Vegas3) enter the network at 10 sec and 20 sec, respectively. The time we calculate our average throughput for each connection is from 100 sec to 200 sec. This is because when a new connection enters the network, the bandwidth will be recalculated by every TCP source, so we have to wait for the network being stable. The simulation lasts for 201 seconds.

Table 4.4 summarizes the performance comparison between Drop-tail routers and RED routers for gradually deploying our RED algorithm to the current Internet.

	Drop-tail	RED		
		R1	R3	ALL
Arrival	48473	48572	48049	48597
Drop	0	118	0	118
S1	1272	728	960	752
S2	152	384	280	376
S3	256	408	336	400
S4	256	416	328	408
Vegas	221	403	315	395
Reno	1272	728	960	752
Fairness	0.1737	0.5536	0.3282	0.5253
Total	1936	1936	1904	1936
Utilization	0.968	0.968	0.952	0.968

Table 4.4 Performance comparison in gradual deployment networks.

In our experiment, R1 and R3 in RED router mean that only one RED router is used, so the Minimum threshold and the Maximum threshold used in RED router R1 are 12 (packets) and 16 (packets) and those used in RED router R3 are 6 (packets) and 8 (packets), according to our RED router algorithm based on the number of connections that we predict in our RED routers.

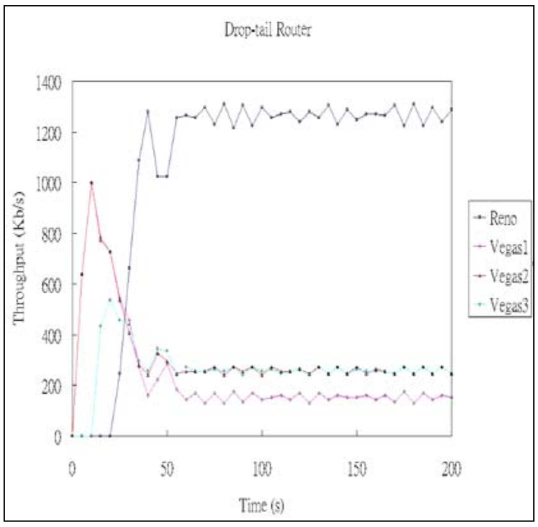
The *Arrival* and *Drop* packets in **Table 4.4** monitored are in the bottleneck link router R1, so there is no packets drop in R3 column, because router R1 have buffer sizes 100 and it is using drop-tail mode. The idle throughput is 2000 Kbps.

We can see that TCP Reno connection (S1) will effect other TCP Vegas connections (S2, S3, S4) when they compete the same bottleneck link (R1-R2). The difference between in R1 and R3 columns is the throughput of S1. This is because in R3 column, the R3 router drops TCP Reno packets earlier than that in R1 column, so S1+S2 will take more bandwidth than S3+S4. It is very clear that even there is only one RED router (R1 or R3), while the others are drop-tail routers, there is still a great improvement, which ranges from 89% to 219%.

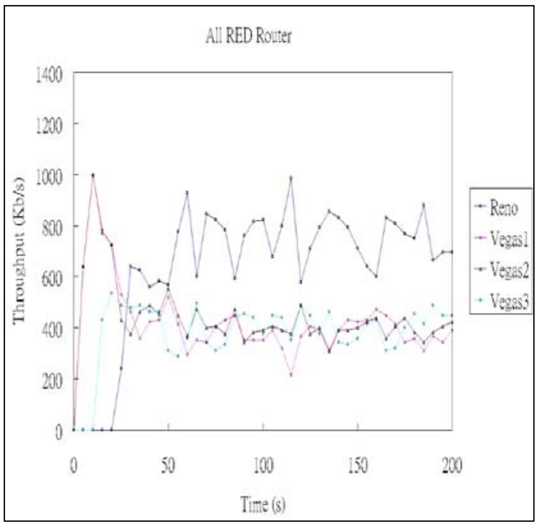
Figure 4.9 illustrates the throughput comparison between TCP Vegas and TCP Reno connections with all Drop-tail routers or all RED routers. The TCP Reno connection in **Fig. 4.9(a)** will be restricted by RED routers in **Fig. 4.9(b)** and give away more bandwidth for sharing with TCP Vegas connections.

Figure 4.10 shows the throughput comparison between TCP Vegas and TCP Reno connections with those four conditions. We will find that there is a big difference between **Fig. 4.10(a)** and **Fig. 4.10(b)**, because the RED router will favor TCP Vegas connection, while there are only R1 or R3 used RED queuing discipline in **Fig. 4.10(c)** and **Fig. 4.10(d)**, our method is still working.

Therefore, we can conclude that even there is only one RED router, if it is on the path of a TCP Reno connection, it will be effective on the fairness problem.

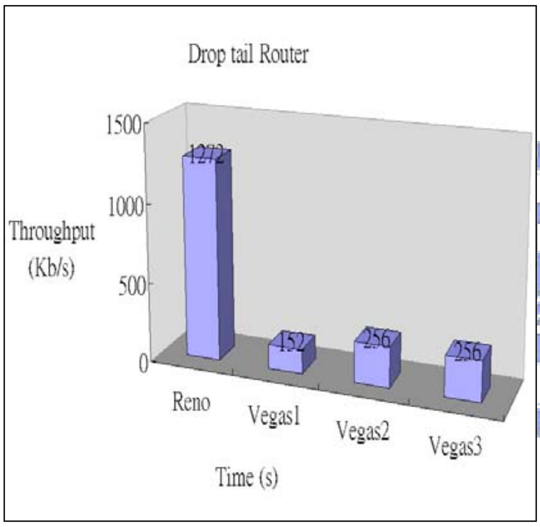


(a)

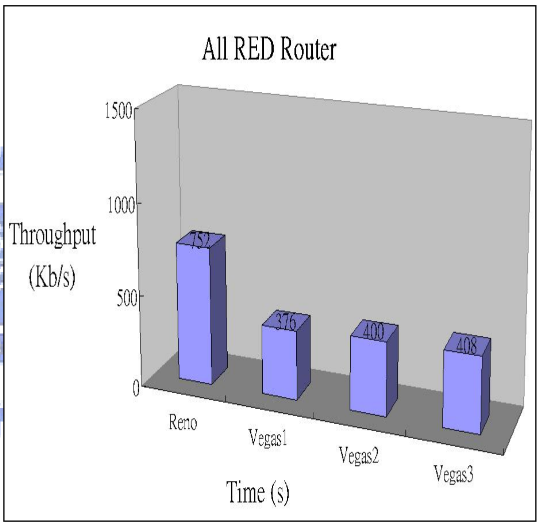


(b)

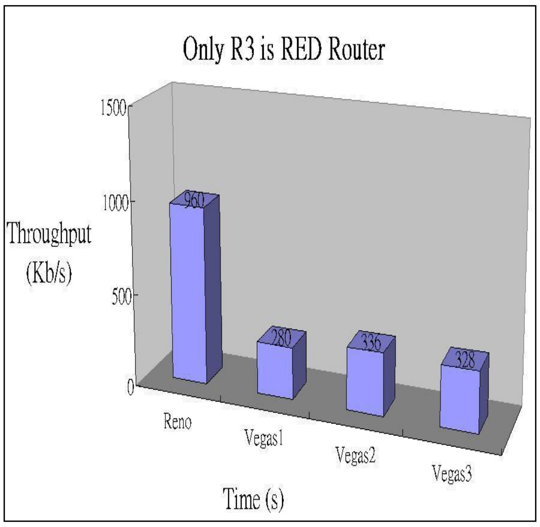
Figure 4.9 Throughput comparison with all Drop-tail Routers or all RED Routers.



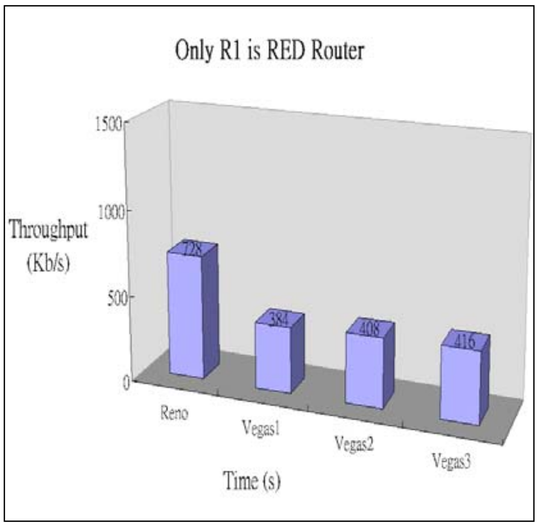
(a)



(b)



(c)



(d)

Figure 4.10 Throughput comparison between Vegas and Reno with four conditions.

CHAPTER 5

Conclusion and future work

Conclusion

In a homogeneous case, TCP Vegas outperforms TCP Reno owing to its higher throughput and stability. However, TCP Vegas is in a squally situation when TCP Reno and Vegas coexist. Because two TCP versions share the same buffer in the routers, the aggressive behavior of TCP Reno and the conservative behavior of TCP Vegas lead to unfairness. This accounts for why TCP Vegas is still unpopular.

In this thesis, we propose a RED Router-based algorithm to improve the fairness problem between TCP Reno and Vegas. The RED Router will give more bandwidth to TCP Vegas by early drop TCP Reno packets. With the function of the RED routers, which can compute the number of connections passing through it, we can set *MinThresh* and *MaxThresh* parameter dynamically. By setting the two thresholds properly, it will lead to great fairness improvement which is shown in our simulations.

There is still a trade-off between fairness and throughput, since lower threshold values result in smaller throughput while the fairness ratio will be improved. We have shown that 3n and 4n are better values for the *MinThresh* and *MaxThresh* parameter, taking the fairness, throughput, and error prediction into consideration. The simulation tells us that even there is only one RED Router along the path using our algorithm with the TCP Reno connections passing through it, our approach is still effective. It is feasible for the RED routers to deploy our algorithm gradually.

Future work

We have shown that RED can improve the fairness to some degree, but there is an inevitable trade-off between fairness and throughput. The main reason is 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 connections become lower, and total throughput gets smaller while we can obtain better fairness. Therefore, for further fairness improvement without throughput degradation, we need two mechanisms:

- How to detect TCP Reno connections
- How to drop more packets from TCP Reno connections

REFERENCES

[1] V. Jacobson, Congestion avoidance and control, ACM SIGCOMM ’88 (1988) 273–288

[2] V. Jacobson, Modified TCP Congestion Avoidance Algorithm, mailing list, end-to-end-interest, April 30, 1999

[3] L.S. Brakmo, L.L. Peterson, TCP Vegas end to end congestion avoidance on a Global Internet, IEEE JSAC 13 (1995) 1465–1480

[4] J. Mo, R.L.V. Anantharam, J. Walrand, Analysis and comparison of TCP Reno and Vegas, in: Proceedings of the IEEE Globecom’99, Rio de Janeiro, Brazil, December 1999

[5] Y.-C. Lai, C.-L. Yao, The performance comparison between TCP Reno and TCP Vegas, in: Proceedings of the Seventh International Conference on Parallel and Distributed Systems, Iwate, Japan, July 2000

[6] U. Hengartner, J. Bolliger, Th. Gross, TCP Vegas revisited, in: Proceedings of the IEEE INFOCOM’2000, Tel Aviv, Israel, March 2000

[7] 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, Helsinki, Finland, June 11–14, 2001

[8] S.H. Low, L. Peterson, L. Wang, Understanding TCP Vegas: a duality model, in: Proceedings of the ACM Sigmetrics’2001, Cambridge, MA, June 17–20, 2001

[9] O. Ait-Hellal, E. Altman, Analysis of TCP Vegas and TCP Reno, IEEE ICC ’97 (1997) 495–499

[10] C.L. Lee, C.W. Chen, Y.C Chen, Weighted Proportional Fair Rate Allocations in a Differentiated Services Networks, IEICE TRANS. Communications, Jan 2002

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

[12] W. Stevens, Slow TCP start, congestion avoidance, fast retransmit, and fast recovery algorithms, RFC 2001 January 1997

[13] Yuan-Cheng Lai, Chang-Li Yao, Performance comparison between TCP Reno and TCP Vegas, Computer Communications 25 (2002) 1765–1773

[14] Andrea De Vendictis, Andrea Baiocchi, Michela Bonacci, Analysis and enhancement of TCP Vegas congestion control in a mixed TCP Vegas and TCP Reno network scenario, INFOCOM Department, University of Roma "La Sapienza", Via Eudossiana 18, 00184 Rome, Italy

[15] K.N. Srijith, Lillykutty Jacob, A.L. Anada, TCP Vegas-A: Solving the fairness and Rerouting Issues of TCP Vegas, IEEE (2003) 309–316

[16] Go Hasegawa, Kenji Kurata and Masayuki Murata, Analysis and improvement of fairness between TCP Reno and Vegas for deployment of TCP Vegas to the Internet, IEEE (ICNP2000) 177-186

[17] K. Fall, S. Floyd, ns-Network Simulator, http://www-mesh.cs.berkeley.edu/ns