A SIMULATION STUDY ON THE FAIRNESS OF TCP VEGAS

by

Tsang, Chi Man Estella

MSc in Information Technology

The Hong Kong Polytechnic University

December 2000

Abstract of dissertation entitled:

A Simulation Study on the Fairness of TCP Vegas

submitted by Tsang, Chi Man Estella

for the degree of MSc in Information Technology

at The Hong Kong Polytechnic University in December 2000.

In addition to throughput, fairness is another important performance criteria of TCP protocols. Fairness is especially important for best effort service, which is still the dominant type of service in the Internet and, predictably, in the years to come. However, the TCP protocols prevailing in the Internet, including TCP Tahoe and TCP Reno, are known to be unfair, especially to connections with larger round-trip delays. Using ns simulator, I have examined the fairness of TCP Vegas with focus on 4 issues : 1) Is TCP Vegas really fair to connections with larger propagation delays, 2) what is the impact of the thresholds, α and β, used in TCP Vegas' congestion avoidance algorithm on fairness, 3) how fair TCP Vegas is when there are many active flows, and 4) does the receiver acknowledgement strategy, namely immediate and delayed acknowledgement, which affects the estimation of the round-trip times, affect the fairness of TCP Vegas ?

For question 1, the simulation results support that, when $\alpha = 1$ and $\beta = 3$, TCP Vegas is still unfair to connections with larger propagation delays but, unlike TCP Reno, the delay bias does not necessarily increase as the delay differences increases. However, if $\alpha = 1$ and $\beta = 3$, the fairness of TCP Vegas is not necessarily better than that of TCP Reno. The unfairness problem can be resolved by Enhanced TCP Vegas that sets $\alpha = \beta = 2$ or 3 but NOT 1.

For question 2, I find that, the fairness is consistently good when $\alpha = \beta = 2$ or 3. When $\alpha = \beta = 1$, the fairness is unstable and may be worse than that when $\alpha = 1$ and $\beta = 3$. Considering a trade-off among fairness, stability and aggressiveness, a value of 3 seems to be an acceptably good choice.

For question 3, when there are many active flows, all of which are running either TCP Reno or TCP Vegas, the fairness of TCP Reno is better and more stable than that of TCP Vegas. The contrary occurs when the number of flows times the threshold of Enhanced TCP Vegas is less than the RED minimum threshold. Intuitively, this suggests that the fairness of TCP Vegas is better when there are few or no retransmissions or timeouts. The result is obtained after amending the ns code to eliminate 2 causes of deadlock that occur after a Reno-type timeout. Moreover, the fairness of TCP Vegas host-only configuration improves when the RED thresholds are higher. The percentage improvement, however, diminishes when the number of flows is large enough.

When there are many active flows with a mixture of TCP Vegas and TCP Reno, the fairness tends to deteriorate as the number of active flows increases. When the RED thresholds are low, TCP Vegas connections are able to capture higher goodput on average. When the RED thresholds are high and the number of flows is low, TCP Reno is the beneficiary. Intuitively, this suggests that TCP Reno performs better when the RED thresholds allow a long enough queue. Moreover, the variance in goodputs within the Vegas community is larger than that within the Reno community. Therefore, while the difference in the average goodput between the Vegas community and the Reno community contributes to unfairness, the difference in goodputs within the Vegas community also contributes significantly. This is due to the record of an under-estimated minimum round-trip time after a Reno-type timeout, causing some connections to run at very low throughputs throughout the rest of the simulation time. Hence, the variance of goodputs within the Vegas community is poor. The findings suggest possible incompatibility between the Reno-type timeout subroutine and the Vegas algorithms. Besides, TCP Vegas may perform unnecessary re-transmission after a Reno-type timeout.

For question 4, I find that fairness does not deteriorate when all the receivers adopt the delayed acknowledgement strategy rather than the immediate acknowledgement strategy. In general, the delayed acknowledgment strategy delays the detection of a proper minimum round-trip time. The first estimation of the minimum round-trip time is always over-estimated. However, since 2 back-to-back TCP segments are sent when the congestion window increases to 2, the receiver returns an acknowledgement nearly immediately after the first of these 2 segments have been received. Therefore, TCP Vegas can eventually capture a minimum round-trip time that is much closer to the true value. Finally, when the receiver adopts the delayed acknowledgement strategy, the calculated expected rate may even be smaller than the actual sending rate.

ACKNOWLEDGMENTS

My many and special thanks go to my supervisor Dr. Rocky K. C. Chang for his valuable advice, comment and ideas, as well as his understanding and help in obtaining computing resources to complete this project. I would also like to thank my co-examiner Dr. Henry C. B. Chan. And last, but not least, for the support from my parents, friends and department manager.

TABLE OF CONTENTS

LIST OF FIGURES

Figure list *Page*

LIST OF TABLES

Tables list *Page*

C h a p t e r 1

INTRODUCTION

1. Introduction

In addition to throughput or goodput, fairness is another important criteria of TCP protocols. Fairness is especially important for best effort service, which is still the dominant type of service in the Internet and, predictably, in the years to come. However, TCP protocols now prevailing in the Internet, including TCP Tahoe and TCP Reno, are widely known to be unfair, especially to connections with larger round-trip delays [LM97, FJ91].

Recently, Brakmo and Peterson [BP95] proposed TCP Vegas, a modified version of TCP Reno. Based on their experiment results, Brakmo and Peterson reported that TCP Vegas was able to achieve higher throughput and retransmit fewer packets than TCP Reno was. Therefore, they implied that the improvement in performance was not based on an aggressive retransmission strategy that stole the bandwidth from the competing TCP flows. Instead, it was based on the modified algorithms that allowed better use of the available bandwidth through reduced retransmission and packet losses. Subsequently, a number of researches confirmed that TCP Vegas performed better in terms of throughput and retransmission.

In this project, I look at the fairness of TCP Vegas, with focus on 4 issues: 1) is TCP Vegas really fair to connections with larger propagation delays, 2) what is the impact of the Vegas thresholds used in the congestion avoidance algorithm on the fairness, 3) how fair is TCP Vegas when there are many active flows, and 4) does the receiver acknowledgement strategy affect the fairness of TCP Vegas ? Hasegawa [HMM99] proposed that a special configuration of TCP Vegas, which he called Enhanced TCP Vegas, was able to achieve better fairness measure than Brakmo and Peterson's TCP Vegas. Hence, I also study the fairness of Enhanced TCP Vegas, where applicable, with comparison to the configuration used in Brakmo and Peterson's paper. The only difference between Hasegawa's Enhanced TCP Vegas and Brakmo's TCP Vegas is that Enhanced TCP Vegas sets $\alpha = \beta$ while Brakmo's TCP Vegas sets $\alpha < \beta$.

My analysis is based on the simulation results obtained with the ns simulator developed at the UC Berkeley. To study the first 2 issues, I vary the propagation delay of the longer connection in a network topology with 2 connections sharing a bottleneck link. To study the next 2 issues, I run simulations with different number of many active flows, with either TCP Reno only, TCP Vegas only or a mixture of TCP Reno and TCP Vegas.

The rest of this project report is organized as follows. In the rest of this chapter, I review the previous work on TCP Vegas. I begin with a discussion of the work on the performance of TCP Vegas. Then, I continue with a discussion of those that focus on the fairness of TCP Vegas. Then, I highlight the contribution of this project. In chapter 2, I provide an overview of the TCP algorithms, including a review of the different algorithms of TCP Vegas as well as the Enhanced TCP Vegas. In chapter 3, I present my simulation results and findings for each of the 4 issues I have mentioned. Finally, in chapter 4, I conclude my findings and summarize possible future work.

2. Previous work

2.1 Development of TCP Vegas

In 1994, Lawrence S. Brakmo and Larry L. Peterson [BP95] at the University of Arizona proposed a new TCP algorithm called TCP Vegas. TCP Vegas was a modified version of TCP Reno (TCP Reno is an implementation of TCP which incorporates the fast retransmission and the fast recovery algorithms.). In their paper, Brakmo and Peterson reported that TCP Vegas was able to achieve 37% to 71% better throughput than TCP Reno on the Internet. It also retransmitted between 1/5 to ½ as much data as TCP Reno did, both in simulations and in live wide-area Internet measurements. Therefore, Brakmo and Peterson implied that the improvement in throughput was not achieved by an aggressive retransmission strategy that stole bandwidth from the competing TCP connections. Instead, the improvement in performance was achieved by a more efficient use of the available bandwidth through reduced packet losses and reduced retransmissions. At the end of the paper, the authors also argued that TCP Vegas was at least as fair as TCP Reno was.

Since then, some researchers have studied TCP Vegas to evaluate Brakmo's claims. In 1995, Ahn et al. [ADLY95] ported Vegas 0.8 to SunOS 4.1.3 and conducted experiments on both an emulated wide-area network and on the live Internet to evaluate the performance of TCP Vegas. (The original TCP Vegas was implemented in the x-kernel protocol network version 3.1.) They were able to confirm claims of higher throughput and reduced retransmission. In addition, they reported that TCP Vegas had shorter RTT averages and variances than TCP Reno did because TCP Vegas kept less data in the network

Employing the x-Sim simulator developed at the University of Arizona, Raghavendra and Kinicki [RK99] were the first to focus on TCP Vegas' performance when it was used with the RED gateway. Raghavendra and Kinicki conducted a series of simulation experiments to study the performance of TCP Vegas when it was used with the RED gateway. (In all their simulations, $\alpha = 1$ and $\beta = 3$ were used.) Their simulation results illustrated that, regardless of whether DropTail or RED gateway was used, TCP Vegas outperformed TCP Reno in terms of effective utilization. Contrasting RED to DropTail gateway, however, they reported that the RED router configurations provided only marginal improvements (about 5% increase) in effective throughput.

TCP Vegas' performance, however, had been under debate as some researchers argued that not all TCP Vegas' techniques were as effective as it was claimed. Recently, Hengartner [HBG00] confirmed that TCP Vegas could achieve significantly higher throughput than TCP Reno could, with improvements ranging from 40% to 120%. He also reported that TCP Vegas retransmitted between 6% to 65% less data than TCP Reno did. However, he further decomposed TCP Vegas' techniques and drew the conclusion that TCP Vegas' techniques for slow-start and congestion recovery had most influence on throughput as they were able to avoid timeouts due to multiple segment losses. TCP Vegas' congestion avoidance algorithm, however, had only minor or even negative effect on throughput.

TCP Vegas has been implemented on Linux version 2.2 and 2.3 by Neal Cardwell and Boris Bak. (The source code can be found at web site http://www.cs.washington.edu/homes/cardwell/linux-vegas/.) The Linux implementation on Linux differs from the original Arizona implementation in a number of ways. According to Cardwell and Bak,

1. Since Linux already recovers from losses reasonably well, using an RTO derived from fine-grained RTT measurements and NewReno or FACK, the loss detection or recovery mechanisms of Linux have not been changed to implement the Vegas adjustment during loss recovery.
2. To avoid considerable performance penalty imposed by increasing the congestion window only every other RTT during slow start, the Linux implementation increases the congestion window every RTT during slow start, just like Reno.
3. Largely to allow continuous congestion window growth during slow start, the actual sending rate is calculated using the rate at which acknowledgements arrive, rather than the rate at which data is sent.
4. To filter out the noise caused by delayed acknowledgements, the Linux implementation uses the minimum RTT sample observed during the last RTT to calculate the actual rate. (According to Cardwell and Bak, this can delay the detection of congestion by up to one RTT or so. However, the author claims that they have not found a better way to filter out the huge RTT spikes caused by delayed acknowledgements.)
5. This Linux implementation uses microsecond-resolution time stamps rather than the millisecond-resolution timestamps used by the x-kernel or the 10ms-resolution timestamps used by Linux/i386.
6. When the sender re-starts from idle, the Linux implementation waits until it has received acknowledgements for an entire flight of new data before making a decision on the adjustment of the congestion window. The original Vegas implementation assumed that the senders never went idle.

2.2 Fairness issue of TCP Vegas

It has long been known to researchers that TCP Reno biases against connections with larger round-trip delays [LM97, FJ91]. For example, Lakshman and Madhow [LM97] provided an intuitive explanation of TCP's bias against connections with larger round-trip delays in a network with high bandwidth-delay product (e.g. WAN where the buffering on the bottleneck link is of the same order of magnitude as, or smaller than, the bandwidth-delay product). They explained that TCP Reno's unfairness was a fundamental consequence of its own window dynamics. In particular, if the queuing delay was small compared to the round-trip delay, then the growth in the window size during the congestion avoidance phase was approximately inversely proportional to the round-trip delay. Moreover, the average throughput was roughly inversely proportional to the square of the round-trip delay. Therefore, the maximum window size was smaller for connections with higher propagation delays so that, with TCP Reno, the initial window size of the longer connection was also smaller after a packet loss. Consequently, throughput suffered.

Floyd and Jacobson [FJ91] also suggested that TCP Reno's bias against longer round-trip time connections was a consequence of TCP's own window increase algorithm. Moreover, they showed that the bias was unrelated to the gateway dropping policy because both the Drop Tail (FIFO) and the Random Drop gateways shared such bias. In their simulations, the improvement was minor even when the RED gateways were used.

Since Brakmo et al. have proposed TCP Vegas, there have been some studies to show that Vegas fairness is better than Reno fairness [CC00, MLAW99] and that TCP Vegas does not suffer from such delay bias as TCP Reno does. Mo et al. [MLAW99] used a simple fluid model to show that TCP Vegas did not suffer from delay bias as TCP Reno did and confirmed their analysis by simulations. By analyzing using a closed fluid model, Mo argued that the congestion window of TCP Vegas converged after some time. Moreover, if the system was quasi-static, the queue size did not depend on the propagation delay. According to FIFO policy, which is adopted in both the DropTail and the RED gateway, the queue occupancy in the router buffer reflects the ratio of bandwidth sharing. In other words, the flow rate is proportional to the connection's queue size in the intermediate routers. Based on these arguments, Mo et al. concluded that if the router exercised FIFO service discipline, the throughput of a connection should be independent of the propagation delay. Hence, TCP Vegas should not suffer from bias in favor of connections with smaller round-trip delay. Mo also ran simulations of a network topology that consisted of 2 connections sharing a bottleneck link. Their result demonstrated that the throughput ratio of TCP Vegas ranged from 1.09 to 1.33 whereas the throughput ratio of TCP Reno ranged from 1.35 to 37.12. Moreover, the simulations showed that the throughput ratio of TCP Reno increased with the increase in the difference of the propagation delays of the 2 connections. In contrast, this phenomenon did not happen with TCP Vegas.

2.2.1 Impact of gateway settings

Raghavendra and Kinicki [RK99] conducted a series of simulation experiments to study the fairness of TCP Vegas when it was used with the RED gateway. TCP Vegas defines 2 thresholds, α and β, in its congestion avoidance mechanism. The original TCP Vegas proposed by Brakmo [BP95] set α and β to 1 and 3 or 2 and 4 respectively. Raghavendra and Kinicki's set α =1 and β = 3 in all of their simulations. Their simulation results illustrated that, independent of whether TCP Reno or TCP Vegas was used, RED gateways provided better and less volatile fairness than DropTail gateway with equivalent configurations. Moreover, the settings of the minimum and maximum thresholds of the RED gateway had little impact on fairness.

2.2.2 Impact of Vegas thresholds α and β

Other researchers pointed out that the difference in the values of α and β caused an unfair bandwidth sharing when connections running TCP Vegas shared a bottleneck link [CC00, HMM99]. They argued that if α and β equal 1 and 3, respectively, TCP Vegas might occupy between 1 to 3 buffer sizes in the intermediate routers. Therefore, in the worst-case scenario, one connection might occupy about 3 router buffer sizes while the other might only occupy 1 router buffer. Consequently, one connection was able to achieve 300% bandwidth of the other. This led to proposals that the values of α and β should be the same.

For instance, Hasegawa et al. [HMM99], pursuant to their discovery of this unfairness through analytical approach, proposed Enhanced TCP Vegas, which set α to be the same as β. They called the control parameter δ and used a value of 3 in their example. Hasegawa claimed that Enhanced TCP Vegas provided better fairness by removing the condition that allowed the window size to stay fixed as long as the difference between the expected sending rate and the actual sending rate lied between a narrow range of values. In their paper, they argued that Enhanced TCP Vegas, instead of allowing the window size to converge to a fixed value that could lie anywhere within a narrow range as with TCP Vegas, forced the window size to oscillate around a central point of value which become completely proportional to the propagation delay. Therefore, although Enhanced TCP Vegas discarded the stability property of TCP Vegas, it could achieve better fairness than TCP Vegas in terms of throughput defined as (window size) / (propagation delay).

Besides, the oscillation range of the window size was small. In the paper, there were examples to illustrate and support the analysis. However, I think there was a lack of extensive simulation results to support the analytical result. In addition, the analysis only focused on a simple steady state model that consisted of 2 TCP Vegas connections sharing a bottleneck link. I am also not aware of any study to show that the best choice of δ is 3.

There were other researchers who agreed that there should be 1 control parameter instead of different values for α and β. For example, based on TCP Vegas, Chen et al. [CC00] proposed a new congestion avoidance mechanism, which they called Vegas Plus. Vegas Plus used a modified congestion avoidance algorithm, based on TCP Vegas. It calculates the VQO (virtual queue occupancy) and compares it against one parameter, α', to decide the direction of window size adjustment. In their simulation to evaluate the fairness of Vegas Plus, Chen et al. set α' to 3. The authors argued that they chose 3 because it was the mean of 2 and 4, the choice of α and β in the original TCP Vegas proposed by Brakmo and Peterson.

2.2.3 Impact of the correctness of the estimated minimum round-trip time

There were proposals that the correctness of the minimum round-trip time also affected fairness. The minimum round-trip time was taken as an estimate of the propagation delay of the connection, i.e. the round-trip time of the connection when the router buffer was empty. In their discussion on unfair treatment of “old” connection (i.e. when 1 connection starts later than the other), Hengartner et al. [HBG00] explained that when the new connection was established, the network might have already been in a congested state. Therefore, the estimate of the minimum round trip time of the new connection would over-estimate the actual value. This would lead to a bigger congestion window for the new connection. Consequently, the new connection would keep more packets in the network than it should. In other words, the new connection could benefit from an unfair share of the bottleneck bandwidth.

Similarly, Hasegawa [HMM99] pointed out that the measured minimum round-trip time of 2 connections might differ although the propagation delays of the 2 connections were the same. Thereby, while the fairness of TCP Vegas was better than that of TCP Reno in the heterogeneous case, its fairness might be worse than that of TCP Reno in the homogenous case. They also claimed that their Enhanced TCP Vegas was able to provide a good estimate of the minimum round-trip time because its modified congestion control algorithm prevented the convergence of the window size. They argued that by forcing the window size to oscillate, the number of segments at the router buffer also fluctuated, and thus the minimum round-trip time of both connections become converged to the same value in the homogenous case. Although not stated explicitly, based on the equation that Hasegawa used to derive the converged window size, I believe Hasegawa did believe that the oscillation enabled Enhanced TCP Vegas to correctly measure the propagation delay in the heterogeneous case as well.

There were also concern that the minimum round-trip time could not be measured accurately in situations with persistent congestion and change of routes. The problem was discussed in [MLAW99].

In summary, I notice 2 causes of unfairness that have been identified in previous work, assuming that there is only TCP Vegas in the network environment. These 2 causes are :

1. the difference in the values of α and β, and

2. an incorrect measured minimum round-trip time.

Recently, Boutremans and Boudec [BB00] provided additional insights into the interaction of the over-estimation of the propagation delay and the values of α and β. The paper focuses on situations where 1 connection joined the network after the previous one has joined for a while. The authors reported that when $\alpha = \beta$, as in Enhanced TCP Vegas,

1. Contrary to what Hasegawa believed, the oscillations of the sending rate, which increased with the value of the sending rate, did not allow TCP Vegas to measure the propagation delay accurately.
2. Any over-estimation of the propagation delay of a given connection would increase its rate and the increase became more pronounced with the over-estimation factor. Therefore, late connections would get more bandwidth than the earlier connections.
3. In return, the different connection's peaks in the oscillations were not synchronized in time. Thus, there was no sub-optimal use of the bottleneck link capacity.

On the other hand, they reported that when $\alpha < \beta$,

1. The sending rate of a connection converged to a stable value that depended on the arrival order of all connections. Therefore, the first connection to be established would be favored when the propagation delays were properly estimated. This can best be explained by Fig. 6 in Boutremans and Boudec's paper.
2. Yet, in later connections, the propagation delays were over-estimated, so their rates were greater than what they should be.

When discussing the interaction of an over-estimation of the propagation delay and the value of α and β on fairness, Boutremans and Boudec argued that these 2 factors tend to counterbalance each other but the effect of an over-estimation of the propagation delay dominated.

2.2.4 Impact of a mixture of TCP Vegas and TCP Reno

Some previous work demonstrated that unfairness existed when TCP Vegas competed for bandwidth with TCP Reno [ADLY95., MLAW99, and RK99]. Moreover, the unfairness occurred because TCP Vegas used a more conservative congestion avoidance mechanism. The explanation is as follows : when a TCP Vegas connection shares a link with a TCP Reno connection, the TCP Reno connection uses most of the buffer space while the TCP Vegas connection, interpreting this phenomenon as a sign of congestion, continues to back off. Therefore, TCP Reno is able to capture higher throughput.

When the buffer is small, however, TCP Vegas outperforms TCP Reno. Mo et al. [MLAW99] provided an intuitive explanation as follows : TCP Reno needs some room in the router buffer for oscillations in order to estimate the available bandwidth. Without the necessary room in the buffer, its performance degrades. TCP Vegas, on the other hand, quickly adapts to the small buffer sizes since it requires buffer space for only a few packets. Mo et al. simulated a network with 2 connections having the same propagation delay and demonstrated that the throughput ratio of Reno to Vegas increased from 0.535 to 11.73 when the router buffer increased from 4 to 50.

Raghavendra and Kinicki [RK99] observed a similar phenomenon when DropTail gateway was used with a mixture of TCP Reno and Vegas network hosts. They ran simulations with 3 connections, either with 3 TCP Vegas only, 3 TCP Reno only, 1 TCP

Vegas and 2 TCP Reno or 1 TCP Reno and 2 TCP Vegas. Based on the simulation results, they argued that when the number of router buffer was small, TCP Vegas was able to benefit from an unfair share of bandwidth. On the contrary, when the number of router buffer was large, TCP Reno would dominate. In addition, they observed that unlike DropTail gateway, RED gateway was able to offer consistent high value of fairness (greater than 0.8).

3. Contribution of this project

In this project, I evaluate the fairness of TCP Vegas with focus on 4 issues: 1) Is TCP Vegas really fair to connections with larger propagation delays, 2) What is the impact of the Vegas thresholds, α and β, on fairness, 3) How fair is TCP Vegas when there are many active flows passing through a bottleneck link, and 4) Does the receiver acknowledgement strategy affect the fairness of TCP Vegas ?

IS TCP REALLY FAIR TO CONNECTIONS WITH LARGER PROPAGATION DELAY

The simulation results that address the 1st question confirm that TCP Vegas is also unfair to connections with larger propagation delays. My work adds to the previous work in the following ways. Mo et. al. in [MLAW99] presents heuristic analysis and simulation results to support that Brakmo and Peterson’s TCP Vegas does not suffer from delay bias as TCP Reno does. Adding to Mo’s work, I also evaluate the fairness of Hasegawa’s Enhanced TCP Vegas that sets $\alpha = \beta$. In particular, I attempt to isolate the impact of an incorrect minimum round-trip time, which is not mentioned in Mo’s paper. My work can also be seen as a continuation of Hasegawa’ work, which argues that Enhanced TCP

Vegas can achieve better fairness than TCP Vegas. Instead of analyzing through analytical approach, my work provides a complementary mean to evaluate the fairness of TCP Vegas through an extensive simulation study. Here are my findings :

1. The original version of TCP Vegas [BP94] proposed by Brakmo and Peterson is still unfair to connections with larger propagation delays, unless the difference in the propagation delays of the connections is small. In some cases, the expected sending rate of the longer connection is even less than the fair share of the available bandwidth. Therefore, the actual sending rate is also less than the fair allocation.
2. Supporting Mo's findings, my simulation results show that, unlike TCP Reno, the degree of bias does not necessarily increase as the difference in propagation delays increases. However, the simulation results in section 4.2.4 of Chapter 3 suggest that the fairness of the original TCP Vegas may even be worse than that of TCP Reno.
3. The problem of unfairness can indeed be much resolved by a version of Hasegawa's Enhanced TCP Vegas that sets $\alpha = \beta = 2$ or 3 but NOT 1.

In my simulations, I monitor the estimation of the minimum round-trip time to identify results with which the minimum round-trip time is not over-estimated. I note that in past work, the simulator is usually configured to start one of the connections after the first connection has been running for, say, 0.5 seconds. In such case, it is highly likely that the minimum round-trip time of the later connection is over-estimated. Therefore, when the interpreting these results, we need to take into consideration the impact of an over-estimation of the minimum round-trip time. My simulation results, however, when stated, is free from the impact of the over-estimation problem.

WHAT IS THE IMPACT OF VARYING THE VEGAS THRESHOLD, α AND β, ON THE FAIRNESS OF TCP VEGAS

The simulation results that address the 2nd question help to evaluate if a single value can be chosen for the Enhanced TCP Vegas thresholds, α and β, to ensure best fairness on the Internet. While most research uses $\alpha = \beta = 3$ with Enhanced TCP Vegas (and $\alpha = 1$ and $\beta = 3$ with TCP Vegas), I am not aware of any critical evaluation in terms of fairness. Being able to fix a single value while maximizing fairness is highly desirable for practical reasons and simplify the protocol implementation. Here are my findings :

1. The fairness of Vegas-1 is unstable but that of Vegas-2 and Vegas-3 are consistently good, regardless of the correctness of the minimum round-trip times. It seems that setting $\delta = 1$ poses a limit on the actual sending rate and does not allow Vegas-1 to compete fairly for bandwidth. The fairness of Vegas-4 ($\alpha = \beta = 4$) and Vegas-10 ($\alpha = \beta = 10$) are consistently good as well.
2. On the other hand, the degree of oscillation decreases as the value of δ for Vegas-δ increases. Therefore, it may be argued that the larger the δ, the better the stability.
3. However, there is a tradeoff : with a larger δ, TCP Vegas tries to force more packets into the network. Therefore, the larger the δ, the more aggressive TCP Vegas is and the more likely that it will cause congestion in case of limited available bandwidth.
4. From the simulation results I obtained in this project, a value of 3 for δ seems an acceptably good value to use.

HOW FAIR IS TCP VEGAS WHEN THERE ARE MANY ACTIVE FLOWS PASS THROUGH THE BOTTLENECK LINK

The simulation results that address the 3rd question provide insight into the fairness of TCP Vegas when there are many active flows. While many of the previous research run simulations with a few connections, I am not aware of any research that study the fairness of TCP Vegas when there are many active flows. On the other hand, I understand that it is highly possible that an Internet router has to support many concurrent flows. However, the behavior of TCP Vegas can be different when there are many flows than the intermediate router buffer can support concurrently. When there are only a few connections, congestion is unlikely and thus TCP Vegas can run smoothly with few timeouts. Under this network condition, there may not be a need to trigger the Reno-type timeout subroutine. When there are many flows, there will be many packet losses and thus timeouts, regardless of whether TCP Vegas or TCP Reno is used. This means that the Vegas timeout subroutine and especially the Reno-type timeout subroutine will be triggered much more often. The simulation results in this project contribute to explore whether Enhanced TCP Vegas is still fair when there are more active flows than the RED thresholds can support.

In case when there is only TCP Vegas or TCP Reno in the network, the simulation results suggest that :

1. In most cases, fairness of TCP Reno is better than that of TCP Vegas. The result is obtained after having amended the ns code for TCP Vegas to eliminate 2 causes of deadlock. In the first scenario, the deadlock occurs after a Reno-type timeout that sets the variable storing the un-inflated window to zero, causing TCP Vegas to wrongly set the congestion window to zero upon exit of the fast recovery phase. The second

scenario occurs after a Reno-type timeout of the first segment sent, which wrongly update the start time of the connection to the time that the first segment is re-transmitted. The update wrongly overwrites the time that the first segment is first sent, which represents the start time of the connection.

2. The fairness of the TCP Vegas host-only configuration improves when RED thresholds are higher. The percentage improvement, however, diminishes when the number of flows is large enough.
3. When analyzing results for the Vegas host-only configuration, I also identify scenario in which TCP Vegas performs unnecessary retransmission. The incidence happens after a Reno-type timeout.

In case when there is a mix of TCP Vegas and TCP Reno sender, the simulation results suggest that :

1. Fairness improves when the RED thresholds are larger.
2. Regardless of the RED thresholds, fairness tends to deteriorate as the number of active flows increases. When the RED thresholds are low and the percentage of TCP Vegas is relatively high, there is large overlap of fairness and the difference seems to diminish.
3. When RED router is used and a mixture of TCP Reno and TCP Vegas are present, TCP Reno tends to be the winner of unfairness when the router thresholds are large and the number of flows is low. In the contrary, TCP Vegas is the beneficiary when the router thresholds are low. Intuitively, this suggests that TCP Vegas is the winner when RED router thresholds allow a long enough queue.
4. The variance of throughputs in the TCP Vegas community is larger that that in the TCP Reno community. Therefore, while the unequal allocation in bandwidth between

the TCP Reno community and the TCP Vegas community contributes to the unfairness, the variance in goodputs in the Vegas community also contributes significantly to the unfairness. One possible cause is the incorrect measurement of the minimum round-trip time after a Reno-type timeout, which results in a limitation of the growth of the congestion window and the actual sending rate.

DOES THE RECEIVER ACKNOWLEDGEMENT STRATEGY AFFECT THE FAIRNESS OF TCP VEGAS

The simulation results that address the 4th question help to evaluate the fairness of TCP Vegas when the receiver adopts the delayed acknowledgement strategy. TCP Vegas expects immediate acknowledgement strategy. On the other hand, delayed acknowledgement is recommended by RFC1122 and is thus expected to be the prevalent acknowledgement strategy in the Internet. In such case, the minimum round-trip time and the actual round-trip time may be over-estimated. Therefore, it is more important to understand the fairness and behavior of TCP Vegas when the receiver adopts the delayed acknowledgement strategy. This project presents simulation results to help understand impact of the delayed acknowledgement strategy on the fairness of TCP Vegas. After analyzing the simulation results, I find that

1. The fairness does not deteriorate if all the receivers adopt the delayed acknowledgement strategy rather than the immediate acknowledgement strategy. This is possibly because the round-trip time tends to be over-estimated. Hence, the chance of an incorrect minimum round-trip time reduces, leading to lower variance in goodputs in the Vegas community. In particular, I observe that TCP Vegas does not

behave detrimentally although the receivers employ the delayed acknowledgement strategy.

2. On the other hand, delayed acknowledgment strategy does delay the detection of a proper minimum round-trip time. By that time, a router queue may have already built up and the minimum round-trip time may be over-estimated.
3. In contrary to the definition, I observe that the measured expected sending rate can be less than the measured actual sending rate. The problem is a direct result of the delayed acknowledgement because the receiver sends an acknowledgement for every 2 segments received. The problem can be resolved by removing the 2nd term in the formula that is used to calculate the expected sending rate.

Chapter 2

AN OVERVIEW OF TCP ALGORIHTMS

In this section, I summarize the 3 major techniques used by TCP Vegas to increase throughput and reduce retransmissions. My summarization is based on the original paper, the original Arizona Vegas code and the simulation traces. In addition, I have inserted my own diagrams, where applicable and appropriate, to help illustrates the techniques of TCP Vegas. Before I discuss the Vegas strategies, I also summarize the algorithms of TCP Tahoe and TCP Reno as I will be referring to these names.

1. TCP Tahoe

TCP Tahoe was released in 1998. The 4.3 TCP Tahoe (1989) release has the following features : slow start, congestion avoidance and fast retransmit.

When a new connection is set up, the sender does :

```
cwnd              =      1 segment = 1 MSS (in bytes)
ssthresh          =      maxwnd (65536 bytes)
```

When an acknowledgement is received before the retransmission timer expires, the sender does :

```
if (cwnd < ssthresh)   cwnd += 1segment;          Slow Start
else                   cwnd += 1/cwnd segment;    Congestion Avoidance
```

When segment loss is detected by a timeout or fast retransmit, the sender does (Slow Start) :

```
ssthresh          =      max (min (cwnd, wnd) /2, 2) segments;
cwnd              =      1 segment
```

where

wnd	=	receiver’s advertised window
cwnd	=	congestion window

The idea of TCP Tahoe is to start the congestion window at the size of a single segment (the MSS) and send it when a connection is established. If the acknowledgement arrives before the retransmission timer expires, add one segment to the congestion window. This is a multiplicative increase algorithm and the window size increases exponentially. The window continues to increase exponentially until it reaches the threshold that has been set. This is the Slow Start Phase.

Once the congestion window reaches the threshold, TCP slows down and the congestion avoidance algorithm takes over. Instead of adding a new segment to the congestion window every time an acknowledgement arrives, TCP increases the congestion window by one segment for each round trip time. This is an additive increase algorithm.

To estimate a round trip time, the TCP codes uses the time to send and receive acknowledgements for the data in one window. TCP does not wait for an entire window of data to be sent and acknowledged before increasing the congestion window. Instead, it adds a small increment to the congestion window each time an acknowledgement arrives. The small increment is chosen to make the increase averages approximately one segment over an entire window.

When a segment loss is detected through timeouts, there is a strong indication of congestion in the network. The slow start threshold is set to one-half of the current window size (the minimum of the receiver’s advertised window and the sender’s congestion window). Moreover, the congestion window is set to 1 segment, which forces slow start.

2. TCP Reno

TCP Reno was released in 1990 and incorporated the fast recovery mechanism as well.

```
When a new connection is set up, the sender does :

    cwnd          =    1 segment = 1 MSS (in bytes)
    ssthresh      =    maxwnd (65536 bytes)

When an acknowledgement is received before retransmission timer
expires, the sender does :

    If (cwnd < ssthresh)   cwnd += 1segment;         Slow Start
    else                   cwnd += 1/cwnd segment;   Congestion Avoidance

When segment loss is detected by timeout, the sender does (Slow Start) :

    ssthresh      =    max (min (cwnd, wnd) /2, 2) segments;
    cwnd          =    1 segment

When a segment is detected by fast retransmit, the sender does (Fast
Recovery):

    ssthresh      =    cwnd /2
    cwnd          =    ssthresh + 3 segments

if a duplicate ACK is received,

     cwnd +=1;

if the non-duplicate ACK corresponding to the retransmitted segment is
received,

    cwnd = ssthresh;
```

In TCP Reno, TCP source behaves in the same way as that in TCP Tahoe. However, when the segment loss is detected by fast retransmission algorithm, the slow start threshold is set to half the current size of the congestion window. The congestion window size is then set to be the same as the slow start threshold plus 3 times the segment size. This is the Fast Recovery Phase,, in which the window size is then increased by one segment when a duplicate ACK is received. When the non-duplicate ACK corresponding to the retransmitted segment is received, the congestion window is restored to the slow start threshold.

3. TCP Vegas : An overview of Vegas strategies

TCP Vegas was proposed by Brakmo et al. in 1994. It was based on modifications to TCP Reno. The 3 major modifications are summarized below. There are other minor modifications, as discussed in [HBG00].

3.1 Extension to Reno fast retransmission algorithm

Initially, TCP Tahoe retransmits only when a timeout occurs. Having incorporated the fast retransmission algorithm, TCP Tahoe and TCP Reno also retransmit on receipt of the third duplicate acknowledgement. In addition, TCP Reno implements the fast recovery algorithm. It was reported that the fast retransmission and the fast recovery mechanisms prevented more than half of the coarse-grain timeouts that occurred in TCP implementations without these mechanisms.

TCP Vegas extends TCP Reno's retransmission mechanism so that it also retransmits under 2 additional conditions. The modification results in further reduction in the time to detect lost packets and in the number of coarse-grain timeouts.

The first condition is met when TCP Vegas receives a duplicate acknowledgement and detects expired segments. TCP Vegas records a more accurate RTT by reading the system clock when it receives an acknowledgement for the segment being timed. It then reads the recorded transmission time of the unacknowledged segment and checks if its RTT is greater than the Vegas timeout value. If it is, then TCP Vegas retransmits the next expected acknowledged segment.

This modified retransmission algorithm allows TCP Vegas to retransmit without having to wait for the 2nd or 3rd duplicate acknowledgement This, however, is only a small saving. Under some specific conditions, e.g. when there are multiple losses in 1 RTT or window, when the window size is very small, or in general, when there is insufficient data to trigger the return of the 2nd or 3rd duplicate acknowledgement, TCP Reno must wait for timeouts before it can retransmit. For instance, it has been reported that [ADLY95] when TCP Reno faces 4 drops in a single RTT, i.e. when there are multiple losses, the 2nd or 3rd duplicate acknowledgement will never come through. Thereby, TCP Reno has to wait for the coarse-grain timeouts before the retransmission can take place. Recall that the coarse-grain retransmission timeout is set to the smoothed RTT plus 4 times its deviation. Since the round-trip times are measured in a graunity of 500ms, a typical timeouts can fall in the region of 2000ms. Therefore, reducing the time to wait for timeouts can significantly help to improve throughputs. The scenario is diagrammatically displayed in figures 2.1 and 2.2 below.

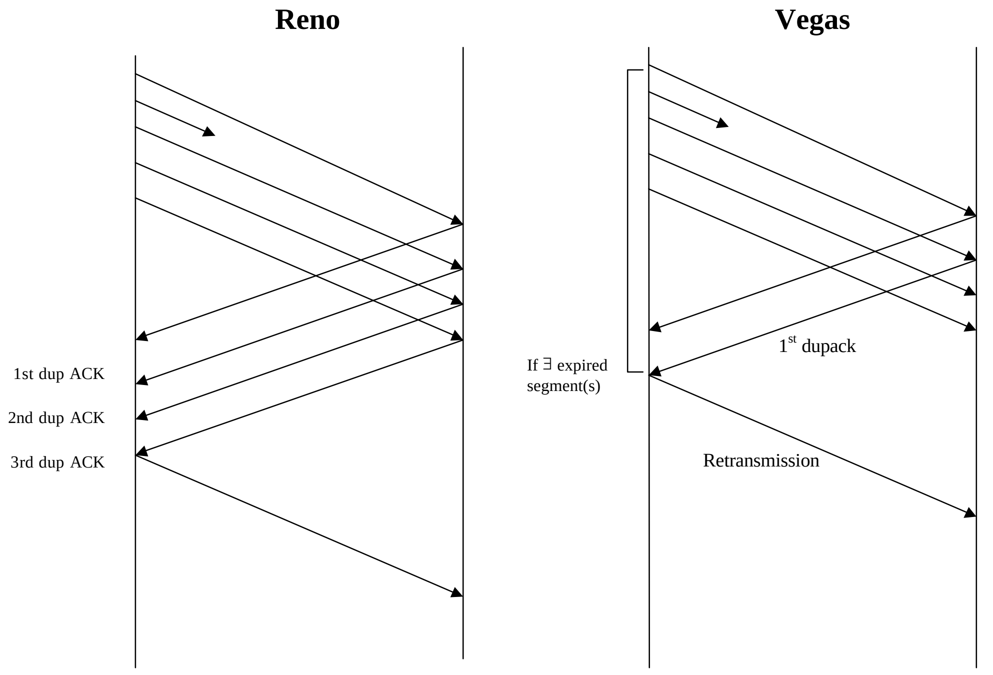


Figure 2.1 : TCP Vegas' enhanced fast retransmission (Illustration 1)
TCP Vegas' retransmission strategy is more aggressive but can result in a more timely decision to retransmit a dropped segment.

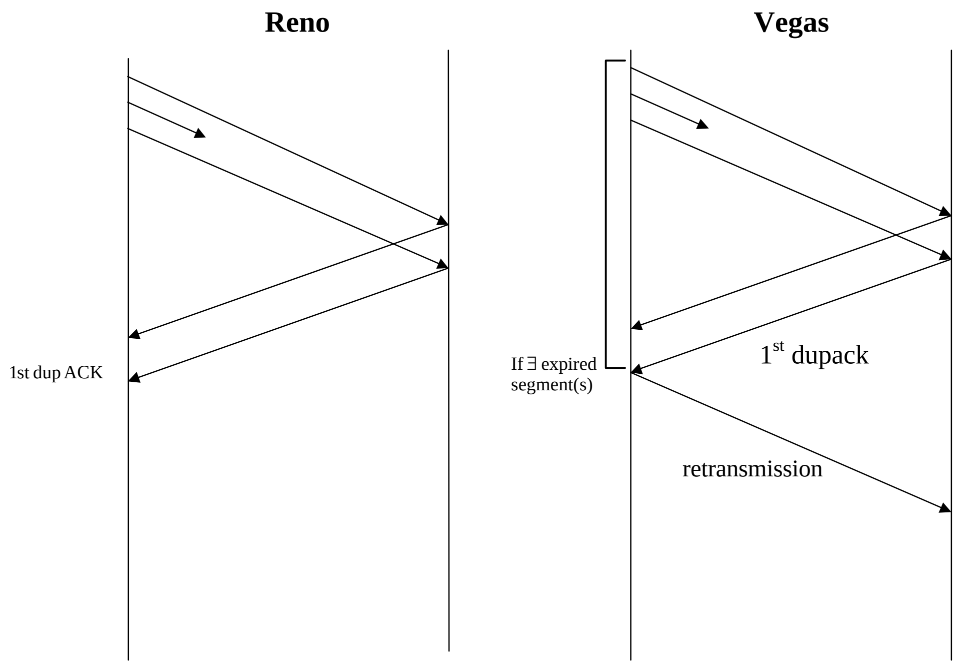


Figure 2.2 : TCP Vegas' enhanced fast retransmission (Illustration 2)
TCP Vegas' retransmission strategy allows it to retransmit a dropped segment although there is not enough data to trigger the return of the 2nd or 3rd duplicate ACK.

The second condition is met when TCP Vegas receives the 1st or 2nd non-duplicate acknowledgement after a retransmission and detects expired segments. On receipt of the 1st or 2nd non-duplicate acknowledgement after a retransmission, TCP Vegas checks to see if the time elapsed since the unacknowledged segment is last sent or retransmitted is larger than the Vegas timeout value. If it is, then TCP Vegas assumes a segment has been dropped. Hence, it retransmits the next expected acknowledged segment. This allows TCP Vegas to catch any other segments that may have been lost prior to the retransmission without having to wait for the duplicate acknowledgements. This is most crucial when there are multiple losses so that TCP inevitably have to wait for timeouts. This scenario is dynamically represented in figure 2.3 below.

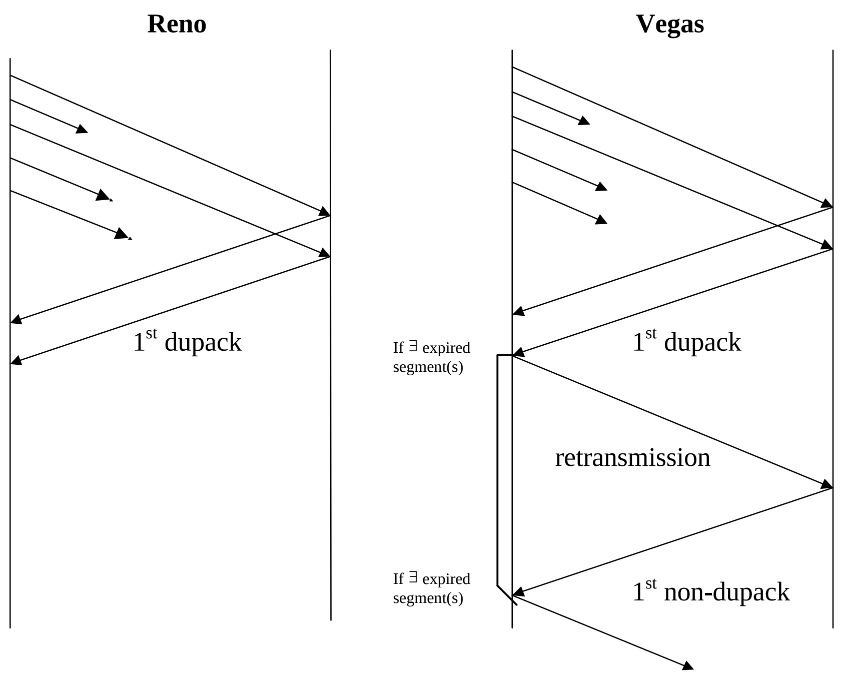


Figure 2.3 : TCP Vegas' enhanced fast retransmission (Illustration 3)
TCP Vegas's retransmission strategy allows it to retransmit a dropped segment after the receipt of the 1st non-duplicate ACK. This results in a more timely decision in case of multiple losses.

In addition, in order to avoid multiple reductions in the congestion window due to multiple losses in 1 RTT, TCP Vegas only decreases the congestion window if the retransmitted segment was previously sent after the last window decrease. This is to ensure that the congestion window is reduced as a result of the losses that occur at the current sending rate, and not due to the losses that happened at an earlier, higher sending rate.

In summary, the major benefit of TCP Vegas' aggressive retransmission strategy is that it enables TCP Vegas to retransmit the dropped segment although there may be no 2nd or 3rd duplicate acknowledgement. Otherwise, TCP will inevitably have to wait for timeouts before the retransmission can take place. Brakmo reported that this mechanism further reduced the number of coarse-grain timeouts in TCP Reno by more than half. Ahn [ADLY95] observed that this strategy was most effective in reducing timeouts in the slow-start phase. However, the retransmission strategy is more aggressive. Hence, it also increases the possibility of unnecessary retransmissions.

3.2 Modified congestion avoidance algorithm

TCP Vegas modifies TCP Reno's *congestion avoidance and control* mechanism that results in TCP's ability to anticipate congestion and adjusts its congestion window and sending rate accordingly. Instead of increasing the congestion window size blindly until losses occur, TCP Vegas tracks the changes in the throughput (or more specifically, changes in the sending rates) and then adjusts the congestion window size. It observes changes in the round-trip times of the segments that the connection has sent before. It then calculates and compares the measured throughput against the expected throughput. If the expected sending rate is higher than the actual sending rate by α or less, TCP Vegas fears that it is not utilizing the bandwidth efficiently by occupying some router buffers and thus increases the congestion window by one. If the expected rate is higher than the actual rate by β or more, TCP Vegas assumes that congestion starts to build up and thus decreases the congestion window by one. Otherwise, the congestion window remains unchanged. TCP Vegas makes this calculation and decision in window adjustment once per RTT. The increment, however, is made on receipt of each acknowledgement, resulting in an increase of about 1 per RTT. The new algorithm is summarized below :

During the congestion avoidance phase, TCP Vegas does :

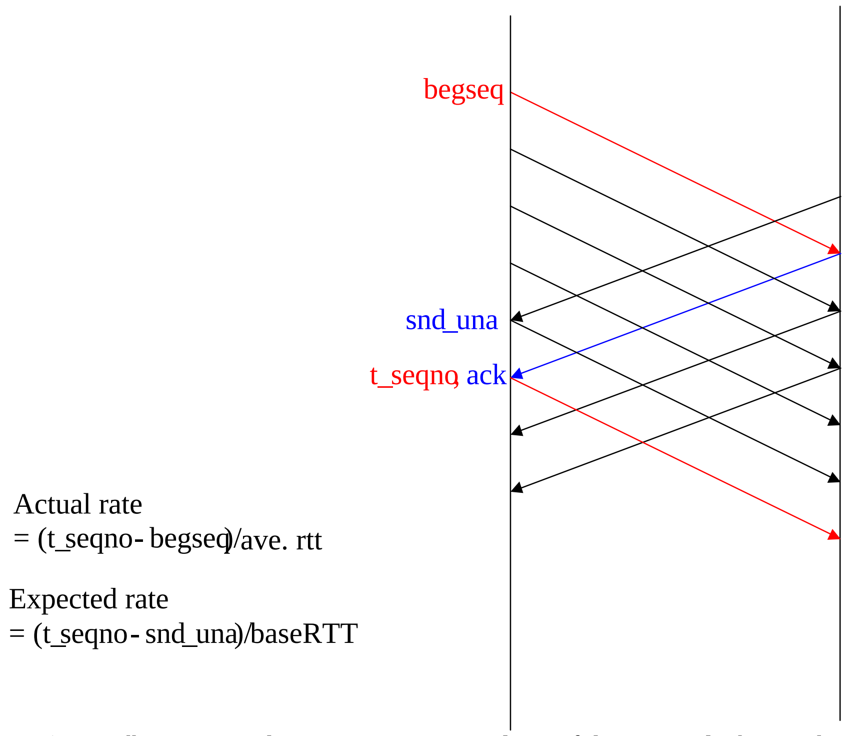
$$\text{cwnd} = \text{cwnd} + 1 \quad \text{if diff} < (\alpha / \text{baseRTT})$$

$$\text{cwnd} = \text{cwnd} \quad \text{if } (\alpha / \text{baseRTT}) \leq \text{diff} \leq (\beta / \text{baseRTT})$$

$$\text{cwnd} = \text{cwnd} - 1 \quad \text{if } (\beta / \text{baseRTT}) < \text{diff}$$

where

diff	=	expected rate – actual rate ≥ 0 by definition
expected rate	=	data in transit / baseRTT
baseRTT	=	the minimum of all measured RTTs, typically the RTT of a packet when the router queue is empty or when the flow is not congested (in seconds)
actual rate	=	(Next send sequence no. – segment timed) / average rtt
rtt	=	observed or actual round trip time (in seconds)
α, β	=	some constant thresholds



(Normally, t_seqno – begseq = t_seqno – snd_una if there is no duplicate acks or timeouts)

As shown in the above diagram, TCP Vegas calculates the number of segments that it has sent since it transmits the timed segment (begseq) by subtracting “begseq” from “t_seqno” (the next send sequence number). It then divides the result by the average of the sampled RTTs (ignoring retransmitted segments) to come up with a measured actual rate. It also calculates the difference between “t_seqno” and the “snd_una” (the next expected acknowledgement number) in order to measure the expected rate. However, since the expected rate is calculated before the “snd_una” variable is updated, “snd_una” refers to the “snd_una” after the previous acknowledgement has just been received. Under normal circumstances (i.e. when there are no duplicate acknowledgements and timeouts), “snd_una” is the same as “ack” (the sequence number being acknowledged). Hence, the difference between “t_seqno” and “snd_una” is the same as that between “t_seqno” and “ack”. In other words, it is the same as that between “t_seqno” and “begseq”. By definition, “baseRTT” is the minimum of all the RTTs. Hence, the expected rate is normally larger than the actual rate. The above calculation ignores the retransmitted segments.

The objective of TCP Vegas' congestion avoidance mechanism is to measure and control the amount of extra data in the network. These extra data inevitably has to be buffered in the router. It can be proved that, in the absence of other connections, the congestion window of TCP Vegas converges to a fixed value with between α to β extra bytes in the network.

In comparison, TCP Tahoe and TCP Reno perform congestion control, i.e. they control congestion once it happens. They treat packet losses as a signal of congestion. Therefore, they consistently cause packet losses to themselves and to other connections, resulting in under-utilization of the bottleneck link. TCP Vegas, on the other hand, performs congestion avoidance from end nodes (i.e. no router involvement is expected). It monitors changes in the sending rate (& RTTs) to predict congestion before losses occur. Empirical results show that TCP Vegas achieves better utilization of the bottleneck link.

3.3 Modified slow-start

TCP Reno doubles its window size every RTT during the slow start phase. TCP Vegas, on the other hand, doubles the window size only every other RTT during slow-start. In between, the congestion window stays fixed so that a valid comparison can be made. TCP Vegas modifies TCP Reno's slow start mechanism to incorporate congestion detection into slow-start. TCP Vegas estimates the available bandwidth by calculating the expected sending rate during the initial slow start period. When the actual sending rate falls below the expected sending rate by the equivalent of one router buffer, TCP Vegas reduces the congestion window by 1/8 and switches from slow start to linear increase/decrease mode (congestion avoidance) immediately. The modified algorithm is very effective in preventing losses and coarse-grain timeouts during the initial slow start phase [HBG00].

4. Enhanced TCP Vegas

In 1999, Hasegawa, Murata and Miyahara proposed Enhanced TCP Vegas as an alternative TCP algorithm to solve the unfairness they observed in TCP Vegas. The major difference between Enhanced TCP Vegas and TCP Vegas is that the control thresholds, α and β, defined in TCP Vegas are set to the same value in Enhanced TCP Vegas. Hasegawa et al. called this δ in their TCP congestion avoidance mechanism, which is summarized below. Other than this, Enhanced TCP Vegas uses the same aggressive retransmission strategy but conservative slow-start strategy as TCP Vegas.

During the congestion avoidance phase, Enhanced TCP Vegas does :

$$\begin{aligned} cwnd &= cwnd + 1, && \text{if } diff < (\delta \,/\, base\_rtt) \\ cwnd &= cwnd - 1\,, && \text{if } diff \geq (\delta \,/\, base\_rtt) \end{aligned}$$

5. Model of Vegas fairness

In their paper, Boutremans and Boudec [BB00] presented a simple analytical study of the steady state congestion window and throughput. The model summarizes 2 causes of unfairness in TCP Vegas and can be useful in explaining the unfairness of TCP Vegas. Therefore, I will present it here but I use 1 set of equations to cover both cases, i.e. when $\alpha < \beta$ and when $\alpha = \beta$.

The model assumes that the router adopts a FIFO discipline and the buffer size is infinite. There is no assumption on the congestion control mechanism at the router. Therefore, I assume that the model should be applicable to both DropTail and RED gateway.

We consider the model in which 2 users share a single bottleneck link. The propagation delay of connection *j* is d_j. The bandwidth of the bottleneck link is *c*. The minimum round-trip delay and the measured round-trip delay of connection *j* are $baseRTT_j$ and RTT_j respectively. I assume that the congestion window converges to a steady state value. I denote the difference between the expected rate and the sending rate as α_j (in packets), i.e.

$$\alpha_j \quad = (\text{expected rate} - \text{actual rate}) * \text{baseRTT}$$

Then, the minimum and the measured round-trip delay can be expressed as

$$BaseRTTj = dj + xj$$

where xj = over-estimation of the propagation delay

$$RTT_j = d_j + \quad _j$$

where $\quad _j$ = queuing delay experienced by the connection j

At steady state,

$$cwndj - (baseRTTj \,/\, RTTj)\ \ cwndj = \alpha_j$$

where $\alpha \le \alpha_j \le \beta$

or

cwndj = α_j (dj + $\ _j$) / ($\ _j$ – xj)

Hence, the expression for actual throughput of connection j is :

ratej = cwndj / RTTj = α_j / ($\ _j$ – xj)

The above expression explains why TCP Vegas can be unfair. One of the causes is the difference in values of α_j of the 2 connections even when xj is zero, i.e. when there is no over-estimation of the propagation delay. In particular, α_j is allowed to converge to any value between α and β. In addition, α_j represents the number of packets queued at the router buffer when the steady state is reached. According to the FIFO policy, the ratio of buffer occupancy in the FIFO gateway reflects the ratio of the bandwidth sharing. If α_j of the 2 connections are different, then TCP will not achieve fairness. There is an open question, though. What factors affect the converged value of α_j ?

Secondly, an over-estimation of the propagation delay of a particular connection (i.e. when $x_j \neq 0$) also results in an increase of its rate. According to Boutremans and Boudec, the effect is more pronounced when the queuing delay is close to the amplitude of the over-estimation of the propagation delay. Therefore, the connections with a more pronounced over-estimation of the propagation delay are favored.

However, this is only a simplified model. In reality, RTT_j can fluctuate even though $\alpha < \beta$. Therefore, $\ _j$, α_j and ratej can also fluctuate.

6. RED gateways

The RED Gateway was first presented by Sally Floyd and Van Jacobson in 1993 [FJ93]. The main goal of the RED gateway is to provide a congestion avoidance service at the gateway so that throughput can be kept high while the delay kept low. RED gateway achieves this by first detecting incipient congestion and then deciding which connections to notify of the congestion. In deciding which connection to notify, the RED gateway also aims to avoid 2 problems that can occur with the DropTail gateway : 1) a bias against bursty traffic, and 2) the global synchronization that results from notifying all connections to reduce their windows simultaneously. Another goal of the RED gateway is to control the average queue size even in the presence of non-cooperating sources that do not reduce throughput in response to congestion notification. The RED gateway achieves this by dropping the arriving packets when the queue size exceeds a maximum threshold.

RED uses 2 separate algorithms to detect incipient congestion and to decide which connections to notify in case of congestion.

DETECTING INCIPIENT CONGESTION

The RED gateway infers incipient congestion by updating the average queue size and comparing it against 2 thresholds : a minimum and a maximum threshold. The rule is as follow :

- When the average queue size is less than the minimum threshold, it infers that there is no congestion and thus no packets are marked (or dropped).

- When the average queue size is greater than the maximum threshold, the RED gateway infers heavy congestion and thus every arriving packet is marked (or dropped). As such, the maximum threshold can be interpreted as the gateway buffer size.
- When the average queue size is between the minimum and the maximum thresholds, each arriving packet is marked with a probability that is a function of the average queue size. Each time a packet is marked, the probability that a packet is marked from a particular connection is roughly proportional to that connection's share of the bandwidth at the gateway.

The RED gateway calculates the average queue size using a low-pass filter with an exponential weighted moving average :

ave_ = (1 – q_weight_) * ave_ + q_weight_ * curq_

where

ave_ = average queue size

curq_ = current queue size

q_weight_ = queue weight

By this equation, a short-term increase in the actual queue size that results from bursty traffic or from transient congestion will not lead to a significant increase in the average queue size. As such, the queue weight affects the size and duration of bursts (the degree of burstiness) that are allowed in the gateway.

The settings of the minimum and maximum thresholds are determined by the desired average queue size. In turn, the desired average queue size reflects the tradeoff between maximizing throughput and minimizing delay. However, the optimal average queue size is dependent on a number of factors, including the network characteristics and the traffic patterns. Therefore, the authors had left the optimal queue size as a question for further research.

In the paper, the authors provided some suggestions on the values of the parameters : the queue weight, the minimum threshold and the maximum threshold.

The queue weight affects the degree of burstiness that is allowed at the gateway. If it is set too large, then the averaging procedure will not filter out transient congestion at the gateway. On the other hand, if it is set too low, then the calculated average queue size will respond too slowly to changes in the actual queue size. In such case, the gateway is unable to detect the initial stages of congestion. In most of the simulations within the paper, the authors used a queue weight of 0.002. The default queue weight in the ns simulator is also 0.002.

The optimal values for the minimum and maximum thresholds depend on the desired average queue size. If the typical traffic is fairly bursty, then the minimum threshold must be sufficiently large in order to maintain the link utilization at an acceptably high level. The optimal maximum threshold depends, in part, on the maximum average delay that can be allowed by the gateway. The authors suggested that the RED gateway functions most effectively when the difference between the minimum and the maximum threshold was larger than the typical increase in the calculated average queue size in 1 round-trip time. Accordingly, they suggested that, as a useful rule of thumb, the maximum threshold be set to at least twice the minimum threshold. The default minimum and maximum thresholds in the ns simulator are 5 and 15 respectively.

PACKET-MARKING PROBABILITY

The algorithm for calculating the packet-marking probability determines how frequently the gateway marks or drops packets, given the current level of congestion. The goal is for the RED gateway to mark packets at fairly evenly spaced intervals, in order to avoid biases against bursty traffic and global synchronization, and to mark packets sufficiently frequently to control the average queue size.

The RED gateway uses randomization to choose which arriving packets to mark or drop. With this method, the probability of marking an arriving packet from a particular connection is roughly proportional to that connection's share of the bandwidth through the gateway.

The initial packet-marking probability, prob1_, is calculated as a linear function of the average queue size.

prob1_ = 1/linterm_ (ave_ - thresh_) / (maxthresh_ - thresh_)

where

prob1	=	initial packet-marking probability
thresh_	=	minimum threshold
maxthresh_	=	maximum threshold
1/linterm_	=	packet-marking probability when ave_ = maxthresh_
	=	an upper bound on the packet-marking probability

The final packet-marking probability is best calculated under the assumption that the number of arriving packets between marked packets is a uniform random variable. In such case, the final-marking probability is

prob2	=	prob1 / (1 – count * prob1)
where		
count	=	number of unmarked packets that have arrived since the last marked packet

According to the original paper, the parameter "linterm_" can be chosen from a fairly wide range because it is only an upper bound on the actual marking probability. If the congestion is so heavy that the gateway cannot control the average queue size by marking at most 1/linterm_ of the packets, then the average queue size will exceed the maximum threshold. Consequently, the gateway will mark every packet until congestion is controlled. For most of the simulations in the paper, the author set linterm_ to 50. The authors also suggested that 1/linterm_ should never be greater than 0.1. The ns default of linterm_ is 10.

Having considered the suggestions in the paper, I use the ns default of the queue weight. Therefore, I will leave the value of the queue weight unchanged in my simulation runs. I set linterm_ to the value of 50, which is adopted in most of the simulations in the original paper of RED. Based on the suggestion in this same paper, I set the maximum threshold to 3 times the minimum threshold. Here is a summary of the RED configuration that I use in my simulations :

q_weight_	0.002
thresh_	different from ns default, max. threshold = 3 x min. threshold
maxthresh_	different from ns default, max. threshold = 3 x min. threshold
linterm_	50
drop_tail_	true (i.e. packet at the tail of the queue will be dropped when the queue overflow

Chapter 3

SIMULATION RESULTS

In this chapter, I illustrate the fairness of TCP Vegas and Enhanced TCP Vegas using the simulation results that I have obtained. Using a network model with 2 TCP Vegas connections, I note that, when $\alpha < \beta$, there is no guarantee that the Vegas fairness is better than the Reno fairness. Previous researchers have reported that the unfairness of TCP Vegas is result of the difference in the values of α and β. When $\alpha < \beta$, TCP Vegas allows α to β extra packets to be buffered in the router. Therefore, if 2 connections share a bottleneck link, then in the worst case, it is possible that one TCP Vegas agent is buffering α extra packets in the router while the other TCP Vegas agent is buffering β extra packets in the router. With a FIFO router such as DropTail and RED, the ratio of throughputs will be β/α. For example, if $\alpha = 1$ and $\beta = 3$, then the ratio of throughputs can be as worst as 3, indicating a higher degree of unfairness. Besides, I also note from the simulation results that when this unfairness happens, there is a bias against connections with larger propagation delays. Luckily, the bias can be resolved by Enhanced TCP Vegas, except when $\alpha = \beta = 1$.

Hasegawa [HMM99] reported that, by setting $\alpha = \beta$, the fairness of TCP Vegas will be improved. He renamed this parameter as δ. I shall call this version of TCP Vegas Vegas-δ. However, I do not notice any study that evaluates the impact of the value of δ on the fairness of Enhanced TCP Vegas. Using a model with 2 Enhanced TCP Vegas connections, I run simulations to evaluate the fairness of Vegas-δ when δ=1, 2 or 3. I discover that the fairness of Vegas-1 is unstable. When the difference in propagation delays is large, the fairness of Vegas-1 is even worse than that of Vegas-1_3 (i.e. when α =

1 and β = 3). On the other hand, the fairness of Vegas-2 and Vegas-3 are stable and consistently good. In fact, the fairness of Vegas-2 and Vegas-3 is always better than that of Vegas1_3. The result seems to support that the fairness of Vegas-2 and Vegas-3 is independent of the propagation delay.

Most of the research work evaluates the fairness of TCP Vegas when there are just a few TCP Vegas connections. In this chapter, I present simulation results that reveal the fairness of TCP Vegas when many active flows share a bottleneck link. When there are many active flows with either TCP Reno or TCP Vegas only, the fairness of TCP Reno is better and more stable than that of TCP Vegas. Moreover, the fairness of Vegas-3 is better when the RED thresholds are higher. The results are obtained after having modified the ns implementation of the Reno-type timeout subroutine and some other parts of the ns code that may cause the TCP Vegas connections to be shut down. One implication is that a slightly different implementation can cause very different performance.

When there are many active flows with a mixture of TCP Vegas and TCP Reno connections, the fairness tend to deteriorate as the number of active flows increases. When the RED thresholds are low, TCP Vegas connections are able to capture higher goodput. When the RED thresholds are high and the number of flows is low, TCP Reno is the winner. Intuitively, this suggests that TCP Reno performs better when the RED thresholds are high enough to support a long enough queue. In such case, TCP Reno is able to open a larger window until packet losses. Besides, the variance of goodputs in the Vegas community is larger than that in the Reno community. Therefore, while the difference in the average goodput between the Vegas community and the Reno community contributes to the unfairness, the difference in goodputs in the Vegas community also contributes significantly to the unfairness. On closer inspection of the ns traces, I find that an incorrect minimum round-trip time recorded after a Reno type timeout causes some of the connections to run at very low throughput in comparison to other TCP Vegas connections. Hence, the variance of the goodput in the Vegas community can be poor. The findings point to possible incomparability between the Reno-type timeout subroutine and the Vegas algorithms. Moreover, I identify that TCP Vegas may perform unnecessary re-transmission.

With regard to the receiver acknowledgement strategy on fairness, I find that the fairness is no worse when all the receivers adopt the delayed acknowledgement strategy. In fact, the fairness is better, except when the RED thresholds are high and the number of flows is low. One possible explanation is that when the receivers adopt the delayed acknowledgement strategy, the minimum round-trip time and the actual round-trip time tend to be over-estimated. Therefore, the chance of an incorrect minimum round-trip time reduces, leading to lower variance of goodputs in the Vegas community and hence better fairness. Looking into the impact of the delayed acknowledgement on the estimation of the minimum round-trip time, I find that the first estimation is always over-estimated. However, since 2 back-to-back TCP segments are transmitted when the congestion window increases to 2, the receiver sends the acknowledgement nearly immediately after the first of these 2 segments have been received. Hence, TCP Vegas can eventually capture a minimum round-trip time that is much closer to the true value for calculation of the expected sending rate. Finally, when the receiver adopts the delayed acknowledgement strategy, the calculated "v_delta_" can be negative. The problem can be corrected by removing the 2nd term used in the calculation of the expected sending rate, similar to what has been mentioned in Hengartner's paper [HBG00].

In summary, the rest of this chapter includes :

1. a description of the network topology used in the simulation study
2. cases to illustrate the dynamics of TCP Vegas when there is only 1 connection
3. cases to illustrate the dynamics of TCP Vegas when 2 connections compete for the bottleneck bandwidth. The bias against connections with larger propagation delays will be mentioned.
4. a more detailed discussion of the unfairness of Vegas-1_3 (i.e. when $\alpha = 1$ and $\beta = 3$) against connections with larger propagation delays
5. a comparison of the fairness of TCP Reno and TCP Vegas
6. a comparison of the fairness of Vegas-1_3 (when $\alpha = 1$ and $\beta = 3$), Vegas-1 (when $\delta = 1$), Vegas-2 (when $\delta = 2$) and Vegas-3 (when $\delta = 3$) and the best choice of δ
7. a discussion of the fairness of TCP Vegas when many active flows share the bottleneck link
8. a brief discussion of the fairness of TCP Vegas when the receiver adopts the delayed acknowledgement strategy

1. Network model

The network topology I use consists of 2 sources (S1 & S2), 2 destinations (D1 & D2), and 2 routers (R1 & R2) as shown in Figure 3.1. Connection 1 starts from S1 to D1 and connection 2 starts from S2 to D2. Both connections are established via R1 and R2. The link between R1 and R2 is shared between the 2 connections. The bandwidth of the shared link is 1.5Mbps. The buffer size of router R1 is 100 packets. The propagation delay between S1 and D1, the shorter link (connection 1), is 6ms by setting x1 = 1ms. Thus, the minimum round-trip time of the shorter link, including processing delay of the packets in the intermediate routers, is 0.013211s. The propagation delay between S2 and D2, the longer link (connection 2), is varied by changing the value of x2. For example, if x2 = 30ms, the propagation delay of connection 2 is 64ms. Therefore, the minimum round-trip time of the longer path, including the transmission delay at the intermediate routers along the path, is 0.071211s.

The behavior and performance of TCP Vegas when the RED router is used is investigated. The RED minimum and maximum thresholds are 15 and 45, respectively. Larger thresholds (instead of the typical values of 5 and 15, respectively) are chosen to ensure that the limit on the average queue length cannot affect the performance of the TCP senders.

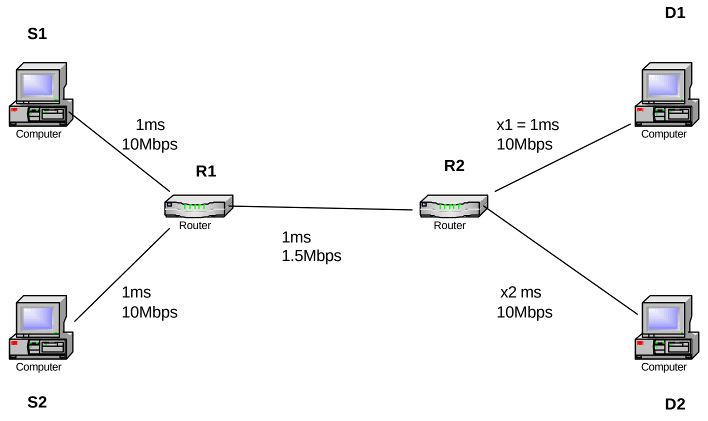


Figure 3.1 : The simulation network model

The maximum window size that a TCP connection can reach will have an impact on its performance. If the maximum window size is too small relative to the available bandwidth of the network path, TCP connection will not be able to fully utilize the available capacity. Alternatively, if the maximum window size is too large for the network to handle, the congestion window will eventually grow to a point where TCP will overwhelm the network with too many segments, some of which will be discarded before reaching the receiver. Fortunately, if the transfer is long enough and if the maximum window size is large enough to overwhelm the network, the TCP congestion control algorithm will find a congestion window that is appropriate for the network path. In my simulation, I set the window size to 30. Moreover, I limit the maximum value of the minimum round-trip time (including the processing delay) to ensure that the window size does not pose a limit on the performance of TCP. For TCP Vegas, α and β are set to 1 and 3, respectively. I shall call this version of TCP Vegas Vegas1_3.

Both connections start their own FTP transfer simultaneously. The sources are greedy sources such that they always have data to send. The packet size is 1000 bytes. Hence, the minimum round-trip time of connection 1, the shorter link, is equivalent to 2.48 packets (0.013211 seconds x 1.5Mbps / 8000 bits). When x2 = 30ms, then the minimum round-trip time of connection 2 is equivalent to 13.35 packets (0.071211 seconds x 1.5Mbps / 8000 bits). If the bandwidth is equally shared among both connections and the bandwidth is fully utilized, then connection 1 should maintain on average about 2.24 to 4.24 packets in the network while connection 2 should maintain on average about 7.67 to 9.67 packets in the network.

Each simulation lasts for 60 seconds. I use the ns simulator developed at the UC Berkeley to run the simulations. 60 seconds is considered an adequate length of time because TCP Vegas congestion window normally converges in less than 5 seconds.

The average throughputs (or average sending rate of the source) of both connections are observed after the simulation has started for 10 ms. I also track the congestion window against time. In case of TCP Vegas, the minimum round-trip time, the actual round-trip time, the actual sending rate, the expected sending rate and the estimated queue backlog (v_delta_) against time are also recorded.

In the second simulation setting, I set x1 to 50ms. Therefore, the propagation delay of connection 1 is 104ms. With this configuration, the propagation delay of connection 2 is smaller than that of connection 1 with some values of x2 but larger than that of connection 1 with other values of x2. This allows me to evaluate the fairness of TCP Vegas with a different configuration.

For easy reference, the following table gives a list of the settings of x1 and x2, together with the corresponding propagation delay and the minimum round-tip times.

x2	**Round-trip propagation delay**	**Minimum round-trip time**
110ms	224ms	0.231211 seconds
100ms	204ms	0.211211 seconds
90ms	184ms	0.191211 seconds
80ms	164ms	0.171211 seconds
70ms	144ms	0.151211 seconds
60ms	124ms	0.131211 seconds
50ms	104ms	0.111211 seconds
40ms	84ms	0.091211 seconds
30ms	64ms	0.071211 seconds
20ms	44ms	0.051211 seconds
10ms	24ms	0.031211 seconds
5ms	14ms	0.021211 seconds
1ms	6ms	0.013211 seconds

Table 3.1 : Table of round-trip propagation delay & minimum RTT vs. x2

2. Performance metrics

2.1 Fairness measure

Jain's fairness index [J91], a very common fairness index, is used to evaluate the fairness among senders with different round-trip times. Given a set of flow throughputs (b_1, b_2, ….., b_n), the fairness index is defined as :

$$\left(\sum_{i=1}^{n} b_i\right)^2 \Big/ n \times \left(\sum_{i=1}^{n} b_i^2\right)$$

b_i denotes an allocation metric that may be throughput, throughput times hops or response time.

Since the throughputs are nonnegative, the fairness index is always between 0 and 1. A lower value implies poorer fairness. If all throughputs are the same, the fairness index is 1.

If the link is shared equally among any k of the n senders (others zero), the Jain's fairness index is equal to k/n. If the bottleneck link is shared by 2 users and the Jain's index is 0.5, then one of them is effectively shut out. If the fairness index is 0.9, then one of the connections is getting 2/3 of the bandwidth. Therefore, a fairness index of less than 0.9 should not be considered a good value.

Figure 3.2 below plots the fairness against the proportion of bandwidth obtained by the connection with lower throughput. The network model is the same as that I have described in section 1 of this chapter. There are only 2 connections in the network model.

Here is a summary of some data points with their approximate fairness measure read from the graph.

Proportion of available bandwidth	Fairness Index
0.50	1.00
0.45	0.99
0.40	0.96
0.35	0.92
0.30	0.84
0.00	0.50

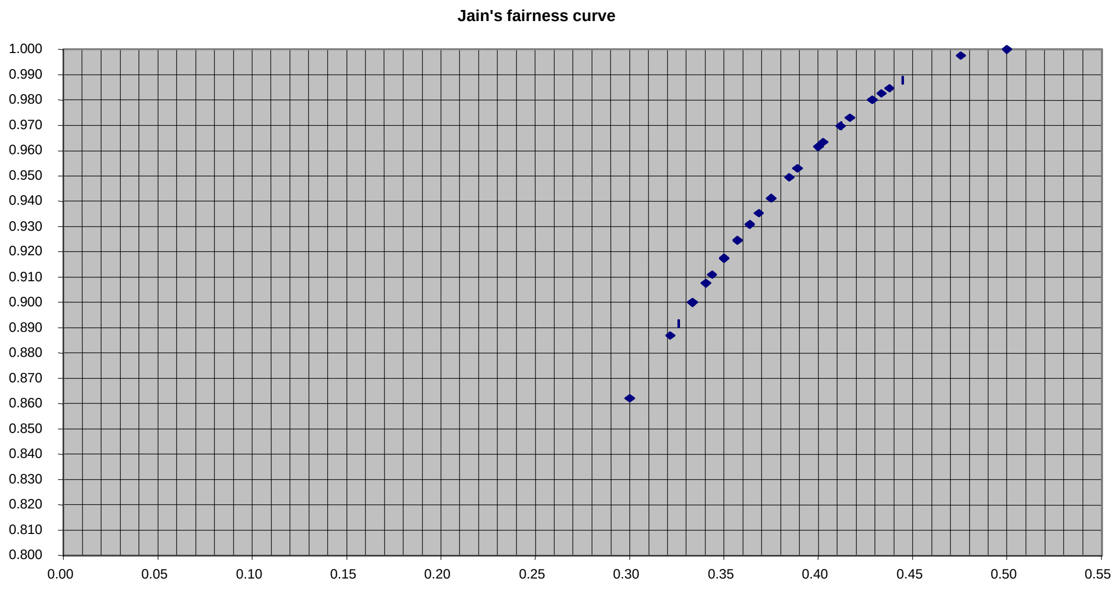


Figure 3.2 : The fairness curve
A plot of the fairness against proportion of bandwidth obtained by the connection with lower throughput

It can be noted that although the proportion of the available bandwidth obtained by the connection with lower throughput drops from 0.5 to 0.4, the fairness index drops only slightly by 0.04, from 1.00 to 0.96. However, when the proportion drops by another 0.1 from 0.4 to 0.3, the fairness index drops by 0.12. The rate of change in the value of the fairness index increases as the proportion of the bandwidth reduces.

The implication is that a relatively high value of the fairness measure, say 0.92, is not a good fairness measure. From figure 3.2, it can be seen that, when the fairness index is 0.92, one of the connections gets only 35% of the available bandwidth.

2.2 Goodput

Throughout the study, I use the average "goodput" instead of the average throughput to calculate the fairness index and to compare the performance. Goodput, also called effective throughput, is defined as the throughput of successfully transmitted packets, excluding the retransmissions of the lost or the corrupted packets. It is determined by the number of bits successfully forwarded to the receiver of the TCP connection and the round-trip time.

I use goodput instead of throughput because I believe the inefficient use of the bandwidth should be discouraged. Goodput is considered as a better measure because it ignores the wasteful bits and thus lead to lower measure when the connection uses the bandwidth ineffectively.

3. Simulation Tool

I use the ***ns*** simulator developed at the UC Berkeley to run the simulations. It is available at the web site http://www.isi.edu/ns/nam or http://www-mash.cs.berkeley.edu. The ***ns*** simulator is an object-oriented simulator, written in C++, and uses the Otcl interpreter as the command and configuration interface. The ***ns*** simulator does not incorporate the real world TCP protocol implementation. For example, there is no SYN/FIN connection establishment or teardown. Moreover, no data is ever transferred, e.g. there is no processing of TCP checksum and urgent data. However, it has the major advantages of having readily available coding for running simulations with TCP Vegas and has been used in research work on TCP Vegas [MLAW99].

For easy reference, I summarize some of the variables used in the ***ns*** code of TCP Vegas here.

• v_baseRTT_	minimum round-trip time
• v_actual_	actual sending rate
• v_expect_	expected sending rate
• v_delta_	(v_expect_ - v_actual_) * baseRTT_
• cwnd_	congestion window
• ssthresh_	congestion threshold
• v_rtt_	Vegas measured RTT of a segment

I have compared it against University of Arizona's x-Kernel-Vegas. Based on observations in the comparison of program codes and the initial set of simulation traces, I have made 2 modifications to the ***ns*** simulator.

1. The original ns code set v_baseRTT_ to the average of the measured v_rtt_ either if 1) average v_rtt_ is less than the current value of v_baseRTT_ or 2) if there is only 1 packet in transit in the last RTT. The Arizona code uses only the 2nd checking, not the first. However, according to the simulation traces, I notice that the 2nd condition is normally satisfied at the beginning of the simulation and thus set the v_baseRTT_ to a value that is normally greater than the true minimum value. Therefore, I have removed the 2nd checking from the IF statement.

2. In ns, the original data type of v_delta_ is “integer”. I have changed it to “double”. When the data type is “integer”, the simulation traces of Enhanced TCP Vegas shows that the congestion window converges to a fixed value. This is different from the behavior of Enhanced TCP Vegas that is described in Hasegawa et al.’s paper. After I have changed the data type of “v_delta” to “double”, the dynamics of the congestion window resembles to that described in Hasegawa et. al.’s paper, i.e. the congestion window oscillates instead of converging to a fixed value as with Brakmo and Peterson’s TCP Vegas. Moreover, it gives a more timely entry from the slow start phase into the congestion avoidance phase and may avoid bandwidth shooting during the slow start phase.

4. Simulation Results

In this section, I use the simulation results to illustrate the dynamics and the characteristics of TCP Vegas. Instead of just focusing on the steady state behavior, I illustrate what happens from the start of the transmission.

I first go through what happens when the propagation delay is 6ms or 64ms. These two simulations serve as the baseline for comparison to the simulations in which 2 connections compete for the bottleneck bandwidth.

4.1 A case with a single connection

In this section, I present the dynamics and the behavior of TCP Vegas when there is only 1 connection, using the simulation data. I hope the pictorial graphs and figures can provide a deeper understanding into the dynamics of TCP Vegas. In addition, the results will serve as the baseline reference in later sections.

4.1.1 A case with a single TCP Vegas connection (when $a = 1$ and $b = 3$)

In this section, I present the simulation data obtained when there is only 1 Vegas1_3 connection. The following result is obtained by setting the propagation delay of connection 1 to 6ms. Including the transmission delay, the minimum round-trip time is 0.013211 seconds. Figure 3.3 shows the dynamics of the congestion window (cwnd_), the difference between the expected sending rate and the actual sending rate (v_delta_) and the actual queue against time. Figure 3.4 shows the actual round-trip time measured by the Vegas algorithms against time. Figure 3.5 shows the expected sending rate and the actual sending rate against time. The data points are connected by lines to merely present the data series more clearly. The congestion window only changes at certain point in time represented by the data point. It does not change continuously as the time moves forward.

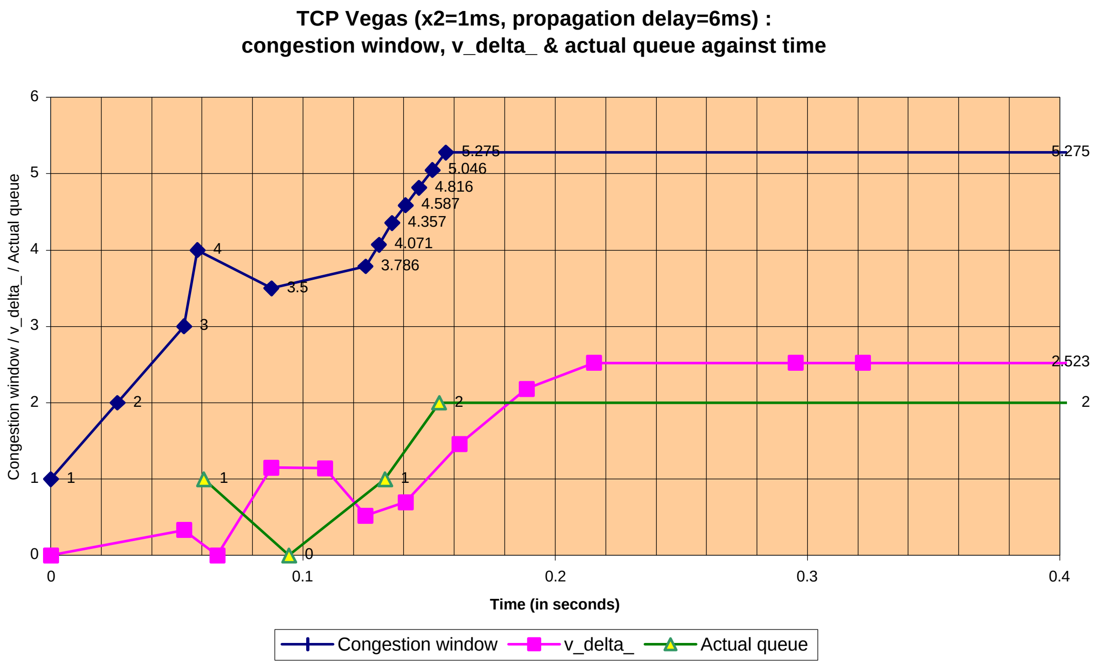


Figure 3.3: Single Vegas1_3 (Propagation delay = 6ms) - cwnd_, v_delta_ & actual queue against time

The slow-start behavior of TCP Vegas can be observed from figure 3.3 above. The congestion window of TCP Vegas grows from 1 to 2 and then from 2 to 4 every other round-trip time (i.e. about 2 x 0.013211 seconds). This is TCP Vegas' modified slow-start phase. When the congestion window reaches 4, TCP Vegas discovers that there is more than 1 extra packets in the intermediate router, which it interprets as a bandwidth overshoot. Therefore, it decreases its congestion window by 1/8 to 3.5. This triggers the congestion avoidance phase.

During the congestion avoidance phase, since $\alpha = 1$, TCP Vegas increases its congestion window by 1/cwnd_ upon the receipt of an acknowledgement as long as v_delta_ is greater than 1. TCP Vegas decides only once per RTT by how much the congestion window should be increased. When the congestion window is 3.5, it detects that it should increase the congestion window by 1/3.5. Hence, the congestion window increases from 3.5 to 4.357, in step of 1/3.5 in the next RTT. Then, at 0.13539 seconds, when it receives an acknowledgement for a Vegas-timed segment, TCP Vegas checks the value of v_delta_ again to decide the direction of window adjustment. Since v_delta_ is still greater than 1 but less than 3, it decides that it should keep on increasing its congestion window. Therefore, it increases its congestion window in step of 1/4.357 from 4.357 to 5.275 in the following RTT. Then, TCP Vegas receives another acknowledgement for a Vegas-timed segment, so it checks the value of v_delta_ again. This time, it finds that v_delta_ is 2.5. Hence, TCP Vegas stops increasing its congestion window because v_delta now lies between 1 and 3. Therefore, the congestion window finally settles at 5.275. Since TCP cannot transmit part of a packet, this means that TCP Vegas keeps on transmitting about 5 packets every round-trip time after the steady state has been attained.

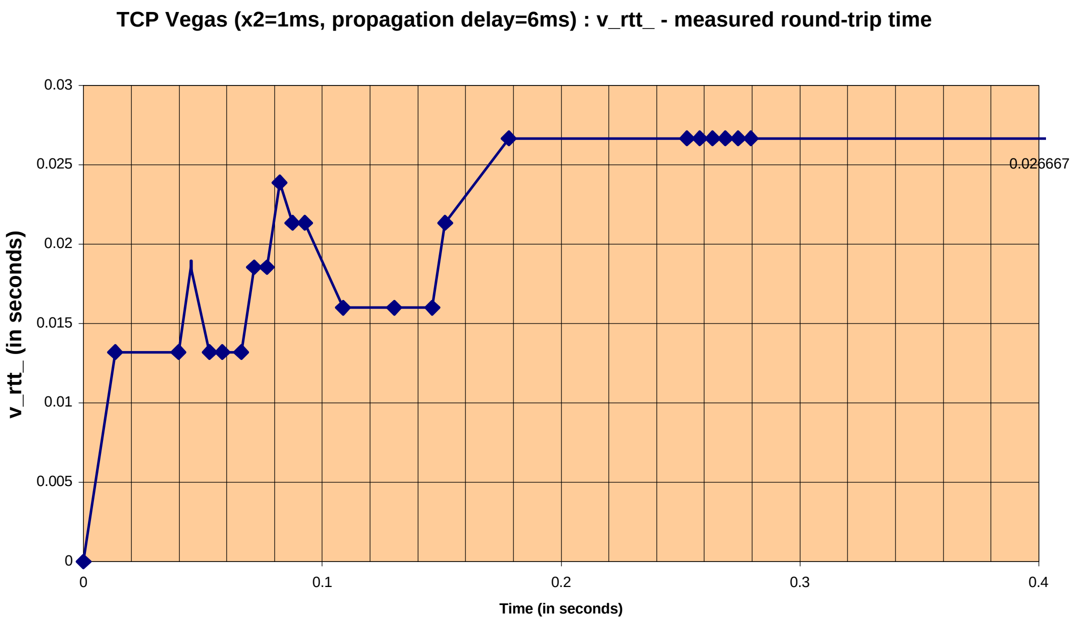


Figure 3.4 : Single Vegas1_3 (Propagation delay = 6ms) - Measured RTT against time

At steady state, the packet's round-trip time is 0.026667 seconds. The queuing delay is thus 0.013456 (0.026667 – 0.013211) seconds. Since the bandwidth of the bottleneck link is 1.5Mbps and the packet size is 1000 bytes, it takes about 0.005333 (8000 / 1.5M) seconds for the router to transmit a packet. Hence, a queuing delay of 0.013456 seconds corresponds to about 2.523 (0.013456 / 0.005333) packets, which is equal to the steady state v_delta_. This should be expected because v_delta_ measures the number of extra packets that TCP Vegas keeps in the network. These extra packets have to be buffered in the intermediate routers.

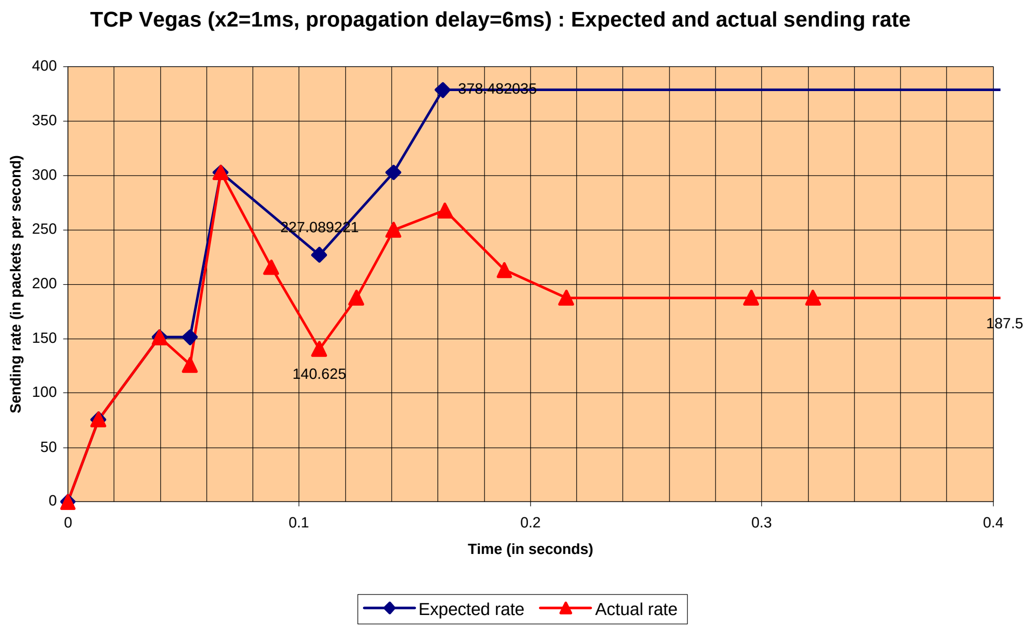


Figure 3.5 : Single Vegas1_3 (Propagation delay = 6ms) - Expected & actual sending rates against time

Figure 3.5 above shows that the expected sending rate and actual sending rate converges to 378.48 packets per second and 187.5 packets per second, respectively. As I have mentioned before, TCP Vegas sends about 5 packets per RTT after the steady state has been reached. With a minimum round-trip time of 0.013211 seconds, the expected rate is indeed 5 / 0.013211 = 378.48 packets per second. Since the round-trip time converges to 0.026667 as shown in figure 3.4, the actual sending rate was indeed 5 / 0.026667 = 187.5 packets per second. TCP Vegas calculates v_delta_ as (expected rate – actual rate) * baseRTT_ to decide the direction of window adjustment. Therefore, at the steady state, v_delta_ = (378.48 – 187.5) x 0.013211 = 2.523. Since v_delta_ is now between $\alpha = 1$ and $\beta=3$, there is no more adjustment of the congestion window.

I have calculated previously that with a minimum round-trip time of 0.013211 seconds and a bottleneck bandwidth of 1.5Mbps, TCP Vegas can keep 2.477 packets in the network without having to occupy any router buffers. With a steady state congestion window of 5, TCP Vegas needs to keep about 3 (5 – 2) packets in the intermediate router. It can be seen from figure 3.3 that the actual queue in the steady state is also fixed at 2. Including the packet being transmitted by the router, there are about 3 packets in the router buffer.

The following result is obtained by setting the propagation delay of connection 1 to 64ms (i.e. x1 = 30ms) instead of 6ms as in the previous case. Again, only connection 1 transmits packets. The minimum round-trip time is 0.071211 seconds. Accordingly, when the pipe is full, TCP Vegas should be keeping about 13.35 packets in the network without having to occupy any router buffers.

Figures 3.6 shows the dynamics of the congestion window (cwnd_), the difference between the expected sending rate and the actual sending rate (v_delta_) and the actual queue against time. Figure 3.7 shows the actual round-trip time measured by the Vegas algorithms against time. Figure 3.8 shows the expected sending rate and the actual sending rate against time.

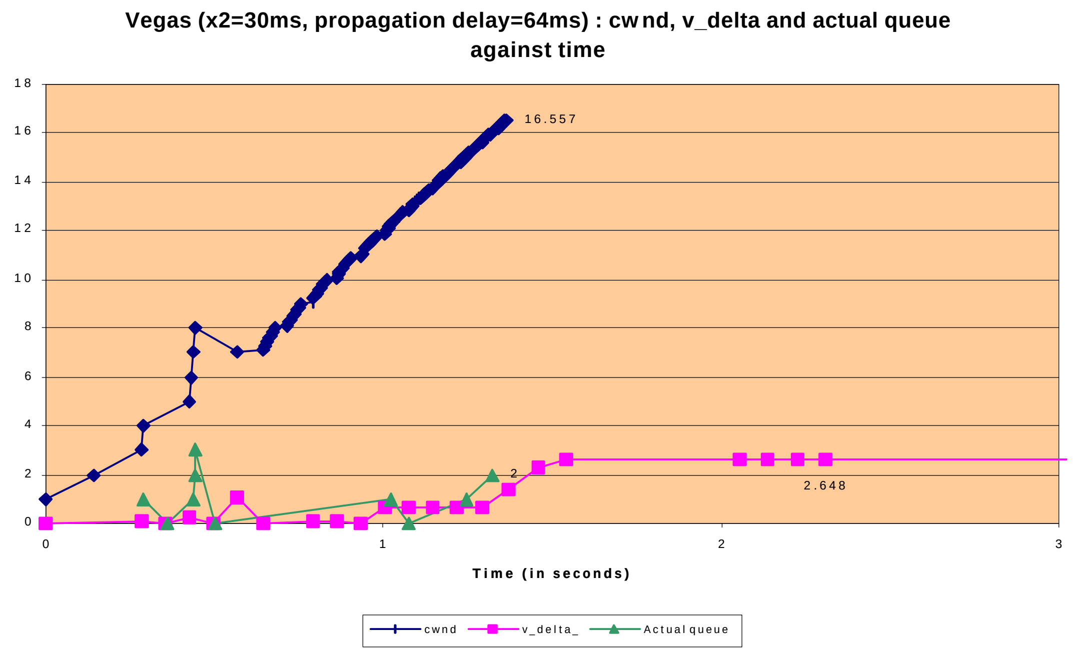


Figure 3.6 : Single Vegas1_3 (Propagation delay = 64ms) – cwnd_, v_delta_ & actual queue against time

It can be observed that the congestion window converges to 16 in this simulation setting. With a recorded minimum round-trip time of 0.071211 seconds, the expected sending rate converges to 16/0.071211 or 224.68 packets per seconds. The converged actual round-trip time is 0.085333 seconds. Accordingly, the actual sending rate is 16/0.085333 or 187.5 packets per seconds. Hence, the converged v_delta_ is (224.68 – 187.5) * 0.071211 or 2.648 packets, which is consistent with that shown in the plot of v_delta_ in figure 3.6 above. Similar to the previous setting, TCP Vegas needs to keep about 3 (16 – 13.35 or 2.65) extra packets in the router buffer.

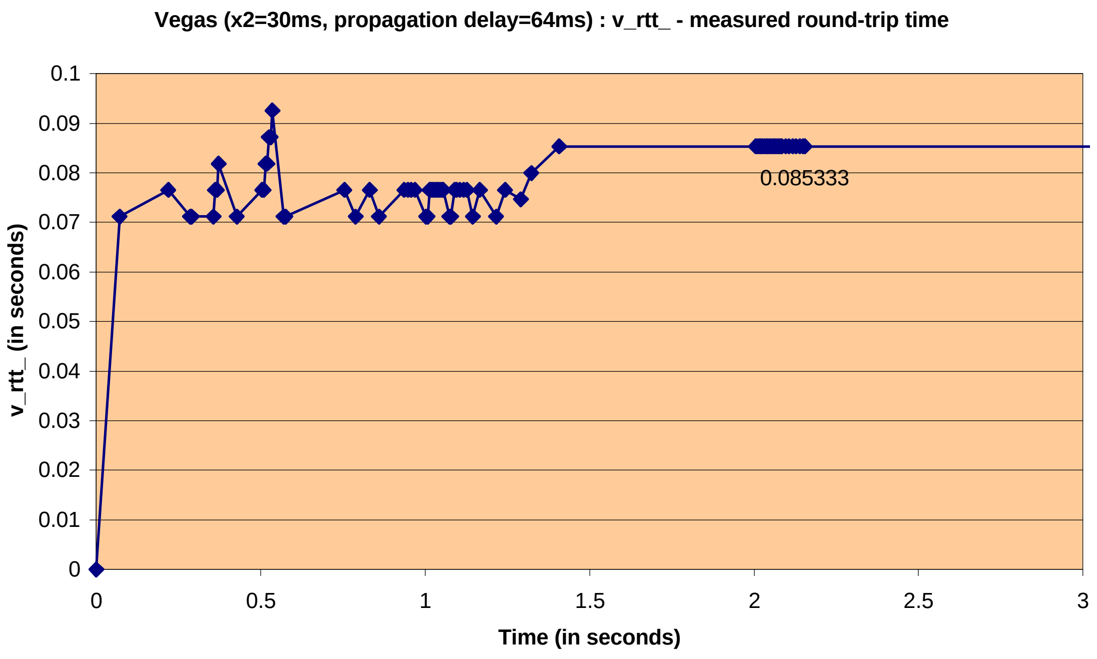


Figure 3.7 : Single Vegas1_3 (Propagation delay = 64ms) - Measured RTT against time

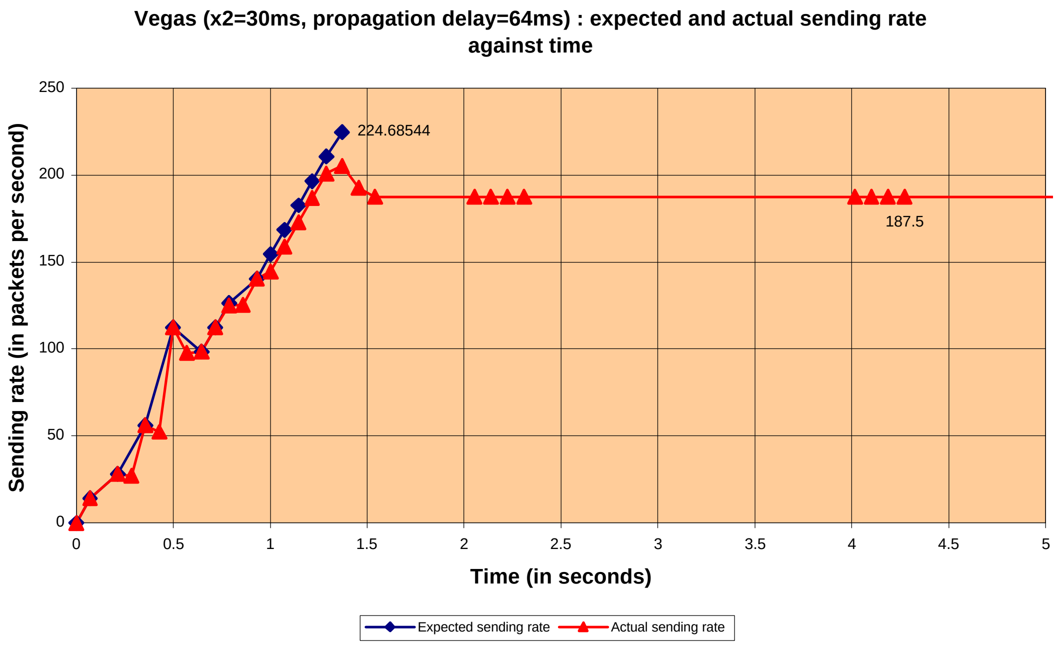


Figure 3.8 : Single Vegas1_3 (Propagation delay = 64ms) - Expected & actual sending rates against time

I notice that the stability of Vegas1_3 is very good. It is clear from the above discussion that, in both cases, the congestion window, the sending rates and the round-trip time eventually converge to a fixed value. The actual number of packets queued at the router also converges and is the same in both cases.

From figure 3.8, it can be noticed that the actual rate matches closely with the expected rate at the beginning. Eventually, the actual sending rate starts to decrease. The change in the congestion window and the sending rate stops when the v_delta_ reaches about 2.5 to 2.6 as it now lies between 1 and 3. In both cases, the congestion window settles at the maximum value that is allowed by the bandwidth-delay product and the parameters of the Vegas agent, i.e. the bandwidth-delay product plus β. The bottleneck link is fully utilized.

Comparing to the case when the propagation delay of the shorter connection is 6ms, I notice that although the expected sending rates converges to different values at steady state, the actual sending rates are the same in both cases. However, the gap between the expected sending rate and the actual sending rate is wider when the propagation delay is smaller. This is understandable because v_delta_, which reflects the difference between the expected rate and the actual rate and hence the number of extra packets that are allowed in the network, is measured in packets. Since the number of extra packets that is allowed in the network are the same in both cases, the difference between the expected sending rate and the actual sending rate will be larger for the shorter connection.

4.1.2 A case with a single Enhanced TCP Vegas connection

In this section, I present the dynamics of Vegas-δ ($\alpha = \beta = \delta$) when there is only one Vegas-δ connection in the network. Again, the simulations use the same network model described before in section 1 of this chapter. However, this time, I set $\alpha = \beta = \delta$ and δ to 1, 2, 3, 4, 5 or 10. The propagation delay is 64ms.

As an example, figures 3.9, 3.10 and 3.11 show the sending rates, the congestion window, v_delta_ and the actual queue when $\alpha = \beta = 3$, respectively. They are all measured after the simulation has started for 20 seconds.

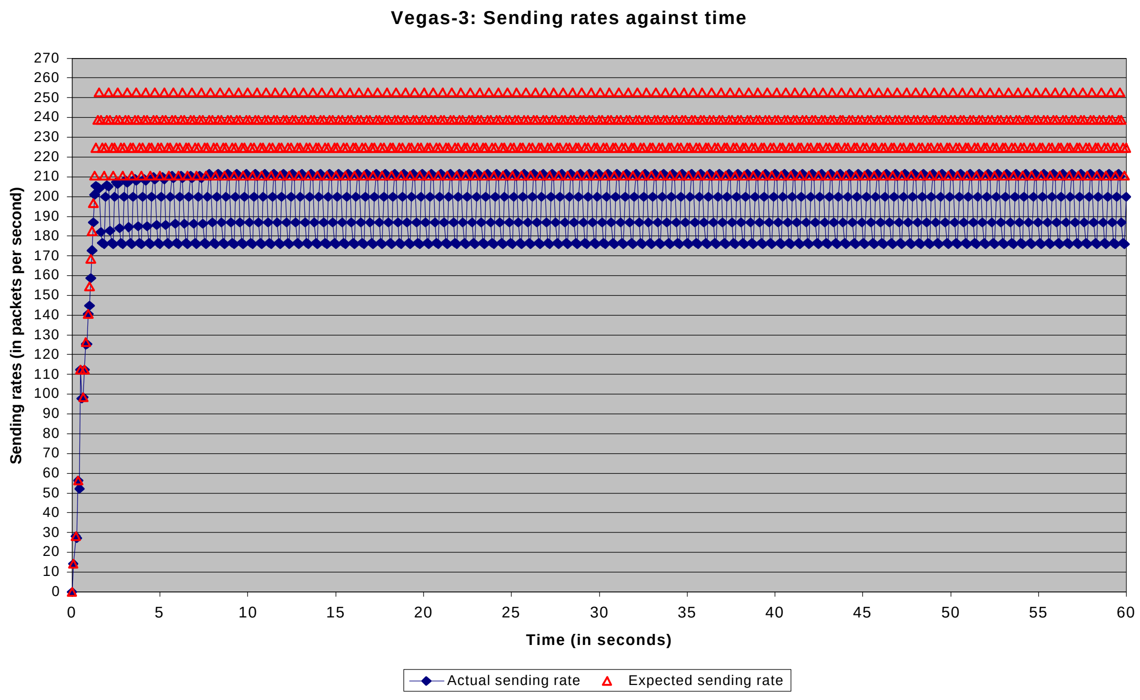


Figure 3.9a : Single Vegas-3 - Expected & actual sending rates against time (whole simulation)

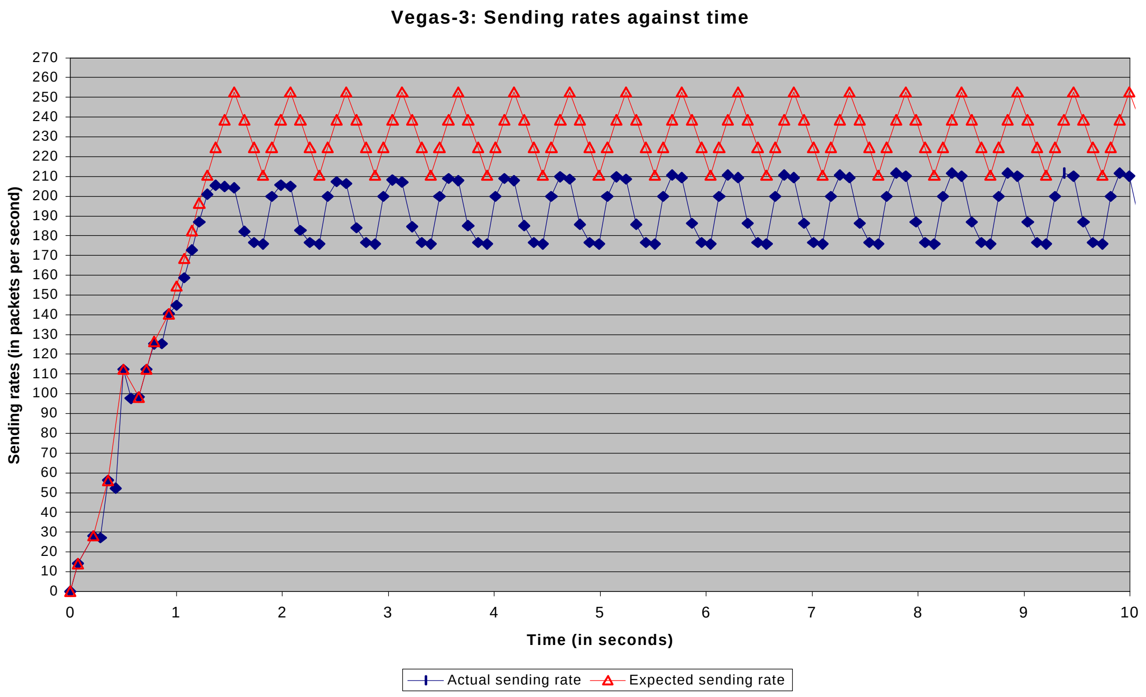


Figure 3.9b : Single Vegas-3 - Expected & actual sending rates against time (first 10 seconds of simulation time)

Vegas-δ	Min. actual sending rate	Max. actual sending rate	(Max. – Min.) actual rate	Min. expected sending rate	Max. expected sending rate	(Max. – Min.) expected rate
Vegas-1	174.11	213.27	39.16	182.56	238.73	56.17
Vegas-2	175.00	213.27	38.27	196.60	252.77	56.17
Vegas-3	175.78	211.62	35.84	210.64	252.77	42.13
Vegas-4	176.47	210.16	33.69	224.68	266.81	42.13
Vegas-5	177.08	208.88	31.80	238.73	280.86	42.13
Vegas-10	179.35	204.14	24.79	308.94	351.07	42.13

Table 3.2 : Maximum / minimum expected & actual sending rates vs. Vegas-δ

Figure 3.9a and 3.9b show that the expected and the actual sending rates no longer converge to fixed values. Instead, they fluctuate. From table 3.2 above, it can be observed that the range of the fluctuation of the actual sending rate becomes narrower as δ increases.

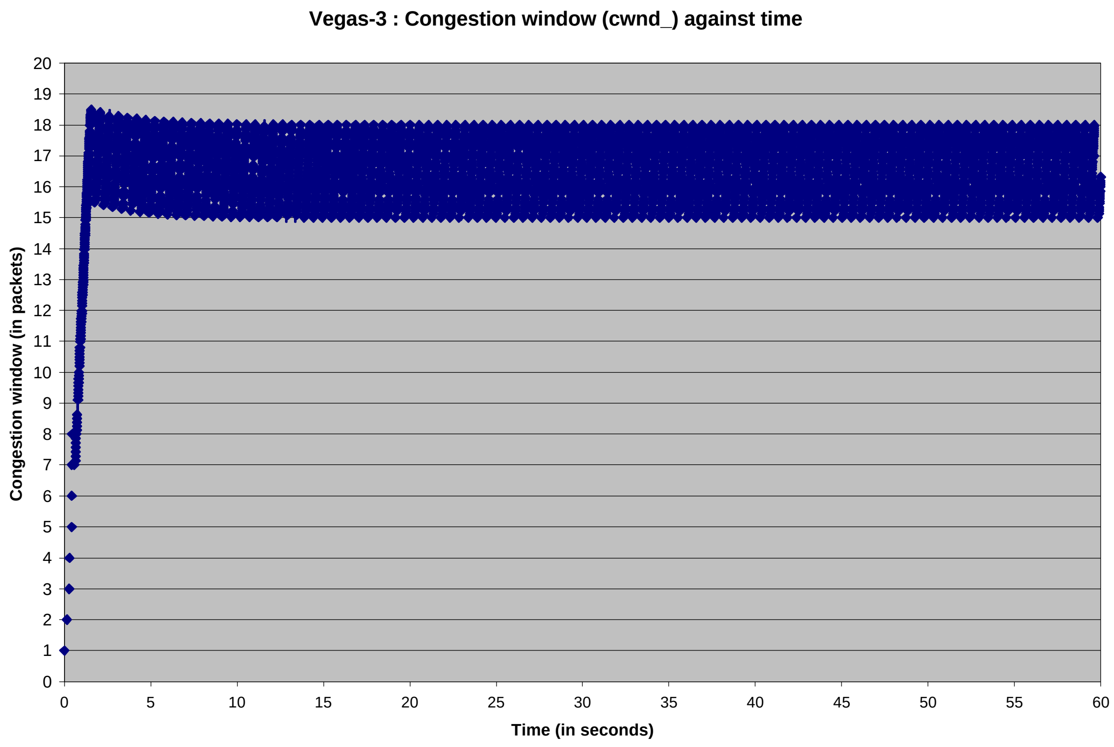


Figure 3.10a : Single Vegas-3 - Congestion window against time (whole simulation)

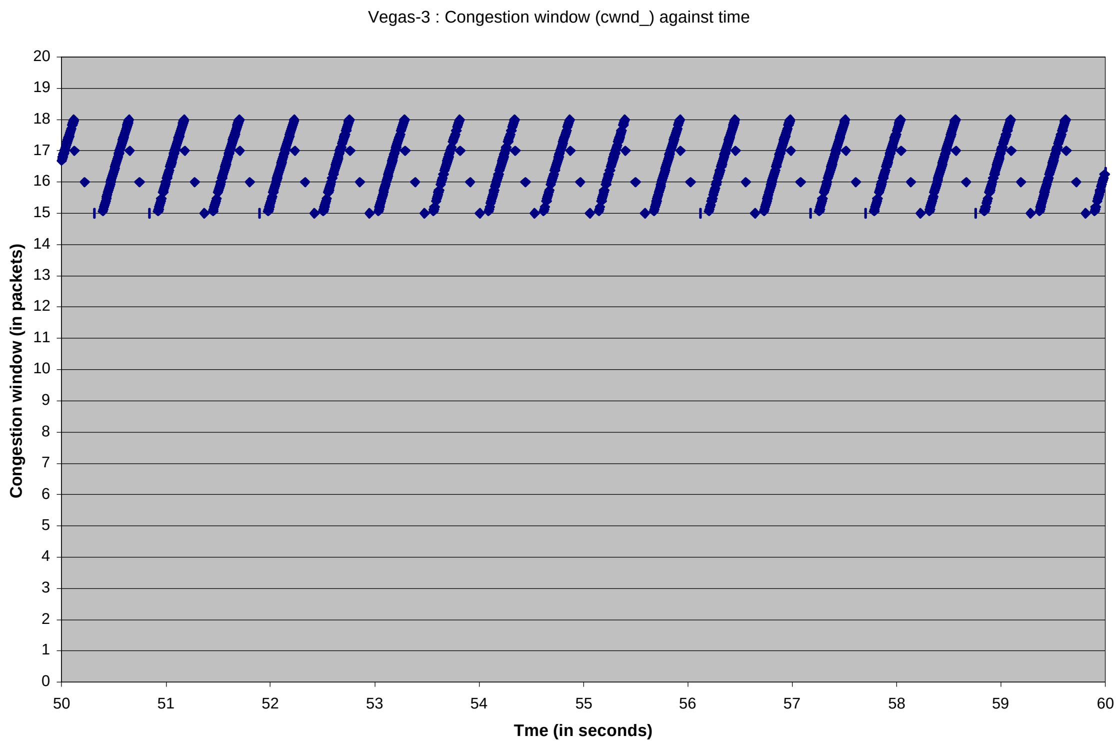


Figure 3.10b : Single Vegas-3 - Congestion window against time (simulation time from 50 to 60 [seconds])

	Min. congestion window	Max. congestion window	Range	Average
Vegas-1	13	17	4	15
Vegas-2	14	18	4	16
Vegas-3	15	18	3	16.5
Vegas-4	16	19	3	17.5
Vegas-5	17	20	3	18.5
Vegas-10	22	25	3	23.5

Table 3.3 : Maximum & minimum congestion window vs. Vegas-δ

It is obvious from figure 3.10a that the congestion window also fluctuates. From figure 3.10b, which shows the movement of the congestion window in the last 10 seconds simulation time, it can be seen that the congestion window increases from 15 to 18 during a RTT. Then, since v_delta_ becomes greater than 3, the congestion window is then decreased by 1 until it reaches 15. The cycle repeats.

Table 3.3 shows that the fluctuation of the congestion window is smaller when δ is greater than or equal 3 than when δ is 1 or 2. Therefore, based on the stability of the congestion window, a value of 3 is more suitable for δ than a value of 2.

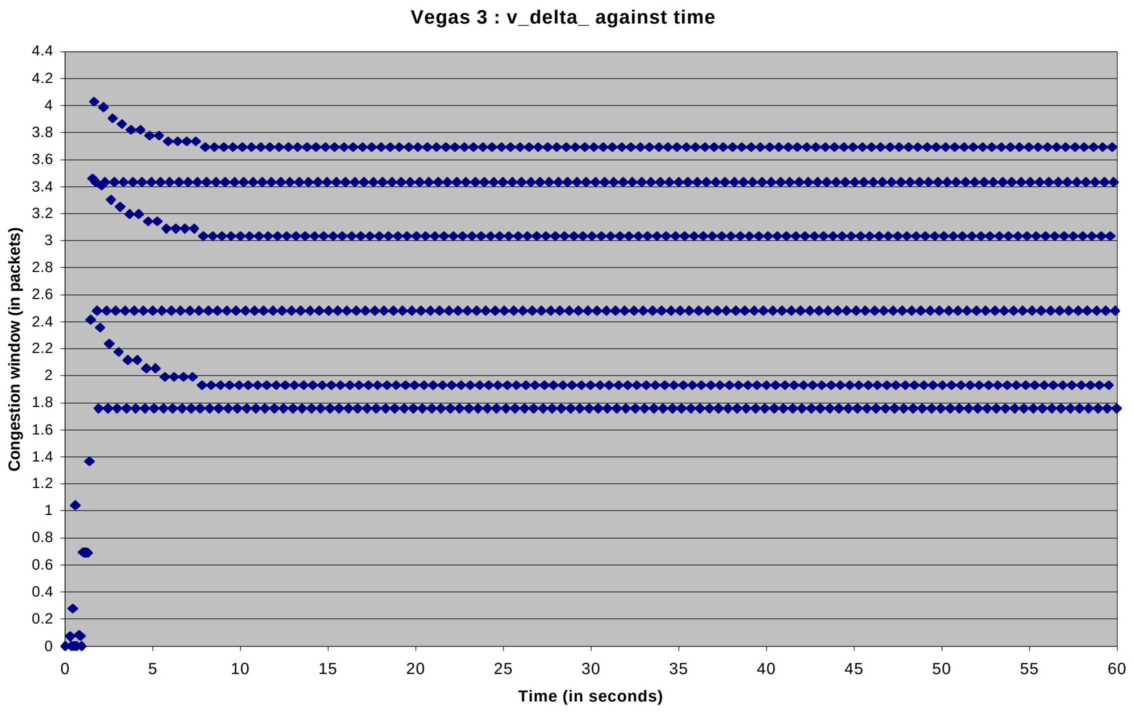


Figure 3.11a : Single Vegas-3 - v_delta_ against time

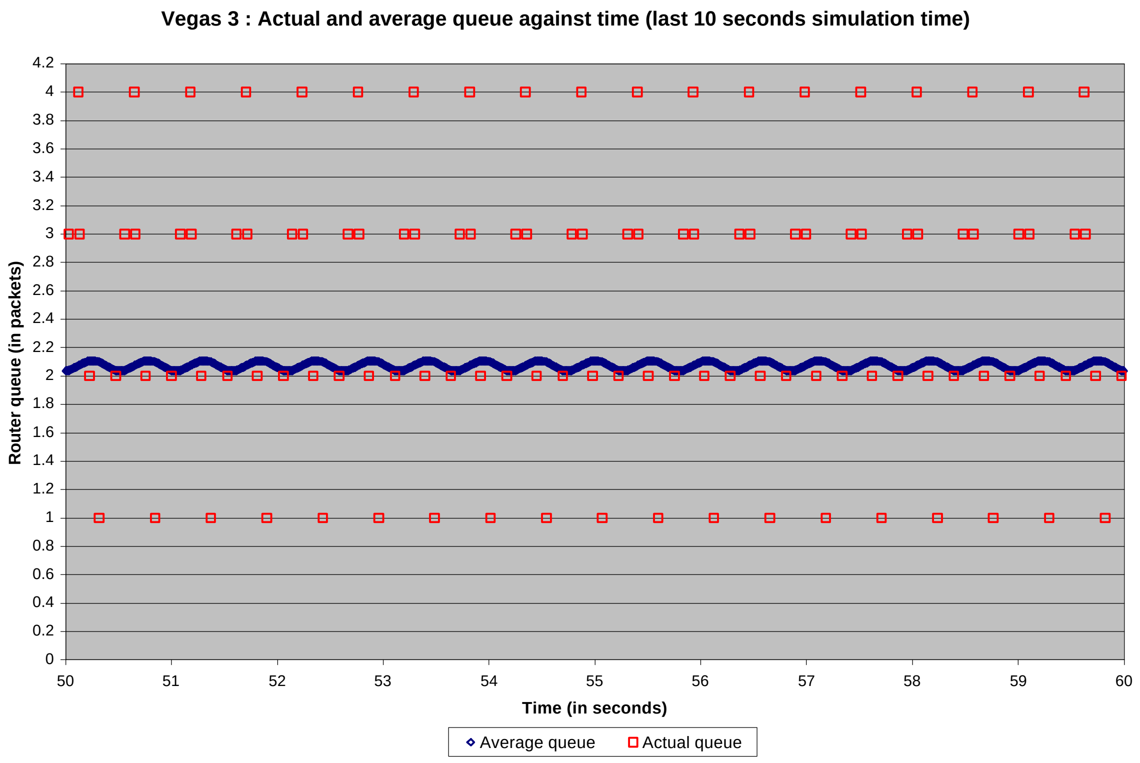


Figure 3.11b : Single Vegas-3 - Average & actual queue against time

Vegas-δ	Min. v_delta_	Max. v_delta_	Range	Average
Vegas-1	0	2.697	2.697	1.27
Vegas-2	0.694	3.691	2.997	2.20
Vegas-3	1.758	3.691	1.933	2.73
Vegas-4	2.814	4.687	1.873	3.75
Vegas-5	3.863	5.683	1.820	4.77
Vegas-10	9.041	10.670	1.629	9.85

Table 3.4 : Maximum & minimum v_delta vs. Vegas-δ

From figure 3.11a and 3.11b, it can be observed that v_delta_ and the actual router queue also fluctuate. It can be observed from table 3.4 that, except with Vegas-1, the range of fluctuation of v_delta_ becomes narrower as δ increases.

I have mentioned that that the stability of Vegas-δ improves as δ increase. Therefore, it can be argued that larger δ should be preferred. However, there is a trade-off : larger δ causes congestion more easily, especially when many TCP connections compete for bandwidth.

4.2 Is TCP Vegas really fair to connections with larger propagation delays

4.2.1 A case with 2 TCP Vegas connections competing for bandwidth

In this section, I discuss the fairness of Vegas-1_3 when 2 of them with different propagation delays share a bottleneck bandwidth. Using the same network model, I set x1 to 1ms and x2 to 30ms. Accordingly, the propagation delay of the shorter connection (connection 1) is 6ms and that of the longer connection (connection 2) is 64ms. The minimum round-trip time of the shorter connection is 0.013211 seconds and that of the longer connection (connection 2) is 0.071211 seconds.

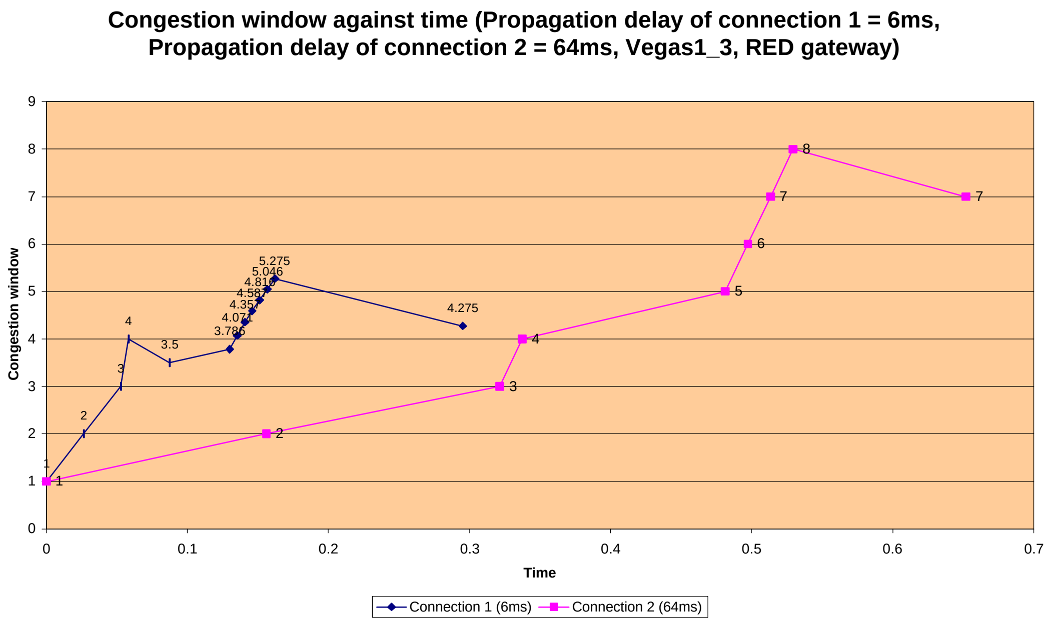


Figure 3.12 : 2 Vegas1_3 (Propagation delay of connection 1 = 6ms, Propagation delay of connection 2 = 64ms) - Congestion window against time

Figure 3.12 shows that the congestion windows of both connections converge quickly in less than 1 second. There is no drop of packets during the whole simulation for both connections.

The dynamics of both TCP Vegas connections is similar to the case when there is only 1 TCP Vegas connection in the network. During the slow-start phase, the congestion window of the shorter connection doubles every other RTT, i.e. for connection 1, from 1 to 2 at 0.026420 seconds, and then from 2 to 4 at 0.058180 seconds. Then, the modified congestion avoidance phase is triggered by a detection of bandwidth over-shoot. When the size of the congestion window reaches 4, TCP Vegas detects a bandwidth over-shoot (i.e. the expected sending rate is greater than the actual sending rate by more than an equivalent of 1 router buffer). Hence, TCP Vegas reduces the congestion window by 1/8 to 3.5 and then enters the congestion avoidance phase. During the congestion avoidance phase, the congestion window keeps on increasing until it reaches 5.275 because the difference between the expected sending rate and the actual sending rate continues to differ by less than $\alpha = 1$.

Contrary to the one-connection scenario, however, when the congestion window reaches 5.275, connection 1 discovers that the difference between the expected sending rate and the actual sending rate differs by more than $\beta = 3$. Therefore, its congestion avoidance algorithm reduces the congestion window by 1 to 4.275.

Finally, the congestion window of the shorter connection converges to about 4 at 0.295390 second and the congestion window of connection 2 converges to 7 at 0.652050 second. The simulation data in section 4.1.1 shows that, when there is only 1 TCP Vegas connection, the congestion window converges to 5 and 16 if the propagation delay is 6ms and 64ms, respectively. Hence, at steady state, the congestion window of connection 1 is only slightly less than that when there is only 1 TCP Vegas connection. However, the congestion window of connection 2 is less than half of that can be attained when there is only 1 TCP Vegas connection with the same propagation delay. The size of the congestion window suggests possible bias against connections with larger round-trip times.

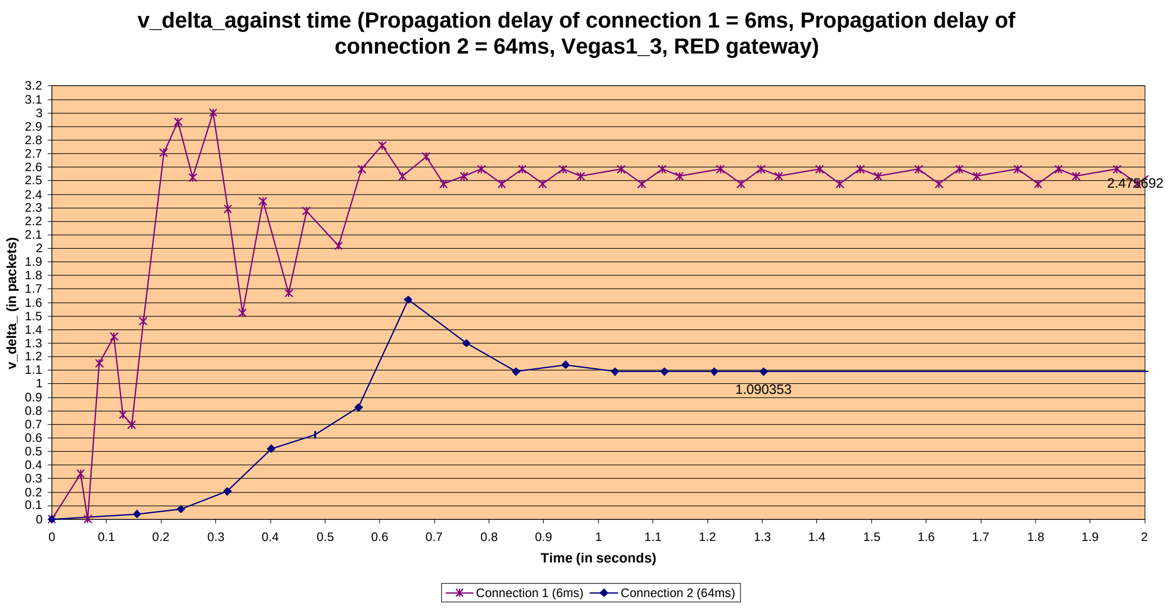


Figure 3.13 : 2 Vegas1_3 (Propagation delay of connection 1= 6ms, Propagation delay of connection 2 = 64ms.) - v_delta_ against time

The converged value of v_delta_ also suggests a bias in throughput against connection 2, which has a larger propagation delay. Figure 3.13 shows the value of v_delta_ against time. The v_delta_ value of connection 1 converges to 2.5 to 2.6 while the v_delta_ value of connection 2 converges to 1.1. This means that connection 1 keeps on average 2.5 to 2.6 packets in the router buffer while connection 2 keeps on average 1.1 packets in the router buffer. This suggests that the throughput of connection 1, the shorter connection, is higher. Therefore, there is an unfair share of the bandwidth.

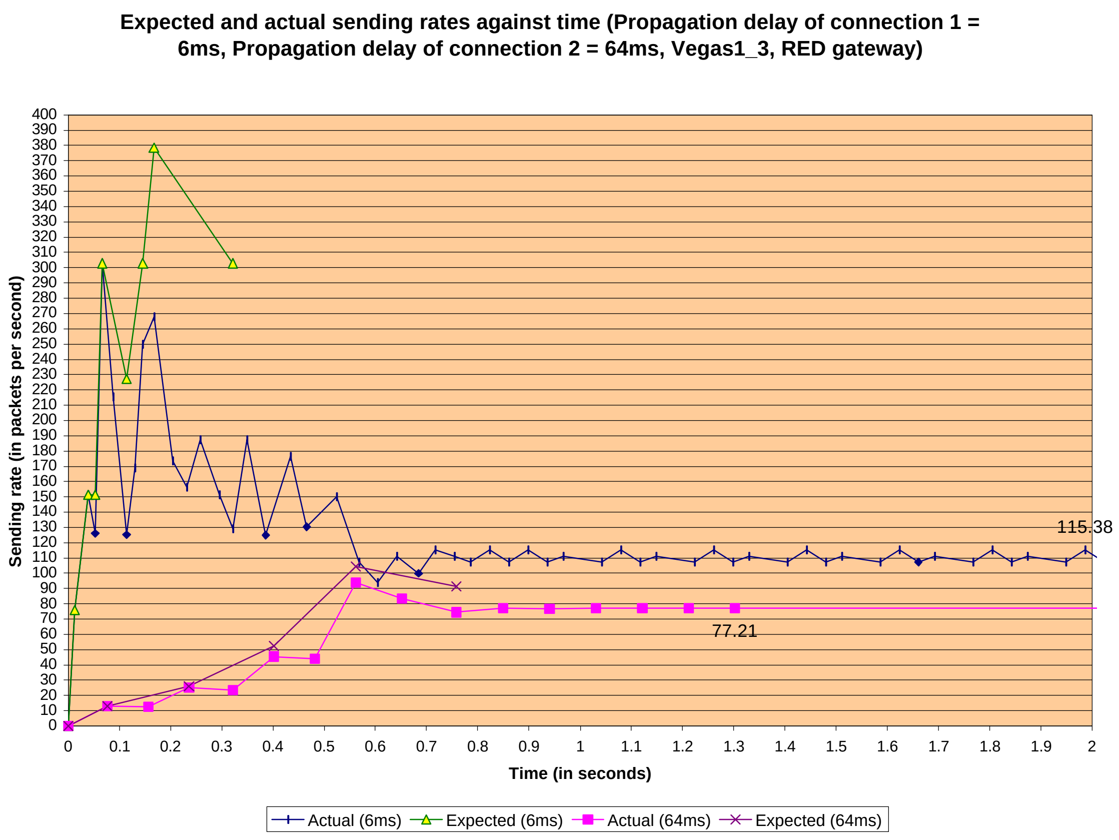


Figure 3.14 : 2 Vegas1_3 case (Propagation delay of connection 1 = 6ms, Propagation delay of connection 2 = 64ms) - Expected & actual sending rates against time

In line with the observation on v_delta_, figure 3.14 shows that the shorter connection also captures a higher throughput. In the simulation, the measured average goodput of the shorter connection is 882,415.16 bps and the measured average goodput of the longer connection is 617,610.61 bps. The fairness index is 0.969.

Under this simulation setting, the longer TCP Vegas connection records a minimum round-trip time of 0.076544 seconds although the true minimum round-trip time is only 0.071211 seconds. Therefore, it calculates its bandwidth-delay product as 1.5 Mbps * 0.076544s (expected sending rate is 7/0.076544) instead of 1.5Mbps * 0.071211s (expected sending rate is 7/0.071211). Accordingly, connection 2 should have pumped more packets into the network pipe because it over-estimates the bandwidth-delay product. However, connection 2 still achieves lower throughput than connection 1.

Even worse, when x2 = 10ms, the fairness index is only 0.9. This means that the shorter connection is transferring at twice the rate of the longer connection. The simulation results suggest a possible unfair treatment against connections with larger RTTs.

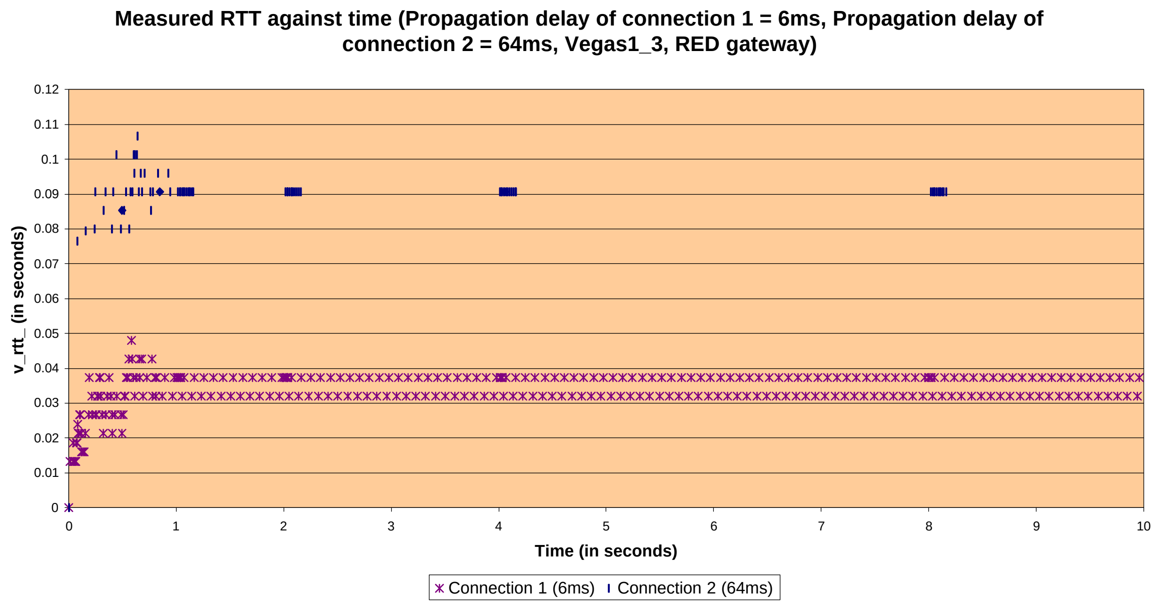


Figure 3.15 : 2 Vegas1_3 (Propagation delay of connection 1 = 6ms, Propagation delay of connection 2 = 64ms) - Measured RTT against time

As a side note, I observe that the actual sending rate does not necessarily converge to a fixed value. Figure 3.14 shows the expected and the actual sending rates of both connections. It can be seen that, while the actual sending rate of the longer connection converges to a fixed value, the actual sending rate of the shorter connection fluctuates within a narrow range. This is due to the fluctuation in the measured round-trip time of the shorter connection. Figure 3.15 shows the measured round-trip times of both connections. It can be seen that the measured round-trip time of the shorter connection is either 0.032 seconds or 0.037333 seconds.

4.2.2 Unfairness of TCP Vegas (when a ¹ b)

To further explore whether there is really a bias against connections with larger propagation delays, I run simulations with 2 connections and set the propagation delay of the shorter connection (connection 1) to 6ms. I vary the propagation delay of the longer connection (connection 2) by changing x2 from 1ms to 110ms (in step of 10ms) in order to investigate the effect of the difference in the propagation delays on the fairness of TCP Vegas. Accordingly, the propagation delays of the longer connections range from 6ms to 224ms. There are no packet losses in all the simulations.

α =1 β = 3	**Vegas 1 : Propagation delay=6ms, Minimum round-trip time = 0.013211 seconds**			**Vegas 2 : Propagation delay = 2 * (x2 + 2)**			Ratio of sending rates (Vegas1 / Vegas2)	Fairness
x2	Converged expected sending rate	Converged actual sending rate	Converged v_delta	Converged expected sending rate	Converged actual sending rate	Converged v_delta_		
110ms	2422285.02	928571.43	2.4666	642586.58	606382.98	1.0705	**1.53**	**0.965**
100ms	2422285.02	925824.17	2.4712	591103.89	545454.54	1.2356	**1.70**	**0.931**
90ms	2422285.02	936107.45	2.4542	610550.31	562500.00	1.1805	**1.66**	**0.941**
80ms	2422285.02	926507.94	2.4700	636363.64	583333.34	1.1667	**1.59**	**0.953**
70ms	2422285.02	857142.86	2.5846	724137.93	642623.72	1.5759	**1.33**	**0.980**
60ms	2422285.02	934407.81	2.4570	644480.90	568965.52	1.2889	**1.64**	**0.945**
50ms	2422285.02	903420.68	2.5082	686436.02	600000.00	1.2592	**1.51**	**0.962**
40ms	2422285.02	928996.34	2.4659	670091.58	571428.57	1.1779	**1.63**	**0.946**
30ms	2422285.02	881562.88	2.5442	731605.35	617647.06	1.0904	**1.43**	**0.970**
20ms	2422285.02	800000.00	2.6789	990379.18	700000.00	2.0524	**1.14**	**0.996**
10ms	2422285.02	1000000.00	2.3487	755794.42	500000.00	1.0153	**2.00**	**0.900**
5ms	1816713.77	750000.00	1.7615	1340183.16	750000.00	1.7615	**1.00**	**1.000**
1ms	1816713.77	750000.00	1.7615	1816713.77	750000.00	1.7615	**1.00**	**1.000**

Table 3.5 : 2 Vegas1_3 – Fairness vs. propagation delay of longer connection, connection 1 starts first (Min. RTT of shorter connection = 0.013211s)

The simulation results in table 3.5 suggest that, when α = 1 and β = 3, there is a bias against connections with larger propagation delays. The converged actual sending rate of the shorter connection is always higher than that of the longer connection, unless the difference in propagation delays between the shorter and the longer connection is close (i.e. when x2 = 1ms or 5ms). When the propagation delays of both connections are close, the fairness is perfect. In other settings, the fairness varies from 0.90 to 0.996 and the converged actual sending rate of the shorter connection is always higher.

However, the degree of bias does not necessarily increase as the propagation delay of the longer connection increases. For instance, the fairness is worst when x2 =10ms, with the shorter connection transmitting at twice the rate of the longer connection.

In line with the bias on sending rate and goodput, the converged v_delta_ of the shorter connection is also larger than that of the longer connection. From table 3.5, it can be found that the average v_delta of connection 1 is about 2.5 while that of the longer connection is normally around 1. Since v_delta_ refers to the number of packets in the router buffer, this implies that, at steady state, the shorter connection is keeping about 2 packets in the intermediate router buffer while the longer connection is keeping about only 1 packet in the router buffer (except when x2 = 1ms, 5ms and 20ms).

X2	True minimum round-trip time (in seconds)	Measured minimum round-trip time (in seconds)	Over-estimation of minimum round-trip time (in seconds)
110ms	0.231211	0.236544	0.005333
100ms	0.211211	0.216544	0.005333
90ms	0.191211	0.196544	0.005333
80ms	0.171211	0.176000	0.004789
70ms	0.151211	0.154667	0.003456
60ms	0.131211	0.136544	0.005333
50ms	0.111211	0.116544	0.005333
40ms	0.091211	0.095509	0.004298
30ms	0.071211	0.076544	0.005333
20ms	0.051211	0.056544	0.005333
10ms	0.031211	0.031755	0.000544
5ms	0.021211	0.023877	0.002666
1ms	0.013211	0.013211	0.000000

Table 3.6 : Over-estimation of minimum RTTs vs. x2

I also observe that, under this simulation setting, the minimum round-trip time of the longer connection is always over-estimated. Exceptions occur when the propagation delays of both connections are the same (i.e. both x1 & x2 = 1ms). Therefore, there is no guarantee that TCP Vegas is able to detect the true minimum round trip. In fact, when the packets belonging to the longer connection arrives at the router, it is likely that some packets belonging to the shorter connection has already occupied the queue at the router. Consequently, the packets belonging to the longer connection has to wait after the packets that are queuing at the router have been transmitted before they can be served. Hence, its estimation of the minimum round-trip time includes the transmission delay and the queuing delay due to packets of the shorter connection queuing at the router. In case when the propagation delays of both connections are the same, the first estimation of the minimum round-trip time of connection 2 is over-estimated as 0.018544 seconds (including the transmission delay of the packet of connection 1). Fortunately, the TCP Vegas agent is able to detect the correct minimum round-trip time at a later stage, i.e. after another RTT.

There is also a bias in the expected sending rate of the TCP Vegas connection. It is obvious from table 3.5 that the expected sending rate of the shorter connection is much higher than that of the longer connection. Since the shorter connection opens its congestion window faster than that of the longer connection, its expected sending rate and actual sending also increase faster than those of the longer connection. Therefore, the rate at which the expected sending rate of the shorter connection increases, relative to the rate at which the expected sending rate of the longer connection increases, seems to affect the steady state equivalent point. In many of the simulations I have done with different value of x2, the expected sending rate of the longer connection is even less than the fair share of its bandwidth, especially when the difference in propagation delays is large. Since the actual sending rate is always lower than the expected sending rate, the actual sending rate of the longer connection will be less that its fair allocation. Therefore, there will be an unfair allocation of the bottleneck bandwidth.

To confirm my observation of the bias against connections with larger round-trip times, I reverse the sequence that the 2 connections are scheduled and thus started in the simulation. This allows me to get a set of simulation data with which the minimum round trip time is correctly estimated. The results, summarized in the table 3.7, show that there is still a bias in bandwidth allocation against connections with larger propagation delays. Again, there are no packet losses in all the simulations.

α =1 β = 3	**Vegas 1 : Propagation delay=6ms, Minimum round-trip time = 0.013211 seconds**			**Vegas 2 : Propagation delay = 2 * (x2 + 2)**			**Ratio of sending rate (Vegas1 / Vegas2)**	**Fairness**
x2	Converged expected sending rate	Converged average sending rate	Converged v_delta	Converged expected sending rate	Converged average sending rate	Converged v_delta_		
110ms	2422285.02	989283.80	2.396937	553607.68	510721.96	1.241872	**1.94**	**0.908**
100ms	2422285.02	1011523.88	2.283397	530276.25	488481.88	1.106325	**2.07**	**0.892**
90ms	2422285.02	975043.74	2.431845	585741.38	525122.02	1.451800	**1.86**	**0.917**
80ms	2422285.02	916667.13	2.483704	654164.85	583537.27	1.515889	**1.57**	**0.953**
70ms	2422285.02	984496.91	2.396937	581969.53	515528.86	1.254000	**1.91**	**0.911**
60ms	2422285.02	1017923.91	2.283397	548735.87	482241.85	1.092214	**2.11**	**0.887**
50ms	2422285.02	1000003.84	2.348667	575484.37	500001.92	1.049333	**2.00**	**0.900**
40ms	2422285.02	1000019.58	2.348667	613963.28	500009.79	1.299333	**2.00**	**0.900**
30ms	2422285.02	882403.39	2.544246	786399.04	617762.37	1.502118	**1.43**	**0.970**
20ms	2422285.02	964164.53	2.404558	781087.27	535842.52	1.570714	**1.80**	**0.925**
10ms	1816713.77	833281.12	1.624445	1025290.50	666720.89	1.399111	**1.25**	**0.988**
5ms	1816713.77	857124.03	1.560244	1131506.16	642883.02	1.295571	**1.33**	**0.980**
1ms	1816713.77	750094.69	1.761500	1816713.77	749934.68	1.761500	**1.00**	**1.000**

Table 3.7 : 2 Vegas1_3 - Fairness vs. propagation delay of longer connection, connection 2 starts first (Min. RTT of shorter connection = 0.013211s)

I also run another set of simulations but set the propagation delay of the shorter connection to 104ms (x1 = 50ms) while varying the value of x2 from 1 ms to 110ms. Under this simulation setting, the minimum round-trip time of both connections are correctly estimated. Moreover, the propagation delay of connection 2 is higher than that of connection 1 in some cases while it is lower than that of connection 1 in other cases. However, the difference in propagation delays is smaller.

Table 3.8 below shows the result with this set of simulation setting. The simulation results also support that there is a bias against connections with larger round-trip delays, except when x2 = 40ms and x2 = 60ms. Therefore, this pattern of bias may not apply when the propagation delays of the 2 connections are close, e.g. when x2 = 40ms to 60ms.

x2	**Vegas 1 : Propagation delay =0.111211 seconds**			**Vegas 2 : Propagation delay = 2 * (X2 + 2)**			**Ratio of sending rates (Vegas1 /Vegas2)**	**Fairness**
	Converged expected sending rate	Converged actual sending rate	Converged v_delta	Converged expected sending rate	Converged actual sending rate	Converged v_delta_		
110ms	1150968.74	989441.68	2.2483	553607.68	510720.87	1.2419	1.94	0.908
100ms	1079033.18	906883.77	2.3920	643906.87	592962.47	1.3434	1.53	0.958
90ms	1079033.18	900013.68	2.4888	669418.72	600009.12	1.6592	1.50	0.962
80ms	1007097.64	900003.89	1.4880	654164.85	600002.59	1.1592	1.50	0.962
70ms	1150968.74	954588.10	2.7187	634875.85	545456.06	1.6902	1.75	0.931
60ms	863226.55	692485.28	2.3760	975530.45	807526.15	2.7547	0.86	0.994
50ms	1079033.18	864652.36	2.9700	791291.00	635369.09	2.1780	1.36	0.977
40ms	1007097.64	857134.54	2.0845	789381.36	643050.91	1.6706	1.33	0.980
30ms	647419.91	540004.98	1.4933	1235769.92	960008.85	2.4582	0.56	0.927
20ms	647419.91	562423.54	1.1805	1249739.64	937639.25	1.9997	0.60	0.941
10ms	647419.91	540002.07	1.4933	1537935.75	960003.69	2.2470	0.56	0.927
5ms	647419.91	562562.52	1.1805	1885843.60	937604.20	2.5144	0.60	0.941
1ms	575484.37	500000.00	1.0493	2422285.02	1000000.00	2.3487	0.50	0.900

Table 3.8 : 2 Vegas1_3 - Fairness vs. propagation delay of longer connection (Min. RTT of shorter connection = 0.111211s)

4.2.3 Fairness of Enhanced TCP Vegas (when $a = b$)

The simulation results presented in the previous section supports that Vegas1_3 biases against connections with larger propagation delays. In this section, I explore whether the bias still exists with Enhanced TCP Vegas.

The simulation setting is similar to that described in the network model. The RED gateway is adopted. Its minimum threshold is 15 and the maximum threshold is 45. The buffer size is 100 packets. The α and β values of the 2 TCP Vegas agents are equal and are set to 1, 2 or 3. The minimum round-trip time of the shorter connection is 0.013211 seconds (including processing delay at the router). However, I vary the propagation delay of the longer connection (i.e. by modifying the value of x2) in an attempt to evaluate the effect of different propagation delays on the fairness. Each of the simulations lasts for 60 seconds. There are no packet losses in all the simulations. The tables and figures below display the result.

α =1 β = 1	**Connection 1 : Propagation delay=6ms, Minimum round-trip time=0.013211 seconds**	**Connection 2 : Propagation delay = 2 * (x2 + 2)**	**Ratio of sending rates (Vegas1/ Vegas2)**	**Fairness**
x2	**Converged actual sending rate**	**Converged actual sending rate**		
110ms	932815.32	526888.65	1.77	0.928
100ms	975849.15	482884.53	2.02	0.898
90ms	977153.15	469455.92	2.08	0.890
80ms	1051843.60	394881.35	2.66	0.829
70ms	847532.29	636169.22	1.33	0.980
60ms	869942.36	622095.99	1.40	0.973
50ms	793147.33	706904.36	1.12	0.997
40ms	750087.04	750087.04	1.00	1.000
30ms	706249.49	793450.66	0.89	0.997
20ms	846240.27	653120.21	1.30	0.984
10ms	833441.12	666720.89	1.25	0.988
5ms	807363.79	692003.25	1.17	0.994
1ms	750246.53	749926.52	1.00	1.000

Table 3.9 : Fairness of Vegas-1 vs. propagation delay of the longer connection

α =2 β = 2	**Connection 1 : Propagation delay=6ms, Minimum round-trip time=0.013211 seconds**	**Connection 2 : Propagation delay = 2 * (x2 + 2)**	**Ratio of sending rates (Vegas1/ Vegas2)**	**Fairness**
x2	**Converged actual sending rate**	**Converged actual sending rate**		
110ms	712482.17	787362.39	0.90	0.998
100ms	764657.94	735377.25	1.04	1.000
90ms	776482.53	723522.36	1.07	0.999
80ms	734889.88	765450.29	0.96	1.000
70ms	723526.32	776486.79	0.93	0.999
60ms	760662.52	739541.89	1.03	1.000
50ms	780643.00	719522.76	1.08	0.998
40ms	742247.13	757287.27	0.98	1.000
30ms	739052.70	761453.08	0.97	1.000
20ms	772803.63	727363.42	1.06	0.999
10ms	768359.46	731877.59	1.05	0.999
5ms	768323.61	731683.44	1.05	0.999
1ms	750254.69	749934.68	1.00	1.000

Table 3.10 : Fairness of Vegas-2 vs. propagation delay of the longer connection

$\alpha = 3$ $\beta = 3$	**Connection 1 :** **Propagation delay=6ms, Minimum round-trip time=0.013211 seconds**	**Connection 2 :** **Propagation delay = 2 * (x2 + 2)**	**Ratio of sending rates (Vegas1 / Vegas2)**	**Fairness**
x2	**Converged actual sending rate**	**Converged actual sending rate**		
110ms	725602.79	774402.97	0.94	0.999
100ms	758422.45	741461.95	1.02	1.000
90ms	762887.32	737287.08	1.03	1.000
80ms	733461.71	766742.70	0.96	1.000
70ms	736652.66	763373.11	0.96	1.000
60ms	759042.91	740962.85	1.02	1.000
50ms	758402.91	741442.85	1.02	1.000
40ms	743842.86	756002.90	0.98	1.000
30ms	729602.80	770242.96	0.95	0.999
20ms	770083.62	730243.43	1.05	0.999
10ms	772641.04	727360.97	1.06	0.999
5ms	755363.55	744803.50	1.01	1.000
1ms	750094.69	749934.68	1.00	1.000

Table 3.11 : Fairness of Vegas-3 vs. propagation delay of the longer connection

The simulation results in table 3.9 suggest that TCP Vegas' bias against connections with larger round-trip delays is still not eliminated with Vegas-1. In some cases, the fairness index is even less than 0.90.

However, the fairness is much improved with Vegas-2 and Vegas-3. It can be found from table 3.10 and 3.11 that the fairness index of Vegas-2 and Vegas-3 is always greater than or equal to 0.998. When Vegas-3 is used, 9 out of the 13 simulations give a fairness index of 1.000 !

4.2.4 Comparison of Vegas fairness and Reno fairness

It has been reported that the fairness of TCP Vegas is better than that of TCP Reno. In addition, the fairness of TCP Vegas does not show the same bias pattern as that of TCP Reno [CC00, MLAW99]. My simulation results in table 3.12 confirm that Reno fairness indeed becomes worse as the difference in propagation delays between the 2 connections increases. However, Vegas fairness can be worse than Reno fairness as well (e.g. when propagation delay of the longer connection is 24ms or 84ms). Therefore, there is no guarantee that the fairness of Vegas1_3 is better than the fairness of TCP Reno.

The simulation setting is similar to that described in the network model. The RED gateway is adopted. Its minimum threshold is 15 and the maximum threshold is 45. The buffer size is 100 packets. The minimum round-trip time of the shorter connection is 0.013211 seconds (including the processing delay at the router). However, I vary the propagation delay of the longer connection (i.e. by modifying the value of x2) in an attempt to evaluate the effect of the difference in propagation delays on fairness. Each of the simulations lasts for 60 seconds. The tables and figures below display the result.

x2	**Propagation delay of the longer connection**	**Reno Fairness**	**Vegas1_3 Fairness (connection 1 starts first)**	**Vegas1_3 Fairness (connection 2 starts first)**	**Vegas-3 Fairness (connection 2 starts first)**
100ms	204ms	0.817	0.931	0.892	1.000
80ms	164ms	0.855	0.953	0.953	1.000
60ms	124ms	0.917	0.945	0.887	1.000
40ms	84ms	0.982	0.946	0.900	1.000
20ms	44ms	0.974	0.996	0.925	0.999
10ms	24ms	0.994	0.900	0.988	0.999
1ms	6ms	0.994	1.000	1.000	1.000

Table 3.12 : Comparison of Reno & Vegas fairness

X2	Reno1	Reno2	Ratio of sending rates (Reno1/Reno2)	Reno Fairness
100ms	1102901.59	394727.73	2.79	0.817
80ms	1060041.96	441137.46	2.40	0.855
60ms	975859.11	525130.28	1.86	0.917
40ms	848803.99	647203.04	1.31	0.982
20ms	872164.10	626882.95	1.39	0.974
10ms	809440.39	690560.33	1.17	0.994
1ms	808815.84	691533.54	1.17	0.994

Table 3.13 : Summary of Reno goodputs

	Connection 1 (Propagation delay = 0.013211 seconds)			Connection 2 (Propagation delay = 2 x (x2 + 2)			
x2	Expected sending rate	Measured average goodput	Converged congestion window	Expected sending rate	Measured average goodput	Converged congestion window	Vegas Fairness
100ms	2422320	954588.26	4.146	591104	545456.15	16.875	0.931
80ms	2422320	916667.13	4.191	636364	583537.27	14.000	0.953
60ms	2422320	931055.14	4.275	686400	568969.25	11.817	0.945
40ms	2422320	928483.57	4.357	670080	571522.19	8.000	0.946
20ms	2422320	800013.01	4.357	990400	700011.38	7.000	0.996
10ms	2422320	1000002.02	4.357	755794	500001.01	3.500	0.900
1ms	2422320	749934.68	3.500	1816714	750094.69	3.500	1.000

Table 3.15 : Summary of Vegas1_3 goodputs

From the simulation results given in table 3.12, it is obvious that, in some cases, the fairness of TCP Vegas is better than the fairness of TCP Reno. However, this is not a guarantee. For example, when x2 is 10ms or 40ms, Reno fairness is much better than that of Vegas1_3. The fairness of Vegas-3, on the other hand, is consistently better than that of TCP Reno.

As expected, the fairness of TCP Reno gets worse as the propagation delay of the longer connection increases. Vegas-1_3 and Vegas-3, on the other hand, does not show the same delay pattern as that of TCP Reno. That is, the degree of bias does not necessarily increases with the increase in the difference in the propagation delays.

In summary, I observe that the original version of TCP Vegas [BP94] is still unfair to connections with larger propagation delays, unless the difference in the propagation delays of the connections is small. In some cases, the expected sending rate of the longer connection is even less than the fair share of the available bandwidth. Therefore, the actual sending rate and average goodput is also less than if the bandwidth is shared equally. However, unlike TCP Reno, the degree of bias does not necessarily increase as the difference in propagation delays increases. The simulation results also confirm that the fairness of TCP Vegas may be worse than that of TCP Reno so that improvement is not a guarantee. The problem of unfairness, however, can be much resolved by setting $\alpha = \beta = 2$ or 3.

4.3 Impact of the Vegas threshold, α and β, on fairness

Hasegawa proposed that by setting $\alpha = \beta$, the fairness of TCP Vegas should be improved. Hasegawa called the control parameter δ. Hasegawa claimed that the improvement was achieved by allowing the window size to fluctuate around a central value that is proportional to the round-trip delay. By doing so, there will be fairness in terms of window size / propagation delay. In addition, the fluctuation allows the minimum round-trip time to be measured accurately.

In this section, I present the simulation results that explore the most appropriate value for δ in terms of fairness. I shall call the corresponding version of Enhanced TCP Vegas, Vegas-δ. For example, Vegas-2 has $\delta = 2$. I also compare their performance against the configuration when α is set to 1 and β is set to 3, which I call Vegas-1_3.

The setting is similar to that stated in our network model. The minimum round trip time of the shorter connection is 0.013211 seconds. The router buffer is 100 packets. The RED gateway is used and the minimum and the maximum thresholds are 15 and 45, respectively. The window sizes of both TCP agents are 30. The simulation is run for 60 seconds. Average goodputs are measured after the simulation has started for 10 seconds. The table and graph below shows the fairness measures of Enhanced TCP Vegas with different value of δ and x2.

Table 3.16 : Fairness of Vegas-1, Vegas-2, Vegas-3 & Vegas1_3, connection 1 starts first (Min. RTT of connections 1 = 0.013211s)

Min. rtt of longer connection	**Vegas-1**	**Vegas-2**	**Vegas-3**	**Vegas-1_3**
0.231211 seconds (x2=110ms)	0.928	0.995	0.992	0.965
0.211211 seconds (x2=100ms)	0.898	1.000	0.997	0.931
0.191211 seconds (x2=90ms)	0.890	1.000	0.996	0.941
0.171211 seconds (x2=80ms)	0.858	0.999	0.995	0.953
0.151211 seconds (x2=70ms)	0.988	0.982	0.989	0.980
0.131211 seconds (x2=60ms)	0.998	0.997	0.997	0.945
0.111211 seconds (x2=50ms)	0.997	0.986	0.996	0.962
0.091211 seconds (x2=40ms)	1.000	0.994	0.995	0.946
0.071211 seconds (x2=30ms)	0.997	0.968	0.991	0.970
0.051211 seconds (x2=20ms)	0.998	0.998	0.998	0.996
0.031211 seconds (x2=10ms)	0.988	0.999	1.000	0.900
0.021211 seconds (x2=5ms)	0.994	0.999	0.999	1.000
0.013211 seconds (x2=1ms)	1.000	1.000	1.000	1.000

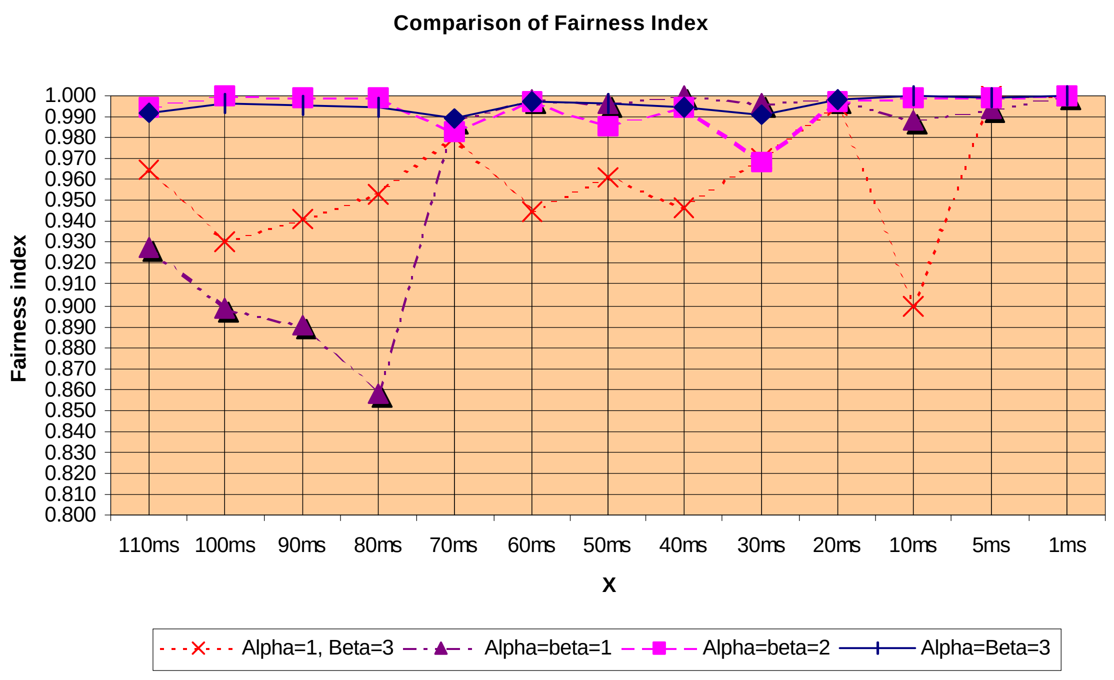


Figure 3.16a : Comparison of the fairness of Vegas-1_3, Vegas-1, Vegas-2 & Vegas-3, connection 1 starts first (Min. RTT of connection 1 = 0.013211s)

Min. rtt of longer connection	**Vegas-1**	**Vegas-2**	**Vegas-3**	**Vegas-1_3**
0.231211 seconds (x2=110ms)	0.928	0.998	0.999	0.908
0.211211 seconds (x2=100ms)	0.898	1.000	1.000	0.892
0.191211 seconds (x2=90ms)	0.890	0.999	1.000	0.917
0.171211 seconds (x2=80ms)	0.829	1.000	1.000	0.953
0.151211 seconds (x2=70ms)	0.980	0.999	1.000	0.911
0.131211 seconds (x2=60ms)	0.973	1.000	1.000	0.887
0.111211 seconds (x2=50ms)	0.997	0.998	1.000	0.900
0.091211 seconds (x2=40ms)	1.000	1.000	1.000	0.900
0.071211 seconds (x2=30ms)	0.997	1.000	0.999	0.970
0.051211 seconds (x2=20ms)	0.984	0.999	0.999	0.925
0.031211 seconds (x2=10ms)	0.988	0.999	0.999	0.988
0.021211 seconds (x2=5ms)	0.994	0.999	1.000	0.980
0.013211 seconds (x2=1ms)	1.000	1.000	1.000	1.000

Table 3.17 : Fairness of Vegas-1, Vegas-2, Vegas-3 & Vegas1_3, connection 2 starts first (Min. RTT of connection 1 = 0.013211s)

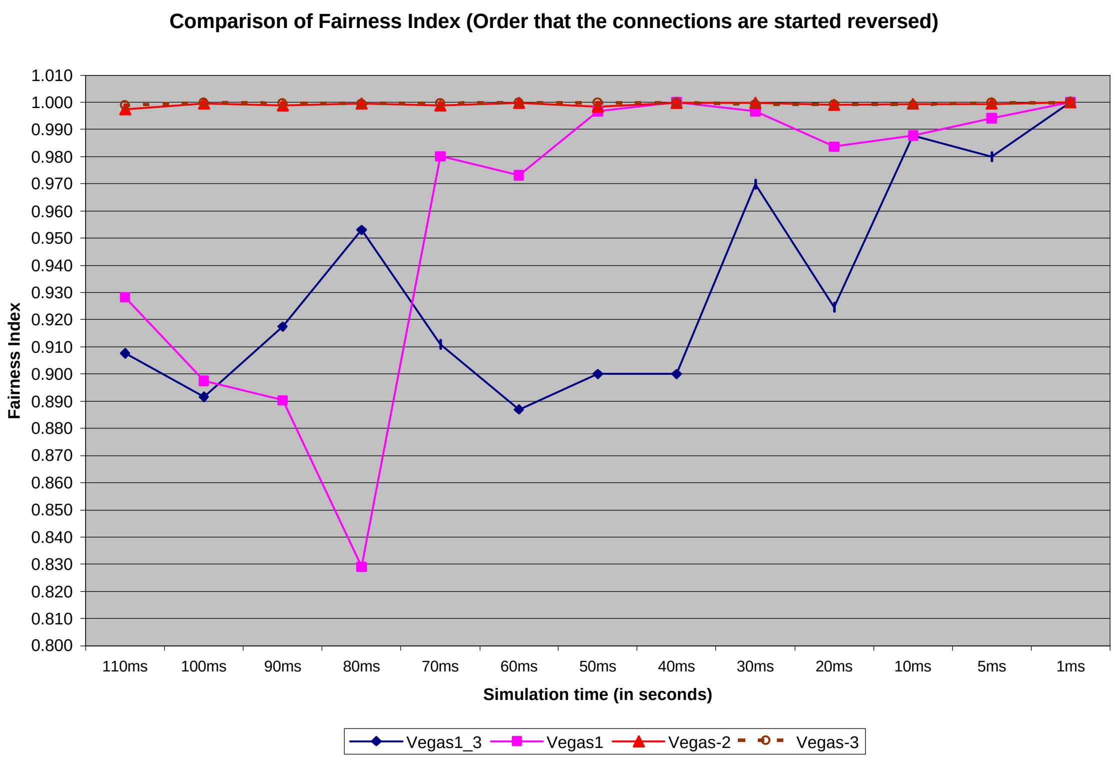


Figure 3.16b : Comparison of the fairness of Vegas-1, Vegas-2, Vegas-3 & Vegas1_3, connection 2 starts first (Min. RTT of connection 1= 0.013211s)

First, I observe from tables 3.16, 3.17 and graphs 3.16a and 3.16b above that, the fairness of Vegas-1 is unstable. When the propagation delay of connection 2, i.e. the longer connection, is relatively low, its fairness is still good. However, when the propagation delay of connection 2 is large enough, the fairness of Vegas-1 gets far worse than that of Vegas-2 and Vegas-3. In some cases, the fairness of Vegas-1 is worse than that of Vegas1_3. For example, when the value of x2 is 80ms (corresponding to a round-trip propagation delay of 164ms), the fairness index of Vegas-1 is far worse that that of Vegas-1_3. It is even less than 0.9.

Comparatively, the fairness of Vegas-2 and Vegas-3 are more stable. In this simulation setting, the fairness index is always greater than 0.98 and, in many cases, very close to 1. The fairness index is very good. The simulation result supports the statement that Vegas-2 and Vegas-3 are fair and its fairness index is independent of the propagation delay. I try running simulations with Vegas-4 and Vegas-10. Their fairness is equally good.

Previously, I mention that the fairness of Vegas-1 is unstable. In general, there is a bias against connections with larger propagation delays (except when x2 = 30ms or 40ms), especially when the delay difference is large. Therefore, I would like to explore why the fairness of Vegas-1 is far worse when the difference in propagation delays is larger. I focus on the case when x2 = 90ms (i.e. when the round-trip propagation delay of the longer connection is 184ms) to illustrate the difference.

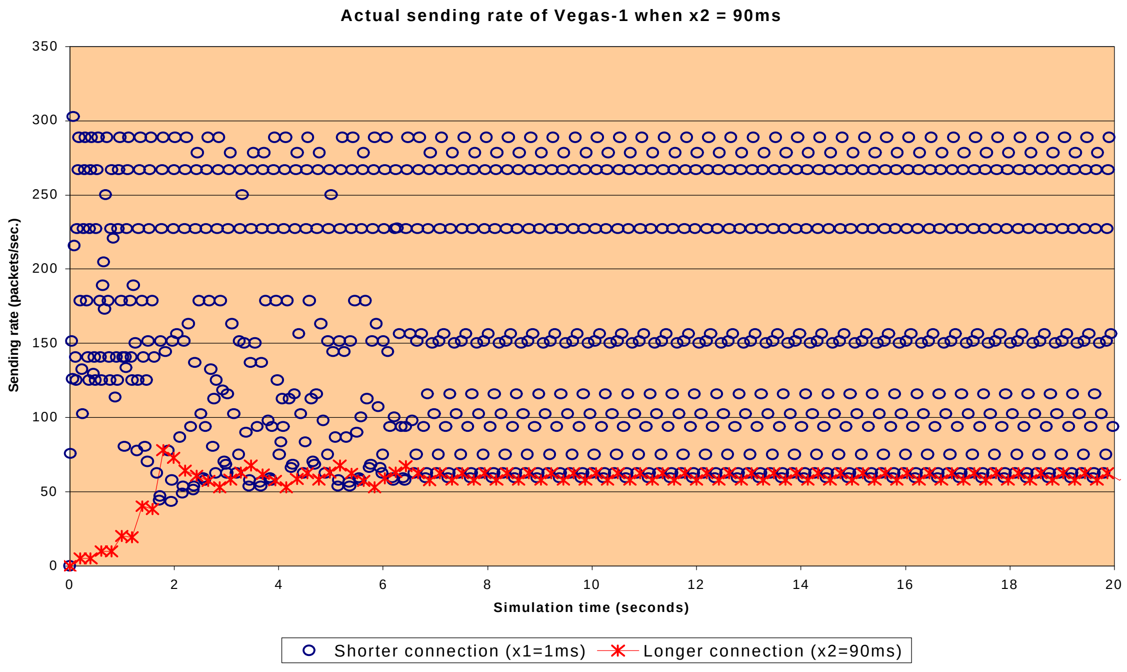


Figure 3.17 : Actual sending rate of Vegas-1 when x2 = 90ms

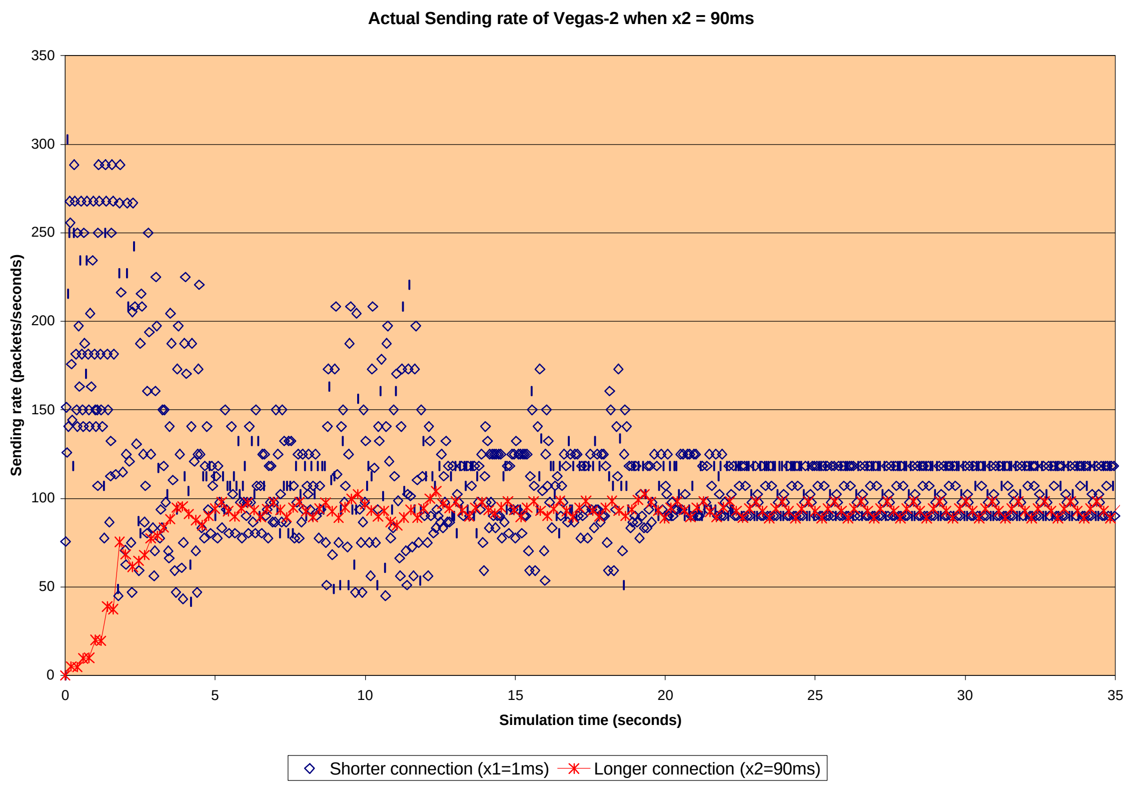


Figure 3.18 : Actual sending rate of Vegas-2 when x2 = 90ms

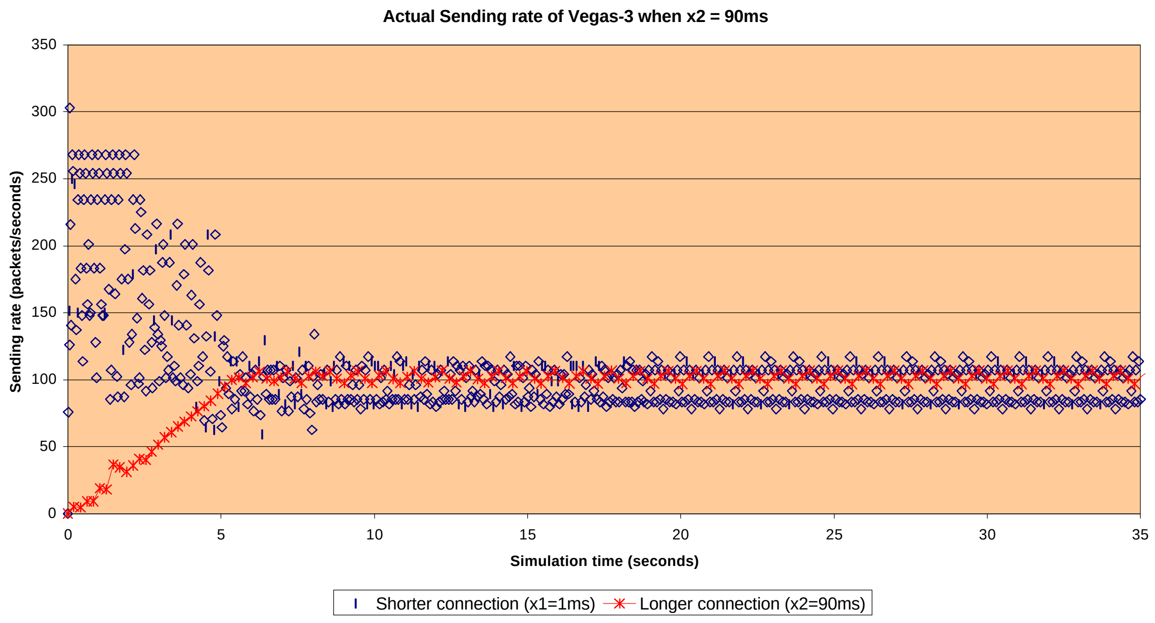


Figure 3.19 : Actual sending rate of Vegas-3 when x2 = 90ms

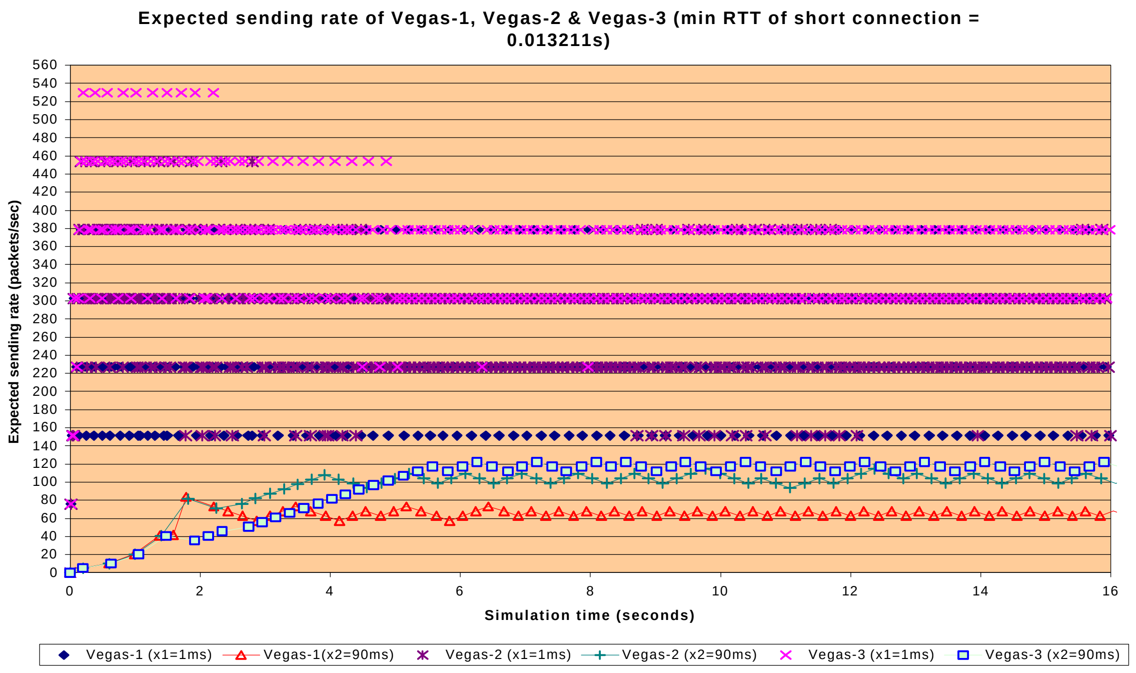


Figure 3.20 : Expected sending rate of Vegas-1, Vegas-2 & Vegas-3 when x2= 90ms (Min. RTT of shorter connection = 0.013211s)

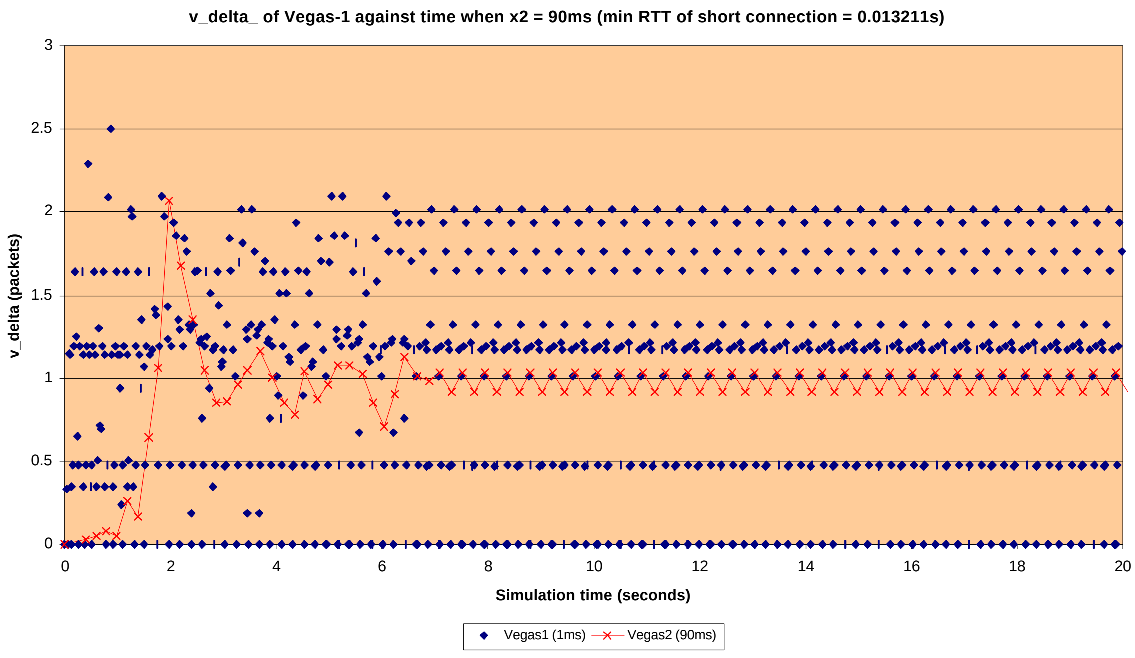


Figure 3.21 : Vegas-1 - v_delta_ against time when x2=90ms

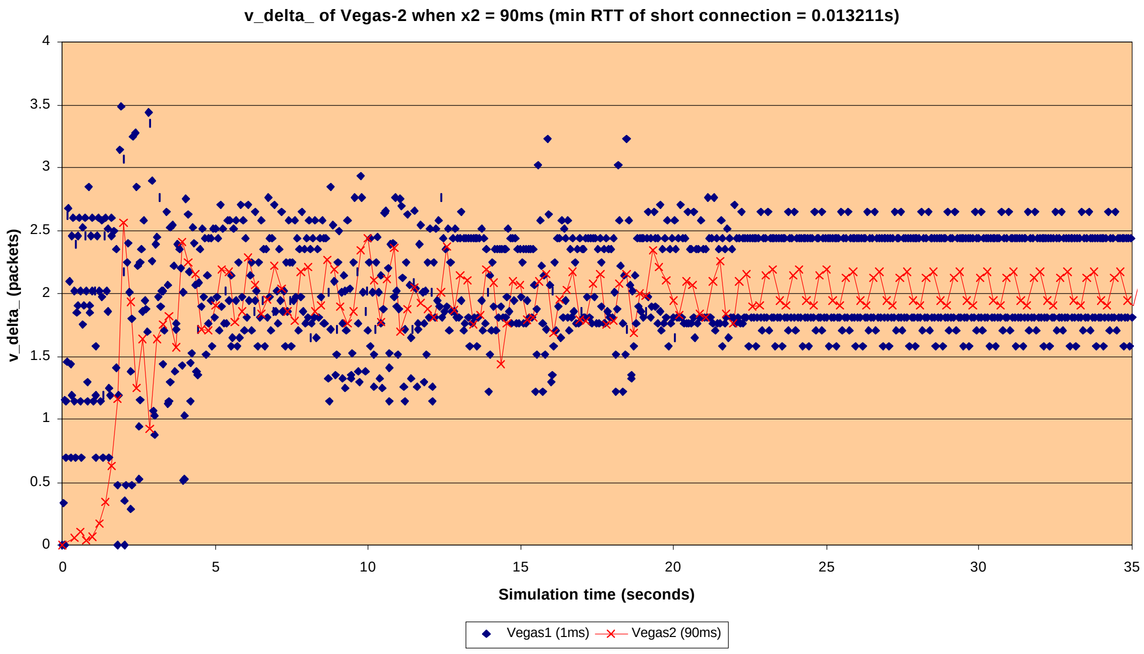


Figure 3.22 : Vegas 2 - v_delta_ against time when x2=90ms

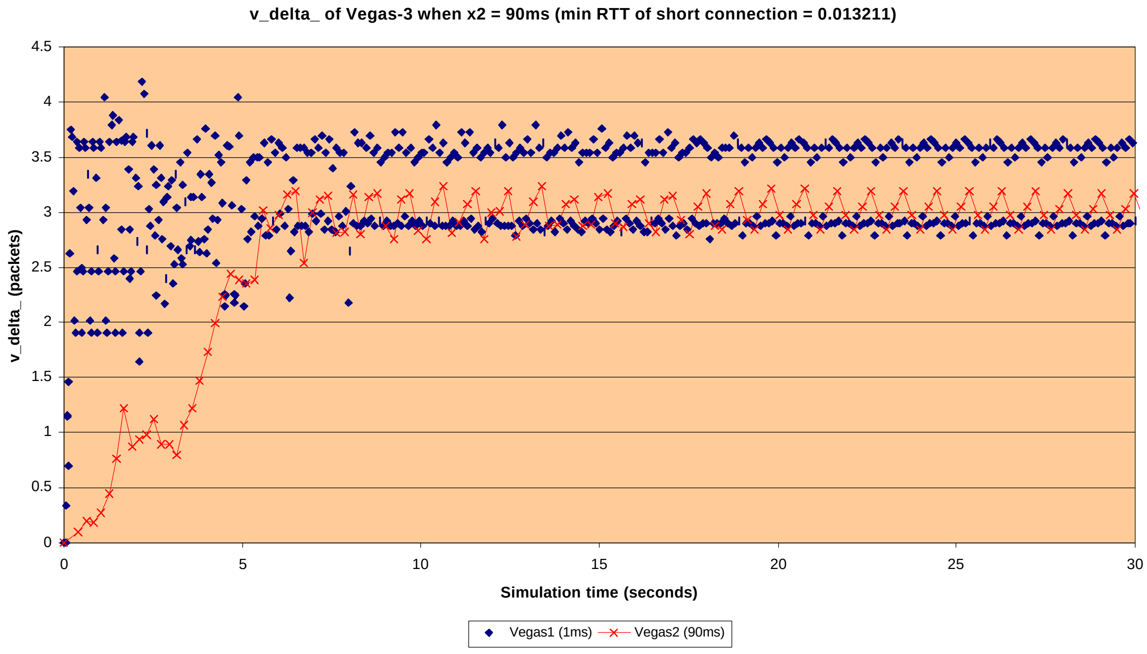


Figure 3.23 : Vegas-3 - v_delta_ against time when x2=90ms

A closer look of the actual sending rate of Vegas-1 in figure 3.17 reveals that, at steady state, the actual sending rate of the longer connection fluctuates within a narrow range around the lowest edge of the fluctuation region of the actual sending rate of the shorter connection. In addition, the range of the actual sending rate of the longer connection is very small. Contrary, the actual sending rate of the shorter connection fluctuates within a very wide range from about 60 packets per second to 288 packets per seconds. Hence, the shorter connection is sometimes able to capture much higher goodput. Therefore, the average goodput of the shorter connection is also higher. In comparison, figures 3.18 and 3.19 show that the actual sending rate of the shorter connection fluctuates between much narrower range with Vegas-2 (90 – 125 packets per second) and Vegas-3 (78 – 117 packets per second), leading to an improvement in fairness.

As can be seen from figure 3.20, the expected sending rate of the longer connection of Vegas-1 fluctuates around 60 packets per second (or about 480,000 bps), a value that is even less than its fair share. This means that the actual sending rate of the longer connection of Vegas-1 will be less than the fair share because the actual sending rate is less than the expected sending rate. The problem is much relieved with Vegas-2 and Vegas-3.

The simulation results suggest that $\delta = 1$ is may not be large enough for a connection with large propagation delay to compete for a fair share of the available bandwidth. Of course, there is another limiting factor, i.e. the advertised window. The advertised window will limit the throughput that the longer connection can achieve.

S1ms	Vegas-1	Vegas-2	Vegas-3	Vegas-1,3
110ms	0.000000	0.000000	0.003456	0.005333
100ms	0.000000	0.002122	0.005333	0.005333
90ms	0.000298	0.000789	0.005333	0.005333
80ms	0.001754	0.000000	0.004789	0.004789
70ms	0.000000	0.003456	0.005333	0.003456
60ms	0.002122	0.005333	0.005333	0.005333
50ms	0.000000	0.005333	0.005333	0.005333
40ms	0.000000	0.004298	0.004298	0.004298
30ms	0.000000	0.003456	0.005333	0.005333
20ms	0.001877	0.005333	0.005333	0.005333
10ms	0.000544	0.000544	0.000544	0.000544
5ms	0.002666	0.002666	0.002666	0.002666
1ms	0.000000	0.000000	0.000000	0.000000

Table 3.17 : Over-estimation of minimum RTT (Propagation delay of shorter connection = 6ms)

In the next simulation setting, I set the minimum round-trip time of the shorter connection to 0.11211 seconds. With this setting, I get a correct estimation of the minimum round-trip time for all values of x2. The table and graph show the fairness index with this simulation setting.

s50ms	Vegas-1	Vegas-2	Vegas-3	Vegas-1,3
110ms	0.998	0.999	1.000	0.910
100ms	1.000	0.999	1.000	0.960
90ms	0.975	0.999	0.998	0.960
80ms	0.995	0.999	0.999	0.960
70ms	0.997	0.999	1.000	0.930
60ms	0.993	1.000	1.000	0.994
50ms	0.998	0.994	1.000	0.977
40ms	0.965	1.000	1.000	0.980
30ms	1.000	1.000	1.000	0.927
20ms	0.976	0.998	0.999	0.941
10ms	0.941	0.994	0.996	0.927
5ms	0.998	1.000	0.999	0.941
1ms	0.997	0.998	0.996	0.900

Table 3.18 : Fairness of Vegas-1, Vegas-2, Vegas-3 & Vegas1_3 (Min. RTT of connection 1 = 0.013211s)

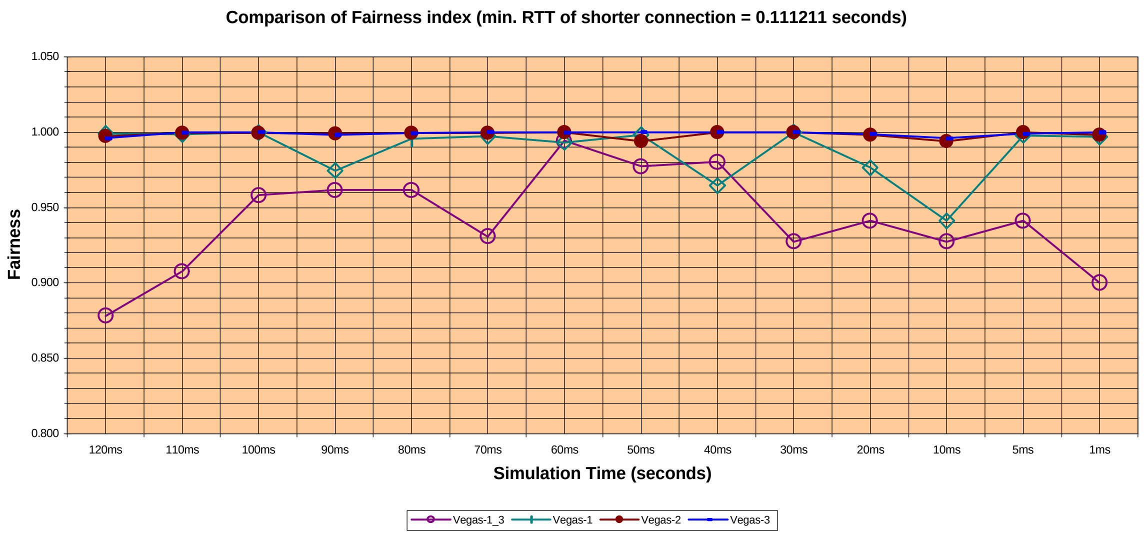


Figure 3.24 : Comparison of fairness of Vegas1_3, Vegas-1, Vegas-2 & Vegas-3 (Min. RTT of the shorter connection=0.0111211s)

I observe from the table and graph shown above that the result is in consistent with what I get when the minimum round-trip time of the shorter connection is 0.013211 seconds. The performance of Vegas-2 and Vegas-3 are consistently good for all values of x2. However, the performance of Vegas-1 is unstable and varies with the values of x2. The fairness of Vegas-2 and Vegas-3 are always better than that of Vegas-1_3.

However, this time, I do not observe any significant bias in fairness. This is probably because the round-trip propagation delay of the shorter connection is 0.11211 seconds instead of 0.013211 seconds. Therefore, the difference in propagation delays of the 2 connections is not significantly large to cause the bias.

In summary, I observe that fairness of Vegas-1 is unstable but that of Vegas-2 and Vegas-3 are consistently good, regardless of the correctness of the minimum round-trip times. It seems that a value of 1 for δ poses a limit on the expected and the actual sending rate and thus does not allow Vegas-δ to compete fairly for the bottleneck bandwidth. The fairness of Vegas-4 and Vegas-10 are good as well. On the other hand, I have mentioned in section 4.1.2 that the degree of oscillation reduces as the value of δ for Vegas-δ increases. Therefore, it may be argued that the larger the δ, the better. However, there is a tradeoff : with a larger δ, TCP Vegas tries to force more extra packets into the network. Therefore, the larger the δ, the more aggressive TCP Vegas is and thus raises the chance of causing congestion in case of limited available bandwidth. From the simulation results I obtained in this project, a value of 3 for δ seems to be an acceptably good value to use.

4.4 Multiple active flows

In this section, I study the fairness of TCP Vegas when a large number of active TCP Vegas flows compete for the bandwidth over the bottleneck link. 2 scenarios are studied, i.e., 1) when there are many active TCP Vegas flows (but no TCP Reno flows); and 2) when TCP Reno flows compete for bandwidth with TCP Vegas flows.

The first scenario is important in understanding the behavior of TCP Vegas when it is fully deployed in the Internet. Analysis reveals that normally many flows pass through an Internet router simultaneously [M97]. Therefore, it is important that TCP Vegas does not exhibit extraneous behavior when there are many flows. The second scenario is important when TCP Vegas is started to be deployed in the Internet but TCP Reno is still the dominant TCP algorithm. In the next section, I will also explore the differences in fairness between 2 receiver acknowledgement strategies : when the TCP receivers employ the immediate acknowledgement strategy or the delayed acknowledgement strategy.

It has been reported that TCP has scaling limit [M97]. Based on simulations with TCP Tahoe, Morris reported that the limit started to appear when the number of active flows exceeded the network’s bandwidth-delay product as measured in packets. The symptoms are high utilization coupled with high packet losses and high variation in delay as seen by the users. Sufficiently high packet losses also cause low efficiency in bandwidth utilization and goodput.

My study differs from Morris' study in a number of ways. Firstly, the TCP version used in Morris' study is TCP Tahoe, a version of TCP that incorporates the fast retransmit algorithm only (TCP Tahoe is described in section 1 of Chapter 2 of this project report.). In this project, I focus on TCP Vegas, with comparison to TCP Reno. Two scenarios are considered, i.e., 1) when the TCP algorithm used by all the connections is TCP Vegas and 2) when there is a mixture of TCP Vegas and TCP Reno. I adopt most of the simulation parameters used in Morris' paper except the value of the maximum window, the router buffer and the RED thresholds. Instead of setting the maximum window to 64 kilobytes, I assume infinite window size so that it will not impose a limit on throughputs. Accordingly, it is set to approximately three times the maximum bandwidth-delay product. With a maximum one-way propagation delay of 50ms, a bottleneck bandwidth of 10 megabits per second, and a packet size of 576, the bandwidth-delay product is equal to an equivalent of 217 (10Mbps x 2 x 50 / 1000 / 8 / 576) packets. Therefore, 3 times the bandwidth-delay product is roughly equal to 650 packets. 2 settings of RED thresholds are used to represent small and large average queue size in the Internet router. When the minimum threshold is 5 and the maximum threshold is 15, the average queue length can be kept between 5 and 15 packets, or roughly (5+15)/2 = 10 packets. This is like a DropTail router with small buffer size. I call this setting of the RED router "RED-5-15". In comparison, when the minimum and maximum thresholds is 54 and 108, the average queue length can be kept between 54 to 108 packets.. This is like a router with relatively large tolerance on average queue size. I call this setting of the RED router "RED-54-108". The router buffer is set to 100 packets for the 1st setting and 217 packets for the 2nd setting. With a router buffer of 100 packets, a maximum of 317 (217 + 100) active flows can co-exist if all of them use the minimum window size of one. With a router buffer of 217 (217 + 217), a maximum of 434 active flows can be supported simultaneously.

Raghavendra and Kinicki also studied the performance of TCP Vegas with RED. Raghavendra and Kinicki ran simulations with 3 sources sending to the same destination node. In their paper, several configurations were studied : 1) a TCP Vegas host dominated configuration (3 connections with TCP Vegas only), 2) a TCP Reno host dominated configuration (3 connections with TCP Reno only) and 3) a mixed configuration (3 connections with either 1 TCP Reno or 1 TCP Vegas connection). They reported that RED fairness was generally higher than the fairness for the equivalent first-come-first-served (FCFS) configuration. They also implied that the RED minimum and maximum thresholds had little effect on the fairness among the TCP Vegas and the TCP Reno sources. Moreover, FCFS routers displayed more volatile fairness behavior while the RED combinations offered consistent higher values of fairness. In particular, when FCFS routers were used with a mixture of TCP Reno and TCP Vegas network hosts, the number of router buffers were critical. If too few buffers were available, the TCP Vegas hosts captured most bandwidth and thus there was an unfair treatment to TCP Reno. When too many router buffers were available, the TCP Reno hosts would dominate and the TCP Vegas hosts received an unfair treatment.

My study differs from Raghavendra and Kinicki's study again in a number of ways. Firstly, I run simulations with many more active flows, i.e. 30, 100 and 200. The focus is on the fairness that can be achieved when TCP Vegas encounters the RED routers. Secondly, Raghavendra and Kinicki set $\alpha = 1$ and $\beta = 3$. As it has been suggested that the fairness can improve when $\alpha = \beta$, I use $\alpha = \beta = 1$ or 2 or 3, with comparison to the setting where $\alpha = 1$ and $\beta = 3$. Previously, I demonstrated that in a 2-connection scenario, the fairness of TCP Vegas could be improved by setting $\alpha = \beta$. As it is highly likely that an Internet router needs to support many active flows, it is important to know whether the fairness of TCP Vegas is still as good when there are many active flows and $\alpha = \beta$. In addition, Raghavendra and Kinicki compared the fairness of TCP Vegas when it worked with different router configurations. Here, my focus is on the fairness of TCP Vegas when there are many flows passing through the bottleneck.

Since TCP Vegas' mechanism relies on the accurate measurement of the minimum round-trip time and the actual round-trip time, it expects a receiver that immediately acknowledges the received TCP segment. On the other hand, according to RFC1122 (Requirements for Internet Hosts -- Communication Layers dated October 1989), TCP SHOULD implement the delayed acknowledgement strategy. Therefore, I expect that the delayed acknowledgement strategy is widely adopted in the Internet TCP algorithm. On the other hand, the estimation of the minimum round trip times and the actual round-trip times may be incorrect if the receiver adopts a delayed acknowledgement strategy. In case of TCP Vegas, this may affect the calculation of the expected and the actual sending rate and thus the adjustment of the congestion window. As such, it is important that TCP Vegas is able to co-operate with a receiver that adopts the delayed acknowledgement strategy without exhibiting any extraneous behavior.

According to RFC1122, a delayed acknowledgment allows TCP to send an ACK for every other TCP segment received. On the other hand, it is understandable that the delay should not be too long because excessive delays on sending acknowledgements can disturb the round-trip timing and packet "clocking" algorithms. Therefore, RFC1122 also points out that an acknowledgement should not be excessively delayed. In particular, the delay must be less than 0.5 seconds, and in a stream of full-sized segments, there should be an acknowledgement for at least every second segment. According to the specification, the maximum distortion to round-trip delay can be expected to be within 0.5 seconds. While it is important to make sure that the delay will not cause TCP Vegas to exhibit extraneous or damaging behavior while communicating with a receiver that adopts the delayed acknowledgement strategy, it is desirable to understand the impact of the delayed acknowledgement strategy on the fairness.

The network model used is similar to that described previously in section 1 of this chapter. Instead of 2 senders, however, there are now N TCP senders transmitting through the bottleneck link to the N receivers. To introduce randomness, each sender starts at a randomly chosen time in the first 5% of the simulation time. With a simulation time of 60 seconds, the TCP senders start within the first 3 seconds. All goodput measurements are taken after the simulation has started for 20 seconds.

All TCP senders are greedy senders using the FTP application protocol. However, to avoid deterministic behavior and phase effects in the case of TCP Reno only situation, 3 low-bandwidth telnet sessions are run to compete with the main TCP Reno and TCP Vegas flows. Typically, the Telnet links represents less than 0.2% of the bottleneck bandwidth. Fairness index is calculated without the goodput measurements of the telnet sessions. Table 3.19 below gives a summary of the simulation parameters.

4.4.1 Simulation parameters

Bottleneck bandwidth	10 megabits per second	
Bottleneck delay	25ms	
Bottleneck router congestion control mechanism	RED – Random early detection gateway	
Router configuration	RED-5-15	RED-54-108
Bottleneck router buffer	100 packets	217 packets
RED minimum threshold	5 packets	54 packets
RED maximum threshold	15 packets	108 packets
RED maximum packet-marking / dropping probability	1/50 = 2%	
RED queue weight	0.002	
Bandwidth of connection from source node to router	10 megabits per second * 10 / N	
Propagation delay from source to destination node	Random, Uniform from 45ms to 50ms	
TCP packet size	576 packets	
Maximum window size	Very large, about 3 x maximum bandwidth-delay product, i.e., 650 packets	
Vegas-δ	δ = 1 or 2 or 3	
TCP timeout granularity	0.5 seconds (Default in ns is 0.1 seconds)	
Simulation time	60 seconds	
Number of active flows	30, 100 or 200	
Start time of each connection	Random, within the first 5% of the whole simulation time (or 3 seconds)	

Table 3.19 : Summary of the simulation parameters for many active flows

4.4.2 Simulation results - Vegas Only

RED-5-15 (VEGAS OR RENO ONLY)

TCP version	**30 active flows**	**100 active flows**	**200 active flows**
Reno	0.981	0.981	0.959
Vegas-1	0.899 (1 shut down)	0.672 (12 shut down)	0.610 (39 shut down)
Vegas-2	0.862 (0 shut down)	0.642 (14 shut down)	0.685 (23 shut down)
Vegas-3	0.923 (0 shut down)	0.660 (18 shut down)	0.668 (33 shut down)

Table 3.20a : Fairness of many active TCP Vegas or TCP Reno flows, using RED-5-15 router

RED 54-108 (VEGAS OR RENO HOSTS ONLY)

TCP version	30 active flows	100 active flows	200 active flows
Reno	0.983	0.971	0.952
Vegas-1	0.998	0.834	0.708
Vegas-2	0.970	0.818	0.639
Vegas-3	0.967	0.863	0.673

Table 3.20b : Fairness of many active TCP Vegas or TCP Reno flows, using RED-54-108 router

Contrast to the findings with only 2 Vegas connections, disappointingly, the initial set of simulation results reveal that, the fairness of TCP Reno host-only configuration is generally better than that of the TCP Vegas host-only configuration. Exception occurs when the number of active flows is 30, the RED minimum threshold is 54, the RED maximum threshold is 108, and the Vegas thresholds, α and β, are both 1.

The number of active flows has a significant impact on the fairness of the TCP Vegas host-only configuration. In contrast, the TCP Reno host-only configuration shows more stable fairness measure as the number of flows increases. For instance, it can be observed that the fairness of TCP Vegas drops significantly when the number of active flows increases from 30 to 100. With a RED-5-15 router, when the number of active flow is 30, the fairness of TCP Vegas is 0.86 to 0.92 while the fairness of TCP Reno is 0.98. When the number of flows increases to 100 or 200, the fairness of TCP Vegas drops significantly to 0.61 to 0.69 while the fairness of TCP Reno remains fairly stable at 0.96 to 0.98.

The difference between the fairness of a TCP Vegas host-only configuration and a TCP Reno host-only configuration is largest when the RED thresholds are low and the number of TCP active flows is high (e.g. in the range of 100 to 200). For instance, with a RED-5-15 router, when the number of flows is 30, the fairness of the TCP Reno host-only configuration is only 6% to 12% better than that of the TCP Vegas host-only configuration. When the number of active flows is 100, the fairness of the TCP Reno host-only configuration is 31% to 35% better than that of the TCP Vegas host-only configuration. When the number of active flows is 200, the fairness of the TCP Reno host-only configuration is 29% to 36% better than that of the TCP Vegas host-only configuration. On the other hand, the difference between the fairness of a TCP Vegas host-only configuration and that of a TCP Reno host-only configuration is smaller when the RED thresholds are higher. For instance, with a RED-54-108 router, when the number of active flows is 100, the fairness of the TCP Reno host-only configuration is only 11% to 16% better than that of the TCP Vegas host-only configuration.

When the number of active flows is 200, the RED thresholds have little impact on the difference in fairness between the TCP Reno and TCP Vegas host-only configuration. For example, with a RED-54-108 router, the fairness of the TCP Reno host-only configuration is 26% to 33% better than that of the TCP Vegas host-only configuration. With a RED-5-15 router, the fairness of the TCP Reno host-only configuration is 29% to 36% better than that of the TCP Vegas host-only configuration. The magnitudes of difference are about the same.

There seems to be no particular pattern to indicate that a particular choice of α and β (or δ) can give better fairness measure than other settings in all cases.

4.4.2.1 Why some connections are shut down

In order to find out why there is a substantial drop in the fairness of the TCP Vegas host-only configuration, I write Unix awk scripts to analyze the distribution of goodputs and enhance the ns program to produce additional variable traces. Unexpectedly, on a closer investigation into the bandwidth allocation to the TCP connections, I find that some of the TCP Vegas connections are shut down even before the 20 seconds from the start of the simulation. This causes extreme low measure of fairness in case of the TCP Vegas-host-only configuration.

On a closer investigation into the TCP agent traces, I identify that the ns code itself causes 2 problems. With regard to the first cause, the problem arises when a Vegas-type timeout is followed by a Reno-type timeout, which causes TCP Vegas to eventually set the congestion window to zero and thus reach a deadlock status. Unfortunately, this event can occur as the number of losses increase. This explains why the fairness of the TCP Vegas host-only configuration is poorest when the RED threshold is low and the number of active flows are high. When the RED thresholds are too low to support the large number of active flows, many TCP flows will face multiple packet drops. Hence, there is a higher probability that these connections encounter timeouts. It can be noted from table 3.20a that more connections are shut down as the number of active flows increases. Consequently, the fairness measures drop.

Vegas-type timeout : According to the original Vegas code, TCP Vegas checks whether any segments has been timed out on receipt of the first duplicate acknowledgement. If expired segments exist, TCP Vegas triggers fast retransmission followed by fast recovery immediately. The idea is to improve the loss recovery process and shorten the time to recover.

TCP Vegas does this by comparing the time elapsed since the unacknowledged segment is last sent to the timeout value kept by TCP Vegas' congestion avoidance mechanism. If any segment has been timed out, it retransmits the expired segment immediately without waiting until the 3rd acknowledgement comes back.

It also adjusts the congestion window. If the existing window is less than or equal to 3, a window size of 2 will be stored in “v_newcwnd_”. Otherwise, if the expired segment has been transmitted only once, ¾ of the current congestion window is stored in “v_newcwnd_”. Otherwise, half the current value is stored in “v_newcwnd_”. The congestion window is inflated by the number of duplicate acknowledgements, dupacks_, i.e. 1. That means the un-inflated value of the congestion window is stored in the variable “v_newcwnd_” while the inflated value of the congestion window is kept in the variable “cwnd_”.

Afterwards,. TCP Vegas sets the number of duplicate acknowledgements directly to 3. Since the number of duplicate acknowledgements is now set to 3, TCP Vegas then continues with the fast recovery phase. That is, on receipt of another duplicate acknowledgements, TCP Vegas increases the number of duplicate acknowledgements by 1 and checks again if there are any expired segments. Suppose there is no expired segment, TCP Vegas will inflate the congestion window by the number of duplicate acknowledgements and transmit a new TCP segment if the congestion window allows.

Finally, on receipt of a non-duplicate acknowledgement, TCP Vegas resets the congestion window to the latest stored value of the variable “v_newcwnd_”. That is, TCP Vegas checks if the number of duplicate acknowledgements “dupack_” is greater than 3 and the congestion window “cwnd_” is greater than the un-inflated window “v_newcnwd_”. If the conditions are satisfied, TCP Vegas resets the congestion window “cwnd_” to “v_newcwnd_”. Normally, “v_newcwnd_” should store the non-inflated window size calculated just before the last fast recovery.

However, in the ns simulator, this is not the case if the Reno-type timeout occurs before the receipt of the non-duplicate acknowledgment. If a Reno-type timeout occurs before a non-duplicate acknowledgement has been received, the ns code of TCP Vegas sets the value of “v_newcwnd_” to zero and retransmits the first unacknowledged segment. Therefore, the variable “v_newcwnd_” no longer stores the non-inflated window but zero.

Unfortunately, the condition of checking, i.e. cwnd_ > v_newcwnd_ = 0 and dupacks_ > 3, is still satisfied. Consequently, upon the receipt of the non-duplicate acknowledgment, the congestion window is wrongly set to zero.

At this stage, TCP Vegas should transmit and start to time a new segment. In addition, the congestion window “cwnd_” (=0) is now less than the threshold value “ssthresh_” of 2. Therefore, TCP Vegas should enter the slow-start phase. However, accordingly to TCP Vegas’ modified slow-start mechanism, the congestion window is increased only once in every other RTT. If it does not increase the congestion window in the current RTT of the slow-start phase, the congestion window will not be increased to a value that is greater than zero. As a result, the segment, v_begseq_, that should be timed cannot be transmitted, which means that an acknowledgement that acknowledges this v_begseq_ will not be received. Without an acknowledgement for a Vegas timed segment, v_begseq_, there will be no adjustment of the congestion window that results in an increase of the congestion window. Consequently, TCP Vegas comes to a deadlock status.

In order to decide whether this is a bug in ns, I next check the Arizona TCP Vegas code. It is found that the original Arizona code uses the original Reno timeout code without any modifications. Hence, it does not reset or change the value of “v_newcwnd_” in the timeout subroutine (v_newcwnd_ is a variable that is used only by the Vegas congestion avoidance mechanism). Therefore, after the receipt of the new acknowledgment, TCP Vegas resets the congestion window to the value of the "v_newcwnd_" stored by the previous Reno-type fast retransmission or Vegas-type fast retransmission algorithm, instead of zero by the ns Reno-type timeout subroutine. Accordingly, I do not expect a deadlock to occur with the original code.

There is a second cause of the deadlock, which also occurs after a Reno-type timeout. In this case, the Reno-type timeout occurs after the first segment has been transmitted, causing TCP Vegas to record a wrong time at which it starts transmitting.

The ns code for TCP Vegas detects the transmission of the first segment by checking the next send sequence number, t_seqno_. If t_seqno_ is zero, it records the send time of the segment in the "firstsent_" variable. . Effectively, the value of "firstsent_" can be treated as time 0 for the connection. Subsequently, the "firstsent_" variable is referred to in a number of calculations to determine the time elapsed since the start time of the TCP connection. Here are a few examples. For instance, whenever a segment is sent, TCP Vegas records its send time by reading the system clock and then calculating the time elapsed since the value of "firstsent_". Within the congestion avoidance algorithm, TCP Vegas calculates the current time by reading the system clock and then subtracting "firstsent_" from the system time. The resulting figure is then used to update the actual round-trip time, the time that the Vegas-timed segment is sent and the time that the congestion window is reduced because of a Vegas-type fast retransmission or Reno-type fast retransmission algorithm. To determine whether a segment has expired, TCP Vegas calculates the time elapsed since the segment is last sent by subtracting "firstsent_" from the system time and then subtracting the time the segment is last sent from the resultant figure. Within the ns Reno-type timeout subroutine for TCP Vegas, "firstsent_" is used in calculating the time the congestion window is adjusted due to a Reno-type timeout. Therefore, the correctness of “firstsent_” affects the TCP Vegas code extensively.

I illustrate the second cause of deadlock by an example as shown in the agent trace in figure 3.27. The first segment of the connection is sent at time 2.19958 seconds because the ns simulator has scheduled this connection to start sending packets at this time. Before the first segment is sent, t_seqno is initialized to zero. Therefore, the time that the first segment is sent is correctly recorded in “firstsent_” as 2.19958. Then, at 8.19958 seconds, a Reno-type timeout occurs and the congestion window is adjusted. Hence, “t_cwnd_changed_”, which records the time the congestion window is last reduced due to a timeout, Vegas-type fast re-transmission or Reno-type fast-retransmission, is recorded as 6 (8.19958 – 2.19958). Since no acknowledgement has ever been received for this connection, the TCP Vegas sender re-transmits the first segments. However, before the re-transmission of the first segment, t_seqno is reset to the last unacknowledged sequence number, which is zero. This triggers TCP Vegas to wrongly record the send time of the re-transmission, i.e. 8.19958, in “firstsent_”. As a result, TCP Vegas wrongly records that the connection starts transmission at time 8.19958 seconds.

From now on, all the calculations that directly or indirectly use “firstsent_” are distorted. For instance, at time 9.51557 seconds, TCP Vegas detects that segment 21 has expired and thus re-transmits it. Normally, TCP Vegas then compares “t_cwnd_changed_” against the time the re-transmitted segment is last sent. If “t_cwnd_changed_” is less than the time the re-transmitted segment is last sent, TCP Vegas updates “v_newcwnd_” in the same way as I have mentioned previously in the discussion of the first cause of deadlock. It also inflates the congestion window and then updates the value of “t_cwnd_changed_” to the current time. Otherwise, “v_newcwnd_”, “cwnd_” and “t_cwnd_changed_” are left unchanged. However, the time the re-transmitted segment is last transmitted is calculated with reference to “firstsent_”, i.e. the start time of the connection. Since the value of “firstsent_” is wrongly set to 8.19958, the time the segment is last re-transmitted is also incorrectly calculated as 9.21605 – 8.19958 = 1.01647, instead of 9.21605 – 2.19958 = 7.01647. Unfortunately, the value of “t_cwnd_changed_” is still 6. Therefore, the time the re-transmitted segment is last sent becomes less than “t_cwnd_changed_”. Consequently, “cwnd_”, “v_newcwnd_” and “t_cwnd_changed_” are not adjusted. In particular, “v_newcwnd_” remains at the initial value of 0 instead of the true un-inflated window.

Afterwards, another duplicate acknowledgement is received at 9.52386 seconds. Therefore, TCP Vegas increases the number of duplicate acknowledgement by 1 to 4. It also inflates the congestion window by 1 to 5.676 because the number of duplicate acknowledgement has now increased by 1.

The problem then occurs. At time 9.64505 seconds, TCP Vegas receives a non-duplicate acknowledgement. Now, same as what happens in the first cause of deadlock, since cwnd_ > v_newcwnd_ = 0 and dupacks_ > 3, TCP Vegas resets the congestion window to the value of “v_newcwnd_”, which is zero. Again, TCP Vegas comes to a deadlock.

This demonstrates how differences in the implementation can cause very different performance. I remove the ns code that set the value of the “v_newcwnd_” to zero. As I have mentioned before, the original Arizona code does not change the Reno-type timeout subroutine in any way. Therefore, I also remove the line of code that updates the “t_cwnd_changed_” due to the occurrence of a Reno-type timeout. More importantly, I modify the ns code so that it does not update the “firstsent_” variable again on re-transmission of the first segment after a Reno-type timeout. The simulation results show much improved performance in fairness, especially when the number of active flows is large. In addition, no TCP Vegas connections are now shut down.

Tables 3.21a and 3.21b give a comparison of the fairness before and after the modification of the ns code. Each of the fairness measure obtained after the modification of the ns code is the average fairness of 7 simulations. Figures 3.25a and 3.25b present the distribution of the fairness index for each simulation configuration.

RED-5-15 (ONLY VEGAS OR RENO HOSTS)

TCP Version	30 active flows		100 active flows		200 active flows	
	Before	**After**	**Before**	**After**	**Before**	**After**
Reno	0.981		0.981		0.959	
Vegas-1	0.899	0.912	0.672	0.822	0.610	0.914
Vegas-2	0.862	0.879	0.642	0.833	0.685	0.892
Vegas-3	0.923	0.900	0.660	0.911	0.668	0.904

Table 3.21a : Fairness of many active flows, before & after modification of the ns code, using RED-5-15 router

RED 54-108 (ONLY VEGAS OR RENO HOSTS)

TCP Version	30 active flows		100 active flows		200 active flows	
	Before	**After**	**Before**	**After**	**Before**	**After**
Reno	0.983		0.971		0.952	
Vegas-1	0.998	0.996	0.834	0.897	0.708	0.910
Vegas-2	0.970	0.963	0.818	0.893	0.639	0.886
Vegas-3	0.967	0.971	0.863	0.930	0.673	0.918

Table 3.21b : Fairness of many active flows, before & after modification of the ns code, using RED-54-108 router

It can be noted that, after the change in the ns code has been applied, TCP Reno still gives better fairness measure than TCP Vegas in my simulation setting. Again, exception occurs when the number of active flow is 30, the minimum threshold is 54, the maximum threshold is 108 and $\alpha = \beta = 1$. I notice that this is the only setting in which the number of flows times δ=1 is less than the minimum thresholds. Therefore, there are no or few retransmissions and timeouts. Intuitively, this suggests that the fairness of TCP Vegas is better when there are no or few retransmissions and timeouts. In section 4.4.3.1, I discuss that the record of an incorrect minimum round-trip time causes a larger variance of goodputs among the TCP Vegas connections. The chances that this happen is higher when there are more timeouts. Therefore, this problem helps to explain why the fairness of TCP Vegas is better when there are no or few retransmissions and timeouts.

Moreover, I observe that the fairness of TCP Vegas is better when the RED thresholds are larger. The percentage improvement in fairness, however, reduces as the number of flows increases and finally diminishes as the number of flows reaches 200. This phenomenon is reasonable because there are fewer transmissions and timeouts when the RED thresholds are higher. Consequently, the fairness of TCP Vegas is also better.

	30 flows	**100 flows**	**200 flows**
Vegas-1	9.2%	9.1%	-0.4%
Vegas-2	9.6%	7.2%	-0.7%
Vegas-3	7.9%	2.1%	1.5%

Table 3.21c : Percentage improvement in fairness of RED-54-108 over RED-5-15
(TCP Vegas or Reno only)

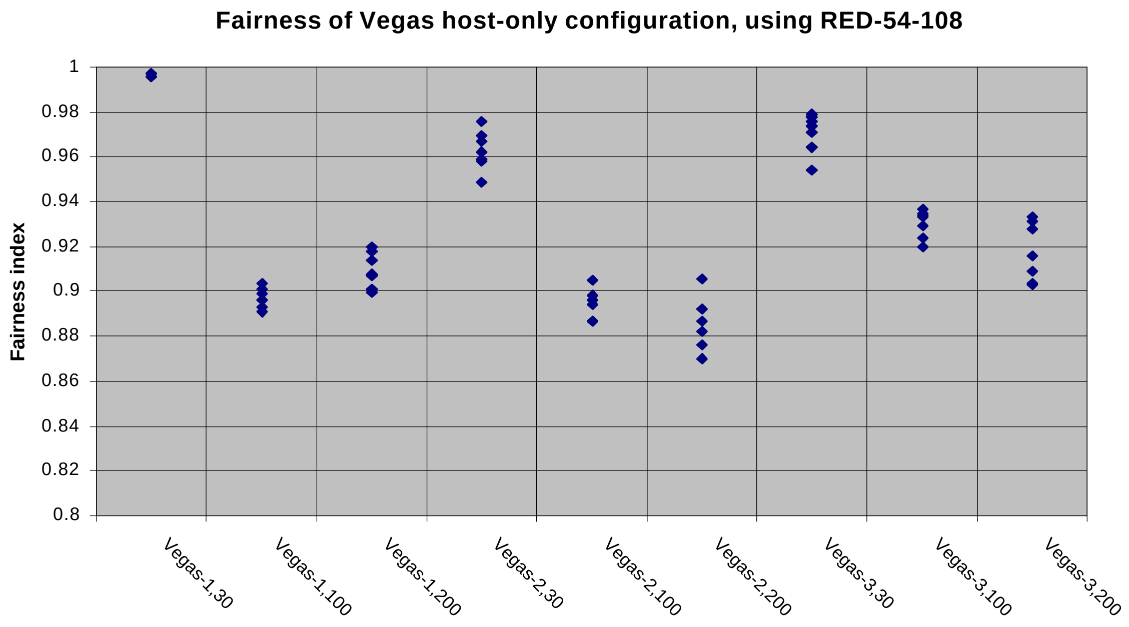


Figure 3.25a : Fairness of Vegas host-only configuration, after modification of the ns code, using RED-54-108 router

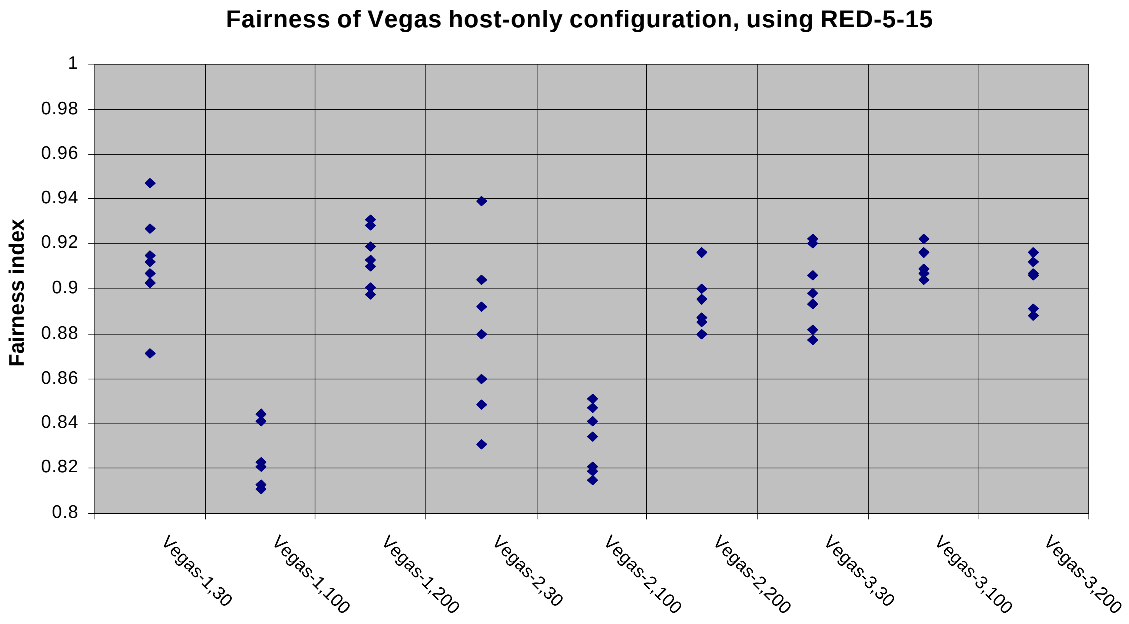


Figure 3.25b : Fairness of Vegas host-only configuration, after modification of the ns code, using RED-5-15 router

Figure 3.26 below gives an example of a TCP agent trace obtained with one of the ns simulations. It demonstrates the first cause of deadlock. Figure 3.27 gives an example of the TCP agent trace that demonstrates the second cause of the deadlock.

```
Receive new ack at 17.22847s
ack = 239
v_delta_ = 1.863830
cwnd_ = 1                                  = --cwnd_
cwnd_ = 2                                  cwnd_< 1 => cwnd_ = 2
v_incr_ = 0
v_begseq_ = 241                            Vegas timed segment 241
v_begtime = 16.687378                      = 17.228472-0.541094
v_timeout_ = 0.147511
ndatapack_ = 272                           Send_much (241,0)
maxseq_ = 241
t_seqno = 242

Receive duplicate ack at 17.334s
ack = 239
dupacks_ = 1
expired = 240                              Vegas-type timeout of segment 240
v_worried = 2
v_timeout_  = 0.165950                     = 0.147511 + 0.147511/8
v_newcwnd_ = 2
cwnd_ = 3                                  = v_newcwnd_ + dupacks_
t_cwnd_changed_ = 16.792901                = 17.334 – 0.5410934
reset_rtx_timer(1,0)                       reset rtx timer without backoff
ndatapack_ = 273                           Output (240, TCP_REASON_DUPACK)
mrexmitpack_ = 31
dupack_ = 3
ndatapack_ = 274                           Send_much (242, 0)
maxseq_ = 242
rtt_seq_ = 242                             Reno timed segment 242
rtt_ts_ = 17.3340
t_seqno = 243

Reno-type timeout at 17.734s (0.4s after segment 242 is sent)
dupacks_ = 0                               Reno-type timeout of segment 242
recover_ = 242
reset_rtx_timer(0,1)                       reset rtx timer with backoff
cwnd_ = 1
ssthresh = 2
cwnd = 2                                   Vegas initializes to 2 after Reno-type timeout
v_newcwnd = 0                              THIS LINE CAUSES DEADLOCK PROBLEM.
t_cwnd_changed_ = 17.192901                17.734 – 0.54109434
ndatapack_ = 275                           Send_much (240, TCP_REASON_TIMEOUT)
nrexmitpack_ = 32
t_seqno = 241
ndatapack_ = 276                           Send_much (241, TCP_REASON_TIMEOUT)
nrexmitpack_ = 33
t_seqno = 242

Receive new ack at 17.84010s
ack = 241
v_delta_ = 0.934156
v_incr_ = 0.5
v_begseq_ = 242                            Vegas timed segment 242
v_begtime_ = 17.299004                     = 17.84010 – 0.54109434
cwnd_ = 2.5
transmits = 2                              Segment 241 has been transmitted twice.
v_worried_ = 2
v_worried_ = 1
expired = 242                              Vegas-type timeout for segment 242 (17.84010 – 17.334 = 0.5061)
dupacks_ = 3
ndatapack_ = 277                           Output ( 242, TCP_REASON_DUPACK)
nrexmitpack_ = 34
ndatapack_ = 278                           Send_much (242 , 0) since 242 < 241 + 2.5 so output (242)
nrexmitpack_ = 35
t_seqno = 243
ndatapack_ = 279                           Send_much (243 , 0) since 243 < 241 + 2.5 so output (243)
maxseq_ = 243
rtt_active_ = 1                            Start Reno re-transmission timer ...
rtt_seq_ = 243                             Reno timed segment 243
rtt_ts = 17.8401
t_seqno = 244

Receive duplicate ack at 17.84517s
ack = 241
dupacks_ =  4
expired = -1
cwnd_ = 3.5                                since dupack_ > 3
ndatapack_ = 280                           Send_much (244 , 0) since 244 < 241 + 3.5 so output (244)
maxseq_ = 244
t_seqno = 245

Receive new ack at 17.95207s
ack = 242
cwnd_ = 0                                  since dupack_ > 3 & cwnd > v_newcwnd_ = 0 => cwnd_ = v_newcwnd_
ssthresh_ = 2                              cause slow start phase
dupacks_ = 0
v_delta_ = 2.802469
slow_start_phase_ = 1
v_incr_flag_ = 0                           During slow-start phase, congestion window increase every other RTT.
v_incr_ = 0
v_begseq_ = 245                            Vegas timed segment 245
v_begtime_ = 17.410978
Since congestion window is zero, segment 245 cannot be sent, so Vegas congestion avoidance mechanism is broken.
```

Figure 3.26 : Agent trace showing how TCP Vegas is shut down after a Reno-type timeout that resets v_newcwnd_ to 0

```
First segment sent at time 2.19958 seconds
maxseq_ = 0
firstsent_ = 2.19958

Reno-type timeout at 8.19958 seconds
nrexmit_  = 1
cwnd_ = 2
t_cwnd_changed_ = 6.000000            = 8.19958 –  firstsent_
nrexmitpack_ = 1                      re-transmit segment with sequence number 0
firstsent_ = 8.19958                  upon re-sending segment with sequence number 0, t_seqno is reset to 0, so TCP Vegas assumes
                                      this is the 1st segment of the connection and re-initialize firstsent_ variable
maxseq_ = 1

.
.
Transmit segment 21 at 9.21605 seconds
maxseq_ = 21                          sendTime of segment 21 = 9.21605 – 8.19958 = 1.01647 instead of 9.21605 – 2.19958 =
7.01647
.
.
.

Detect and re-transmit an expired segment at 9.51557 seconds
dupack_ = 2
expired = 21
nrexmitpack_ = 4                      re-transmit expired segment 21 but since sendTime of segment 21 < t_cwnd_changed_ of 6,
                                      cwnd_ is not reduced. Also, since sendTime < t_cwnd_changed of 6, v_newcwnd_ is not
                                      updated and so remains as the initial value of 0
dupacks_ = 3                          since sendTime < t_cwnd_changed_ of 6, cwnd_ is not inflated by dupacks_ of 3.

Receive another duplicate acknowledgement at 9.52386 seconds
dupacks_ = 4
expired = -1
cwnd_ = 5.676                         inflate congestion window by 1 since 1 more duplicate acknowledgement has been received
maxseq_ = 25

Receive a non-duplicate acknowledgement at 9.64505 seconds
cwnd_ = 0                             since dupacks_ > 3 and cwnd_ of 5.676 > v_newcwnd_ of 0, cwnd_ is wrongly set to 0
dupacks_ = 0
```

Figure 3.27 : Agent trace showing how TCP Vegas is shut down, due to a Reno-type timeout that occurs only after the 1st TCP segment has been sent.

4.4.2.2 *Unnecessary retransmission*

After analyzing the ns agent traces, I discover that the ns version of the TCP Vegas sometimes performs unnecessary re-transmission. Again, the problem occurs after a Reno-type timeout. I illustrate the phenomenon by an example.

An example is shown with the ns trace in figure 3.28. It can be seen from figure 3.28 that TCP segment with sequence number 241 is transmitted at 17.22847 seconds. At 17.334 seconds, TCP Vegas detects that the segment 240 has been timed out. Thus, it re-transmitted this expired segment. In addition, it transmits a new segment with sequence number 242 because the congestion window is now inflated by 1 duplicate acknowledgement. At 17.734 seconds, however, a Reno-type timeout occurs. Since the last unacknowledged segment is segment 240, it re-transmits segment 240. In addition, since TCP Vegas reduces the congestion window to 2 instead of 1 after a Reno-type timeout, TCP Vegas is able to send another TCP segment with sequence number 241. The next send sequence number is changed to 242 accordingly. Then at time 17.84517 seconds, TCP Vegas sender receives a non-duplicate acknowledgement for segment 241. According to TCP Vegas' modified recovery strategy, it needs to check whether any segment has been timed out. However, upon checking, it detects that segment 242 has expired. Therefore, it re-transmits segment 242. Following the retransmission, the output subroutine needs to check if the congestion window allows the transmission of more new segments (i.e. segments that have not been transmitted before). However, since the next send sequence number is now 242, the expired segment is transmitted again at the same timing, causing unnecessary re-transmission and a waste of the bandwidth.

```
Receive new ack at 17.22847s
ack                     239
v_delta_                1.863830
cwnd_                   2
v_begseq_               241
ndatapack_              272                 transmit new segment 241
maxseq_                 241
t_seqno_                242

Receive duplicate ack at 17.33400s
dupacks_                1
expired                 240
cwnd_                   3
ndatapack_              273                 re-transmit expired segment 240
dupacks_                3
ndatapack_              274                 transmit new segment 242
maxseq_                 242
t_seqno_                243

Reno-type timeout at 17.73400s
dupacks_                0
ssthresh_               2
cwnd_                   2
v_newcwnd_              0
ndatapack_              275                 re-transmit 240, since last unacknowledged segment is
240
t_seqno_                241
ndatapack_              276                 re-transmit 241
t_seqno_                242                 next send sequence number is updated to 242

Receive new ack at 17.84010s
ack_                    241
v_delta_                0.934156
v_begseq_               242
cwnd_                   2.5
expired                 242
dupacks_                3
ndatapack_              277                 re-transmit  expired segment 242
ndatapack_              278                 unnecessary re-transmission <= send_much (242,0)
                                            since t_seqno is 242, TCP Vegas assumes segment
                                            242 is the next new segment
t_seqno_                243
ndatapack_              279                 transmit new segment 243
maxseq_                 243
t_seqno_                244
```

Figure 3.28 : Agent trace showing that TCP Vegas performs unnecessary retransmission

Obviously, one solution is to change the TCP Vegas code so that it will check the next send sequence number against the sequence number of the expired segment sent. If the next send sequence number is less than or equal to the sequence number of the expired segment sent, the TCP Vegas should increment the next send sequence number to the sequence number of the expired segment that has already re-transmitted plus 1.

4.4.3 Simulation results - A mixture of Vegas and Reno connections

Next, I consider cases where there is a mixture of TCP Vegas and TCP Reno connections. I try different mixes of TCP Reno and TCP Vegas connections, i.e. 10%, 30%, 50%, 70% and 90% of the connections run TCP Vegas. Two Vegas configurations are compared : 1) $\alpha = \beta = 3$ and 2) $\alpha = 1$ & $\beta = 3$. Tables 3.22a and 3.22b below give the fairness with 2 settings of RED thresholds :. 1) RED-5-15 : when the RED minimum threshold is 5 and the maximum threshold is 15, or 2) RED-54-108 : when the RED minimum threshold is 54 and the maximum threshold is 108. The fairness measures are obtained by calculating the average fairness each from 7 simulation runs.

RED-5-15	**0% Vegas**		**10% Vegas**		**30% Vegas**		**50 % Vegas**		**70% Vegas**		**90% Vegas**		**100% Vegas**	
NO. OF ACTIVE FLOWS	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3
30	0.984		0.974	0.956	0.941	0.945	0.933	0.930	0.907	0.907	0.917	0.879	**0.900**	0.873
100	0.975		0.958	0.961	0.936	0.920	0.925	0.896	0.913	0.872	0.912	0.852	0.911	0.845
200	0.955		0.912	0.927	0.891	0.886	0.881	0.895	0.891	0.892	0.900	0.896	0.904	0.900

Table 3.22a : Average fairness of different mixes of TCP Vegas & TCP Reno, using RED-5-15 router

RED-54-108	**0% Vegas**		**10% Vegas**		**30% Vegas**		**50 % Vegas**		**70% Vegas**		**90% Vegas**		**100% Vegas**	
NO. OF ACTIVE FLOWS	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3
30	0.976		0.981	0.969	0.976	0.909	0.979	0.852	0.972	0.786	0.980	0.739	0.971	0.778
100	0968		0.960	0.951	0.947	0.945	0.939	0.928	0.929	0.908	0.926	0.884	0.930	0.872
200	0.954		0.929	0.947	0.912	0.919	0.903	0.905	0.899	0.898	0.909	0.890	0.918	0.891

Table 3.22b : Average fairness of different mixes of TCP Vegas & TCP Reno, using RED-54-108 router

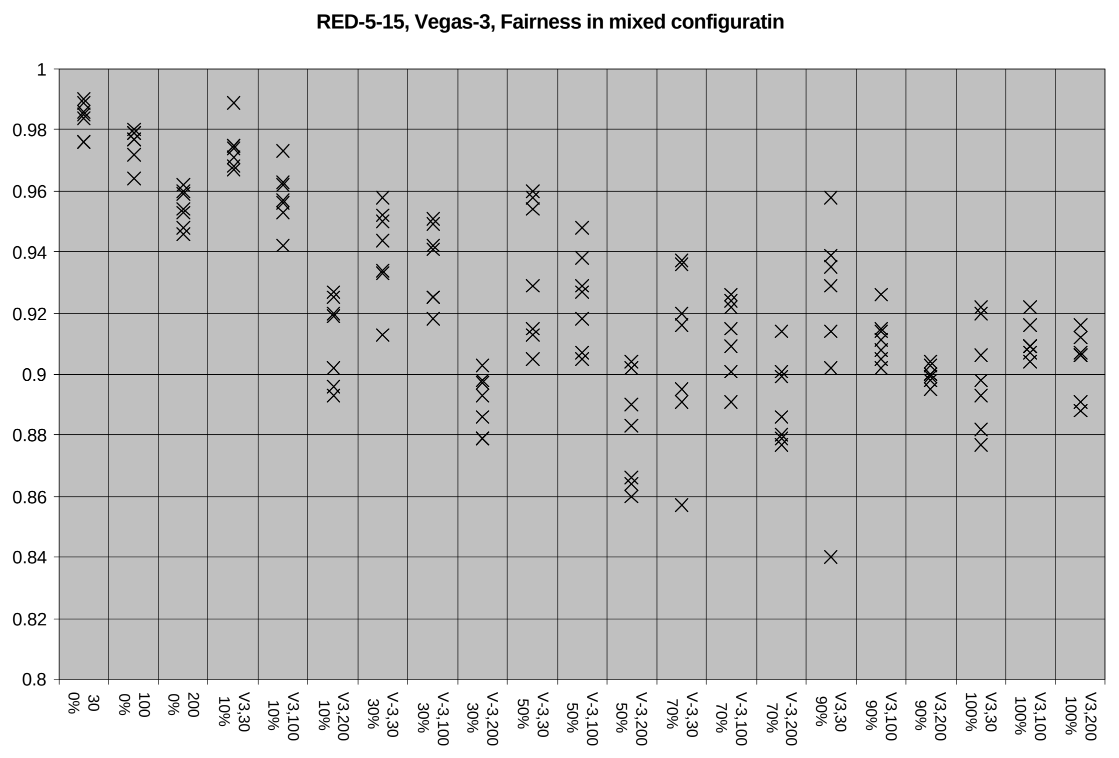


Figure 3.29a : Fairness of different mixes of Vegas-3 & Reno, using RED-5-15 router

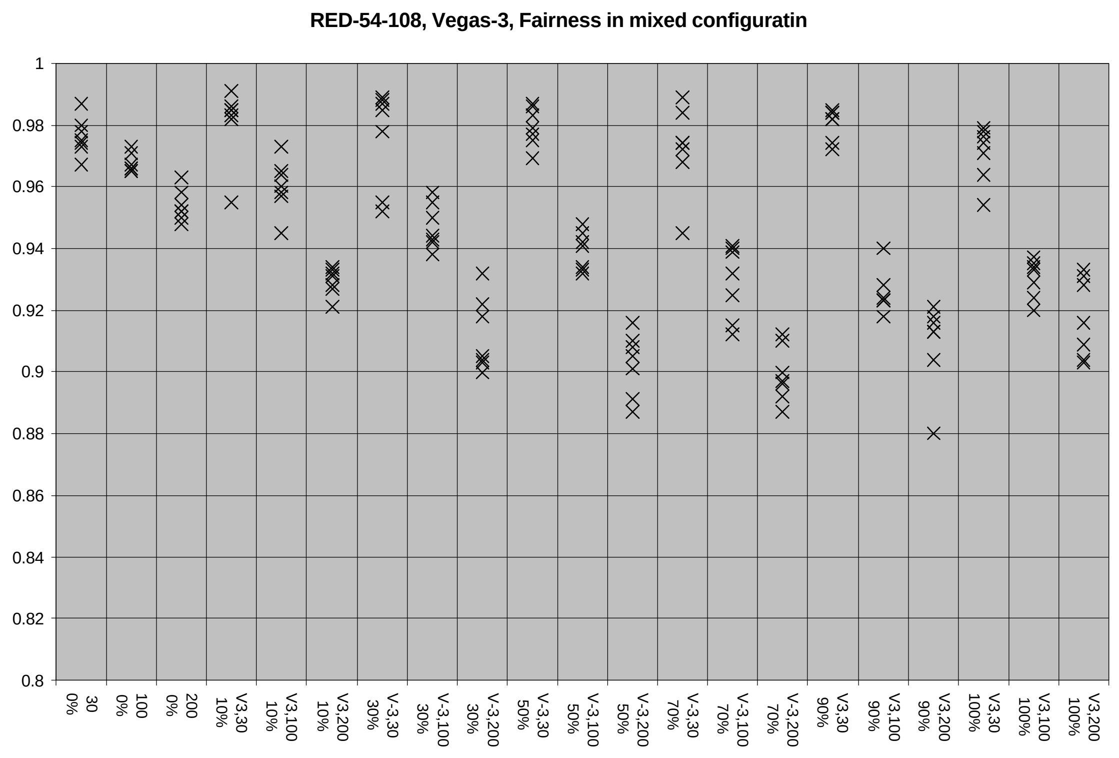


Figure 3.29b : Fairness of different mixes of Vegas-3 & Reno, using RED-54-108 router

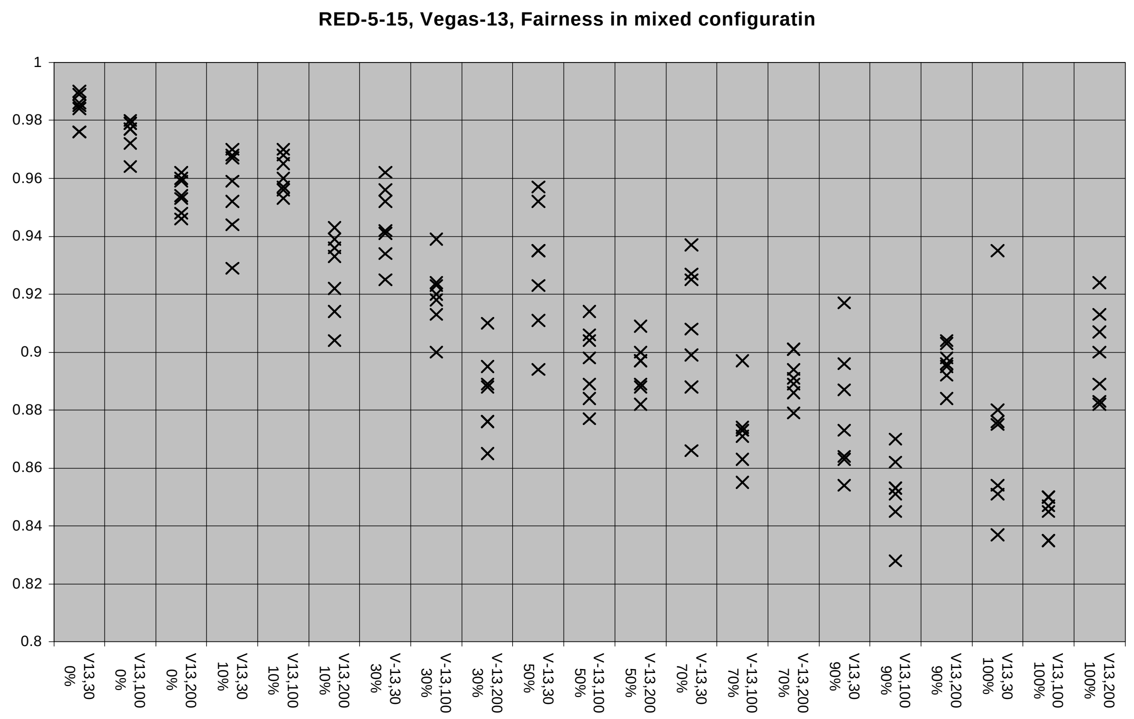


Figure 3.29c : Fairness of different mixes of Vegas1_3 & Reno, using RED-5-15 router

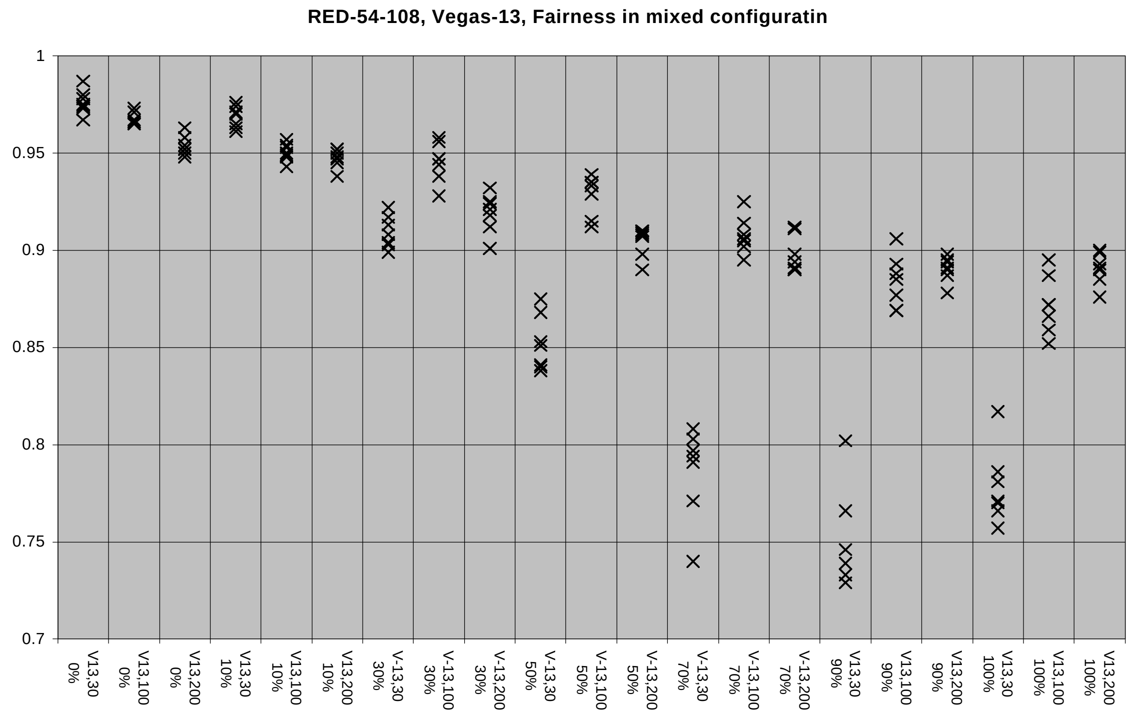


Figure 3.29d : Fairness of different mixes of Vegas1_3 & Reno, using RED-54-108 router

Impact of RED thresholds : When $\alpha = \beta = 3$, the average fairness is higher when the RED thresholds are higher, regardless of the simulation setting (excluding the case in which there are only TCP Reno connections in the network). The difference is larger when the number of active flows is low and the percentage of Vegas-3 is high. Table 3.23 below shows the percentage improvements in fairness of RED-54-108 over RED-5-15.

NO. OF ACTIVE FLOWS ($\alpha = \beta = 3$)	**0% Vegas**	**10% Vegas**	**30% Vegas**	**50 % Vegas**	**70% Vegas**	**90% Vegas**	**100% Vegas**
30	-0.8%	0.7%	3.7%	4.9%	7.2%	6.9%	7.9%
100	-0.7%	0.2%	1.2%	1.5%	1.8%	1.5%	2.1%
200	-0.1%	1.9%	2.4%	2.5%	0.9%	1.0%	1.5%

Table 3.23 : Percentage improvement in average fairness of RED-54-108 over RED-5-15, when there is a mixture of many Vegas-3 & Reno flows

Impact of the number of active flows : When $\alpha = \beta = 3$, the fairness tends to deteriorate as the number of active flows increases, regardless of the setting of the router thresholds. However, when the RED threshold is low and the percentage of Vegas-3 is relatively high (e.g. ƒ 70%), there is a large overlap in the fairness index and the difference seems to diminish. In these cases, the variance of the fairness measure is larger when there are 30 flows than when there are 100 or 200 flows. In particular, the fairness gets worse if more connections get into the problem of an under-estimated minimum round-trip time (This problem will be discussed in section 4.4.3.1.). On the other hand, when $\alpha = 1$ and $\beta = 3$, this relationship may not hold. For example, when the router buffer is high (e.g. using RED-54-108), the fairness with 30 active flows is worse than that with 100 active flows, unless the percentage of TCP Vegas1_3 connections is low (e.g. 10%).

RED-5-15	**0% Vegas**		**10% Vegas**		**30% Vegas**		**50 % Vegas**		**70% Vegas**		**90% Vegas**		**100% Vegas**	
NO. OF ACTIVE FLOWS	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3
100 over 30	-0.9%		-1.6%	0.5%	-0.5%	-2.6%	-0.9%	-3.7%	0.7%	-3.9%	-0.5%	-3.1%	1.2%	-3.2%
200 over 100	-2.1%		-4.8%	-3.5%	-4.8%	-3.7%	-4.8%	-0.1%	-2.4%	2.3%	-1.3%	5.2%	-0.8%	6.5%

Table 3.24a : Percentage change in average fairness vs. number of flows, using RED-5-15 router

RED-54-108	**0% Vegas**		**10% Vegas**		**30% Vegas**		**50 % Vegas**		**70% Vegas**		**90% Vegas**		**100% Vegas**	
NO. OF ACTIVE FLOWS	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3	α=3 β=3	α=1 β=3
100 over 30	-0.8%		-2.1%	-1.9%	-3.0%	4.0%	-4.1%	8.9%	-4.4%	15.5%	-5.5%	19.6%	-4.2%	12.1%
200 over 100	-1.4%		-3.2%	-0.4%	-3.7%	-2.8%	-3.8%	-2.5%	-3.2%	-1.1%	-1.8%	0.7%	-1.3%	2.2%

Table 3.24b : Percentage change in average fairness vs. number of flows, using RED-54-108 router

Impact of the percentage of TCP Vegas : Although there are large overlaps in the fairness measures, it can be said that the fairness tends to deteriorate as the percentage of TCP Vegas-3 connections increases. In case when the RED thresholds are high and 30 flows are present, the decrease is not significant. When there are 200 active flows, the result is a bit mixed.

When the RED thresholds are low and 30 or 100 flows are present, or when the RED thresholds are high and 100 flows are present, the fairness index decreases as the percentage of TCP Vegas-3 increases from 0% to 50%. This is probably because the variance of the goodputs in the Vegas community is larger than that in the Reno community. Therefore, as the percentage of TCP Vegas-3 connections increases, the fairness as a whole also deteriorates. However, the fairness index then levels off as the percentage of TCP Vegas-3 continues to increase. Thereby, a further increase in the percentage of TCP Vegas-3 connections has minimal impact on the fairness measure of the whole network.

When the number of flows is 200, the fairness of TCP Vegas-3 also deteriorates as the percentage of TCP Vegas increases. The decrease continues until the percentage of TCP Vegas-3 reaches about 50-70% but then tends turn upwards as the percentage of TCP Vegas-3 continues to increase. One observation is that the difference between the goodputs of the TCP Vegas-3 connections and that of the TCP Reno connections is larger when there are 200 flows than when there are 100 flows. Therefore, in the case of 200 flows, the difference in goodputs between the TCP Vegas-3 community and the TCP Reno community is possibly so larger that the fairness becomes better as the percentage of the TCP Reno flows reduces.

Impact of a and b : Comparatively, the drop in fairness is more significant when α =1 and β =3. Thus, it seems that the fairness measure is more stable when $\alpha = \beta = 3$ than when $\alpha = 1$ and $\beta = 3$.

RED-54-108

$\alpha = \beta = 3$	30% Vegas	50 % Vegas	70% Vegas
30 active flows	0.977 (234163/343386)	0.969 (293599/371919)	0.964 (323614/416524)
100 active flows	0.972 (114486/97053)	0.945 (107171/90024)	0.922 (99236/93750)
200 active flows	0.919 (67306/47123)	0.910 (56724/40596)	0.900 (49539/35516)

Table 3.25a : Simulation results when there is a mixture of Vegas-3 & TCP Reno, using RED-54-108 router

RED-54-108

$\alpha = 1, \beta = 3$	30% Vegas	50 % Vegas	70% Vegas
30 active flows	0.960 (189427/347960)	0.844 (210509/454940)	0.759 (271019/888998)
100 active flows	0.964 (73244/101549)	0.920 (90460/106675)	0.883 (96956/116536)
200 active flows	0.949 (50734/48863)	0.916 (51865/45755)	0.891 (49217/41432)

Table 3.25b : Simulation results when there is a mixture of Vegas1_3 & TCP Reno, using RED-54-108 router

RED-5-15

$\alpha = \beta = 3$	30% Vegas	50 % Vegas	70% Vegas
30 active flows	0.962 (354355/297238)	0.937 (301939/320963)	0.926 (320042/308864)
v100 active flows	0.949 (110335/89808)	0.920 (110986/81527)	0.931 (103800/79588)
200 active flows	0895 (60808/41305)	0.888 (57531/36773)	0.901 (50991/35585)

Table 3.25c : Simulation results when there is a mixture of Vegas-3 & TCP Reno, using RED-5-15 router

RED-5-15

$\alpha = 1, \beta = 3$	30% Vegas	50 % Vegas	70% Vegas
30 active flows	0.913 (237901/339379)	0.949 (312307/316247)	0.842 (290288/352538)
100 active flows	0.923 (113476/88255)	0.891 (101272/91257)	0.889 (97329/94748)
200 active flows	0.890 (58963/42945)	0.886 (54818/40014)	0.900 (51134/38425)

Table 3.25d : Simulation results when there is a mixture of Vegas1_3 & TCP Reno, using RED-5-15 router

The fairness measures in the above 4 tables are quoted from one set of the simulation runs. The figures next to the fairness measure give the average goodput of all the TCP Vegas connections and that of all the TCP Reno connections, respectively. For instance, with RED-54-108, $\alpha = \beta = 3$ and when 10% of the 200 connections are Vegas connections, the fairness index is 0.919. The corresponding average goodput of all the Vegas connections is 67,306 bps while the average goodput of all the Reno connections is 47,123 bps.

The simulation results show that, when RED thresholds are high and the number of active flows is low (30 when $\alpha = \beta =3$, and 30 or 100 when $\alpha = 1$ & $\beta = 3$), TCP Reno is the winner. However, when the number of active flows is large (100 or 200 when $\alpha = \beta =3$. and 200 when $\alpha = 1$ & $\beta =3$), TCP Vegas becomes the beneficiary of the unfairness problem.

When the RED thresholds are low and the number of active flows is relatively low (e.g. 30 when $\alpha = \beta =3$, and 100 when $\alpha = 1$ & $\beta = 3$), even each TCP Vegas connection may capture higher goodput than the TCP Reno connections on average. Intuitively, this suggests that when the RED thresholds are high enough, TCP Reno is able to benefit from a larger window and thus becomes the winner in the unfairness problem.

Due to the setting of the thresholds, the RED-5-15 gateway starts to drop packets when its queue is longer than 5 packets and drops all arrived packets when its queue is longer than 15 packets. When there are 100 connections, it is highly likely that these queue lengths are met and thus there will be a large number of packet drops. The retransmission of dropped packets is unavoidable. When there are too many losses, timeouts are also inevitable.

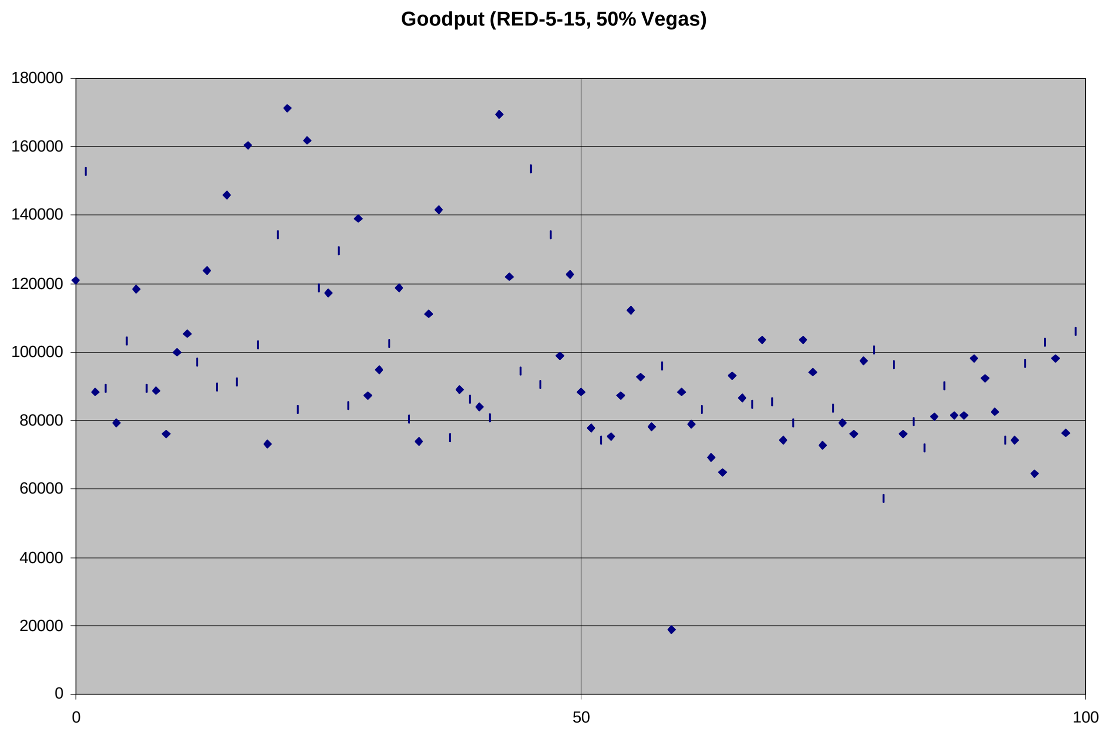


Figure 3.30 : Distribution of goodput when there is a mixture of Vegas-3 & Reno
Connection 0 – 49 : TCP Vegas, Connection 50 – 99 : TCP Reno

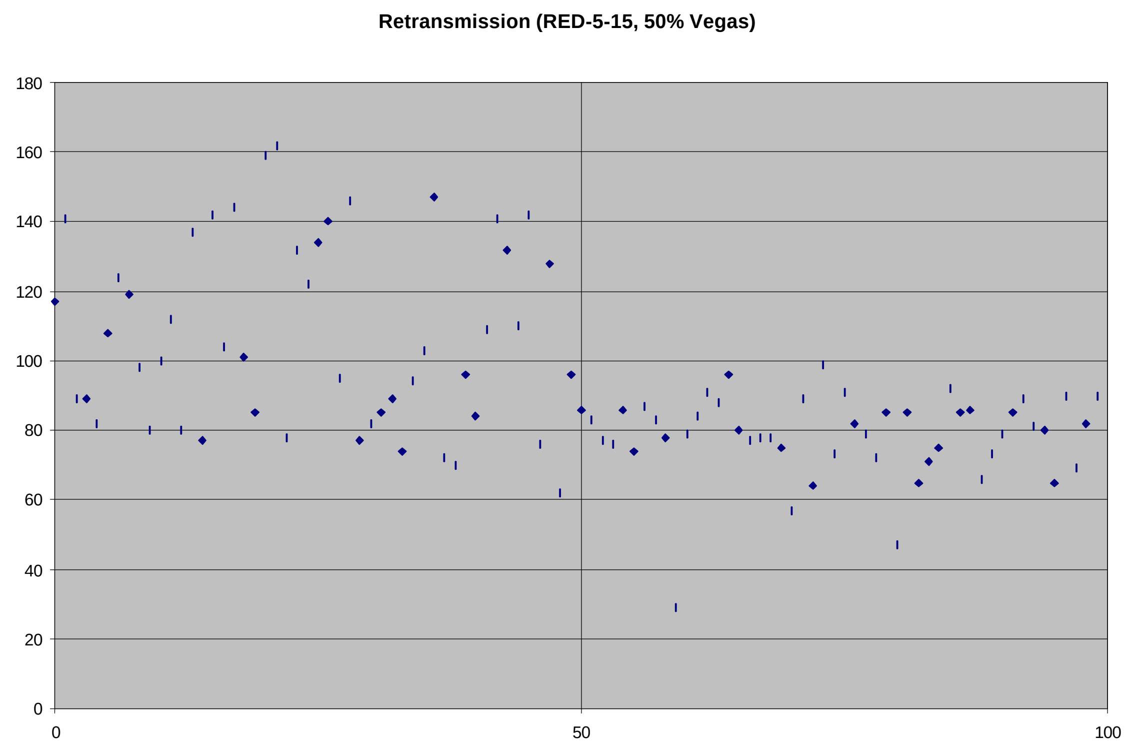


Figure 3.31 : Distribution of retransmission counts when there is a mixture of Vegas-3 & Reno
Connection 0 – 49 : TCP Vegas, Connection 50 – 99 : TCP Reno

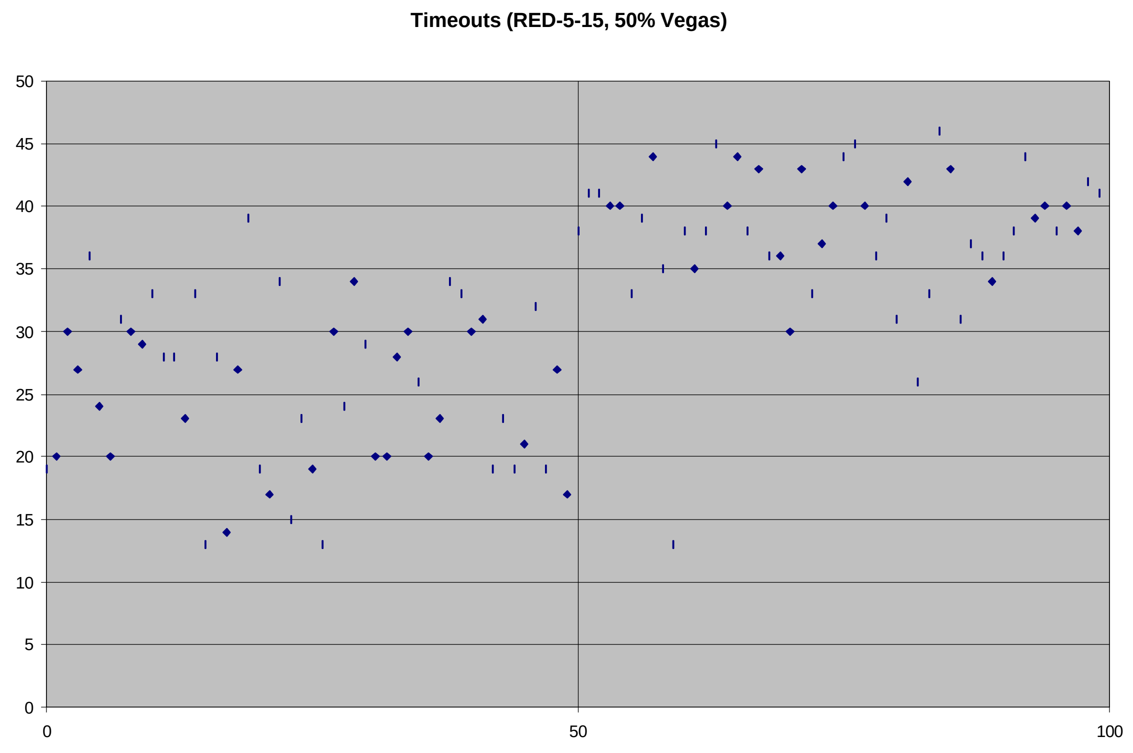


Figure 3.32 : Distribution of timeouts counts when there is a mixture of Vegas-3 & Reno
Connection 0 – 49 : TCP Vegas, Connection 50 – 99 : TCP Reno

Figures 3.30, 3.31 and 3.32 above show the distribution of the goodput, the retransmissions and the timeouts obtained from one of the simulation runs. Connections 0 to 49 are TCP Vegas connections while the rest are TCP Reno connections.

The figures reveal that TCP Vegas retransmits more aggressively but has less timeouts. By re-transmitting before timeouts occur, TCP Vegas is able to save time in waiting and thus transmit more bytes. Therefore, it is reasonable that TCP Vegas is able to capture more bandwidth. However, it can also be observed that the variance of the goodputs in the Vegas community is larger than that in the Reno community. Therefore, while the unequal allocation of bandwidth between the Vegas community and the Reno community contributes to the unfairness, the unequal allocation of the bandwidth in the Vegas community also contributes significantly to the unfairness

Interestingly, all of the TCP Vegas connections with low goodputs also re-transmit less. TCP Reno connections still have more timeouts than the TCP Vegas connections on average.

4.4.3.1 Under-estimated baseRTT_ after a Reno-type timeout

Time	Event	t_seqno	ack_	v_baseRTT_	v_actual	v_expect_	v_delta_	v_begseq_	cwnd_	v_sumRTT_/ v_cntRTT	t_seqno
				0.095726				1056	3.672	0.210125/2	1058
20.87894	new	1058	1055	-	-	-	-	-	4.004	0.31703/3	1060
21.27894	Timeout	1060	-	-	-	-	-	-	2	-	1058
21.38648	new	1058	1056	-	(1058 – 1056) / (0.31703 / 3) = 18.92563	(1058 – 1056) / 0.095726 =20.89300	0.188329	1058	2.5	not recorded for retransmitted packets	1059
21.38830	new	1059	1059	**21.38830 – 21.38648 = 0.001822**	(1060 – 1058) / **0.001822** = 1097.902	(1060 – 1059) / **0.001822** = 548.9511	**-1.000188**	1060	2.9	**0.509364/1**	1062
21.48934	new	1062	1060	-	(1062 – 1060) / 0.509364 = 3.926463	(1062 – 1060) / 0.001822 = 1097.902	1.992847	1062	3.245	0.101033/1	1064
21.49164	new	1064	1061	-	-	-	-	-	3.59	0.204371/2	1065
21.59442	new	1065	1062	-	(1065 – 1062) / (0.204371 / 2) = 29.35838	(1065 – 1062) / 0.001822 = 1646.853	2.946519	1065	3.868	0.105084/1	1066

Figure 3.33 : An example of under-estimated minimum RTT after a Reno-type timeout

During the simulation, I notice that the goodput of some of the Vegas connections can be as low as one fifth of the maximum goodput even within the Vegas community. On a closer investigation into the ns traces, I discover that the unfairness is due to an incorrect measurement of the minimum round-trip time that occurs after a Reno-type timeout. The impact is a limitation in the growth of the congestion window size. Figure 3.33 above shows an example from one of the simulation runs. The phenomenon shows the incompatibility between the Vegas mechanisms and the Reno-type timeouts.

The example shows that TCP segments with sequence number up to 1059 has been transmitted. Then, at time 21.27894 seconds, a Reno-type timeout occurs. Before the occurrence of the timeout, the next send sequence number t_seqno_ is 1060. Due to a timeout, the next send sequence number is initially reset back to 1056 because the highest acknowledged sequence number is 1055. At the same time, the congestion window is reset to 2. Then 2 TCP segments 1056 and 1057 are sent. Therefore, the next send sequence number is increased to 1058 but is still less than the highest sequence number sent, i.e. 1059.

Problem then arises. When a new acknowledgement is received later at 21.38648 seconds for a segment v_begseq_ that is currently being timed by the Vegas fined-grained timer, TCP Vegas updates both the actual and expected sending rate and then re-transmits segment 1058. It also starts to time segment 1058. Note that under normal circumstances, TCP Vegas should start to time a new segment, not a re-transmitted one. At time 21.38830 seconds, an acknowledgement that acknowledges all previously sent segment, including the segment 1058, arrives. Upon receipt of an acknowledgement that acknowledged the segment being timed, TCP Vegas needs to update the minimum round-trip time, and then the actual and the expected sending rate. Segment 1058 is first transmitted at time 20.87894, i.e. 0.50936 seconds earlier. However, since TCP Vegas starts to time the retransmission of segment 1058 at 21.38648 seconds, it calculates the minimum round-trip time as 21.38830 seconds – 21.38648 seconds = 0.00182 seconds. Hence, although the actual round-trip propagation delay of the connection is 0.084552 seconds, TCP Vegas records a much lower minimum round-trip time measurement of 0.01822 seconds.

The sequence of variable update causes a large distortion in the measured minimum round-trip delay and the subsequent update of the sending rates. At time 21.59442 seconds, another acknowledgement for the segment (v_begseq_ = 1062) being timed by TCP Vegas arrives. Therefore, the actual sending rate is updated to 29.35838 packets per seconds and the expected rate is updated to 1646.853 packets per second. The congestion window is only about 3.6. However, the value of v_delta_ has already reached 2.946 packets. Therefore, although the size of the congestion window is only 3, TCP Vegas assumes that it already has about 3 extra packets in the network. The problem is caused by an under-estimated minimum round-trip time, causing TCP Vegas to assume a much lower bandwidth-delay product.

The ns trace of the congestion window shows that the size of the congestion window of the problematic connection never gets larger than 5 after the incident has occurred. Therefore, the unreasonably small value of the minimum round-trip time has resulted in a limitation in the growth of the congestion window.

Figures 3.34 and 3.35 below show the sending rate and the congestion window of the problematic connection and one of the other TCP Vegas connections in the same simulation run. The fair share of throughput should be 333,333 bits per second (72 packets per second). The average goodput of the problematic connection is only 131,904 bit per second (28.6 packets per second) while the average goodput of the other connection is 593,856 bits per second (128.875 packets per second). The other connection is able to achieve 4.5 times the goodput of the problematic connection. It can clearly be seen from the figures that both its actual sending rate and the size of the congestion window are limited.

In figure 3.34, which shows the actual sending rate against time, there are 2 exceptionally high sending rates. These 2 data points are caused by a wrong estimation of the round-trip time, as what I have just mentioned. At that time, TCP Vegas receives an acknowledgement for a single segment, the retransmitted segment that is being timed by TCP Vegas' congestion avoidance mechanism. Therefore, the estimated actual round-trip time becomes the time elapsed since the time the timed segment is re-transmitted, not the time the timed segment is first sent. The estimated actual round-trip time is also distorted to a very small value.

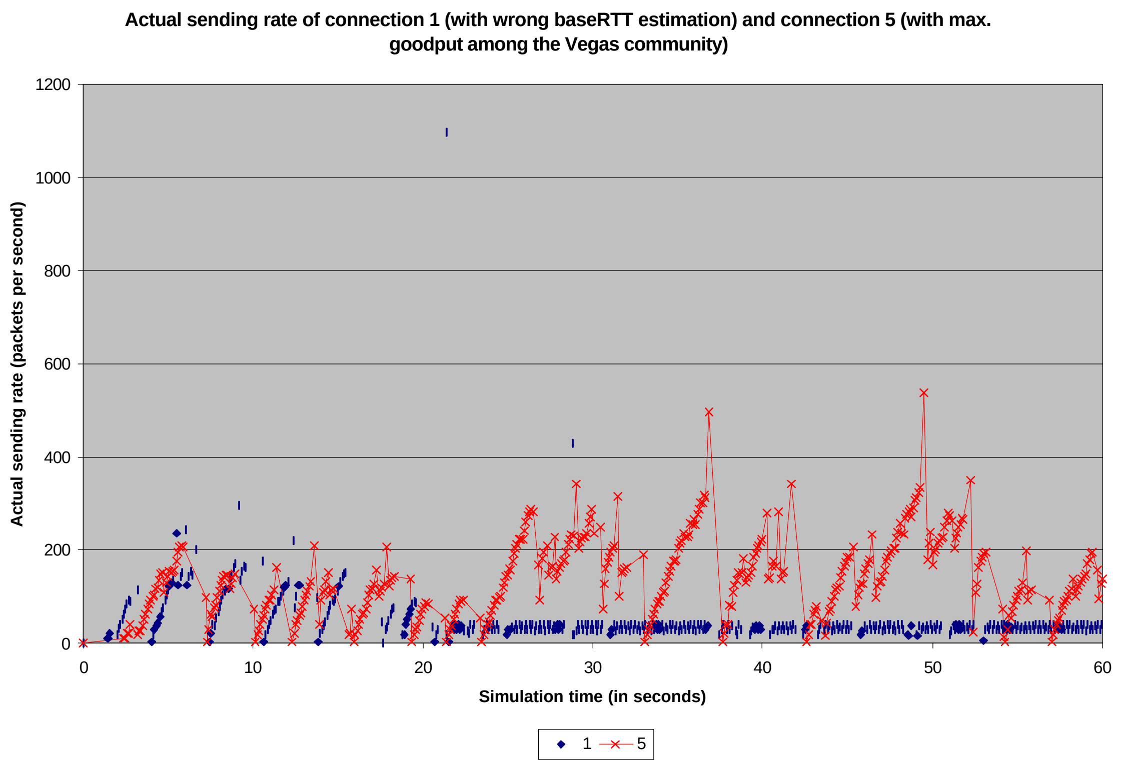


Figure 3.34 : Actual sending rate of connection 1 (with under-estimated min. RTT) & connection 5 (with maximum goodput within the Vegas community)

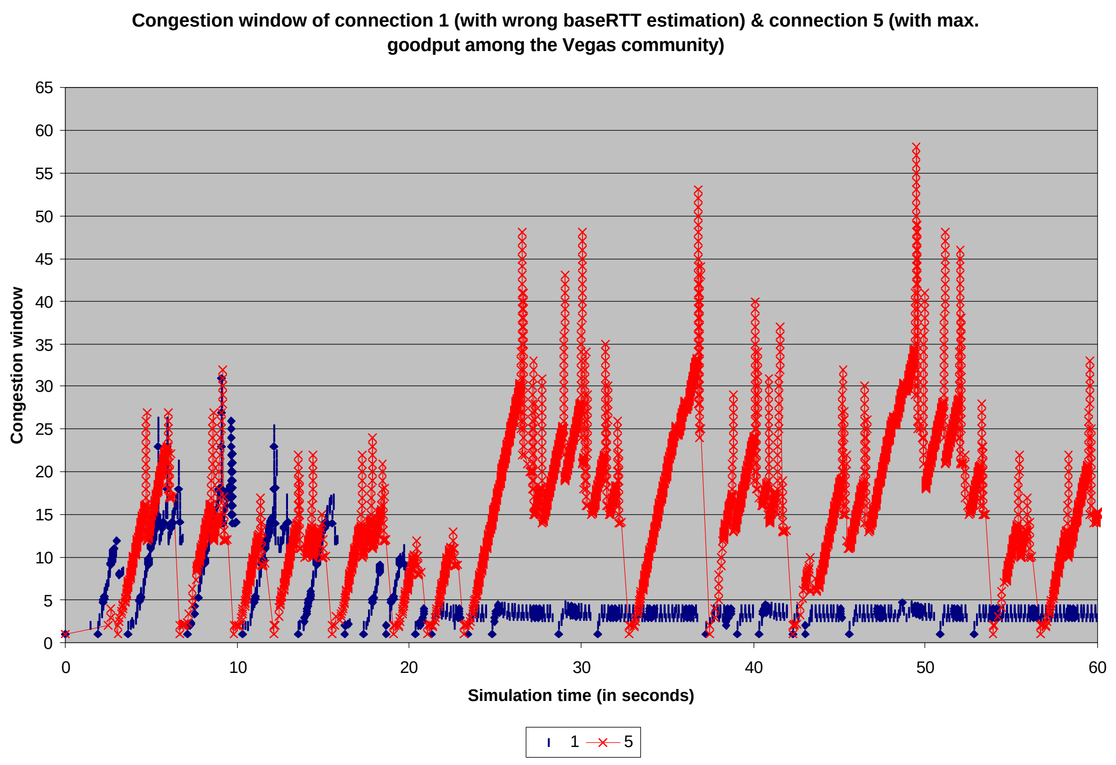


Figure 3.35 : Congestion window of connection 1 (with under-estimated min. RTT) & connection 5 (with maximum goodput within the Vegas community)

In comparison, the minimum goodput within the Reno community is 224,755 bits per second and the maximum goodput within the Reno community is 395,481 bits per second. Therefore, the fairness within the Reno community is better. The phenomenon demonstrates the incompatibility of the Reno-type timeout mechanism and the Vegas mechanisms.

Figure 3.36 gives an ns traces which shows how TCP Vegas eventually records a wrong minimum round-trip time.

Receipt of new ack at 20.87894 seconds

ack_	1055	
cwnd_	4.004	
v_sumRTT_	0.31703	
v_cntRTT	3	
maxseq_	**1058**	**new segment with sequence number 1058 sent**
t_seqno_	1059	next send sequence number 1059
maxseq_	**1059**	**new segment with sequence number 1059 sent**
t_seqno_	1060	next send sequence number 1060

Receipt of 1st duplicate ack at 20.97571 seconds

dupacks_	1	

Receipt of 2nd duplicate ack at 20.9755 seconds

dupacks_	2	

Timeouts at 21.27894 seconds

dupacks_	0	
t_seqno_	**1056**	
cwnd_	2	
nrexmitpack_	**71**	**retransmit 1056**
t_seqno_	1057	
nrexmitpack_	**72**	**retransmit 1057**
t_seqno_	1058	

Receipt of new ack at 21.38648 seconds

ack_	1056	Receipt of new ack for v_begseq_ 1056
v_sumRTT_	0	
v_cntRTT_	0	
v_baseRTT_	0.095726	
v_actual_	18.92563	(1058 – 1056) / (0.31703 / 3)
v_expect_	20.89300	(1058 – 1056) / 0.095726
v_delta_	0.188329	
v_begseq_	**1058**	
v_begtime_	**20.08923**	**(Start at 1.297254)**
cwnd_	2.5	
nrexmitpack_	73	re-transmit 1058
t_seqno_	1059	

Receipt of new ack at 21.38830 seconds

ack_	1059	
t_seqno_	1060	
v_rtt_	**0.001822**	
v_baseRTT_	**0.001822**	**21.38830 – 21.38648, first transmit at 20.87894s**
v_actual_	**1097.902**	**(1060 – 1058) / 0.001822**
v_expect_	**548.9511**	**(1060 – 1059) / 0.001822**
v_delta_	**-1.000188**	
v_begseq_	**1060**	
cwnd_	**2.9**	
v_sumRTT_	**0.509364**	**21.38830 – 20.87894**
v_cntRTT_	**1**	
maxseq_	1060	
t_seqno_	1061	
maxseq_	1061	
t_seqno_	1062	

Receipt of new ack 21.48934 seconds

ack_	**1060**	
v_sumRTT_	0	
v_cntRTT_	0	
v_actual_	**3.926463**	**(1062 – 1060) / 0.509364**
v_expect_	**1097.902**	**(1062 – 1060) / 0.001822**
v_delta_	**1.992847**	
v_begseq_	1062	
cwnd_	3.245	

v_sumRTT_	0.101033	
v_cntRTT_	1	
maxseq_	1062	
t_seqno_	1063	
maxseq_	1063	
t_seqno_	1064	
Receipt of new ack at 21.49164 seconds		
ack_	1061	
cwnd_	3.59	
v_sumRTT_	0.204371	
v_cntRTT_	2	
maxseq_	1064	
t_seqno_	1065	
Receipt of new ack at 21.59442 seconds		
ack_	1062	Receipt of ack for v_begseq
v_sumRTT_	0	
v_cntRTT_	0	
v_actual_	**29.35838**	**(1065 – 1062) / (0.204371 / 2) = (1065 – 1062) / 0.1021855**
v_expect_	**1646.853**	**(1065 – 1062) / 0.001822**
v_delta_	**2.946519**	
v_begseq_	1065	
cwnd_	3.868	
v_sumRTT_	0.105084	
v_cntRTT_	1	
maxseq_	1065	
t_seqno_	1066	

Figure 3.36 : Agent trace showing how TCP Vegas records an under-estimated minimum RTT

4.5 Impact of delayed acknowledgement strategy on fairness

4.5.1 Simulation results

In this section, I explore whether there is any impact on the fairness if the TCP receiver adopts the delayed acknowledgement strategy instead of the immediate acknowledgement strategy. If the receiver adopts the delayed acknowledgement strategy, then the minimum round-trip time (baseRTT_) and the actual round-trip time may be larger than the actual value. Therefore, it is beneficial to understand if there is any significant impact. Tables 3.26a and 3.26b summarize the average fairness index. Each average is obtained from 7 simulation runs.

RED-5-15 NO. OF ACTIVE FLOWS (α=β=3)	**10% Vegas**		**30% Vegas**		**50 % Vegas**		**70% Vegas**		**90% Vegas**		**100% Vegas**	
	IA	**DA**	**IA**	**DA**	**IA**	**DA**	**IA**	**DA**	**IA**	**DA**	**IA**	**DA**
30	0.974	0.958	0.941	0.950	0.933	0.940	0.907	0.950	0.917	0.957	0.900	0.967
100	0.958	0.936	0.936	0.904	0.925	0.908	0.913	0.930	0.912	0.949	0.911	0.960
200	0.912	0.925	0.891	0.894	0.881	0.894	0.891	0.907	0.900	0.914	0.904	0.931

Table 3.26a : Average fairness vs. receiver ACK strategy, using RED-5-15 router
IA=Immediate Acknowledgement, DA=Delayed Acknowledgement

RED-54-108 NO. OF ACTIVE FLOWS (α=β=3)	**10% Vegas**		**30% Vegas**		**50 % Vegas**		**70% Vegas**		**90% Vegas**		**100% Vegas**	
	IA	**DA**	**IA**	**DA**	**IA**	**DA**	**IA**	**DA**	**IA**	**DA**	**IA**	**DA**
30	0.981	0.977	0.976	0.985	0.979	0.986	0.972	0.984	0.980	0.985	0.971	0.970
100	0.960	0.959	0.947	0.944	0.939	0.941	0.929	0.945	0.926	0.962	0.930	0.971
200	0.929	0.936	0.912	0.904	0.903	0.911	0.899	0.915	0.909	0.931	0.918	0.939

Table 3.26b : Average fairness vs. receiver ACK strategy, using RED-54-108 router
IA=Immediate Acknowledgement, DA=Delayed Acknowledgement

To explore the impact of the receiver acknowledgement strategy on the fairness of TCP Vegas, I focus on the case when all the connections run Vegas-3. Interestingly, from table 3.26a and 3.26b, the fairness is better when the receiver adopts the delayed acknowledgement strategy instead of the immediate acknowledgement strategy. In addition, the difference seems to be larger when the RED thresholds are low. When the RED thresholds are high and the number of active TCP Vegas flows is only 30, there seems to be no significant difference.

One possible explanation is that, when the receiver adopts the delayed acknowledgement strategy, the round-trip times tend to be over-estimated. Therefore, the probability of an under-estimated minimum round-trip time reduces, resulting in lower variance of goodputs within the Vegas community. Consequently, the fairness is also better.

4.5.2 Impact on the estimation of the minimum RTT

The following table shows the recorded minimum round-trip times and the goodput of 10 of the 15 connections in one of the simulation runs, in which 50% of the connections run TCP Vegas. It can be seen that while the initial value of the recorded minimum round-trip time is large, TCP Vegas eventually detects a value that is much closer to the true propagation delay. Therefore, although the receiver adopts the delay acknowledgement strategy, the value of the estimated minimum round-trip time is not adversely affected.

Connection number	**1st recorded baseRTT**	**last recorded baseRTT**	**Propagation delay**	**Difference**	**Goodput**
8	0.599058	0.100396	0.09561	0.00479	335116.80
1	0.601500	0.095294	0.09184	0.00345	339264.00
5	0.597561	0.097561	0.09411	0.00345	341222.40
9	0.597671	0.097671	0.09422	0.00345	345024.00
6	0.601617	0.101617	0.09817	0.00345	347212.80
7	0.601262	0.101262	0.09781	0.00345	347212.80
3	0.603339	0.103339	0.09989	0.00345	354355.20
0	0.595219	0.095219	0.09177	0.00345	355968.00
4	0.602864	0.102864	0.09941	0.00345	358387.20
2	0.630239	0.112818	0.09105	0.02177	485107.20

Table 3.27 : Recorded minimum RTT when receivers use delayed ACK strategy

Figure 3.37 shows the agent trace of one of the TCP Vegas connections. It can be seen that TCP Vegas sends 1 segment and waits for the return of its acknowledgement. However, since the receiver adopts the delayed acknowledgement strategy, it must wait until the delay timer expires before it can transmit the acknowledgement for the 1st segment. Therefore, the first record of the minimum round-trip time is estimated to be in the range of 0.6 seconds (the delay timer is set to expire in 0.5 seconds). However, eventually, the congestion window increases to 2 and thus 2 back-to-back TCP segments are transmitted. Hence, the acknowledgement is transmitted nearly immediately after the first of these 2 back-to-back segments has reached the receiver. Therefore, the next record of the minimum round-trip time is much closer to the true minimum round-trip time. In this configuration, it seems that the delayed acknowledgement strategy does not have much impact on the estimation of the minimum round-trip time. In fact, table 3.27 shows that 8 of the 10 connections measure the minimum round-trip time accurately. The transmission delay of the data segment and the acknowledgement is 0.00345 seconds.

To further explore the impact of the receiver acknowledgement strategy on fairness, I next run 2 more sets of simulations, using the simple network model described in the section 1 of this chapter. In the first simulation, both receivers adopt the delayed acknowledgement strategy. In the second simulation, one of the receivers adopts the delayed acknowledgement strategy while the other adopts the immediate acknowledgement strategy. The propagation delays of both connections are the same and are set to 6ms (i.e. x1 = x2 = 1ms).

In the first simulation, there is unfairness in bandwidth allocation. In the second simulation, the fairness is better as the goodput of both connections are comparable. The unfairness is caused by an over-estimation of the minimum round-trip time. In the first simulation, the minimum round-trip time of the connection that sends to a receiver that adopts the immediate acknowledgment strategy is correct. On the other hand, the minimum round-trip time of the connection that sends to a receiver that adopts the delayed acknowledgement strategy is much over-estimated. Consequently, the goodput of this connection is much higher. When both connections send to receivers that adopt the delayed acknowledgement strategy, the fairness is much better. However, the minimum round-trip times of both connections are still over-estimated. Fortunately, their difference is so small that the bandwidth is still fairly shared.

	Average goodput	1st estimation of the min. RTT	Final estimation of the min. RTT
1st connection : IA	750094.69 bps		0.013211 seconds
2nd connection : IA	749934.68 bps		0.013211 seconds
1st connection : DA	993133.51 bps	0.518544 seconds	0.032000 seconds
2nd connection : IA	507366.90 bps	0.013211 seconds	0.013211 seconds
1st connection : DA	749921.68 bps	0.518544 seconds	0.021333 seconds
2nd connection : DA	749921.68 bps	0.513211 seconds	0.018544 seconds

Table 3.28 : Impact of delayed acknowledgement strategy on goodput & minimum RTT

Based on the simulation results, I tend to believe that the delayed acknowledgement strategy delays the proper record of the minimum round-trip time by at least 2 RTTs. In my simulation setting, the first estimation of the minimum round-trip time and actual round-trip are over-estimated if the receiver adopts the delayed acknowledgement strategy. Thereafter, TCP Vegas is able to detect a better minimum RTT that is much closer to the true value. However, it is hard to tell whether TCP Vegas is able to detect the true minimum round-trip time. The interaction of different network topologies and configurations cannot be ignored because they, as a whole, can affect whether the TCP sender is able to detect the correct minimum round-trip time.

4.5.3 *Impact on the calculation of queue backlog*

By definition, the expected sending rate is always higher than the actual sending rate. During the analysis, I discover that the value of v_delta_ can be negative, implying that the expected sending rate is less than the actual sending rate. This is caused by the delayed acknowledgement strategy. Again, I use an example to illustrate the phenomenon. The agent trace in figure 3.37 shows how this can happen.

```
1.35390  8  0  43 0  maxseq_ 0
1.35390  8  0  43 0  t_seqno_ 1

1.95295  8  0  43 0  v_baseRTT_ 0.599057633638
1.95295  8  0  43 0  ack_ 0
1.95295  8  0  43 0  v_actual_ 1.669288468835
1.95295  8  0  43 0  v_expect_ 1.669288468835
1.95295  8  0  43 0  v_inc_flag_ 0
1.95295  8  0  43 0  v_begseq_ 1
1.95295  8  0  43 0  v_begtime_ 0.599058
1.95295  8  0  43 0  maxseq_ 1
1.95295  8  0  43 0  t_seqno_ 2

2.55453  8  0  43 0  ack_ 1
2.55453  8  0  43 0  v_actual_ 1.662310638531
2.55453  8  0  43 0  v_delta_ 0.004180
2.55453  8  0  43 0  v_inc_flag_ 1
2.55453  8  0  43 0  v_begseq_ 2
2.55453  8  0  43 0  v_begtime_ 1.200630
2.55453  8  0  43 0  cwnd_ 2.000
2.55453  8  0  43 0  v_sumRTT_ 0.601572279465
2.55453  8  0  43 0  v_cntRTT_ 1
2.55453  8  0  43 0  maxseq_ 2
2.55453  8  0  43 0  t_seqno_ 3
2.55453  8  0  43 0  maxseq_ 3
2.55453  8  0  43 0  t_seqno_ 4

2.65497  8  0  43 0  ack_ 3
2.65497  8  0  43 0  v_sumRTT_ 0.000000000000
2.65497  8  0  43 0  v_cntRTT_ 0
2.65497  8  0  43 0  v_actual_ 3.324621277061
2.65497  8  0  43 0  v_delta_ -0.991640
2.65497  8  0  43 0  v_inc_flag_ 0
2.65497  8  0  43 0  v_begseq_ 4
2.65497  8  0  43 0  v_begtime_ 1.301070
2.65497  8  0  43 0  v_sumRTT_ 0.100440033638
2.65497  8  0  43 0  v_cntRTT_ 1
2.65497  8  0  43 0  v_baseRTT_ 0.100440033638
2.65497  8  0  43 0  maxseq_ 4
2.65497  8  0  43 0  t_seqno_ 5
2.65497  8  0  43 0  maxseq_ 5
2.65497  8  0  43 0  t_seqno_ 6
```

Figure 3.37 : Agent trace showing negative v_delta_

From figure 3.37, it can be observed that, at time 2.65497 seconds, the value of v_delta_ is negative, implying that the expected sending rate is lower than the actual sending rate. In the ns simulator, TCP Vegas calculates the expected and the actual sending rate using the following formulae :

Expected sending rate = (t_seqno_ - last_ack_) / v_baseRTT_

Actual sending rate = (t_seqno_ - v_begseq_) / average rtt_

According to the above traces, the sending rates are thus :

Expected sending rate = (4 – 3) / v_baseRTT_ = 1 / v_baseRTT_

Actual sending rate = (4 – 2) / v_rtt_ = 2 / v_rtt_

Next, I analyze the Arizona code and find that the same problem can occur. According to the Arizona code, the expected sending rate is calculated using the following formula :

```
pred_rate = (tp->snd_nxt - tp->snd_una + MIN(tp->t_maxseg - acked, 0))*10
                                    / tp->v_cong_detect_baseRTT;
```

where

pred_rate	=	expected sending rate = v_expect_ in ns
snd_nxt	=	next send sequence number = t_seqno_ in ns
snd_una	=	next expected acknowledgement sequence number immediately after the last ACK has been received
t_maxseq	=	1
acked	=	number of segments acknowledged by this ACK

v_cong_detect_baseRTT = v_baseRTT_ in ns

If this formula is used, the expected sending rate will be :

pred_rate = (4 – 2 + MIN((1 – 2), 0)) / v_baseRTT_ or 1 / v_baseRTT_

However, if I remove the 2nd term in the numerator in the Arizona's formula, the equation becomes :

```
pred_rate = (tp->snd_nxt - tp->snd_una) *10 / tp->v_cong_detect_baseRtt;
```

Then, the expected sending rate will be correct :

pred_rate = (4 – 2) / v_baseRTT_ or 2 / v_baseRTT_

In ns, to correct the problem, the formula can be changed to :

Expected sending rate = (t_seqno_ - (last_ack_ - acked + 1)) / v_baseRTT_

where acked = last_ack_ - oldack

or

Expected sending rate = (4 – (3 – 2 + 1)) / v_baseRTT_

Chapter 4

CONCLUSION AND FUTURE WORK

1. Conclusion

The purpose of this project is to address 4 specific issues of the fairness of TCP Vegas : 1) Is TCP Vegas really fair to connections with larger propagation delays, 2) what is the impact of the thresholds, α and β, used in TCP Vegas' congestion avoidance algorithm on fairness, 3) how fair is TCP Vegas when there are many active flows, and 4) does the receiver acknowledgement strategy, namely the immediate and the delayed acknowledgement strategy, which affects the calculation of the round-trip times, also affects the fairness of TCP Vegas ?

My observations on the fairness index obtained from the simulation data yield the conclusion that, when $\alpha = 1$ and $\beta = 3$, TCP Vegas is unfair to connections with larger propagation delays, unless the propagation delays of the 2 connections are close. In a more general context, I believe this applies to cases in which $\alpha < \beta$ and only TCP Vegas is present. While previous literature has already pointed out that unfairness may arise when $\alpha < \beta$, my conclusion additionally suggests that the bias is in favor of connections with shorter propagation delay. Unlike TCP Reno, the bias does not necessarily increase as the difference in propagation delays between the connections increases. However, I also notice that the fairness of TCP Vegas can be worse than that of TCP Reno so that an improvement in the fairness over that of TCP Reno is not a guarantee. In my simulation setting, the problem of unfairness is resolved with Vegas-2 or Vegas-3.

My assessment of the impact of the Vegas thresholds on fairness yields the conclusion that fairness is stable and good when $\alpha = \beta = 2$ or 3. When $\alpha = \beta = 1$, fairness is unstable and may even be worse that when $\alpha = 1$ and $\beta = 3$. Also, my simulation results support previous researcher's observations that the stability of TCP Vegas is better when the values of α and β are larger. Of course, there is a tradeoff : when the available bandwidth per connection is low, larger thresholds will cause TCP Vegas to be more aggressive, adding additional load to the network and thus making congestion a more likely event. Considering a balance among stability, fairness and aggressiveness, $\alpha = \beta = 3$ seems to be an acceptably good choice should a single value be required.

While analyzing the fairness of many TCP flows, I discover that some TCP Vegas connections are shut down. As one reason, it is caused by an implementation of the ns Reno-type timeout subroutine, which wrongly set the variable that stores the un-inflated window to zero, causing TCP Vegas to eventually reach a deadlock status. As a second reason, it is caused by the part of the ns code that incorrectly updates the start transmission time of the TCP connection to the re-transmission time of the first segment. Consequently, the fairness measure is very poor. This demonstrates how differences in the TCP implementation can cause substantial differences in TCP performance.

After amending the ns code to eliminate the causes of deadlock, I discover from the simulation results that, when there are many active flows passing through the bottleneck link, the fairness of the TCP Reno host-only configuration is better than that of the TCP Vegas host-only configuration. An exception occurs when the RED thresholds are high, the number of flows is low and Vegas-1 is used. This is the only setting with which the number of flows times δ (i.e. 1) is less than the RED minimum threshold. Intuitively, this suggests that the fairness of TCP Vegas is better when there is few or no timeouts.

Moreover, I observe from the ns agent traces that TCP Vegas may perform unnecessary retransmission after a Reno-type timeout. One possible solution is to change the TCP Vegas code so that it will check the next send sequence number against the sequence number of the expired segment sent. If the next send sequence number is less than or equal to the sequence number of the expired segment that has just been sent, TCP Vegas should increment the next send sequence number to the sequence number of the expired segment that has already re-transmitted plus 1.

With regard to the impact of the RED thresholds on fairness when there are many active flows, fairness improves when the RED thresholds are higher. In addition, the fairness measure is more stable when $\alpha = \beta = 3$ than when $\alpha = 1$ and $\beta = 3$.

The simulation results also suggest that, when there is a mixture of TCP Vegas and TCP Reno connections, the RED thresholds are high and the number of active flows is low, TCP Reno is the winner. However, when the number of active flows is large, TCP Vegas is the winner. Intuitively, this suggests that, if the RED router thresholds are large enough to support a large enough congestion window, TCP Reno is able to benefit from an unfair share of the bottleneck bandwidth.

While the difference in bandwidth allocation between TCP Vegas and TCP Reno contributes to unfairness, I discover that the variance in goodputs within the Vegas community is also larger than that within the Reno community. Therefore, while the difference in goodputs between TCP Reno and TCP Vegas contributes to unfairness, the variance in goodputs within the TCP Vegas community also contributes significantly to the unfairness. On a closer investigation into the ns traces, I then discover that an incorrect measurement of the minimum round-trip time that occurred after a Reno-type timeout contributes to the unfairness. In such case, the minimum round-trip time under-estimated. The impact is a limitation in the growth of the congestion window. The phenomenon suggests possible incompatibility between the Vegas mechanisms and the Reno-type timeouts.

My assessment of the receiver acknowledgement strategy on fairness yields the conclusion that fairness does not deteriorates when all the receivers adopts the delayed acknowledgement strategy. In particular, I observe that TCP Vegas does not behave detrimentally although the receiver employs the delayed acknowledgement strategy. In general, the first estimation of the minimum round-trip time is always over-estimated. However, since 2 back-to-back segments are eventually transmitted when the congestion window increases to 2, the receiver sends the acknowledgement nearly immediately after the first of these 2 segments has been received. Therefore, TCP Vegas is eventually able to detect a minimum round-trip time that is much closer to the true value. However, in contrary to the definition, I also observe that the measured expected sending rate can be less than the measured actual sending rate. The problem is a direct result of the delayed acknowledgement strategy because the receiver sends an acknowledgement for every 2 segments received. The problem can be resolved by removing the 2^{nd} term in the formula that is used to calculate the expected sending rate.

2. Future work

In summary, I observe that the fairness of Vegas-3 is consistently good. Moreover, setting δ to 3 is acceptably good after considering a balance among the criteria of fairness, stability and aggressiveness. Therefore, I suggest Vegas-3 be used as the baseline version of TCP Vegas in future work.

Besides, I suggest that future work be pursued to improve the protocol implementation of Enhanced TCP Vegas and, in general, TCP Vegas, especially if it is eventually to be deployed in the Internet. This is very important because different implementation can cause very different performance. For example, I have observed that the original ns implementation may cause TCP Vegas connections to be shut down. After amending the ns code to correct the problem, the fairness of TCP Vegas improves substantially.

More importantly, there are areas for improvement in the implementation of TCP Vegas. Firstly, I observe that TCP Vegas may perform unnecessary re-transmissions after a Reno-type timeout. In particular, this is not caused by a pre-mature timeouts but by a reset of the next send sequence number. To improve the protocol implementation, there may be a need to evaluate its impact and to eliminate the unnecessary retransmission in the real-world protocol implementation.

Secondly, I identify that the minimum round-trip time may be under-estimated after a Reno-type timeout, which I believe, after a review of the original Arizona code, is a problem in the original TCP Vegas implementation as well. I expect an effective correction will substantially improve fairness because it shall reduce the variability of goodputs within the Vegas community. Besides, a number of factors can cause an incorrect measurement of the minimum round-trip time, either over-estimated or under-estimated, and thus give rise to unfairness problem. To improve fairness, I think it would be beneficial to identify how the estimation of the minimum round-trip time can be improved.

Thirdly, I notice that the estimated queue backlog may be negative when the receiver adopts the delayed acknowledgement strategy. This is in contradiction to the definition, i.e. the estimated queue backlog should be positive because the expected sending rate is larger than the actual sending rate. Accordingly, future work may be required to further evaluate the appropriateness of the Vegas formulae, especially in scenarios that have not been studied before.

In fact, the first and second issues I have just mentioned are related to the Reno-type timeout subroutine. This points to possible incompatibility between the Vegas algorithms and the Reno-type timeout subroutine. In particular, I notice that 2 sets of timer are maintained for the purpose of retransmission, including the original Reno retransmission timer and the new Vegas finer-grain timer. To save processing overhead and improve the protocol implementation, it would be beneficial to combine the 2 sets of timer, potentially using the timestamp option.

In addition, I think future work should be pursued to further investigate the performance of Enhanced TCP Vegas when there are many flows, when the router buffer is insufficient or, in general, in situations when there are many timeouts. Normally, only a few flows are present in past analysis. However, the behavior of TCP Vegas can be very different when there are many flows. In particular, I observe that the minimum round-trip times of Enhanced TCP Vegas may be under-estimated when there are many flows and timeouts. The problem needs to be corrected because an improvement in fairness and performance is expected. After the correction of the problem, the fairness and the performance of Enhanced TCP Vegas in situation where there are many active flows can be re-visited.

REFERENCE

[ADLY95] Ahn, J. S., Danzig, P. B., Liu, Z., and Yan, L. "Evaluation of TCP Vegas: emulation and experiment". *ACM SIGCOMM Computer Communication Review*, October 1995, 25(4), 185-195.

[B89] Braden, R. "RFC 1122: Requirements for internet hosts – communication layers". October 1989.

[BB00] Boutremans, Catherine. and Boudec, Jean-Yves Le Boudec. "A Note on the Fairness of TCP Vegas". In *Proceedings of International Zurich Seminar on Broadband Communications*, 2000, 163-170.

[BP] Brakmo, L. TCP Vegas. Available via ftp://ftp.cs.arizona.edu/xkernel/new-protocols/Vegas.Tar.Z.

[BP95] Brakmo, L.S. and Peterson, L.L. "TCP Vegas: end to end congestion avoidance on a global Internet". *IEEE Journal on Selected Areas in Communications*, October 1995, 13(8), 1465-1480.

[CC00] Chen, J. R. and Chen, Y. C. "Vegas Plus: improving the service fairness". *IEEE Communication Letters*, May 2000, 4(5), 176-178.

[CJ89] Chiu, D. M. and Jain R. "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks". *Computer Networks and ISDN Systems*, 1989, No.17, 1-14.

[DLY94] Danzig, P. B., Liu Z. and Yan, L. "An evaluation of TCP Vegas by live emulation". *Technical Report 94-588*, Los Angeles: Computer Science Department, USC, 1994.

[F91] Floyd, S. "Connections with multiple congested gateways in packet-switched networks, Part 1: one-way traffic". *ACM Computer Communication Review*, August 1991, 21(5), 30-47.

[FJ91] Floyd, S. and Jacobson, V. "Traffic phase effects in packet-switched gateways". *Computer Communication Review*, April 1991, 21(2), 26–42.

[FJ93] Floyd, S. and Jacobson, V. “Random early detection gateways for congestion avoidance”. *IEEE/ACM Transactions on Networking*, August 1993, 1(4), 397-413.

[HBG00] Hengartner, U., Bolliger, J. and Gross, Th. “TCP Vegas revisited”. In *Proceedings of IEEE INFOCOM 2000*, March 2000, 1546-1555. Also available via http://www.cs.cmu.edu/~uhengart.

[HMM99] Hasegawa, G., Murata, M., and Miyahara, H. “Fairness and stability of congestion control mechanisms of TCP”. In *Proceedings of IEEE INFOCOM ’99*, March 1999, Vol. 3, 1329-1336.

[J88] Jacobson, V. “Congestion avoidance and control”. In *Proceedings of SIGCOMM ’88*, August 1988, 314-329.

[J91] Jain, R. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Canada : John Wiley & Sons, Inc., 1991.

[LM97] Lakshman, T. V. and Madhow, U. “The performance of TCP/IP for networks with high bandwidth-delay products and random loss”. *IEEE/ACM Transactions on Networking*, June 1997, 5(3), 336-350.

[NS] NS network simulator, Available via http://www~mash.cs.berkeley.edu/ns.

[NS00] Fall, K. and Varadhan, K. *The ns Manual*. UC Berkeley, 2000. Available via http://www.isi.edu/nsnam/ns/ns-documentation.html.

[M97] Morris, Robert. “TCP behavior with many flows”. In *Proceedings of IEEE International Conference on Network Protocols*, October 1997, 205 – 211.

[MLAW99] Mo, J., La, R. J., Anantharam, V. and Walrand, J. “Analysis and comparison of TCP Reno and Vegas”. In *Proceedings of IEEE INFOCOM ‘99*, June 1999, 1556 - 1563.

[PD91] Peterson, Larry L., and Davie, Bruce S. *Computer Networks : A System Approach*. Canada : John Wiley & Sons, Inc., 1991.

[S94a] Stevens, W. R. *TCP/IP Illustrated, Volume 1: The Protocols*. Massachusetts : Addison-Wesley, 1994.

[S94b] Stevens, W. R. and Wright, G. R. *TCP/IP Illustrated, Volume II, The Implementation.* Massachusetts : Addison-Wesley, 1994.

[S97] Stevens, R. “RFC 2001: TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms”. Jan 1997.

[RK99] Raghavendra, Aditya M. and Kinicki, Robert E. “A simulation performance study of TCP Vegas and Random Early Detection”. In *Proceedings of IEEE International Computing and Communications Conference on Performance, 1999*, 169-176.