IST2005
INTERNATIONAL SYMPOSIUM ON TELECOMMUNICATIONS
September 10-12, 2005 Shiraz, Iran

A Fuzzy Based TCP Congestion Controller

Adel Nadjaran Tousi, Mohammad Hossien Yaghmaee
Computer Dept., Faculty of Engineering, Ferdowsi University of Mashhad
Institute for Studies in Theoretical Physics and Mathematics (I.P.M)
Emails: ad_na85@stu-mail.um.ac.ir, hyaghmae@ferdowsi.um.ac.ir

Abstract

One of the most important problems in current computer networks is the congestion control. Computer networks have experienced an explosive growth over the past few years and with the growth have come severe congestion problems. The Transmission Control Protocol (TCP) was proposed and implemented to prevent the congestion collapses. TCP has been the most popular and widespread protocol since the arrival of the Internet. In this paper we propose a fuzzy technique inspired by TCP Vegas for better adjustment of congestion window in sight of better throughput and reducing packet loss.

Keywords: Congestion, Congestion Window, Fuzzy Vegas, Fuzzy TCP, TCP, Fuzzy, Vegas.

Introduction

During the past few years, it has gone through several phases of improvement, and many new features such as fast retransmit and fast recovery have been added. Over the years, several different flavors of TCP have been introduced, such as TCP Tahoe [1], TCP Reno [2], TCP New Reno [3] and TCP Vegas [4].

This paper attempts to go beyond this earlier work; to provide some new insights into congestion control, and to propose modifications to the implementation of TCP.

The main results reported in this paper show better throughput compared with all pervious different flavors of TCP and more efficient use of the available bandwidth by reducing packet loss and packet retransmission.

The main objective of current study is to use fuzzy logic capabilities to dynamically tune the congestion window. Note that in our proposed approach, the algorithms for the sending side have only been treated.

This paper is organized in seven sections. The first section outlines TCP Variants except TCP Vegas. Section 2 then describes the techniques employed by TCP Vegas coupled with the insights that led us to the devised technique, the next section, provides the basis for our technique and explains fuzzy based congestion window controller, Section 4 provides some simulations scenarios and evaluation of our approach and shows the results of the conducted experiments in NS2. Finally, sections 5 and 6 make some concluding remarks and issue future fields of work.

1. TCP Variants

1.1. TCP Tahoe

Tahoe is distributed as a standard with 4.3BSD UNIX; hence one of the most widely adopted versions of TCP. In addition, this is the original implementation of Van Jacobson's proposed mechanisms [1], a follow up of the original TCP that was standardized in RFC 793. Tahoe includes three main features: Slow Start, Congestion Avoidance, and Fast Retransmit.

Under Slow Start, the connection starts out with congestion window size, cwnd, at cwnd=1 (packet).Thereafter, the cwnd is incremented for every ACK received indicating the receipt of a new packet. This results in an exponential increase for the cwnd size, doubling every round trip time (RTT). We abbreviate the Slow Start threshold as ssthresh, and perform Slow Start operation while cwnd <ssthresh.

The sender defines the window size W as the minimum of (cwnd, RMSS), where RMSS is the receiver's maximum segment size. During Congestion Avoidance (cwnd = ssthresh), we assume that packet losses imply network congestion. Tahoe deals with such situations in the following manner:

1. If loss occurs when cwnd=W, then the network can presumably handle somewhere between 0.5W and W packets.

2. ssthresh is adjusted to ssthresh=0.5W, and the state of TCP is reverted to Slow Start where cwnd = 1.

3. upon the receipt of a new ACK with cwnd>ssthresh, then we increase cwnd by cwnd+=1/cwnd – resulting in an additive increase

Fast Retransmit saves Tahoe from resetting on timeouts due to lost packets. Duplicate ACKs (repeated ACKs with the same sequence number, dupACKs) indicate a possible packet loss, and Tahoe reverts to the Slow Start stage. It begins retransmission immediately upon receiving three or more dupACKs instead of waiting for timeout before realizing packet losses.

1.2. TCP Reno

Reno came about in 1990, and includes all the mechanisms previously implemented in Tahoe. However, Reno extends the congestion control scheme by coming with the addition of Fast Recovery, as well as delayed ACKs and header prediction (common-case code in-lined)[2].

Like Tahoe, the Congestion Avoidance state starts when cwnd=ssthresh. As cwnd exceeds ssthresh, cwnd is increased by just 1/cwnd for every ACK received, growing linearly. Fast retransmit works similarly in that a retransmission is initiated upon the receipt of three dupACKs. What is different from Tahoe is when Reno transitions to Fast Recovery. When the third dupACK is received, we adjust ssthresh=0.5cwnd and cwnd=ssthresh+3 (since three ACKs, even dupACKs, indicate that the receiver has absorbed three additional packets after the first lost packet). Reno then works with cwnd, incrementing it as each additional dupACK is received. As Reno finally receives a proper ACK of the retransmitted packet, we reset cwnd to cwnd=ssthresh again (half of its value prior to Fast Recovery).

In short, each of the dupACKs notifies the TCP sender that a single packet has cleared the network (hence generating an ACK response). Fast Recovery temporarily inflates cwnd during the recovery of the lost segment, while sending new packets with each subsequent dupACK to maintain selfclocking.

When the lost packet is recovered, it again deflates cwnd=cwnd/2. As a result, Reno uses Fast Recovery to smoothly transition to Congestion Avoidance. One glaring disadvantage of Reno is the way it handles multiple packet losses within a data window W. Reno will initiate the Fast Recovery procedure multiple times due to the multiple losses, hence affecting the cwnd and ssthresh values continuously. In the end, this often leads to timeouts, resulting in a cwnd reset (cwnd=1) and diminishing throughputs. Additionally, Reno does not handle burst traffic (such as Pareto sources) very well – under its ACK framework, the protocol retransmits just one lost packet per RTT.

1.3. TCP New Reno

New Reno intelligently improves on the mishaps of Reno, in particular the Slow Start and Fast Recovery aspects. It adapts more gradually to a new window, and also addresses multiple losses in one window gracefully [3].

Before we proceed with the explanation of New Reno, let us first define partial ACKs, or parACKs.

A partial ACK is an ACK for some but not all of the packets that were outstanding at the start of the Fast Recovery period. ParACKs do not transition New Reno out of Fast Recovery. Instead, the parACKs received during Fast Recovery are treated as an indication that the packets immediately following the ACKed packets are lost and should be retransmitted. Thus, when multiple packets are lost, New Reno can recover without a retransmission timeout. New Reno shines when multiple packet losses are experienced during a given data window W. During Fast Recovery, parACKs greatly reduces the possibility of timeouts, since the running count is reset periodically (for each and every ACK – termed slow-but-steady; or for the first in a series of ACKs only – dubbed impatient variant).

At next section we explain TCP Vegas as a basis of our approach and motivation of A Fuzzy Based TCP Congestion Controller.

2. TCP Vegas

TCP Vegas was first introduced by Brakmo et al. in [4].It primarily enhances the Congestion Avoidance and Fast Retransmission algorithms of TCP Reno [2]. There are several changes made in TCP Vegas.

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

Congestion avoidance mechanism: TCP Vegas does not continually increase the congestion window during congestion avoidance. Instead, it tries to detect

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

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

The congestion avoidance mechanism that TCP Vegas uses is quite different from that of TCP Tahoe or Reno. TCP Reno uses the loss of packets as a signal that there is congestion in the network and has no way of detecting any incipient congestion before packet losses occur. Thus, TCP Reno reacts to congestion rather than attempts to prevent the congestion. TCP Vegas uses the difference between the estimated throughput and the measured throughput as a way of estimating the congestion state of the network.

TCP Vegas sets BaseRTT to the smallest measured Round Trip Time (RTT), and the expected throughput is computed according to following equation:

$$Expected = \frac{cwnd}{BaseRTT} \quad (1)$$

Where *cwnd* is the current window size. With each packet being sent, TCP Vegas records the sending time of the packet by checking the system clock and computes the round trip time by computing the elapsed time before the ACK comes back. It then computes Actual throughput using this estimated RTT according to following equation:

$$Actual = \frac{cwnd}{RTT} \quad (2)$$

Then, TCP Vegas compares Actual to Expected and computes the difference *Delta* as below:

$$delta = Actual - Expected \quad (3)$$

The *Delta* is used to adjust the window size. To achieve this, TCP Vegas defines two threshold values, α, β (α<β). If *Delta* < α, the window size is increased linearly during the next RTT. If *Delta* > β, then TCP Vegas decreases the window size linearly during the next RTT. Otherwise, it leaves the window size unchanged.

As mentioned above, the TCP Vegas tunes the congestion windows linearly. The main problems of TCP Vegas is that it only compare the *Delta* with two fixed thresholds and it does not consider the distance between *Delta* and two thresholds α,β. The main objective of current study is to use fuzzy logic capabilities to dynamically tune the congestion window. To do this, if the distance between *Delta* and α,β is very low, low, high or very high , we can change the window size very low, low, high or very high, respectively.

Another Fuzzy approach that focused on congestion control using fuzzy control for end-to-end TCP has been proposed in [5]. It uses an on-line adaptive fuzzy system at the source to find the best possible weighted combination among the available window allocation policies.

3. Fuzzy based TCP congestion controller

In the pervious sections we introduces the linear adjustment of congestion window in Vegas and explained that we can use the distance of delta from α or β as a factor for more accurate adjustment of congestion window(cwnd) in TCP. This is a motivation for using fuzzy controller to adjust cwnd.

We propose a fuzzy based TCP congestion controller using a single-input single-output fuzzy controller. The input parameter to the controller is *delta*. The output of fuzzy controller, *Adjust*, is used to tune the value of congestion window accurately. The input and output Fuzzy sets of the linguistic variables are shown in figure 1 and figure 2, respectively.

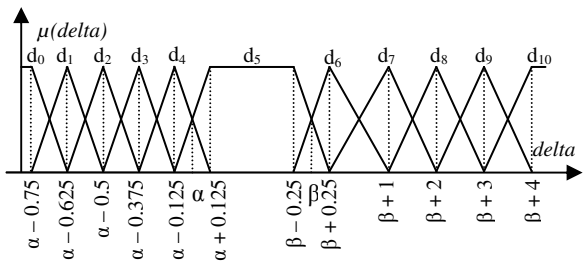

Fig. 1 The fuzzy sets of *delta*

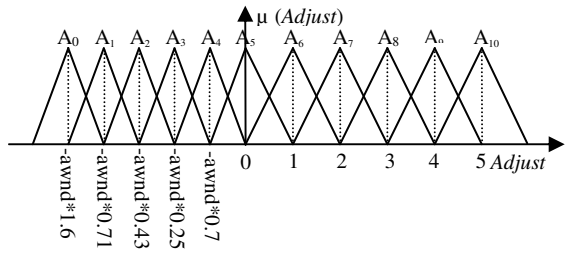

Fig.2 The fuzzy sets of *Adjust*

In figure 2, the awnd parameter represents the average of the send window size. The output of fuzzy controller (*Adjust*) is added to the current window size and the new window size is calculated. The proposed

fuzzy controller uses the following rules in its rule base:

Rule 0: if delta is d_0 then Adjust A_{10}

Rule 1: if delta is d_1 then Adjust A_9

Rule 2: if delta is d_2 then Adjust A_8

Rule 3: if delta is d_3 then Adjust A_7

Rule 4: if delta is d_4 then Adjust A_6

Rule 5: if delta is d_5 then Adjust A_5

Rule 6: if delta is d_6 then Adjust A_4

Rule 7: if delta is d_7 then Adjust A_3

Rule 8: if delta is d_8 then Adjust A_2

Rule 9: if delta is d_9 then Adjust A_1

Rule 10: if delta is d_{10} then Adjust A_0

Using the single tone fuzzifier, product inference engine and center of average defuzzifier, the final output of fuzzy controller is calculated as below:

$$Adjust = f(delta) = \frac{\sum_{l=0}^{10} \overline{A}^{l} . \mu_{d^l}(delta)}{\sum_{l=0}^{10} \mu_{d^l}(delta)} \qquad (4)$$

Where l is the rule number and $\overline{A}^{l}$ is the center of fuzzy set used in the Then part of l th rule.

4. Performance Evaluation

To evaluate the performance of the proposed fuzzy controller, we used the NS2 simulator [6]. The proposed model which is called Fuzzy Vegas was implemented in the NS2 simulator.

Figure 3, shows the structure of first simulated network.

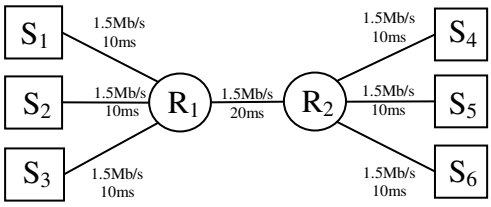

Fig. 3 The Simulated Network

In the simulation scenario, the connection between (S_1, S4) is TCP and for (S_2, S_5), (S_3, S_6) the UDP protocol is used. The S_1, S_2 and S_3 transmit data and the S_4, S_5, S_6 receive the data packets. All queues in the nodes are based on Drop Tail strategy.

An FTP traffic source is attached to node S_1 and Constant Bit Rate (CBR) traffic with 1.5Mb/s sending rate attached to S_2, S_3. S_2, S_3 transmit data at t=5s and are stopped respectively at t=10s and t=30s. S_3 restarts to send packets at t=50s and it is stopped after 10s, S_3 restarts to send at t=60s and stops at t=70s the same as S_2. The node S_1 starts at t=0s and is stopped at the end of the simulation.

In this scenario S1 is once simulated with TCP Vegas agent and another time with the Fuzzy Vegas. The network throughput for connection between S_1 and S_4 is plotted versus the simulation time in figure 4. In figure 5, the variation of *cwnd* is plotted versus the simulation time for both of them. As it can be seen, the proposed Fuzzy Vegas can tune the *cwnd* in a much better form than TCP Vegas does. In table 1, for both TCP Vegas and Fuzzy Vegas, the number of duplicate ACK packets received by the sender has been shown.

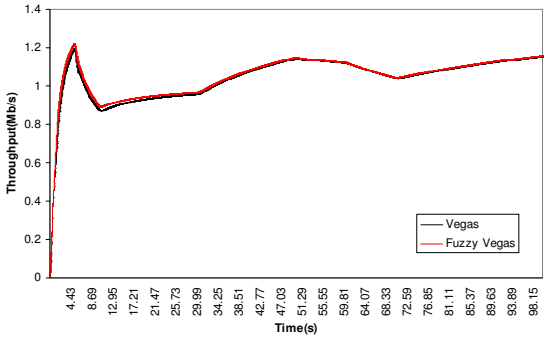

Fig. 4 Throughput versus simulation time

Drop tail strategy at R_1

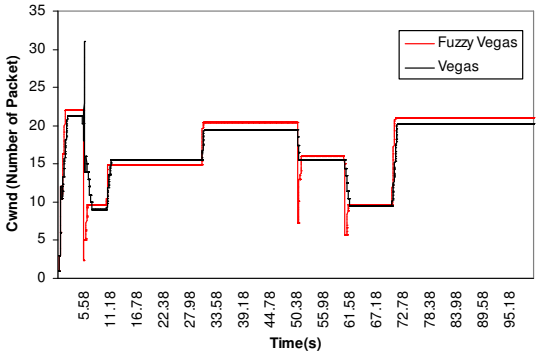

Fig. 5 The *cwnd* versus simulation time

Drop tail strategy at R_1

Based on the results shown in figures 4, 5 and table 1, it can easily be seen that the proposed Fuzzy Vegas has a better performance than the traditional TCP Vegas.

Table 1 Total Number of Duplicate Acknowledge Received by the Sender

	Fuzzy Vegas	Vegas
Drop Tail	36	282
RED	2153	2217

We repeated the above scenario on the simulated network shown in figure 3, with queue based on RED strategy for R_1. Results of this experiment includes throughput for connection between S_1 and S_2, cwnd adjustment and duplicate ACK packet received with both TCP Vegas and Fuzzy Vegas present in figure 6, 7 and table 1.

Based on the results shown in figures 4-7 and table 1, it can be easily seen that the proposed Fuzzy Vegas has a better performance compared with the traditional TCP Vegas.

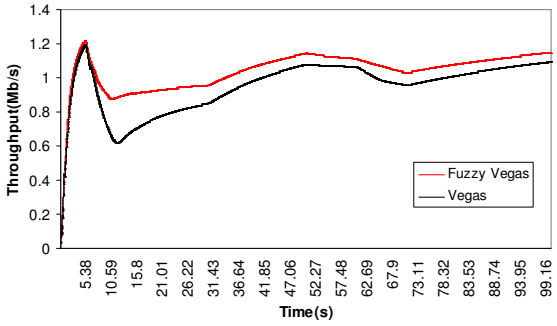

Fig. 6 Throughput versus simulation time

RED strategy at R_1

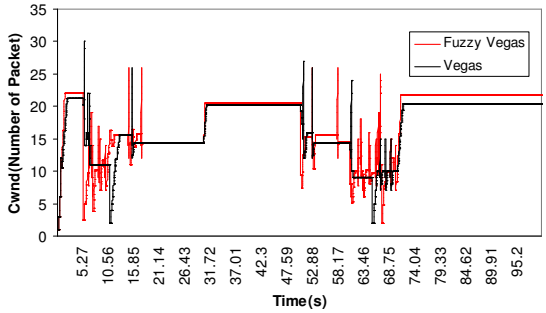

Fig. 7 The *cwnd* versus simulation time

RED strategy at R_1

The Structure of the second simulation is similar to first simulation with some changes at connection bandwidth and delays, and also the length of the scenario. The structure of the second simulation network is shown in figure 8.

All queues in the nodes are based on Drop Tail strategy.

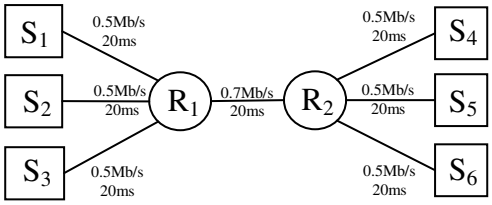

Fig. 8 The Simulated Network

S2 and S3 are again UDP source and S5, S6 are sinks respectively. The Connection between S1 and S4 is TCP. S2 and S3 transmit data at constant bit rate of 0.5Mb/s, S2 starts sending packets at t=0s and doesn't stop before the end of the simulation. S3 starts at t=15 and stops at t=25. S1 is a source of FTP traffic, it starts sending packets at t=0s and continues till the end.

S1 was attached to the TCP variants presented in section 1 and the throughput of the TCP connection between S1 and S2 was measured each time. The measured throughput is plotted versus time in figure 9.

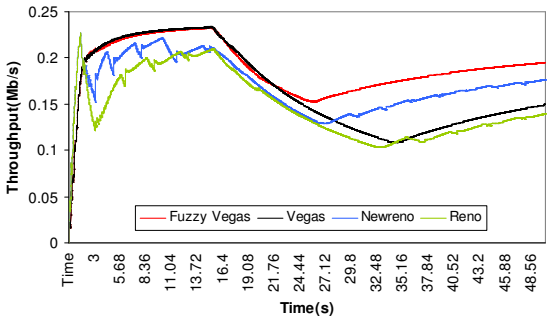

Fig. 9 Throughput versus simulation time

Table 2 shows the total duplicates acknowledge received by sender.

Table 2 Total Number of Duplicate Acknowledge Received by Sender

	Duplicate Acknowledge
TCP Reno	5128
TCP New Reno	12117
TCP Vegas	4032
TCP Fuzzy Vegas	322

The Final simulation was conducted to evaluate the influence of increasing the number of nodes on the simulated network on the performance of the proposed model. Here we increase the total number of nodes in the simulated network to 20.

Figure 10 shows the details of the simulated network. All queues in the nodes are based on Drop Tail strategy again.

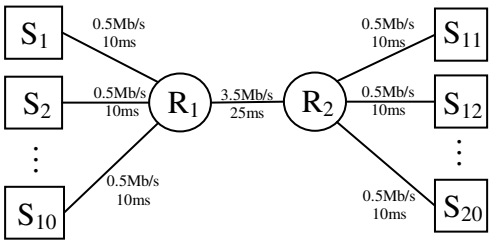

Fig. 10 The Simulated Network

S_1 and S_{11} are TCP connections and connection of other pairs of nodes are UDP, S_1 starts sending FTP traffic at t=30s and stops at t=150s, others start sending packets at constant bit rate at t=5s, t=10s … t=45s respectively, and stop accordingly at t=85s, t=90s, t=125s.

The throughput for connection between S_1 and S_4 is plotted versus simulation time in figure 11.

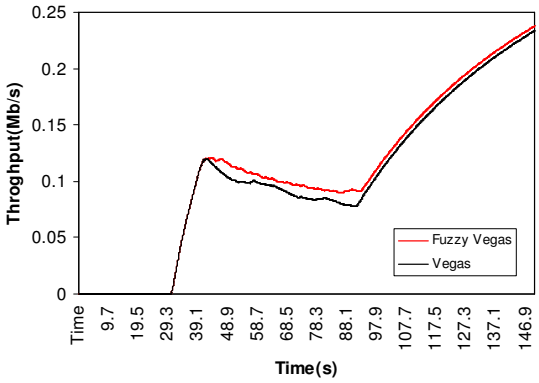

Fig. 11 The Throughput versus simulation time

5. Future Works

As mentioned in section 3 and 4, Vegas and our proposed approaches use static thresholds (α, β).

Our future work will focus on dynamically adjusting these thresholds with fuzzy controller to achieve better performance, also we will continue to study on more fuzzy variables for the proposed fuzzy based TCP congestion controller in this paper.

More accurate evaluating can be performed for the proposed fuzzy based TCP congestion controller at other simulation situations or real network test beds for better challenges.

Conclusions

Fuzzy congestion window controller achieved better performance by integrating a fuzzy controller and nonlinear adjusting end to end congestion windows in TCP.

Our experiments and simulations show better throughput and lower packet loss and more sensitivity of congestion window to the parameters of network.

Acknowledgment

The authors would like to thank the anonymous reviewers and their insightful comments and suggestions helped to make this a much better paper in particular by Ebrahim Bagheri.

Also we appreciate the helpful suggestions and comments of our dear colleague Mohsen Amini.

References

[1] V. Jacobson, “Congestion avoidance and control” In Proceedings of SIGCOMM ’88

[2] L. S. Brakmo, S. W. O’Malley and L. Peterson, “TCP Vegas: End to End Congestion Avoidance on a Global Internet,” IEEE Journal on selected areas in communication, Vol. 13, No. 8, October 1995

[3] M. Allman, V. Paxson and W. and Stevens, “TCP Congestion Control.” Request for Comments (Standards Track) RFC 2581, Internet Engineering Task Force, April 1999

[4] S. Floyd, and T. Henderson., “The New Reno Modification to TCP's Fast Recovery Algorithm.” Request for Comments (Experimental) RFC 2582, Internet Engineering Task Force, April 1999.

[5] P.Carbonell, Z. P. Jiang, S. S. Panwar, “Fuzzy TCP: A Preliminary Study”, Proceedings Of the 15th IFAC World Congress (IFAC 2002), Barcelona, Spain, July 21-26, 2002.

[6] NS-2 Network simulator http://www.isi.edu/nsnam/ns/