AdaVegas: Adaptive Control for TCP Vegas

Amir Maor Yishay Mansour

***Abstract*— In this paper we introduce AdaVegas, an adaptive congestion control mechanism based on TCP Vegas. TCP Vegas has several parameters which control the way it increases the sending rate. While TCP Vegas holds theses parameters constant, AdaVegas sets these values dynamically. In this way AdaVegas is able to change its increment strategy dynamically and better adapt to the current environment. Using simulations we both evaluate AdaVegas and compare it to TCP Vegas. Our simulations show that AdaVegas achieves significantly better results than TCP Vegas with a fairly low overhead.**

I. Introduction

The Internet is practically everywhere today, and its Transmission Control Protocol (TCP) is the most widely used for reliable data transmission. A critical design aspect of TCP is its flow control that allows the protocol to adjust the end to end communication rate to the available bandwidth at the bottleneck link. This challenge has become even more critical as the Internet grew to a mass collection of heterogeneous networks and gateways.

TCP is a window based end to end protocol, and its flow control is based on adjusting the window size, given the feedback collected from the network. The initial versions of TCP tried to detect an ”optimal” window size when establishing the connection, focusing mainly on the receiver's buffer limitations. Setting the window size only during the session initialization has a fundamental drawback, since the traffic load fluctuates over the life of the session.

A major breakthrough was achieved when Jacobson [1] introduced a TCP flow control mechanism that allows the end stations to adjust their window size dynamically. The main idea is to use packet loss as an implicit notification that the sending rate is too high. The sender continuously increases the sending rate, and upon a detection of a packet loss it sharply decreases the window size. The loss event is recognized implicitly when either a timeout occurs or a certain number of duplicate acknowledgments are received. More specifically, the mechanism is composed of two main parts: The ”Slow Start”, where the sender expands its window at an exponential rate, and “congestion avoidance”, where it increases the window linearly every round trip time, and upon a packet loss halves the window size. This general scheme falls in the category of Additive Increase Multiplicative Decrease (AIMD) which have been shown in general to guarantee fairness [2]. The move from slow start to congestion avoidance occurs when the window size exceeds a certain threshold. (For a detailed description see [1].).

Department of Computer Science, Tel-Aviv University, email amaor@math.tau.ac.il

Department of Computer Science, Tel-Aviv University, email mansour@math.tau.ac.il

TCP Vegas [3] extends the mechanisms of TCP Reno. In TCP Vegas the user tries to estimate the end to end delay and it responds to changes in the end to end delay. Such a mechanism has the potential to detect congestion earlier, and hopefully both avoid the cost associated with packet loss and achieve an earlier response.

This work extends the adaptiveness of TCP beyond that of TCP Vegas and TCP Reno. The basic idea is to have the history (namely, the previous network responses) guide our window size changes, which can be interpreted as dynamically setting the parameters of the AIMD flow control mechanism. In this work we focus on the benefits of dynamically changing the additive increase parameters.We show that making these parameters history dependent, even in a very weak way, makes the users react faster and better to changing environments, and therefore improves the overall performance. As a motivating example consider the case, when the available bandwidth increases substantially, the users should use a more aggressive increment strategy. Using successful transmission as feedback, our algorithm decides whether to switch to a more aggressive strategy.

Although we focus on TCP Vegas and specifically on the increment parameter, we believe that our concept may be applied both to other parameters in TCP Vegas and to other mechanisms, most notably TCP Reno and others. A broader view of this research is to try and incorporate current machine learning techniques in the TCP flow control in particular and network protocols in general. It is for this reason that we regard this work not only as a presentation of an improved algorithm of TCP Vegas, but also as a step in incorporating learning algorithms in network protocols. In our case, one can cast the flow control problem in decision theoretic terms, however, as we discuss in Section II, setting a “good” optimization criteria may be tricky.

Related work: Yang and Lam [4] studied AIMD flow control and derived the relationship between the parameters so that the flow would be TCP friendly [5]. Bansal and Balakrishnan [6] evaluate a class of non linear congestion control algorithms. Both suggest different update parameters for the flow control, however, the parameters do not change over time. In contrast, we suggest that the algorithm modify the parameters dynamically.

The rest of this paper is organized as follows: In Section II we give an overview of our results, comparing AdaVegas with current TCP Vegas. In Section III we briefly describe the relevant components of TCP Vegas and describe in detail the new mechanisms of AdaVegas. In Section IV we describe our simulation model for testing AdaVegas. Section V describes our results in details. Section VI considers the performance of AdaVegas in a heterogeneous environment, both a case of different

round trip time and a case of a combination of AdaVegas and TCP Vegas users are considered. We conclude with a summary of some ideas for future work in Section VII.

II. Results Overview

To gain intuition let us start by defining two different variants of TCP Vegas that use two different values for the increase parameter. The first, Vegas_1, is identical to TCP Vegas with the default increase rate, the second, Vegas_8, has an increase rate which is 8 times faster than that of Vegas_1 [1]. First we demonstrate that the two are incomparable, i.e., in some scenarios Vegas_1 outperforms Vegas_8 while in others Vegas_8 outperforms Vegas_1. This motivates our design of AdaVegas, that dynamically sets the increase parameters, and therefore has the potential to match the performance of the better of the two.

Our first set of scenarios has initially only two users that share one bottleneck link. At some point of time one user stops sending packets, which implies that the other user's fair share is then doubled. We study how fast the remaining user increases its window size to capture the available bandwidth. Since we want to make things as simple as possible in this first set of scenarios, we define a large queue size (180 packets, packet size is 1500 bytes) which implies that no packets are dropped.

We consider two sets of scenarios, in one the bottleneck link has relatively high bandwidth, while in the other it has a relatively low bandwidth. (The results of our simulations are shown in Figure 1.)

When the fair share is high (the total bandwidth is either 4Mb

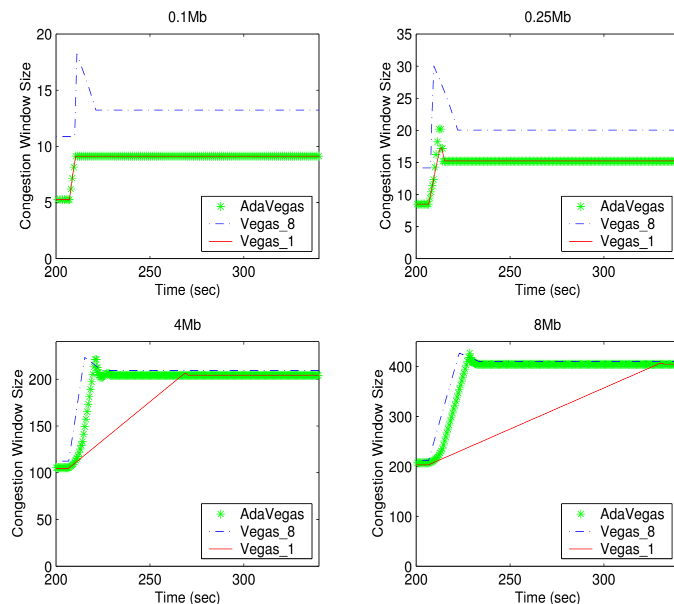

Fig. 1. The changes in the window size of the three mechanisms Vegas_1, Vegas_8 and AdaVegas in the case where one of two users leaves the bottleneck link.

or 8Mb) Vegas_1 converges very slowly to the appropriate window size, while Vegas_8 converges much faster. Since Vegas_8 converges much faster than Vegas_1, it also utilizes much better the available bandwidth and achieve higher throughput. When the fair share is low (0.1Mb and 0.25Mb) Vegas_8 significantly overshoots the window size, and take longer to converge. However, the slow convergence time of Vegas_8 is not the main problem in this case. In this scenario we used a large queue size, therefore the overshoot did not result in dropped packets. In more realistic scenarios, this overshoot would result in packet loss, which in turn would cause lower throughput for Vegas_8. (We demonstrate this in the more complex and realistic scenarios described later.)

The above discussion shows that there is no clear winner, and every scenario has different preferable increase values. This motivates our design of AdaVegas which sets these values dynamically, depending on the specific scenario.

One can view this flow control as a learning problem. The agent (AdaVegas) takes an action (increasing the window size by a certain constant or decrementing the window by a certain factor). The agent receives a feedback (whether packets have been lost or how many acknowledgments were received) and selects an appropriate action. The decision to perform a certain action may prove wrong (for example, a packet may be dropped). However, the agent "decides", on the basis of prior successes and failures, which action to perform. This is an almost classical decision theoretical model, with a stochastic environment.

Based on this view, we designed a Markov Decision Process (MDP) [7] for the flow control problem. The MDP had to chose between the different increment actions. The reward for each action was the throughput achieved in the next round trip time and the goal was to maximize the total user throughput. We hoped that in any given situation the users will converge to the actions that maximize their throughput. Because our reward was the agent's **own** throughput and not the total throughput, the simulations became a live demonstration of Game Theory famous prisoner's dilemma. Since every user wanted to maximize its own throughput, each user chose the most aggressive action, which of course, caused the total throughput of the system to deteriorate.

Keeping this in mind we then continued working on our design which resulted in AdaVegas. AdaVegas changes its increment strategy to a more aggressive one as it discovers more available bandwidth. A detailed description of the mechanism will be described in Section III, for now we settle for a brief overview. Since there is no direct knowledge of the available bandwidth AdaVegas needs to use its history, and records the number round trip times it has incremented the window consecutively successfully. Whenever this number of consecutive successful increments goes over a predefined threshold the increase parameters are changed. The idea is that if the increment was successful a few times in a row it might be the case that there is enough available bandwidth, and it is worthwhile to estimate that there is enough bandwidth to move to a more aggressive increase strategy. There are of course several level of "aggressiveness". We designed the mechanism for this paper such that the most moderate increment will correspond to Vegas_1 and the most aggressive to Vegas_8. (We also made a slight change in the decrement strategy, which will also be

[1] The constant 8 was chosen to demonstrate the advantage of using a fast increment rate rather than slow increment rate, however, other constants could have also been used.

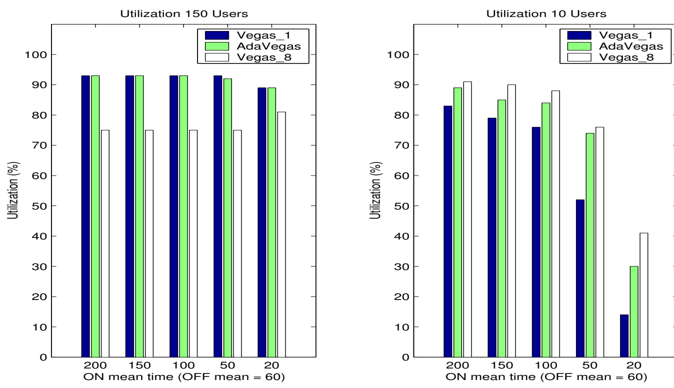

Fig. 2. Link Utilization comparison of Vegas_1, Vegas_8 and AdaVegas on scenarios with 150 Users and 10 Users

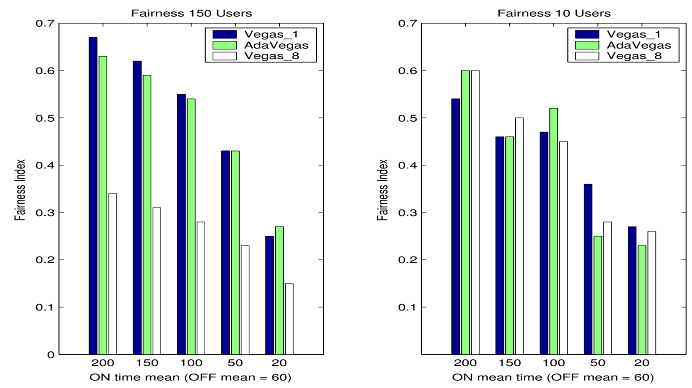

Fig. 4. Fairness index comparison of Vegas_1, Vegas_8 and AdaVegas on scenarios with 150 users and 10 users.

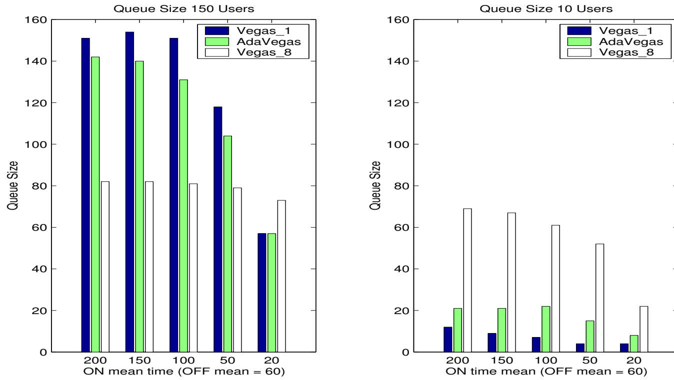

Fig. 3. Queue size comparison of Vegas_1, Vegas_8 and AdaVegas on scenarios with 150 Users and 10 Users

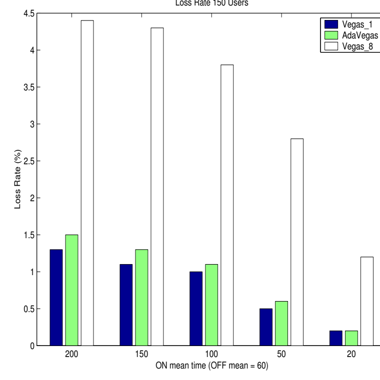

Fig. 5. Loss rate at bottleneck link for 150 users. Each triplet shows loss rate for the three mechanism Vegas_1, AdaVegas and Vegas_8. The loss rate for 10 users was negligible and therefore not presented.

detailed in Section III.)

First, we simulated AdaVegas on the two users scenarios described before, and the results are shown in Figure 1. For a low bandwidth bottleneck, AdaVegas behavior is similar to that of Vegas_1, and for a high bandwidth bottleneck, AdaVegas behavior is similar to that of Vegas_8. We can see how AdaVegas changes from moderate increment (Vegas_1) to an aggressive increment (Vegas_8) in a very smooth way. Our simulations show how AdaVegas is able to adapt to the environment in which it is operating. When the available bandwidth is high, it increases in a high rate, and when the available bandwidth is low, it uses a moderate increment rate.

The two user scenarios are rather synthetic, and were given to demonstrate the principles of AdaVegas behavior. Now that the basic motivation for AdaVegas is understood, we move to more "realistic" scenarios, and demonstrate that also in these situations our basic claims still hold. Namely, AdaVegas will match the better of Vegas_1 and Vegas_8. (A detailed description of the simulation and results will be given in Sections V and VI).

We set up a two sets of scenarios, in one 10 users are sharing a bottleneck link, and in the other 150 users are sharing the same bottleneck link. Each user alternates between ON periods, in which it tries to send packets greedily, and OFF periods, in which it is not active. We use a heavy tailed distribution for both the ON period and the OFF period. To evaluate the results we consider line utilization, queue size, packet loss rate and fairness index.

Figure 2 shows the link utilization. When 150 users share the line, Vegas_1 outperforms Vegas_8, and AdaVegas utilization is almost identical to that of Vegas_1 (and significantly better than Vegas_8). When 10 users are sharing the line, Vegas_8 outperforms Vegas_1 and the gap widens as the ON mean time is shorter. In this case AdaVegas' utilization is between that of Vegas_1 and Vegas_8 (and slightly closer to that of Vegas_8).(More details are given in Section V.)

Figure 3 shows the queue size. When 150 users share the line, Vegas_1 and AdaVegas are similar, however AdaVegas is slightly lower. When 10 users share the line, Vegas_8 queues are significantly larger than those of both Vegas_1 and AdaVegas, and AdaVegas queues are slightly larger than those of Vegas_1. (The reason for which Vegas_8 has such large queues is related to some of the parameters setting, and is explained and discussed in Sections IV and V.)

Figure 4 shows the fairness index which is defined as

$$FairnessIndex \equiv \frac{(\sum_{i=1}^{n} W_i)^2}{n * \sum_{i=1}^{n} W_i^2}$$

where W_i is the window size of user i (out of n users). When 150 users share the line, Vegas_1 is more fair than Vegas_8, and AdaVegas is close to Vegas_1. When 10 users share the

bottleneck, the differences between Vegas_1 and Vegas_8 are smaller.

The loss rate for 150 users is given in Figure 5. As expected the packet loss rate of Vegas_8 is the highest and in general the loss rate increases with the duration of the ON period. For the case of 10 users the loss rate of all three is very low.

The above simulations show that AdaVegas is able to approach the behavior of the better suited mechanism. In cases in which Vegas_1 reaches better utilization, AdaVegas is closer to it, and when Vegas_8 is better, AdaVegas is closer to it. This is the main benefit of using AdaVegas with its "adaptive mechanism". Although AdaVegas does not always perform as well as the best mechanism, its performance is very similar to that of the best mechanism.

III. Adaptive Vegas Algorithm

A. TCP Vegas: Quick Overview

In this section we overview the parts in TCP Vegas which are related to this paper. In both this subsection and subsection III-B we refer for simplicity to window size, transmission rate and queue sizes in terms of segments. (For a detailed description of TCP Vegas see [3].)

The main idea behind TCP Vegas congestion avoidance and Slow Start is that a connection continuously tries to estimate the amount of *extra* data it has queued at the bottleneck link. This means the amount of data that would not have been sent, if the sending rate was equal to the connection bandwidth, and therefore is queued. Every round trip time (RTT) TCP Vegas picks a distinguished segment and computes its round trip time. This is done by recording the segment sending time, and recording the time its acknowledgment is received, the difference is called `sampled_RTT`. `Base_RTT` is defined as the round trip time of a segment when the line is not congested, namely, TCP Vegas sets `Base_RTT` to the minimum value of all the `sampled_RTT`. Every RTT, using `Base_RTT`, the `sampled_RTT` and congestion window size, TCP Vegas estimates the expected and actual throughput.

The *Expected throughput* is the sending rate that would have been achieved by using the current congestion window size, if the line had not been congested. The expected throughput is set to $\frac{congestion_window_size}{\texttt{Base_RTT}}$. TCP Vegas keeps track of the amount of data sent during the time between the distinguished packet was sent and its acknowledgment received. The actual throughput is then computed by dividing this amount by the `sampled_RTT`. This way, every RTT,TCP Vegas has an evaluation of the `Expected` and `Actual` throughput. Given this information TCP Vegas considers the difference $\Delta =$ `Expected-Actual`, which is its estimation of the extra data sent. During congestion avoidance TCP Vegas compares every RTT the value of Δ to two predefined thresholds α and β. If $\Delta < \alpha$ then TCP Vegas concludes that there is a light load and therefore increases the congestion window size by one MSS (Maximum Segment Size) during the next round trip time. If $\Delta > \beta$ then TCP Vegas concludes that the line is congested and decreases the congestion window size by one MSS during the next round trip time. If $\alpha < \Delta < \beta$ TCP Vegas concludes that the amount of extra data is reasonable, and does not change the congestion window size. If a packet is considered dropped the window is cut in half, regardless of the value of Δ.

B. Our algorithm: AdaVegas

Every RTT AdaVegas, like TCP Vegas, computes Δ, and decides which action to take according to α, β and Δ. Our crucial control parameter in AdaVegas is inc which is set in TCP Vegas to be 1 (namely one MSS). Like TCP Vegas, AdaVegas increases the congestion window size when $\Delta < \alpha$. However, instead of using an increment of $1 \times MSS$, AdaVegas uses $inc \times MSS$, where inc is set dynamically in the following way:

1) A set of predefined values for inc, α and β are defined. Each set corresponds to a different increment strategy (from moderate to aggressive).
2) AdaVegas remembers the number of successive RTT in which $\Delta < \alpha$, and refers to this value as $succ$.
3) A number of ranges of $succ$ are defined, each of which corresponds to one of the increment strategies. The larger $succ$, the more aggressive the corresponding strategy.

Whenever $\Delta < \alpha$ AdaVegas updates $succ$ and then sets α, β and inc to the corresponding values. The parameters we used are given in Table I.

	α	β	inc
$0 \leq succ \leq 3$	1	3	1
$4 \leq succ \leq 7$	2	4	2
$4 \leq succ \leq 15$	2	4	4
$16 \leq succ$	4	8	8

TABLE I

AdaVegas dynamic values for α, β and inc and the corresponding range for $succ$.

As can be seen from Table I we double the increment rate when switching from one strategy to the next. Theoretically, doubling the increment rate results in logarithm convergence time, in contrast to TCP Vegas which converges linearly. However, in our simulations we limit the increment rate to $8 \times MSS$ per RTT, which allows us to exhibit a faster convergence time, but not a logarithmic one.

Setting α and β in TCP Vegas influences both the throughput and the fairness. We therefore slightly modify α and β according to the increment rate. This enables smooth increment of the sending window as seen in Figure 1, while maintaining fairness as can be seen in Figure 4.

Our basic methodology has been to leave other parts of TCP Vegas algorithm unchanged. However, since in high increment rates AdaVegas may overshoot, we slightly modified the decrement algorithm to better cope with such situations. When increasing the congestion window size, AdaVegas remembers the increase rate used for a certain period. This way, when congestion is detected, TCP Vegas decreases the window size according to the last increment rate used. More specifically, whenever AdaVegas expands the window (i.e., $\Delta \leq \alpha$) , it sets $lastInc \leftarrow inc$. After two RTTs of steady state (i.e., $\alpha < \Delta < \beta$) AdaVegas resets $lastInc$ ($lastInc \leftarrow$

1). When congestion is detected (i.e., $\Delta \geq \beta$) AdaVegas sets $lastInc \leftarrow max(lastInc/2, 1)$ and decreases the window by $lastInc$. The logic behind this is that we assume that if congestion occurred after incrementing the window, it is probably because of an overshoot. We therefore estimate that the right window size is between $congestionWindow$ and $congestionWindow - lastInc$, and we move to the average of the two sizes. In our simulations we observed that this decrease mechanism is especially crucial for high increment rates.

We would like to emphasize that when a packet is dropped AdaVegas cuts the window in half, similar to TCP Vegas and TCP reno.

IV. Simulation Model

We consider the following simulation model. A number of different users, denoted by N, are connected through a single bottleneck link, each to a different destination. The generic topology is shown in Figure 6, with the default delay and bandwidth values. Each source and destination is connected to a router through a fast link of 100Mb/sec and 1msec delay. The routers are using a drop tail queue of size 180 segments. The bottleneck link has a delay of 300msec and bandwidth of 20Mb/sec. Naturally, congestion occurs at router 1.

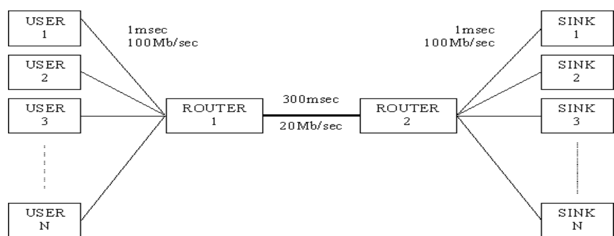

Fig. 6. Topology used in our simulations

We use ON/OFF users with heavy tailed distribution for both the ON periods and OFF periods. During the ON period each user tries to send as many packets as possible. At the end of an ON period, the user stops sending packet and starts an OFF period that ends in a new ON period. The duration of both the ON and OFF periods are heavy tailed distributions (specifically Pareto distribution with parameter 1.5). In different scenarios we use different means for the heavy tailed distribution used by the users, and we explicitly state them.

The simulations where done using the software of ns-2 Network Simulator [8].

Evaluation Criteria

Our analysis focused on the following important parameters:

1) Line Utilization:
   We define the line utilization as the amount of **successfully** transmitted data (*goodput*), divided by the product of the link bandwidth and simulation time which clearly gives an upper bound on the throughput. Formally, $Utilization \equiv$

$$\frac{(number\ of\ acked\ packets) \times (packet\ size)}{(link\ bandwidth) \times (time)}$$

2) Queue Size:
   We monitor the queue size at the bottleneck link, which is shared by all the connections.
3) Loss rate:
   We look at the queue on the bottleneck link and compute the ratio between the number of segments dropped and the number of segments that arrived at the queue (namely segments either transmitted or dropped).
4) Fairness:
   Our aim is to test whether our improved performance comes at the cost of lower fairness. Since all the RTT are identical, it is enough to consider the window sizes. In a perfectly fair situation, all active user should have exactly the same window size. At the other extreme a single user overtakes the bottleneck link's bandwidth while all other active users experience starvation (very small window size). We use the following formula for the fairness index [2], [9]:

$$FairnessIndex \equiv \frac{(\sum_{i=1}^{n} W_i)^2}{n \times \sum_{i=1}^{n} W_i^2}$$

   Where W_i is the window size of the i-th active user out of n active users. Note that the fairness index is bound from below by $1/n$ and from above by 1. The fairness index equals to 1 when all the users have the exact same window size.

Both fairness and queue size are sampled at least once a second throughout the simulations which are about 1000sec long.

V. Simulation Results

A. Simulation Description

We use two sets of scenarios, which differ only in the number of users. In the first set we have 150 users while in the second we have 10 users. The idea is to test AdaVegas in different scenarios, once with high fair share per user and once with low fair share per user. In the simulations we use the simulation model described in Section IV.

Each set of simulations (150 users and 10 users) consists of 5 scenarios which differ only in the distributions of the duration of the ON period. In all scenarios the duration of the OFF period is distributed with a Pareto distribution with parameter 1.5 and mean 60 sec. The mean of the duration of the ON period are 200sec, 150sec, 100sec, 50sec and 20sec. The simulations run for 1010sec, and the results of each scenario are the average of five runs. In the rest of the section we will refer to the simulations with 150 users as the *low fair share* simulations and to the 10 users simulations as the *high fair share* simulations.

B. Low Fair Share Results

The line utilization and queue size for the low fair share scenarios are shown in Table II and Table III, and appear in Figure 2 and Figure 3.

The results show that when fair share is relatively low, Vegas_8 has poor line utilization. Only when users activity is low (20-ON 60-OFF) does Vegas_8 improve its utilization. As can

	200-on	150-on	100-on	50-on	20-on
Vegas_1	93%	93%	93%	93%	89%
Vegas_8	75%	75%	75%	75%	81%
AdaVegas	93%	93%	93%	92%	89%

TABLE II
LINE UTILIZATION FOR 150 USERS

	200-on	150-on	100-on	50-on	20-on
Vegas_1	151/ 33	154/ 30	151/ 31	118/ 47	57/ 53
Vegas_8	82/ 71	82/ 70	81/ 70	79/ 68	73/ 63
AdaVegas	142/ 43	140/ 44	131/ 47	104/ 56	57/ 54

TABLE III
QUEUE SIZE FOR 150 USERS. EACH ENTRY GIVES FIRST THE AVERAGE QUEUE SIZE AND THEN THE STANDARD DEVIATION.

	200-on	150-on	100-on	50-on	20-on
Vegas_1	0.67/ 0.07	0.62/ 0.07	0.55/ 0.06	0.43/ 0.05	0.25/ 0.05
Vegas_8	0.34 0.07	0.31/ 0.07	0.28/ 0.06	0.23/ 0.06	0.15/ 0.06
AdaVegas	0.63/ 0.07	0.59/ 0.06	0.54/ 0.06	0.43/ 0.06	0.27/ 0.05

TABLE V
FAIRNESS INDEX FOR 150 USERS. EACH ENTITY SHOWS THE AVERAGE VALUE OF THE FAIRNESS INDEX AND THE STANDARD DEVIATION.

	200-on	150-on	100-on	50-on	20-on
Vegas_1	83%	79%	76%	52%	14%
Vegas_8	91%	90%	88%	76%	41%
AdaVegas	89%	85%	84%	74%	30%

TABLE VI
LINE UTILIZATION FOR 10 USERS

be seen from the queue size, apparently the users were too aggressive, resulting in drops which caused the congestion window to underflow as can be deducted from the high variance. Vegas_1 has high utilization which is also evident from the queue capacity. As for AdaVegas, it achieves results almost identical to those of Vegas_1. This is since for low fair share AdaVegas users operate mostly with the low increment rate.

The packet loss rate at the bottleneck link are shown in Table IV. The results show that Vegas_1 and AdaVegas have similar loss rate values. Vegas_8 has a significantly higher loss rate which is the result of its increment strategy which is far too aggressive for this low fair share value.

	200-on	150-on	100-on	50-on	20-on
Vegas_1	1.3%	1.1%	1%	0.5%	0.2%
Vegas_8	4.4%	4.3%	3.8%	2.8%	1.2%
AdaVegas	1.5%	1.3%	1.1%	0.6%	0.2%

TABLE IV
LOSS RATE AT BOTTLENECK LINK FOR 150 USERS. THE LOSS RATE IS DEFINED AS $\frac{packets_dropped}{packets_arrived}$ AND MEASURED AT THE BOTTLENECK LINK QUEUE

The fairness index results are shown in Table V and Figure 4. The results show that Vegas_1 and AdaVegas have a very similar value of the fairness index, while Vegas_8 has significantly lower value. The fact that AdaVegas performs the same as Vegas_1 is no surprise, given our argument that in this environment the AdaVegas operates very much like Vegas_1, i.e., incrementing at a rate of one MSS per RTT.

C. High Fair Share Results

The line utilization and queue size for the high fair share scenarios are shown in Table VI, Table VII, Figure 2 and Figure 3.

All three mechanisms exhibit lower performance as the activity (i.e. ON period) decreases. Vegas_8 achieves the best results, showing the most moderate decrease in line utilization as activity decreases, while Vegas_1 performs the worst. Apparently the default increment rate is too slow in adapting to the large changes in the fair share, while the more aggressive Vegas_8 takes advantage of the changes and therefore reaches better utilization. AdaVegas performance is, as expected, somewhere between that of Vegas_1 and Vegas_8.

The advantage of Vegas_8 should not be related only to the high increase rate (which is eventually used by AdaVegas) but also to the queue utilization. Due to the setting of the parameters α and β, in steady state each user using Vegas_8 aims for 4-8 segments in the queue while Vegas_1 aims for at 1-3 (see Section III-B). Since in steady state AdaVegas converges to Vegas_1 it aims for 1-3 segments in the buffer as well. This enables Vegas_8 to handle better cases of sudden increase in the fair share.

The loss rate in this scenario is very low. Both AdaVegas and Vegas_1 have almost no packet loss, and Vegas_8 has a packet loss rate between 10^{-4} and 10^{-5}.

The fairness index results for the high fair share are shown in Table VIII and Figure 4. The results show the average fairness index and the standard deviation. The fairness index in high fair share degrades as activity decreases. The relative performance of the mechanisms seems quite random. The low utilization results lead us to believe that users don't converge to the optimal window, but rather expand the window constantly due to the high available bandwidth. As activity decreases window sizes (and hence fairness index) depend more on the ON time of each user than on the interaction between different users.

VI. HETEROGENOUS ENVIRONMENTS

In this section we focus on two issues in particular. The first deals with the case in which different connections have different

	200-on	150-on	100-on	50-on	20-on
Vegas_1	12/	9/	7/	4/	4/
Vegas_1	13	12	11	9	10
Vegas_8	69/	67/	61/	52/	22/
Vegas_8	37	40	41	45	27
AdaVegas	21/	21/	22/	15/	8/
AdaVegas	19	20	22	20	13

TABLE VII

QUEUE SIZE FOR 10 USERS, AVERAGE SIZE AND STANDARD DEVIATION

	200-on	150-on	100-on	50-on	20-on
Vegas_1	0.54/	0.46/	0.47/	0.36/	0.27/
	0.12	0.14	0.12	0.1	0.12
Vegas_8	0.60/	0.5/	0.45/	0.28/	0.26/
	0.11	0.13	0.15	0.1	0.11
AdaVegas	0.60/	0.46/	0.52/	0.25/	0.23/
	0.12	0.13	0.09	0.1	0.10

TABLE VIII

FAIRNESS INDEX FOR 10 USERS. EACH ENTITY SHOWS THE AVERAGE VALUE OF THE FAIRNESS INDEX AND ITS STANDARD DEVIATION.

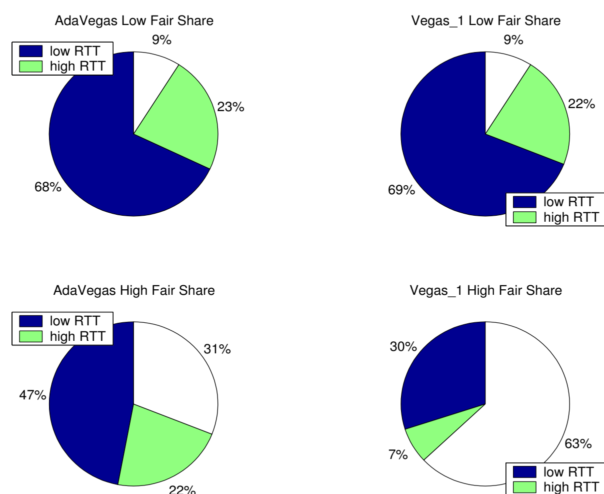

Fig. 7. Simulating users with different RTT. The high fair share shows the simulation results with 10 users of which 5 users have RTT of 600ms and the other 5 users have RTT of 1200ms. The low fair share shows the simulation results with 150 users of which 75 have RTT of 600ms and 75 have RTT of 1200ms. The results show the relative share of the low RTT sessions, high RTT sessions and the unused bandwidth.

round trip times (RTT). The second deals with a mixture of TCP Vegas and AdaVegas users.

A. Heterogenous round trip time

Since AdaVegas (like TCP Vegas) increases the window once every RTT, the increment rate is different for connections with different round trip time. In this set of simulations we use the topology described in Section IV with the following modification: we set the RTT of a half of the connections to 600msec and the RTT of the other half to 1200msec. All users have ON mean time of 50sec and OFF time mean is 60sec (using a Pareto distributions with parameter 1.5). We look both at the case of high fair share (10 users) and low fair share (150 users). The results of the experiments are shown in Figure 7.

As expected, in all scenarios the throughput of the connections with the low RTT was significantly higher. For a relatively low fair share the results of AdaVegas and Vegas_1 are similar. Again, this comes as no surprise since we assume that in this case AdaVegas basically runs TCP Vegas. For a relatively high fair share AdaVegas shows a significant improvement in link utilization, and in the throughput of all sessions. In addition the fairness between the different users has also improved in AdaVegas. We find these results quite encouraging, due to the significant advantage gained by AdaVegas.

B. Heterogenous users

We examine a mixture Vegas_1 and AdaVegas. Since in certain scenarios AdaVegas becomes more aggressive than Vegas_1, this issue can be problematic. We examine simulations with different mixtures of AdaVegas users and Vegas_1 users. Like in earlier simulations, we did simulations with low fair share (150 users) and with high fair share (10 users). We use the topology described in Section IV with the following modifications: All users work with 50sec mean ON time, and 60sec mean OFF time Pareto distribution. In each scenario there is a different mixture of AdaVegas users and Vegas_1 users. (Round trip time of all the users is identical.) The results of the experiments are shown in Figure 8.

For relatively low fair share, the difference between the throughput of the AdaVegsa users and the Vegas_1 users is small and the line utilization is stable around 92%. Again, this is no surprise since in low fair share environments AdaVegas users act like Vegas_1 users.

For high fair share, the advantage of AdaVegas becomes apparent. As the fraction of AdaVegas users increases, the line utilization increases from around 55% to about 70%. In addition, the performance of Vegas_1 users slightly deteriorates as the fraction of AdaVegas users increases.

VII. CONCLUSION

We have introduced AdaVegas, a flow control mechanism based on TCP Vegas. By setting its increment parameters dynamically, AdaVegas is able to better adapt to the changing network environments. We have studied AdaVegas behavior in various scenarios and showed that it is able to adapt well. We compared AdaVegas performance with that of TCP Vegas, and have shown that AdaVegas achieves an improvement.

We simulated AdaVegas in environments were different users had different RTTs, and in environments were some users were AdaVegas and others TCP Vegas. Both cases show the advantage of using AdaVegas.

Although we focused on TCP Vegas, we believe learning mechanism can also be used in other variants of TCP, namely TCP New Reno, and in other network protocols. Therefore,we regard this paper not only as a research to improve TCP Vegas, but as an example of how learning algorithms can be used in

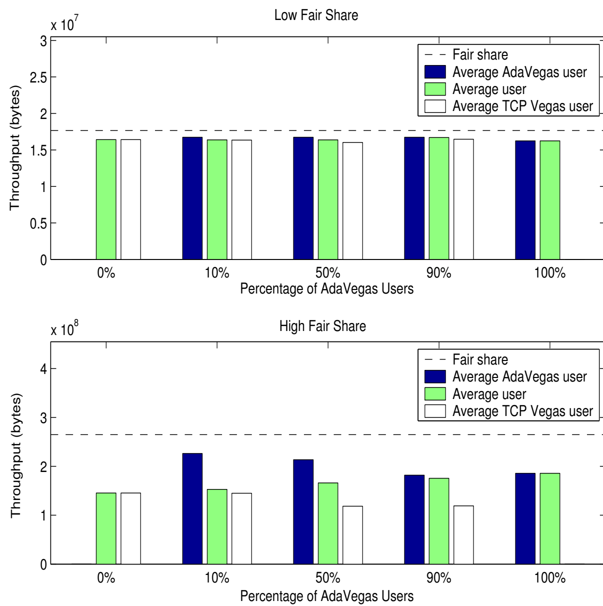

Fig. 8. Different mixture of AdaVegas users and Vegas_1 users. The top graph has 150 users, and the bottom has 10 users. We plot the throughput of the average AdaVegas user, the average user (both TCP Vegas and AdaVegas) and the average TCP Vegas (Vegas_1) user. The fair share is shown by the dashed line. The throughput is measured in bytes.

TCP flow control specifically and network protocol in general to achieve better performance.

In the future we plan to incorporate in TCP New Reno adaptive mechanisms to control both the increase and decrease parameters. As for AdaVegas, we would like to add an adaptive mechanism for the decrement which will be similar to the one used for the increment. In addition, we would like to allow the increment value to be unbound, and our preliminary experiments were encouraging. However, using higher increment rates raises the question of how to set the α and β parameters of TCP Vegas due to the delicate interaction between those parameters and the increase parameter.

REFERENCES

[1] V. Jacobson, "Congestion avoidance and control," in *Proceedings of the ACM, SIGCOMM '88*, August 1988, pp. 314 – 329.

[2] D. Chiu and R. Jain, "Analysis of the increase and decrease algorithm hms for congestion avoidance in computer networks," in *Computer Networks and ISDN Systems V.17*, 1989, pp. 1 – 14.

[3] L. Brakmo, S.W. O'Malley, and L. Peterson, "Tcp vegas: New techniques for congestion detection and avoidance," in *Proceedings of SIGCOMM '94*, 1994, pp. 24 – 35.

[4] Yang Richard Yang and Simon S. Lam., "General aimd congestion control," Tech. Rep. TR-2000-09, University of Texas at Austin, may 2000.

[5] M. Floyd, "Tcp-friendly unicast rate-based flow control," 1997.

[6] D. Bansal and H. Balakrishnan, "Binomial congestion control algorithms," 2001.

[7] Martin L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, John Wiley & Sons, Inc., April 1994.

[8] ," The Network Simulator - ns-2 available at http://www.isi.edu/nsnam/ns/.

[9] T. H. Henderson, E. Sahouria, S. McCanne, and R. H. Katz, "On improving the fairness of TCP congestion avoidance," *IEEE Globecomm conference, Sydney*, 1998.