Problems in TCP Vegas and TCP Reno

Omar Ait-Hellal and Eitan Altman
INRIA B.P. 93,
2004 Route des Lucioles
06902 Sophia Antipolis
FRANCE
email address: {oaithel,altman}@sophia.inria.fr
Phone: (33) 93 65 77 86, Fax: (33) 93 65 77 65

Abstract

This paper outlines some problems observed, by simulation in Reno and Vegas TCP versions. We describe the two protocols and give examples (obtained by simulation) where these problems can occur. The most serious problem causes in an erroneous estimation of the RTT, and can occur in both Vegas and Reno, however, in case of Vegas, we propose a solution which allows to decrease the probability that this problem occurs. Also, we show by an example that the technic which consists on retransmitting packets when we receive the first or second acknowledgment after retransmission can't work in many cases when the retransmission timer (coarse grain timeout) is not sufficiently large.

1 Introduction

The congestion control mechanism has been studied for long time for the TCP/IP protocols, and many algorithms were proposed in the literature, the most known is that proposed by Van Jacobson [7] and dubbed Tahoe. Revised two years after by his author [8], by including *Fast Recovery* and *Fast Retransmit* [11], the new version called Reno achieves more bandwidth utilization and retransmits less data than its predecessor. As will be shown bellow, due to multiple losses, TCP Reno doesn't improve the bandwidth utilization in some cases [2].

Brakmo and Peterson [4] proposed a new congestion avoidance algorithm dubbed Vegas based on some modifications on the source behavior of Reno. This new algorithm is able to interoperate with any other valid implementation of TCP and was reported in [4] to achieve between 37% and 71% better throughput than Reno, to retransmit between one fifth and one half as much data as does Reno and hence to use more efficiently the available bandwidth. In [5] The authors confirm these improvements, and report a gain between 4%-20% over Reno in throughput.

The "Fast retransmit" and "Fast recovery" mechanisms are implemented in both Reno and Vegas, and are known to be efficient in improving the throughput, by detecting losses faster, and hence using more efficiently the bandwidth. However, they can be the main sources of some anomalies, and thus lead to some problems. In [6], a successive fast retransmit problem is explained, and a solution is proposed to avoid this problem.

In the present paper, we illustrate some other problems due, essentially, to the "Fast recovery" mechanism. To do that, we use packet traces from simulations done by the REAL simulator network [10] in a simple environment: one source sending packets to the sink and sharing a bottleneck with an exogenous traffic (Poisson source with parameter λ). Some analytic results, especially Theorem 3.2, from [1] enable us to identify parameters for which the performances of Vegas degrades (has the same behavior as Reno see [1]).

2 Description and comments on Reno and Vegas

Reno and Vegas versions of TCP flow control are two dynamic window schemes that use two types of mechanisms to detect packet losses: duplicate Ack and timeouts (see details below). The congestion window (Cwnd) is defined as the number of allowed packets (or bytes) sent but not yet acknowledged (bytes in transit).

The control mechanism (Reno or Vegas) increases or decreases the window size depending on whether the packet is acknowledged or lost (its timeout has expired or the source received duplicate Acks). In the first case it is set to min{receive window, congestion window}. The size of the receive window is defined by the destination and determines the number of packets it can receive (packets out of sequence are buffered at the destination); we assume that the receive window is always larger than the congestion window.

2.1 TCP Reno

Reno adjusts its window according to the phase it performs:

Slow start: It is an "exponential" increase of the window; the source sends two packets every time an Ack is received. A Slow start phase is initiated after a loss is detected through a time-out mechanism and it ends when the window reaches a certain level called *slow start threshold* W_{th} (which is set in our experiments, at the beginning to 32 packets).

Congestion avoidance: This phase begins at the end of the slow start phase or after a loss is detected by duplicate Acks. TCP continues to increase its congestion window by $1/Cwnd$ every time it receives an Ack, till it reaches the receive window or till a loss occurs (it uses the loss of packets to detect an eventually congested network), whichever occurs first.

When the sender sends a packet, it starts a retransmission timer. If the timer expires (i.e if it exceeds some value, called the coarse grain timeout) and the packet is not yet acknowledged, the packet is considered to be lost and the slow start phase is performed again by setting the slow start threshold to one-half the current window size and the congestion window to one. Reno checks for timeout every 500 ms and determines the coarse grain timeout by keeping an estimator of the round trip time.

We next describe the fast retransmit and fast recovery mechanisms.

A loss can also be detected by receiving three duplicate Acks; in this case we retransmit *only* the packet that is supposed to be lost but not the subsequent ones, without waiting for a retransmission timer to expire (this is the *Fast Retransmit* mechanism), and the congestion avoidance phase is again performed:

1. The Slow Start threshold (W_{th}) is set to one-half the current window size.

2. The congestion window ($Cwnd$) is set to W_{th} plus three times the packet size (MSS, i.e. the Maximum Segment Size).

3. Each time the sender receives a duplicate Ack, it increments $Cwnd$ by one packet and sends a new packet (if it is allowed by the new $Cwnd$ size).

4. When the first non-duplicated Ack arrives, we set $Cwnd$ to W_{th}.

1-4 is the *fast recovery* mechanism.

The *fast retransmit* mechanism is known, historically, to be efficient and was introduced in order to avoid restarting the slow-start phase each time a loss occurs (problem which results in drastic slow of the throughput). It reaches its goals only in case where there is no multiple drops. Furthermore, in case of multiple drops, Reno halves its window every time a loss is detected by duplicate Acks (it can recover in some cases up to three drops but it is unlikely). When there is no arriving Ack (duplicate), and still there are drops not yet detected, Reno waits for a retransmission timer to expire and restarts the slow-start phase after setting the slow start threshold (W_{th}) to half the current window size. The slow-start threshold (W_{th}) obtained equals to the size of the window when a first loss is detected divided by 2^{1+n}, where n is the number of detected losses by duplicate Acks. On the other hand, in case of Tahoe, when there are multiple drops, on the first detected loss, the window is set to one and the slow-start threshold (W_{th}) to half the window size at which a loss has occured (max_window). Thus, Tahoe performs the slow-start phase (and doing so, it recovers from the other losses) quicker than Reno and with a threshold (W_{th}) larger than the one in case of Reno. This is the reason why Tahoe outperforms Reno version in case of multiple drops see [2] (Ex: if there are three drops in the same window of data, and the two first ones are detected by duplicate Acks and the third one by timeout, then Reno restarts the slow-start phase with a threshold (W_{th}) equal to the size of the window when a first loss is detected divided by 8. In Tahoe version the slow-start is performed with a threshold four times greater (max_win/2)).

2.2 TCP Vegas

Vegas also uses the fast retransmit and fast recovery mechanisms. It differs from Reno in its manner to detect losses, to detect the available bandwidth and it also has a different slow start behavior. The congestion window is updated according to the phase being performed:

Congestion avoidance: In its congestion avoidance phase, Vegas estimates the amount of data buffered at switches and according to that, it decides whether it increases or decreases its congestion window by one packet. This estimation is done one time every RTT: Vegas sets $BaseRTT$ to the minimum of all measured round trip times (including those of previous RTTs) and computes the $Expected$

rate as $Expected = Cwnd/BaseRTT$ where the $Cwnd$ is the number of bytes in transit. During this period, Vegas records the sending time of a distinguished packet and how many bytes are transmitted between this time and time when its acknowledgment is received and it computes the sending rate as $Actual = Cwnd/RTT_d$, where RTT_d is the round trip time of the distinguished packet. The sender defines three thresholds α_{th}, β_{th} and γ_{th} (in our experiments, we set $\alpha_{th} = 2$, $\beta_{th} = 4$ and $\gamma_{th} = 1$ ($Vegas_{2,4}$)). If $Expected - Actual \leq \alpha_{th}/BaseRTT$ then we increase the congestion window by $1/Cwnd$ every time we receive an Ack (as does Reno). However, if $Expected - Actual > \beta_{th}/BaseRTT$ we decrease the congestion window by one packet size, finally if the difference $Expected - Actual$ falls between the thresholds (α_{th} and β_{th}), we keep the congestion window unchanged.

Recall that in the implemented version (Vegas release 0.8 [3]), RTT_d is considered as the average round trip time over all packets acknowledged between sending this packet till receiving its Ack.

Slow start: The principle is similar to that of Reno, However, Vegas doubles its congestion window every other RTT, as opposed to Reno's every RTT. The slow start phase ends, once Vegas detects queue buildup. This is done, when the congestion window reaches a value W_{th} for which the difference $Expected - Actual$ exceeds a certain threshold $\gamma_{th}/BaseRTT$ and then Vegas transitions to its congestion avoidance phase. This helps to avoid losses in the slow start phase, where they always may be in the order of half the current window size in case of Reno.

Losses detection: Here we describe the retransmission mechanism of Vegas and comment the different techniques used for detecting losses.

Vegas extends Reno's retransmission mechanism as follows: Vegas reads and records the system clock each time a packet is sent. When an Ack is received, Vegas reads the clock and does the RTT calculation by computing the mean round trip time, the mean deviation and the fine grain timeout value (*v_ timeout*), which is computed in the same way as does Reno but not adjusted to a multiple of 500 ms. Recall that coarse grain timeout used by Reno, remains effective under Vegas. Vegas decides to retransmit in the following situations:

- When a duplicate Ack is received, Vegas checks to see if the round trip time of the first (in sequence) unacknowledged packet is greater than the fine grain timeout value (v_timeout). If it is, then Vegas retransmit the packet without waiting for three duplicate Acks (required by Reno).

Quite often, this technique is not useful and does not allow to detect a loss faster. This is because the value of the fine grain timeout should be at least 2RTT (see [9]), where as the time between the first and the third duplicate Ack are typically much shorter. In other words, if there is a loss, the time between the Ack of the well received packet and the third duplicate Ack does not often exceed RTT.

- When a non duplicate Ack is received, if it is the first or the second one after a retransmission, Vegas again checks to see if the time interval since the packet, not yet acknowledged, was sent exceeds the timeout value. If it is, then Vegas retransmits the packet.

This technique allows to detect losses faster and to recover from multiple drops without restarting the slow-start phase if the retransmission timer doesn't expire before (Figure 3). Hence it permits to deal with a problem from which Reno suffers too much, namely, multiple drops in the same window of data as explained above.

3 Problems and suggestions for improving Reno and Vegas

3.1 Simulator

To get the packet traces we used REAL [10] network simulator,which simulates a packet switched network. We included Vegas TCP version by modifying jk_ftp.c source (source code for Tahoe in REAL), the modifications we have provided, are inspired from Vegas' code [3](vtcp_input.c, vtcp_timer.c,...). We have not included the solution proposed by S. Floyd [6] for avoiding the successive fast retransmit problem since we have not observed this problem in Vegas and Reno.

We have allowed a maximum window size of 128 packets corresponding, approximatively, to 64 Kbytes and it is never reached in our examples. W_{th} is set to 32 packets at the beginning for Reno, and all packets have a default fixed size of 576 bytes (including IP and TCP headers), so that a congestion window is a multiple of 576 bytes.

To illustrate the different problems, we considered a simple model with one controlled ftp source sending to the sink and sharing with another source (poisson source) a bottleneck node with B buffers and transmission rate μ. T is the round trip time, including the service time in an empty system. The duration of all the simulations bellow is 100 seconds.

3.2 Description of the problems

- First problem

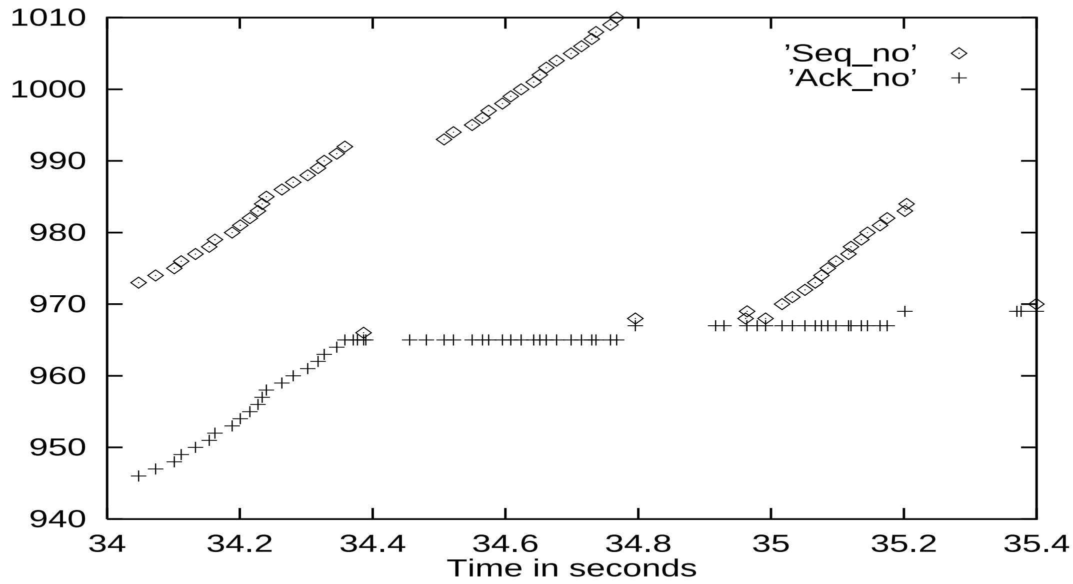

Figure 1: Problem of fast recovery after coarse grain timeout occurs

The first problem which appears in Vegas and Reno, is of unnecessary retransmissions that occur when we perform the *fast recovery* mechanism after the retransmission timer expires. This is shown in Figure 1 (Vegas), which plots the times at which packets are sent and at which their acknowledgments return to the sources (the latter is denoted by the crosses). The vertical axis denotes the sequence number of the packets. When more than one acknowledgment have the same sequence number, these correspond to duplicate Acks, indicating that a loss is detected.

The scenario is as follows: at time 34.38 the sender receives three duplicate Acks acknowledging packet 965; this means that packet 966 is lost, and it is thus retransmitted; the window is then decreased to 21+3. The window increases while duplicate Acks (still acknowledging packet 965) arrive (fast recovery); at each such arrival another packet is transmitted if the actual size of the window allows it. We send packets 993-1010 (992 was the last packet sent before receiving the first duplicate Ack). At time 34.8, a first non duplicate Ack acknowledging packet 967 arrives, the window is set to 21; Vegas now checks whether the fine grain timeout for packet 968 has expired; since it is so, in our case, packet 968 is retransmitted. After that, coarse grain timeout occurs for packet 969 at time 34.95(the first in sequence unacknowledged packet is 968. However, since it has just been retransmitted, its corresponding timeouts are reset to zero). The window is set to 2, we retransmit packets 968-969. On the third duplicate Ack after the window was set to 2, packet 968 is again considered to be lost and retransmitted for the fourth time. By applying the fast recovery mechanism, each time a duplicate Ack is received, a "new" packet is sent and then packets 970-982 are retransmitted in a fast recovery.

We see that we retransmit packet 968 four times (while it was in fact only lost once). Moreover, many packets (970-982) that were well received are retransmitted due to the fast recovery mechanism.

One can avoid the last retransmission of packet 968 by setting the number of received duplicate Acks to three just after a coarse grain timeout expires. (This can be written in the TCP Vegas code release 0.8 [3] $tp \rightarrow t\_dupacks = tp \rightarrow t\_rexmtthresh$ in vtcp_timer.c).

The solution given by S. Floyd [6] for avoiding the problem of successive fast retransmit is also a solution in this case; it doesn't allow to retransmit two lost packets, detected by duplicate Acks, and belonging to the same window of data (see [1] for definition of a window of data), that were detected by duplicate Acks. The drawback of this solution is that under Reno and Vegas, actually, there is no problem of successive fast retransmit; we have not observed such problem, and this solution means that if there are two or more losses in the window, the second loss will be detected by timeout, which decreases, substantially, the throughput since the new slow-start threshold will be one-forth the maximum window size (the window at which a loss occurs) instead of one-half in Tahoe. Taking into account the bursty nature of the traffic generated on the Internet, successive losses are very likely to occur, and hence Reno may often fail to improve the performances with respect to Tahoe [2].

The problem of the retransmission of packets 970-982 can be solved by not performing fast recovery after the retransmission (i.e. coarse grain) timer expires. This can be done by setting a flag (fast recovery) when the coarse grain timeout expires and clear it once we receive a non duplicate Ack (in the TCP Vegas code release 0.8 [3] we increment the window when duplicate Acks arrive only if ($tp \rightarrow t\_dupacks > tp \rightarrow t\_rexmtthresh$ && !fast_recovery)).

Under Reno, this problem, can also occur when the timeout expires. In our example, if we suppose that the scenario is the same till receiving the first non duplicate Ack (time 34.8), Reno receives two duplicate Acks (acknowledging packet 967) before the retransmission timer expires for packet 969. When the retransmission timer expires, the window is set to one and packet 968 is retransmitted again. This packet (968) will be retransmitted for the third time when the third duplicate Ack after timeout arrives. Since duplicate Acks arrive, the scenario is the same as that under Vegas except that we receive less duplicate Acks than in case of Vegas and hence retransmit less data (Reno halves the window while Vegas reduces it by quarter when a loss occurs. Reno sets its window to 14+3 instead of 21+3 in case of Vegas when the first loss is detected. Hence we retransmit, only, packets 969-974). The solution is the same as that proposed for Vegas.

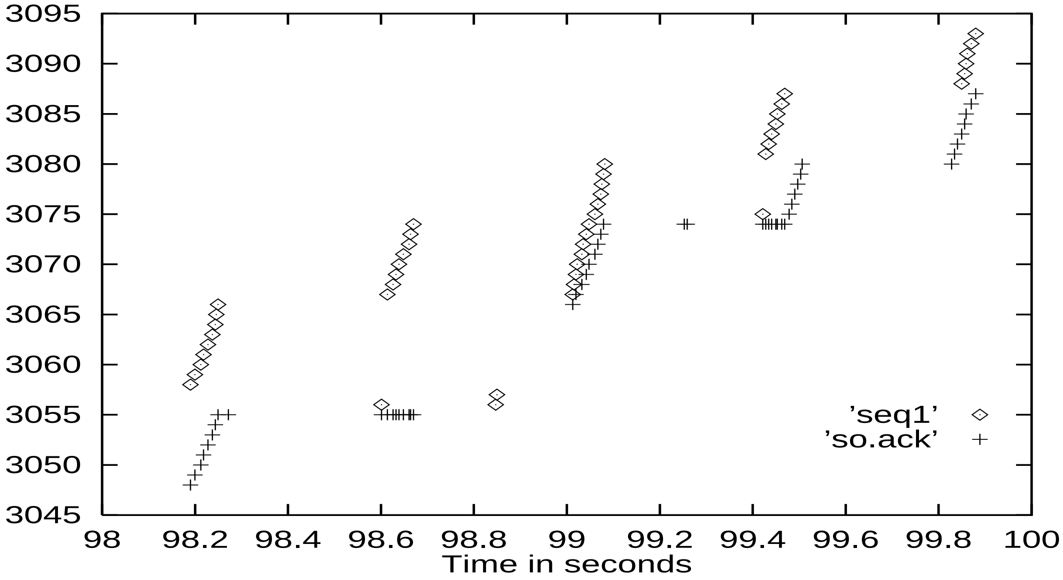

Figure 2: Problem of erroneous estimation of the round trip time (RTT)

- Second problem

The second problem is of an erroneous estimation of the RTT. It occurs in the same situation : fast recovery followed by coarse grain timeout. An example is given in Figure 2. The scenario is as follows: The window size was 12, when we receive the second duplicate Ack at time 98.6 (acknowledging packet 3055, the last sent was 3066), packet 3056 is retransmitted (Vegas' timeout has expired for this packet) and the window is then set to 9+2. Packets 3067-3074 are sent, due to fast recovery mechanism (we receive 8 duplicate Acks, for each duplicate Ack, we send a new packet).

Num Ack	*Win size*	*Packet sent*	*Comments*
3066	2	3067-3068	slow start phase
3067-3068	4	3069-3072	the window is doubled
3069-3072	4	3073-3076	the window remains constant
3073-3074	8	3077-3080	the window is doubled before the first dup Ack

Table 1: Comments on Figure 2

Coarse grain timeout expires for packet 3057 at time 98.85: the window is set to 2 (first packet in sequence unacknowledged is 3056. Setting the window to 2 packets instead of 1 as is the case in Reno is not explained by the authors) we send packets 3056 and 3057, then we receive acknowledgment of packet 3066 at time 90.019, we retransmit packet 3067 and we receive 0.00685 sec after the Ack of the previous transmission of that some packet. This time period of 0.00685 sec is considered, erroneously, to be the RTT of packet 3067. Recall that the slow start phase is performed at this moment. The retransmission of packets 3067-3074 is described in Table 1 and the erroneous measured RTT (round trip time) for these packets varies from 0.00685-0.0312 instead of ~ 0.40 and the smoothed RTT goes till 0.193, it can be worst if there were many received duplicate Acks after the lost packet was detected by duplicate Acks.

This problem, also, can occur under Reno, however, we retransmit less data than in case of Vegas (because the window is reduced by one half in Reno and by 1/4 in Vegas). If we suppose the same scenario till the first duplicate Ack arrives (time 98.27), packet 3056 will be sent on the third duplicate Ack (not on the second as under Vegas) and the window will be set to 6+3 (9+2 for Vegas), and then only *five* packets (967-971) will be retransmitted due to fast recovery and will have a very small, erroneously, measured round trip time instead of *eight* in case of Vegas.

The solution for this problem under Reno seems to be more difficult, however, under Vegas, one can decrease the probability that this problem occurs by reducing the probability that the retransmission timer expires before receiving the Acks of packets retransmitted due to the first or second non duplicate Ack after retransmission (see bellow). This implies a large coarse grain timeout as a solution.

- Minimum values for the coarse grain timeout allowing to recover from multiple losses (Vegas)

Since the coarse grain time out is set in the current implementation to around twice rtt, then when there are two or more (multiple) losses in the same window of data, then the coarse grain timeout is likely to occur, before detecting the 2nd loss by comparing the time elapsed since the packet was sent to Vegas' fine grain timeout. Figure 3 shows an example where the minimum values for the coarse grain timeout which allow to recover from two or three multiple losses are given. Restarting the slow-start phase when more than one loss has occured (as it is, generally, the case in Reno) has the undesirable consequence of reacting (systematically) in an extreme way to multiple losses, by decreasing the window to 2. In many simulations we have done, it seems that the reaction of reducing the window by half (instead of 1/4) due to the duplicate Acks could have well sufficed and gives better performances than the 1/4 reduction (see table 2). Since TCP/IP is likely to operate in an environment of nonregular exogenous traffic causing multiple drops, the performance would improve if we increased the coarse time-out to around 4 times RTT. More generally, if we choose the coarse time-out to be around $n+1$ times RTT, then we would be able to recover, by the new mechanism for losses detection, from n drops per window without restarting the slow-start phase (this will be made clear below). In Vegas, the objective is typically to recover from $n = 3$ multiple losses by introducing Vegas' timeout (first loss is detected by duplicate Acks, the second *resp* the third by receiving the first *resp* the second nonduplicate Ack after the first retransmission).

To illustrate this, an example is given in figure 3, where there are three drops (packets 2, 4, 7) in the same window of data, the first one will be detected when we receive the third duplicate Ack acknowledging packet 1. At this time, the window is reduced by quarter (i.e. by 2 packets, since it was 8) and packet 2 is then retransmitted. Due to fast recovery, the window is increased by 3 times the packet size (the increase corresponds to the number of duplicate Acks) i.e. it becomes 6+3, and then packets 9-10 are sent. When the fourth duplicate Ack arrives, the window is increased by one and packet 11 is sent. Packet 3 will be acknowledged $T1$ after it was sent, so its coarse timeout should be at least $T1 > 2RTT$, else the retransmission timer expires for this packet before its Ack returns back and slow-start phase is then performed. Once Ack3 is received, the window is set to 6 (no duplicate Acks) and since it is the first nonduplicate Ack after retransmission, Vegas checks to see if Vegas' timeout has expired for the first unacknowledged packet (packet 4). As it is the case (it was sent more than two RTTs ago), it decides to retransmit it with setting its retransmission timer to zero.

The second loss (packet 4) is then detected at this time. Due to fast recovery, the window reaches the size 9, and then packet 12 is sent because the first unacknowledged one is Packet 4 and the last sent is 11 (11-4+1=8). The second loss (4) is considered to be recovered when the Ack 6 arrives (packet 6 is safely sent), acknowledging (among other) packet 5. This happens $T2 \geq 3RTT$ after packet 5 was sent. Then for the same reasons as above, the timeout for packet 5 should be at least $T2 \geq 3RTT$. When Ack 6 arrives, the window is set to 6 (no duplicate Acks), and since it is the second nonduplicate Ack after the first retransmission (packet 2), Vegas checks to see if the time elapsed since the first unacknowledged packet was sent (packet 7; it was sent more than 2RTTs ago) exceeds vegas' fine grain timeout, since it is so, packet 7 is considered to be lost and thus, retransmitted with setting its retransmission timer to zero. Packet 13 is sent when the second Ack6 is received, this is due to the fast recovery mechanism (12-7+1=6).

Packet 8 will be considered as safely sent, when Ack 11 arrives. However if the retransmission timer expires before (for packet 8), slow-start will be performed and packet 7 will be retransmitted for the third time. Hence, in order to recover from the three losses (2,4,7) without restarting the slow-start phase, the timeout for packet 8 should be at least $T3$ which is around 4RTTs. So, as mentioned above, more generally, in order to recover from n multiple losses in the same window of data, the coarse grain timeout should be around $n + 1$ times RTT.

The proposed solution (i.e large coarse grain timeout) does not have undesirable effects on Vegas' functioning, since Vegas uses the fine grain timeout to recover from losses (Vegas' timeout) and the retransmission timer is used only in extreme cases where the fine grain timeout fail (e.g. four losses in the same window of data).

Note that all these problems occur when the round trip time RTT is close to the retransmission timer resolution.

3.3 Numerical results

In table 2, we summarize some performance measures of Vegas and Reno TCP versions obtained by simulation with and without introducing the improvements discussed above. We used the same model described above, with a Poisson source as exogenous traffic. The versions are: V1 which represents the original version from [3], V2 is the V1 version with the solution proposed to avoid the first problem, V3 is V2 version with a large coarse grain timeout value (it is set to twice the original value i.e timeout=2 (RTT+2D) where D is the mean deviation of the round trip time and RTT is the mean round trip time (see [11]). The obtained value for the timeout is increased by 500ms and adjusted to a multiple of 500ms, this is done by REAL in order to be in accordance with the timer resolution) and, finally, V4 is V3 version with reducing the window by half when a loss is detected by duplicate Acks instead of quarter in the original version. We defined, also, Reno1 and Reno2 to be, respectively, the original version of Reno and its improved version after introducing the modifications avoiding the first described problem. As it is expected, we can see from table 2 that Reno2 is better than Reno1.

From V1 to V4 the improvements are "gradual", and according to these results, the last version V4 is the most efficient one, it retransmits less data while it sends, relatively, the same amount of data than the others. The slow-start phase in the improved version (V4) is performed rarely compared to the original version and all the problems described above are not observed in this version. The differences in throughput (Thp) and retransmissions (Retxs) between V3 and V4 show the need of decreasing the window by half when a loss is detected instead of quarter. Even if we include the solutions for the problems described above, V3 is not more efficient than the original version and this is due to the fact that Vegas reduces its window only by quarter when a loss is detected by duplicate Acks, so that by applying the "Fast recovery" mechanism, many new packets are transmitted just after a loss is detected causing a network saturation. So, reducing the window by half decreases the number of new packets likely to be transmitted due to fast recovery, and hence keeps less data in the network.

Let us consider another example where the "*star war*" traces are used as exogenous traffic in the single

Parameters								
μ	λ	T	*Buffer*	*Version*	*Thp (pkt/s)*	*Losses*	*Retxs*	*Losses (ex)*
320	268	0.3260	12	V1	54.61	49	97	75
				V2	49.27	47	76	64
				V3	56.61	54	60	89
				V4	56.54	35	39	91
				Reno1	52.65	33	59	57
				Reno2	57.23	30	53	76
80	50	0.5137	5	V1	12.48	59	103	110
				V2	12.55	59	93	105
				V3	10.43	80	95	96
				V4	12.39	49	57	92
				Reno1	10.29	36	46	96
				Reno2	10.29	36	46	96

Table 2: Performances of TCP Vegas and TCP Reno, the original versions and their improved versions

bottleneck model, described above. We assume that the exogenous source (sending *star war*) divides frames into packets with a fixed size of 576 Bytes, and sends one frame every 30 msec. During the 100 seconds simulation, the average rate of the exogenous source is 934.136 $pkts/sec$. The capacity of the bottleneck is set to 1500 $pkts/sec$, the buffering capacity is of 20 packets and the line speeds of the sources are equal to 3000 $pkts/sec$. We run two sets of simulation, in the first set, Vegas source and exogenous source have, respectively, a round trip time RTT of 81.2 and 81.2. In the second set Vegas has an RTT of 61.2 ms, while the exogenous source is closer 41.20 $msec$ (table 3). The timer resolution is set, in both examples, to 100 msec (the resolution proposed for the TCP over ATM). The average available bandwidth is about 565.864 $pkts/sec$.

Table 3 shows the different results obtained for the versions defined above, the utilization (fraction of the throughput of the source to the average available bandwidth), losses for the two sources and the number of retransmissions. The first observation is that Vegas outperforms Reno in all the examples, and the solution for the first problem improves Reno's performances in terms of retransmissions and exogenous losses (Reno becomes less aggressive). The losses and retransmissions for vegas are less important while the version V4 is executing, however the throughput, slightly decreases, due to the reduction of the window by half. In fact, the optimal amount of data that the window should be reduced by, strongly depends on the nature of the exogenous traffic and the available bandwidth. In these examples the window size reaches 80 packets, and hence reducing the window by quarter (20 packets) could well suffice.

Vegas technic for losses detection works well by increasing the coarse grain timeout (twice the original value) and allows to detect losses, without restarting the slow-start phase, which may slow the throughput and cause unnecessary retransmissions. As we can see from the results obtained when the version V3 is executing, V3 allows a hight utilization while it retransmits less data comparing to the original version. Note, also, that Vegas and Reno performs well when the source is closer to the router.

Version	*Throughput (pkt/s)*	*Utilization*	*Losses*	*Retransmissions*	*Losses for video*
V1	464.82	82.1 %	83	323	991
V2	454.45	80.3 %	77	118	1031
V3	461.56	81.5 %	90	118	972
V4	433.15	76.54 %	71	80	763
Reno1	392.66	69.39 %	123	385	853
Reno2	397.39	70.22 %	138	232	821
V1	429.76	75.94 %	112	287	593
V2	427.97	75.63 %	114	178	617
V3	432.02	76.34 %	141	155	676
V4	417.53	73.78 %	105	117	554
Reno1	381.90	67.48 %	147	487	680
Reno2	360.91	63.78 %	155	248	589

Table 3: Performances of TCP Vegas and TCP Reno, in presence of bursty traffic

References

[1] O. Ait-Hellal, E. Altman "Evaluation of TCP-Vegas and TCP Reno", *IEEE International Conference on Communications (ICC'97)*, Montreal, 8-12 june 1997.

[2] T. V. Lakshman, U. Madhow, B. Suter "Window-based error recovery and flow control with a slow acknowledgment channel: a study of TCP/IP performance" submitted.

[3] Lawrence Brakmo, "TCP Vegas Release 0.8", November 15, 1994. available from anonymous ftp from ftp://ftp.cs.arizona.edu /xkernel/new-protocols/Vegas.Tar.Z.

[4] L.S Brakmo, L.L. Peterson, "TCP Vegas : End to End Congestion Avoidance on a Global Internet," *IEEE Journal on selected Areas in communications,* vol. 13, pp. 1465-1480, 1995.

[5] J.S. Ahn, P. B. Danzig, Z. Liu and L. Yan, "Experience with TCP Vegas: Emulation and experiment," in *Proc. SIGCOMM'95 Symp.,* Aug. 1995, to be published.

[6] S. Floyd, "TCP and successive fast retransmits". *Lawrence Berkeley Laboratory, Technical Report*, February 24, 1995.

[7] Van Jacobson. "Congestion avoidance and control". *ACM SIGCOMM 88*, pages 273-288, 1988. ftp.ee.lbl.gov/papers/congavoid.ps.Z

[8] Van Jacobson. "modified TCP congestion avoidance algorithm", *mailling list, end2end-interest* received on mon, 30 Apr 1990.

[9] Van Jacobson, L. Peterson, L. Brakmo, S. Floyd "problems with Arizona's Vegas" *mailling list, end2end-tf* (discussion on TCP Vegas), 1994.

[10] S. Keshav, "REAL: A network simulator", *Department of Computer Science, UC Berkeley, Technical Report* 88/472, 1988.

[11] W. Richard Stevens. "TCP/IP illustrated", *volume 1. Addison Wesley*, 1994.

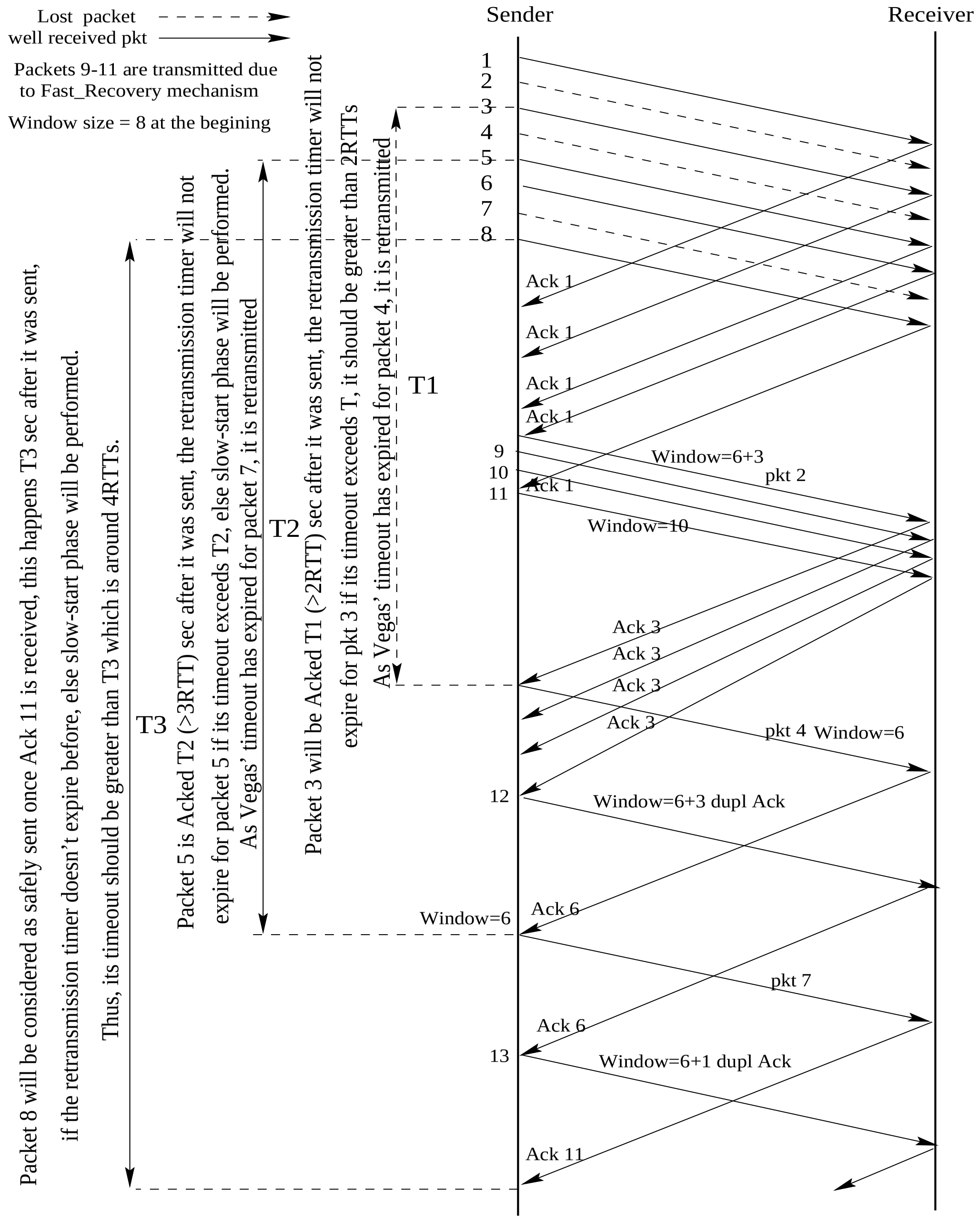

Figure 3: Losses detection in Vegas