**TCP VENO:**

Distinguishing between congestion control loss and random loss, and providing different measures to deal with them, is a fundamental problem that remains unsolved for TCP. Veno makes use of a mechanism similar to that in Vegas to estimate the state of the connection. If packet loss is detected while the connection is in the congestive state, Veno assumes the loss is due to congestion; otherwise, it assumes the loss is random.

**Distinguishing Between Congestive and Non-congestive States**

In Vegas [10], the sender measures the so-called Expected and Actual rates:

Expected = cwnd/BaseRTT

Actual = cwnd/RTT

where cwnd is the current TCP window size, BaseRTT is the minimum of measured round-trip times, and RTT is the smoothed round-trip time measured . The difference of the rate is

Diff = Expected - Actual

When RTT > BaseRTT, there is a bottleneck link where the packets of the connection accumulate. Let the backlog at the queue be denoted by N. We have

RTT = BaseRTT + N/Actual

That is, we attribute the extra delay to the bottleneck link in the second term of the right side above. Rearranging, we have

N = Actual \* (RTT – BaseRTT) = Diff \* BaseRTT.

If N < β when a packet loss is detected, Veno will assume the loss to be random rather than congestive, and a different window-adjustment scheme than that in Reno will be used; otherwise, Veno will assume the loss to be congestive, and the Reno window adjustment scheme will be adopted**.**

**Slow-Start Algorithm**

At the start-up phase of a connection, Reno sets the window size, cwnd, to one and sends out the first packet. Thereafter, each time a packet is acknowledged, the window size increases by one. Thus, cwnd increases from 1 to 2 after the first round-trip time, from 2 to 4 after the second round trip time (because of the reception of two acks), and so on. This results in an exponential increase of the sending rate over time until packet loss is 7 experienced. Veno employs the same initial slow-start algorithm as Reno’s without modifications.

**Additive Increase Algorithm**

If ( N < β ) // available bandwidth not fully utilized set cwnd=cwnd+1/cwnd when each new Ack is received

Else if (N ≥ β ) // available bandwidth fully utilized set cwnd=cwnd+1/cwnd when every other new Ack is received

The first part is the same as the algorithm in Reno. In the second part, the backlog packet (N) in the buffer exceeds β, and we increase the window size by one every two round-trip times so that the connection can stay in this operating region longer.

**Multiplicative Decrease Algorithm**

1. if (N < β) //random loss due to bit errors is most likely to have occurred

ssthresh = cwnd \* (4/5) ;

else

ssthresh = cwnd /2 ; // congestive loss is most likely to have occurred

1. Each time another dup Ack arrives, increment cwnd by one packet.
2. When next Ack acknowledging new data arrives, set cwnd to ssthresh (value in step 1)

**IMPLEMENTATION:**

**HEADER FILES USED:**

#include <linux/mm.h>

#include <linux/module.h>

#include <linux/skbuff.h>

#include <linux/inet\_diag.h>

#include <net/tcp.h>

**VARIABLES USED:**

#define [V\_PARAM\_SHIFT](http://lxr.free-electrons.com/ident?i=V_PARAM_SHIFT) 1

static const int [beta](http://lxr.free-electrons.com/ident?i=beta) = 3 << [V\_PARAM\_SHIFT](http://lxr.free-electrons.com/ident?i=V_PARAM_SHIFT);

*/\* Veno variables \*/*

struct [veno](http://lxr.free-electrons.com/ident?i=veno) {

[u8](http://lxr.free-electrons.com/ident?i=u8) doing\_veno\_now; */\* if true, do veno for this rtt \*/*

[u16](http://lxr.free-electrons.com/ident?i=u16) cntrtt; */\* # of rtts measured within last rtt \*/*

[u32](http://lxr.free-electrons.com/ident?i=u32) minrtt; */\* min of rtts measured within last rtt (in usec) \*/*

[u32](http://lxr.free-electrons.com/ident?i=u32) basertt; */\* the min of all Veno rtt measurements seen (in usec) \*/*

[u32](http://lxr.free-electrons.com/ident?i=u32) [inc](http://lxr.free-electrons.com/ident?i=inc); */\* decide whether to increase cwnd \*/*

[u32](http://lxr.free-electrons.com/ident?i=u32) diff; */\* calculate the diff rate \*/*

};

**FUNCTION USED WITH DESCRIPTION:**

There are several situations when we must re-start Veno:

1. When a connection is established
2. After an RTO
3. After fast recovery
4. When we send a packet and there is no outstanding unacknowledged data (restarting an idle connection)
5. **Static void veno\_enable(struct sock \* sk)**

This function is used to enable TCP veno, by assigning veno\_doing\_now = 1.

1. **Static void veno\_disable(struct sock \*sk)**

This function turns off the Veno by assigning veno\_doing\_now = 0.

1. **Static void tcp\_veno\_init(struct sock \*sk)**

This function initialises Veno by initialising the value of

Veno->baseRTT = 0x7fffffff;

Veno->inc = 1;

Veno\_enable(sk);// by calling veno\_enable

/\* Do RTT sampling for Veno \*/

1. **Static void tcp\_veno\_pkts\_acked(struct sock \*sk, u32 cnt, s32 rtt\_us)**

This function finds the min rtt value during the last rtt value.

If(rtt\_us < 0)

Return;

/\* Never allow zero rtt or baseRTT\*/

Vrtt = rtt\_us+1;

/\* filter to find the propogation delay \*/

/\* find the min rtt during the last rtt to find the current propogation delay + queuing delay.

Veno->minrtt = min(veno->minrtt, vrtt);

Veno->cntrtt++;

1. **Static void tcp\_veno\_state(struct sock \*sk, u32 ca\_state)**

This function enables or disables the Veno based on the state of TCP.

If the connection is idle and we restarting, then we don’t want to do any Veno calculations until we get fresh rtt samples. So when we restart, we reset our Veno state to a clean state. After we get ACKs for this flight of packets, then we can make Veno calculations again.

1. **Static void tcp\_veno\_cwnd\_event(struct sock \*sk, u32 ack, s32 acked)**

This function initialises the Veno when we want to restart or start by checking the state.

1. **Static void tcp\_veno\_cong\_avoid(struct sock \*sk, u32 ack, s32 acked)**

This function mainly implements the Veno algorithm and sets the size of congestion window (cwnd) accordingly.

/\* if not doing Veno, then we’ll use TCP Reno \*/

If(!veno->doing\_veno\_now)

Tcp\_reno\_cong\_avoid(sk,ack, acked);

/\* we do the Veno calculations only if we get enough rtt samples. \*/

If(veno-<cntrtt <= 2)

/\* we do not have enough rtt samples to do the Veno calculation so we’ll behave like Reno \*/

Tcp\_reno\_cong\_avoid(sk, ack, acked);

Else

/\* we have enough rtt samples, so using the Veno algorithm, we determine the state of the network. \*/

If(tp->snd\_cwnd <= tp->snd\_ssthresh)

/\* slow start \*/

Tcp\_slow\_start(tp, acked);

Else

/\* congestion avoidance \*/

If(veno->diff < beta)

/\* in the “congestive state”, increase cwnd every other rtt. \*/

If(tp->snd\_cwnd < 2)

Tp->snd\_cwnd = 2;

/\* wipe the slate clean for the next rtt. \*/

/\* veno->rtt = 0;\*/

Veno->minrtt = ox7fffffff;

1. **Static u32 tcp\_veno\_ssthresh(struct sock \*sk)**

This is for the multiplicative decrease algorithm.