**TCP VEGAS**

TCP Vegas is a TCP congestion avoidance algorithm that emphasizes packet delay, rather than packet loss, as a signal to help determine the rate at which to send packets. It was developed at the University of Arizona by Lawrence Brakmo and Larry L. Peterson.

**ALGORITHM:**

TCP Vegas adopts a more sophisticated bandwidth estimation scheme. It uses the difference between expected and actual flow rates to estimate the available bandwidth in the network. The idea is that when the network is not congested, the actual flow rate will be close to the expected flow rate. Otherwise, the actual flow rate will be smaller than the expected flow rate. TCP Vegas, using this difference in flow rates, estimates the congestion level in the network and updates the window size accordingly. This difference in the flow rates can be easily translated into the difference between the window size and the number of acknowledged packets during the round trip time, using the equation,

Diff = (Expected – Actual) BaseRTT,

Where Expected is the expected rate, Actual is the actual rate, and BaseRTT is the minimum round trip time. The details of the algorithm are as follow:

1. First, the sender computes the expected flow rate Expected = CWND/BaseRTT,

Where CWND is the current window size and BaseRTT is the minimum round trip time.

1. Second, the sender estimates the current flow rate by using the actual round trip time according to Actual = CWND/RTT, where RTT is the actual round trip time of a packet.
2. The sender, using the expected and actual flow rates, computes the estimated backlog in the queue from diff = (Expected-Actual) BaseRTT.
3. Based on diff, the sender updates its window size as follows:

CWND = CWND+1 if diff < α

CWND = CWND-1 if diff > β

CWND otherwise

TCP Vegas tries to keep at least α packets but no more than β packets in the queues. The reason behind this is that TCP Vegas attempts to detect and utilize the extra bandwidth whenever it becomes available without congesting the network. This mechanism is fundamentally different from that used by TCP Reno. TCP Reno always updates its window size to guarantee full utilization of available bandwidth, leading to constant packet losses, whereas TCP Vegas does not cause any oscillation in window size once it converges to an equilibrium point.

**IMPLEMENTATION:**

Header files included:

#include <linux/mm.h>

#include <linux/module.h>

#include <linux/skbuff.h>

#include <linux/inet\_diag.h>

#include <net/tcp.h>

#include "[tcp\_vegas.h](http://lxr.free-electrons.com/source/net/ipv4/tcp_vegas.h)"

HEADER FILE TCP.H INCLUDES FOLLOWING THINGS:

#ifndef [\_\_TCP\_VEGAS\_H](http://lxr.free-electrons.com/ident?i=__TCP_VEGAS_H)

#define [\_\_TCP\_VEGAS\_H](http://lxr.free-electrons.com/ident?i=__TCP_VEGAS_H) 1

*/\* Vegas variables \*/*

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

[u32](http://lxr.free-electrons.com/ident?i=u32) beg\_snd\_nxt; */\* right edge during last RTT \*/*

[u32](http://lxr.free-electrons.com/ident?i=u32) beg\_snd\_una; */\* left edge during last RTT \*/*

[u32](http://lxr.free-electrons.com/ident?i=u32) beg\_snd\_cwnd; */\* saves the size of the cwnd \*/*

[u8](http://lxr.free-electrons.com/ident?i=u8) doing\_vegas\_now;*/\* if true, do vegas 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 Vegas RTT measurements seen (in usec) \*/*

};

void [tcp\_vegas\_init](http://lxr.free-electrons.com/ident?i=tcp_vegas_init)(struct [sock](http://lxr.free-electrons.com/ident?i=sock) \*sk);

void [tcp\_vegas\_state](http://lxr.free-electrons.com/ident?i=tcp_vegas_state)(struct [sock](http://lxr.free-electrons.com/ident?i=sock) \*sk, [u8](http://lxr.free-electrons.com/ident?i=u8) ca\_state);

void [tcp\_vegas\_pkts\_acked](http://lxr.free-electrons.com/ident?i=tcp_vegas_pkts_acked)(struct [sock](http://lxr.free-electrons.com/ident?i=sock) \*sk, [u32](http://lxr.free-electrons.com/ident?i=u32) [cnt](http://lxr.free-electrons.com/ident?i=cnt), [s32](http://lxr.free-electrons.com/ident?i=s32) rtt\_us);

void [tcp\_vegas\_cwnd\_event](http://lxr.free-electrons.com/ident?i=tcp_vegas_cwnd_event)(struct [sock](http://lxr.free-electrons.com/ident?i=sock) \*sk, enum [tcp\_ca\_event](http://lxr.free-electrons.com/ident?i=tcp_ca_event) [event](http://lxr.free-electrons.com/ident?i=event));

void [tcp\_vegas\_get\_info](http://lxr.free-electrons.com/ident?i=tcp_vegas_get_info)(struct [sock](http://lxr.free-electrons.com/ident?i=sock) \*sk, [u32](http://lxr.free-electrons.com/ident?i=u32) [ext](http://lxr.free-electrons.com/ident?i=ext), struct [sk\_buff](http://lxr.free-electrons.com/ident?i=sk_buff) \*[skb](http://lxr.free-electrons.com/ident?i=skb));

#endif */\* \_\_TCP\_VEGAS\_H \*/*

FUNCTIONS USED WITH DESCRITPTION:

1. static void [vegas\_enable](http://lxr.free-electrons.com/ident?i=vegas_enable)(struct [sock](http://lxr.free-electrons.com/ident?i=sock) \*sk)

This function enables the VEGAS algorithm.

*/\* Begin taking Vegas sample*

[vegas](http://lxr.free-electrons.com/ident?i=vegas)->doing\_vegas\_now = 1;

*/\* Set the beginning of the next send window. \*/*

[vegas](http://lxr.free-electrons.com/ident?i=vegas)->beg\_snd\_nxt = [tp](http://lxr.free-electrons.com/ident?i=tp)->snd\_nxt;

1. static inline void [vegas\_disable](http://lxr.free-electrons.com/ident?i=vegas_disable)(struct [sock](http://lxr.free-electrons.com/ident?i=sock) \*sk)

this function disables the VEGAS

*Stop taking Vegas samples for now.*

[vegas](http://lxr.free-electrons.com/ident?i=vegas)->doing\_vegas\_now = 0;

1. void [tcp\_vegas\_init](http://lxr.free-electrons.com/ident?i=tcp_vegas_init)(struct [sock](http://lxr.free-electrons.com/ident?i=sock) \*sk)

initialises the value of the baseRTT value to value of int.

*/\* Do RTT sampling needed for Vegas.*

*\* Basically we:*

*\* o min-filter RTT samples from within an RTT to get the current*

*\* propagation delay + queuing delay (we are min-filtering to try to*

*\* avoid the effects of delayed ACKs)*

*\* o min-filter RTT samples from a much longer window (forever for now)*

*\* to find the propagation delay (baseRTT)*

*\*/*

1. void [tcp\_vegas\_pkts\_acked](http://lxr.free-electrons.com/ident?i=tcp_vegas_pkts_acked)(struct [sock](http://lxr.free-electrons.com/ident?i=sock) \*sk, [u32](http://lxr.free-electrons.com/ident?i=u32) [cnt](http://lxr.free-electrons.com/ident?i=cnt), [s32](http://lxr.free-electrons.com/ident?i=s32) rtt\_us)

In this function we find the value of minimum BaseRTT. We never allow zero value of baseRTT, if so give it the value 1.

vrtt = rtt\_us + 1;

*Filter to find propagation delay:*

if (vrtt < [vegas](http://lxr.free-electrons.com/ident?i=vegas)->baseRTT)

[vegas](http://lxr.free-electrons.com/ident?i=vegas)->baseRTT = vrtt;

*Find the minimum RTT during the last RTT to find the current propogation delay+ queuing delay*

[vegas](http://lxr.free-electrons.com/ident?i=vegas)->minRTT = [min](http://lxr.free-electrons.com/ident?i=min)([vegas](http://lxr.free-electrons.com/ident?i=vegas)->minRTT, vrtt);

[vegas](http://lxr.free-electrons.com/ident?i=vegas)->cntRTT++;

1. void [tcp\_vegas\_state](http://lxr.free-electrons.com/ident?i=tcp_vegas_state)(struct [sock](http://lxr.free-electrons.com/ident?i=sock) \*sk, [u8](http://lxr.free-electrons.com/ident?i=u8) ca\_state)

This function enables and disables the VEGAS according to the state.

If the connection is idle and we are restarting then we’ll not do any vegas calculations until we get any fresh RTT samples. So when we restart, we reset our Vegas state to a clean state. After we get acks for this flight of packets, then we can make Vegas calculations again.

1. void [tcp\_vegas\_cwnd\_event](http://lxr.free-electrons.com/ident?i=tcp_vegas_cwnd_event)(struct [sock](http://lxr.free-electrons.com/ident?i=sock) \*sk, enum [tcp\_ca\_event](http://lxr.free-electrons.com/ident?i=tcp_ca_event) [event](http://lxr.free-electrons.com/ident?i=event))

This function initialises the Vegas when we want to start or restart.

1. static inline [u32](http://lxr.free-electrons.com/ident?i=u32) [tcp\_vegas\_ssthresh](http://lxr.free-electrons.com/ident?i=tcp_vegas_ssthresh)(struct [tcp\_sock](http://lxr.free-electrons.com/ident?i=tcp_sock) \*[tp](http://lxr.free-electrons.com/ident?i=tp))

This function returns the threshold value.

1. static void [tcp\_vegas\_cong\_avoid](http://lxr.free-electrons.com/ident?i=tcp_vegas_cong_avoid)(struct [sock](http://lxr.free-electrons.com/ident?i=sock) \*sk, [u32](http://lxr.free-electrons.com/ident?i=u32) [ack](http://lxr.free-electrons.com/ident?i=ack), [u32](http://lxr.free-electrons.com/ident?i=u32) acked)

This function applies the main congestion avoidance algorithm of TCP Vegas.

If we do not apply Vegas, then we’ll implement TCP Reno.

If we are using Vegas,

//Do the Vegas once-per RTT cwnd adjustment. Save the extent of the current window so we can use this at the end of the next RTT.//

/\* we do the Vegas calculations only if we get enough RTT samples that we can reasonably sure that we got at least one RTT sample that wasn’t from delayed ACK.

If we had 2 samples total, which means we are getting only 1 ACK per RTT, which means they are almost delayed ACK’s.

If we have 3 samples we are OK.

**if ([vegas](http://lxr.free-electrons.com/ident?i=vegas)->cntRTT <= 2)**

[**tcp\_reno\_cong\_avoid**](http://lxr.free-electrons.com/ident?i=tcp_reno_cong_avoid)**(sk,** [**ack**](http://lxr.free-electrons.com/ident?i=ack)**, acked);**

//if 2 samples we act like RENO

**else {**

**// we have enough RTT samples, so using the Vegas algorithm, if**

**we should increase or decrease cwnd, and by how much.**

**Pluck out the RTT we are using for the Vegas calculations. This is the min RTT seen during the last RTT. Taking the min filters out the effects of delayed ACKs, at the cost of noticing congestion a bit later.**

**diff =** [**tp**](http://lxr.free-electrons.com/ident?i=tp)**->snd\_cwnd \* (rtt-**[**vegas**](http://lxr.free-electrons.com/ident?i=vegas)**->baseRTT) /** [**vegas**](http://lxr.free-electrons.com/ident?i=vegas)**->baseRTT;**

**if (diff >** [**gamma**](http://lxr.free-electrons.com/ident?i=gamma) **&&** [**tp**](http://lxr.free-electrons.com/ident?i=tp)**->snd\_cwnd <=** [**tp**](http://lxr.free-electrons.com/ident?i=tp)**->snd\_ssthresh) {**

***/\* Going too fast. Time to slow down***

***\* and switch to congestion avoidance.***

***\*/***

***/\* Set cwnd to match the actual rate***

***\* exactly:***

***\* cwnd = (actual rate) \* baseRTT***

***\* Then we add 1 because the integer***

***\* truncation robs us of full link***

***\* utilization.***

***\*/***

[**tp**](http://lxr.free-electrons.com/ident?i=tp)**->snd\_cwnd =** [**min**](http://lxr.free-electrons.com/ident?i=min)**([tp](http://lxr.free-electrons.com/ident?i=tp)->snd\_cwnd, (**[**u32**](http://lxr.free-electrons.com/ident?i=u32)**)target\_cwnd+1);**

[**tp**](http://lxr.free-electrons.com/ident?i=tp)**->snd\_ssthresh =** [**tcp\_vegas\_ssthresh**](http://lxr.free-electrons.com/ident?i=tcp_vegas_ssthresh)**([tp](http://lxr.free-electrons.com/ident?i=tp));**

**} else if ([tp](http://lxr.free-electrons.com/ident?i=tp)->snd\_cwnd <=** [**tp**](http://lxr.free-electrons.com/ident?i=tp)**->snd\_ssthresh) {**

***/\* Slow start. \*/***

[**tcp\_slow\_start**](http://lxr.free-electrons.com/ident?i=tcp_slow_start)**([tp](http://lxr.free-electrons.com/ident?i=tp), acked);**

**} else {**

***/\* Congestion avoidance. \*/***

***/\* Figure out where we would like cwnd***

***\* to be.***

***\*/***

**if (diff >** [**beta**](http://lxr.free-electrons.com/ident?i=beta)**) {**

***/\* The old window was too fast, so***

***\* we slow down.***

***\*/***

[**tp**](http://lxr.free-electrons.com/ident?i=tp)**->snd\_cwnd--;**

[**tp**](http://lxr.free-electrons.com/ident?i=tp)**->snd\_ssthresh**

**=** [**tcp\_vegas\_ssthresh**](http://lxr.free-electrons.com/ident?i=tcp_vegas_ssthresh)**([tp](http://lxr.free-electrons.com/ident?i=tp));**

**} else if (diff <** [**alpha**](http://lxr.free-electrons.com/ident?i=alpha)**) {**

***/\* We don't have enough extra packets***

***\* in the network, so speed up.***

***\*/***

[**tp**](http://lxr.free-electrons.com/ident?i=tp)**->snd\_cwnd++;**

**} else {**

***/\* Sending just as fast as we***

***\* should be.***

***\*/***

**}**

**}**

**if ([tp](http://lxr.free-electrons.com/ident?i=tp)->snd\_cwnd < 2)**

[**tp**](http://lxr.free-electrons.com/ident?i=tp)**->snd\_cwnd = 2;**

**else if ([tp](http://lxr.free-electrons.com/ident?i=tp)->snd\_cwnd >** [**tp**](http://lxr.free-electrons.com/ident?i=tp)**->snd\_cwnd\_clamp)**

[**tp**](http://lxr.free-electrons.com/ident?i=tp)**->snd\_cwnd =** [**tp**](http://lxr.free-electrons.com/ident?i=tp)**->snd\_cwnd\_clamp;**

[**tp**](http://lxr.free-electrons.com/ident?i=tp)**->snd\_ssthresh =** [**tcp\_current\_ssthresh**](http://lxr.free-electrons.com/ident?i=tcp_current_ssthresh)**(sk);**

**}**

***/\* Wipe the slate clean for the next RTT. \*/***

[**vegas**](http://lxr.free-electrons.com/ident?i=vegas)**->cntRTT = 0;**

[**vegas**](http://lxr.free-electrons.com/ident?i=vegas)**->minRTT = 0x7fffffff;**

**}**

***/\* Use normal slow start \*/***

**else if ([tp](http://lxr.free-electrons.com/ident?i=tp)->snd\_cwnd <=** [**tp**](http://lxr.free-electrons.com/ident?i=tp)**->snd\_ssthresh)**

[**tcp\_slow\_start**](http://lxr.free-electrons.com/ident?i=tcp_slow_start)**([tp](http://lxr.free-electrons.com/ident?i=tp), acked);**