**TCP WESTWOOD:**

TCP Westwood is a sender-side only modification of the TCP Reno/NewReno classic congestion control protocol stack that optimizes the performance of TCP congestion control especially over wireless networks. TCPW is based on end-to-end bandwidth estimation to set congestion window and slow start threshold after a congestion episode, that is, after three duplicate acknowledgments or a timeout.

**ALGORITHM:**

The pseudo code of Westwood+ algorithm is reported below:

1. on ACK reception: cwnd is increased accordingly to the Reno algorithm; the end-to-end bandwidth estimate BWE is computed;

2. when 3 DUPACKs are received: ssthresh=max (2,(BWE\*RTTmin)/seg\_size);cwnd=ssthresh;

3. when coarse timeout expires: ssthresh=max (2, (BWE\*RTTmin)/seg\_size); cwnd=1.

**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>**

**VARIABLES USED:**

***\* TCP Westwood structure \*/***

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

[**u32**](http://lxr.free-electrons.com/ident?i=u32) **bw\_ns\_est; */\* first bandwidth estimation..not too smoothed 8) \*/***

[**u32**](http://lxr.free-electrons.com/ident?i=u32) **bw\_est; */\* bandwidth estimate \*/***

[**u32**](http://lxr.free-electrons.com/ident?i=u32) **rtt\_win\_sx; */\* here starts a new evaluation... \*/***

[**u32**](http://lxr.free-electrons.com/ident?i=u32)[**bk**](http://lxr.free-electrons.com/ident?i=bk)**;**

[**u32**](http://lxr.free-electrons.com/ident?i=u32) **snd\_una; */\* used for evaluating the number of acked bytes \*/***

[**u32**](http://lxr.free-electrons.com/ident?i=u32) **cumul\_ack;**

[**u32**](http://lxr.free-electrons.com/ident?i=u32) **accounted;**

[**u32**](http://lxr.free-electrons.com/ident?i=u32) **rtt;**

[**u32**](http://lxr.free-electrons.com/ident?i=u32) **rtt\_min; */\* minimum observed RTT \*/***

[**u8**](http://lxr.free-electrons.com/ident?i=u8) **first\_ack; */\* flag which infers that this is the first ack \*/***

[**u8**](http://lxr.free-electrons.com/ident?i=u8) **reset\_rtt\_min; */\* Reset RTT min to next RTT sample\*/***

**};**

**#define** [**TCP\_WESTWOOD\_RTT\_MIN**](http://lxr.free-electrons.com/ident?i=TCP_WESTWOOD_RTT_MIN) **(**[**HZ**](http://lxr.free-electrons.com/ident?i=HZ)**/20) */\* 50ms \*/***

**#define** [**TCP\_WESTWOOD\_INIT\_RTT**](http://lxr.free-electrons.com/ident?i=TCP_WESTWOOD_INIT_RTT) **(20\***[**HZ**](http://lxr.free-electrons.com/ident?i=HZ)**) */\* maybe too conservative?! \*/***

**FUNCTIONS USED WITH DESCRIPTION:**

🡺From this function the algorithm begins:-

1)static void tcp\_westwood\_init(struct sock \*sk)

{

struct westwood \*w = inet\_csk\_ca(sk);

w‐>bk = 0;

w‐>bw\_ns\_est = 0;

w‐>bw\_est = 0;

w‐>accounted = 0;

w‐>cumul\_ack = 0;

w‐>reset\_rtt\_min = 1;

w‐>rtt\_min = w‐>rtt = TCP\_WESTWOOD\_INIT\_RTT;

w‐>rtt\_win\_sx = tcp\_time\_stamp;

w‐>snd\_una = tcp\_sk(sk)‐>snd\_una;

w‐>first\_ack = 1;

}

🡺 Low‐pass filter. Implemented using constant coefficients.

2) static inline u32 westwood\_do\_filter(u32 a, u32 b)

{

return ((7 \* a) + b) >> 3;

}

3) static void westwood\_filter(struct westwood \*w, u32 delta)

{

*/\* If the filter is empty fill it with the first sample of bandwidth \*/*

if (w‐>bw\_ns\_est == 0 && w‐>bw\_est == 0) {

w‐>bw\_ns\_est = w‐>bk / delta;

w‐>bw\_est = w‐>bw\_ns\_est;

} else {

w‐>bw\_ns\_est = westwood\_do\_filter(w‐>bw\_ns\_est, w‐>bk / delta);

w‐>bw\_est = westwood\_do\_filter(w‐>bw\_est, w‐>bw\_ns\_est);

}

}

🡺westwood\_pkts\_acked Called after processing group of packets.but all westwood needs is the last sample of srtt.

4) static void tcp\_westwood\_pkts\_acked(struct sock \*sk, u32 cnt, s32 rtt)

{

struct westwood \*w = inet\_csk\_ca(sk);

if (rtt > 0)

w‐>rtt = usecs\_to\_jiffies(rtt);

}

🡺westwood\_update\_window

It updates RTT evaluation window if it is the right moment to do

it. If so it calls filter for evaluating bandwidth.

5) static void westwood\_update\_window(struct sock \*sk)

{

struct westwood \*w = inet\_csk\_ca(sk);

s32 delta = tcp\_time\_stamp ‐ w‐>rtt\_win\_sx;

🡺 Initialize w‐>snd\_una with the first acked sequence number in order

to fix mismatch between tp‐>snd\_una and w‐>snd\_una for the first

bandwidth sample

if (w‐>first\_ack) {

w‐>snd\_una = tcp\_sk(sk)‐>snd\_una;

w‐>first\_ack = 0;

}

🡺 See if a RTT‐window has passed.

Be careful since if RTT is less than

50ms we don't filter but we continue 'building the sample'.

This minimum limit was chosen since an estimation on small

time intervals is better to avoid...

Obviously on a LAN we reasonably will always have

right\_bound = left\_bound + WESTWOOD\_RTT\_MIN

if (w‐>rtt && delta > max\_t(u32, w‐>rtt, TCP\_WESTWOOD\_RTT\_MIN)) {

westwood\_filter(w, delta);

w‐>bk = 0;

w‐>rtt\_win\_sx = tcp\_time\_stamp;

}

}

6) static inline void update\_rtt\_min(struct westwood \*w)

{

if (w‐>reset\_rtt\_min) {

w‐>rtt\_min = w‐>rtt;

w‐>reset\_rtt\_min = 0;

} else

w‐>rtt\_min = min(w‐>rtt, w‐>rtt\_min);

}

🡺westwood\_fast\_bw

It is called when we are in fast path. In particular it is called when

header prediction is successful. In such case in fact update is

straight forward and doesn't need any particular care.

7) static inline void westwood\_fast\_bw(struct sock \*sk)

{

const struct tcp\_sock \*tp = tcp\_sk(sk);

struct westwood \*w = inet\_csk\_ca(sk);

westwood\_update\_window(sk);

w‐>bk += tp‐>snd\_una ‐ w‐>snd\_una;

w‐>snd\_una = tp‐>snd\_una;

update\_rtt\_min(w);

}

🡺westwood\_acked\_count

This function evaluates cumul\_ack for evaluating bk in case of

delayed or partial acks.

8) static inline u32 westwood\_acked\_count(struct sock \*sk)

{

const struct tcp\_sock \*tp = tcp\_sk(sk);

struct westwood \*w = inet\_csk\_ca(sk);

w‐>cumul\_ack = tp‐>snd\_una ‐ w‐>snd\_una;

If cumul\_ack is 0 this is a dupack since it's not moving

tp‐>snd\_una.

if (!w‐>cumul\_ack) {

w‐>accounted += tp‐>mss\_cache;

w‐>cumul\_ack = tp‐>mss\_cache;

}

if (w‐>cumul\_ack > tp‐>mss\_cache) {

🡺Partial or delayed ack

if (w‐>accounted >= w‐>cumul\_ack) {

w‐>accounted ‐= w‐>cumul\_ack;

w‐>cumul\_ack = tp‐>mss\_cache;

} else {

w‐>cumul\_ack ‐= w‐>accounted;

w‐>accounted = 0;

}

}

w‐>snd\_una = tp‐>snd\_una;

return w‐>cumul\_ack;

}

🡺 TCP Westwood

Here limit is evaluated as Bw estimation\*RTTmin (for obtaining it

in packets we use mss\_cache). Rttmin is guaranteed to be >= 2

so avoids ever returning 0.

9) static u32 tcp\_westwood\_bw\_rttmin(const struct sock \*sk)

{

const struct tcp\_sock \*tp = tcp\_sk(sk);

const struct westwood \*w = inet\_csk\_ca(sk);

return max\_t(u32, (w‐>bw\_est \* w‐>rtt\_min) / tp‐>mss\_cache, 2);

}

10) static void tcp\_westwood\_event(struct sock \*sk, enum tcp\_ca\_event event)

{

struct tcp\_sock \*tp = tcp\_sk(sk);

struct westwood \*w = inet\_csk\_ca(sk);

switch (event) {

case CA\_EVENT\_FAST\_ACK:

westwood\_fast\_bw(sk);

break;

case CA\_EVENT\_COMPLETE\_CWR:

tp‐>snd\_cwnd = tp‐>snd\_ssthresh = tcp\_westwood\_bw\_rttmin(sk);

break;

case CA\_EVENT\_LOSS:

tp‐>snd\_ssthresh = tcp\_westwood\_bw\_rttmin(sk);

🡺Update RTT\_min when next ack arrives

w‐>reset\_rtt\_min = 1;

break;

case CA\_EVENT\_SLOW\_ACK:

westwood\_update\_window(sk);

w‐>bk += westwood\_acked\_count(sk);

update\_rtt\_min(w);

break;

default:

*/\* don't care \*/*

break;

}

}

🡺Extract info for Tcp socket info provided via netlink

11) static void tcp\_westwood\_info(struct sock \*sk, u32 ext,

struct sk\_buff \*skb)

{

const struct westwood \*ca = inet\_csk\_ca(sk);

if (ext & (1 << (INET\_DIAG\_VEGASINFO ‐ 1))) {

struct tcpvegas\_info info = {

.tcpv\_enabled = 1,

.tcpv\_rtt = jiffies\_to\_usecs(ca‐>rtt),

.tcpv\_minrtt = jiffies\_to\_usecs(ca‐>rtt\_min),

};

nla\_put(skb, INET\_DIAG\_VEGASINFO, sizeof(info), &info);

}

}

12)static struct tcp\_congestion\_ops tcp\_westwood \_\_read\_mostly = {

.init = tcp\_westwood\_init,

.ssthresh = tcp\_reno\_ssthresh,

.cong\_avoid = tcp\_reno\_cong\_avoid,

.cwnd\_event = tcp\_westwood\_event,

.get\_info = tcp\_westwood\_info,

.pkts\_acked = tcp\_westwood\_pkts\_acked,

.owner = THIS\_MODULE,

.name = *"westwood"*

};

13) static int \_\_init tcp\_westwood\_register(void)

{

BUILD\_BUG\_ON(sizeof(struct westwood) > ICSK\_CA\_PRIV\_SIZE);

return tcp\_register\_congestion\_control(&tcp\_westwood);

}

14) static void \_\_exit tcp\_westwood\_unregister(void)

{

tcp\_unregister\_congestion\_control(&tcp\_westwood);

}

module\_init(tcp\_westwood\_register);

module\_exit(tcp\_westwood\_unregister);

MODULE\_AUTHOR(*"Stephen Hemminger, Angelo Dell'Aera"*);

MODULE\_LICENSE(*"GPL"*);

MODULE\_DESCRIPTION(*"TCP Westwood+"*);