Linux Cross Reference

Free Electrons

Embedded Linux Experts

• *source navigation* • diff markup • identifier search • freetext search •

Version:
2.0.40 2.2.26 2.4.37 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 3.10 3.11 3.12 3.13 3.14 3.15 3.16 *3.17*

Linux/net/ipv4/tcp_hybla.c

```
/*
 * TCP HYBLA
 *
 * TCP-HYBLA Congestion control algorithm, based on:
 *   C.Caini, R.Firrincieli, "TCP-Hybla: A TCP Enhancement
 *   for Heterogeneous Networks",
 *   International Journal on satellite Communications,
 *                                      September 2004
 *    Daniele Lacamera
 *    root at danielinux.net
 */

#include <linux/module.h>
#include <net/tcp.h>

/* Tcp Hybla structure. */
struct hybla {
        bool  hybla_en;
        u32   snd_cwnd_cents; /* Keeps increment values when it is <1, <<7 */
        u32   rho;            /* Rho parameter, integer part  */
        u32   rho2;           /* Rho * Rho, integer part */
        u32   rho_3ls;        /* Rho parameter, <<3 */
        u32   rho2_7ls;       /* Rho^2, <<7     */
        u32   minrtt_us;      /* Minimum smoothed round trip time value seen */
};

/* Hybla reference round trip time (default= 1/40 sec = 25 ms), in ms */
static int rtt0 = 25;
module_param(rtt0, int, 0644);
MODULE_PARM_DESC(rtt0, "reference rout trip time (ms)");


/* This is called to refresh values for hybla parameters */
static inline void hybla_recalc_param (struct sock *sk)
{
        struct hybla *ca = inet_csk_ca(sk);

        ca->rho_3ls = max_t(u32,
                            tcp_sk(sk)->srtt_us / (rtt0 * USEC_PER_MSEC),
                            8U);
        ca->rho = ca->rho_3ls >> 3;
        ca->rho2_7ls = (ca->rho_3ls * ca->rho_3ls) << 1;
        ca->rho2 = ca->rho2_7ls >> 7;
```

```
}

static void hybla_init(struct sock *sk)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct hybla *ca = inet_csk_ca(sk);

        ca->rho = 0;
        ca->rho2 = 0;
        ca->rho_3ls = 0;
        ca->rho2_7ls = 0;
        ca->snd_cwnd_cents = 0;
        ca->hybla_en = true;
        tp->snd_cwnd = 2;
        tp->snd_cwnd_clamp = 65535;

        /* 1st Rho measurement based on initial srtt */
        hybla_recalc_param(sk);

        /* set minimum rtt as this is the 1st ever seen */
        ca->minrtt_us = tp->srtt_us;
        tp->snd_cwnd = ca->rho;
}

static void hybla_state(struct sock *sk, u8 ca_state)
{
        struct hybla *ca = inet_csk_ca(sk);

        ca->hybla_en = (ca_state == TCP_CA_Open);
}

static inline u32 hybla_fraction(u32 odds)
{
        static const u32 fractions[] = {
                128, 139, 152, 165, 181, 197, 215, 234,
        };

        return (odds < ARRAY_SIZE(fractions)) ? fractions[odds] : 128;
}

/* TCP Hybla main routine.
 * This is the algorithm behavior:
 *     o Recalc Hybla parameters if min_rtt has changed
 *     o Give cwnd a new value based on the model proposed
 *     o remember increments <1
 */
static void hybla_cong_avoid(struct sock *sk, u32 ack, u32 acked)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct hybla *ca = inet_csk_ca(sk);
        u32 increment, odd, rho_fractions;
        int is_slowstart = 0;

        /*  Recalculate rho only if this srtt is the lowest */
        if (tp->srtt_us < ca->minrtt_us) {
                hybla_recalc_param(sk);
                ca->minrtt_us = tp->srtt_us;
        }

        if (!tcp_is_cwnd_limited(sk))
                return;

        if (!ca->hybla_en) {
                tcp_reno_cong_avoid(sk, ack, acked);
                return;
        }

        if (ca->rho == 0)
                hybla_recalc_param(sk);

        rho_fractions = ca->rho_3ls - (ca->rho << 3);

        if (tp->snd_cwnd < tp->snd_ssthresh) {
                /*
                 * slow start
                 *      INC = 2^RHO - 1
                 * This is done by splitting the rho parameter
                 * into 2 parts: an integer part and a fraction part.
                 * Inrement<<7 is estimated by doing:
                 *             [2^(int+fract)]<<7
                 * that is equal to:
                 *             (2^int)  *  [(2^fract) <<7]
                 * 2^int is straightly computed as 1<<int,
                 * while we will use hybla_slowstart_fraction_increment() to
                 * calculate 2^fract in a <<7 value.
                 */
                is_slowstart = 1;
                increment = ((1 << min(ca->rho, 16U)) *
                        hybla_fraction(rho_fractions)) - 128;
        } else {
                /*
                 * congestion avoidance
                 * INC = RHO^2 / W
                 * as long as increment is estimated as (rho<<7)/window
                 * it already is <<7 and we can easily count its fractions.
                 */
                increment = ca->rho2_7ls / tp->snd_cwnd;
                if (increment < 128)
                        tp->snd_cwnd_cnt++;
        }

        odd = increment % 128;
        tp->snd_cwnd += increment >> 7;
        ca->snd_cwnd_cents += odd;

        /* check when fractions goes >=128 and increase cwnd by 1. */
        while (ca->snd_cwnd_cents >= 128) {
                tp->snd_cwnd++;
                ca->snd_cwnd_cents -= 128;
                tp->snd_cwnd_cnt = 0;
        }
        /* check when cwnd has not been incremented for a while */
        if (increment == 0 && odd == 0 && tp->snd_cwnd_cnt >= tp->snd_cwnd) {
                tp->snd_cwnd++;
                tp->snd_cwnd_cnt = 0;
        }
        /* clamp down slowstart cwnd to ssthresh value. */
        if (is_slowstart)
                tp->snd_cwnd = min(tp->snd_cwnd, tp->snd_ssthresh);

        tp->snd_cwnd = min_t(u32, tp->snd_cwnd, tp->snd_cwnd_clamp);
}

static struct tcp_congestion_ops tcp_hybla __read_mostly = {
        .init           = hybla_init,
        .ssthresh       = tcp_reno_ssthresh,
        .cong_avoid     = hybla_cong_avoid,
        .set_state      = hybla_state,

        .owner          = THIS_MODULE,
        .name           = "hybla"
};

static int __init hybla_register(void)
{
        BUILD_BUG_ON(sizeof(struct hybla) > ICSK_CA_PRIV_SIZE);
        return tcp_register_congestion_control(&tcp_hybla);
}

static void __exit hybla_unregister(void)
{
        tcp_unregister_congestion_control(&tcp_hybla);
}

module_init(hybla_register);
module_exit(hybla_unregister);

MODULE_AUTHOR("Daniele Lacamera");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("TCP Hybla");

```