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_bic.c

```
/*
 * Binary Increase Congestion control for TCP
 * Home page:
 *      http://netsrv.csc.ncsu.edu/twiki/bin/view/Main/BIC
 * This is from the implementation of BICTCP in
 * Lison-Xu, Kahaled Harfoush, and Injong Rhee.
 *  "Binary Increase Congestion Control for Fast, Long Distance
 *  Networks" in InfoComm 2004
 * Available from:
 *  http://netsrv.csc.ncsu.edu/export/bitcp.pdf
 *
 * Unless BIC is enabled and congestion window is large
 * this behaves the same as the original Reno.
 */

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


#define BICTCP_BETA_SCALE    1024       /* Scale factor beta calculation
                                         * max_cwnd = snd_cwnd * beta
                                         */
#define BICTCP_B                4        /*
                                          * In binary search,
                                          * go to point (max+min)/N
                                          */

static int fast_convergence = 1;
static int max_increment = 16;
static int low_window = 14;
static int beta = 819;          /* = 819/1024 (BICTCP_BETA_SCALE) */
static int initial_ssthresh;
static int smooth_part = 20;

module_param(fast_convergence, int, 0644);
MODULE_PARM_DESC(fast_convergence, "turn on/off fast convergence");
module_param(max_increment, int, 0644);
MODULE_PARM_DESC(max_increment, "Limit on increment allowed during binary search");
module_param(low_window, int, 0644);
MODULE_PARM_DESC(low_window, "lower bound on congestion window (for TCP friendliness)");
module_param(beta, int, 0644);
MODULE_PARM_DESC(beta, "beta for multiplicative increase");
module_param(initial_ssthresh, int, 0644);
MODULE_PARM_DESC(initial_ssthresh, "initial value of slow start threshold");
module_param(smooth_part, int, 0644);
MODULE_PARM_DESC(smooth_part, "log(B/(B*Smin))/log(B/(B-1))+B, # of RTT from Wmax-B to Wmax");

```

```

/* BIC TCP Parameters */
struct bictcp {
        u32     cnt;            /* increase cwnd by 1 after ACKs */
        u32     last_max_cwnd;  /* last maximum snd_cwnd */
        u32     loss_cwnd;      /* congestion window at last loss */
        u32     last_cwnd;      /* the last snd_cwnd */
        u32     last_time;      /* time when updated last_cwnd */
        u32     epoch_start;    /* beginning of an epoch */
#define ACK_RATIO_SHIFT 4
        u32     delayed_ack;    /* estimate the ratio of Packets/ACKs << 4 */
};

static inline void bictcp_reset(struct bictcp *ca)
{
        ca->cnt = 0;
        ca->last_max_cwnd = 0;
        ca->last_cwnd = 0;
        ca->last_time = 0;
        ca->epoch_start = 0;
        ca->delayed_ack = 2 << ACK_RATIO_SHIFT;
}

static void bictcp_init(struct sock *sk)
{
        struct bictcp *ca = inet_csk_ca(sk);

        bictcp_reset(ca);
        ca->loss_cwnd = 0;

        if (initial_ssthresh)
                tcp_sk(sk)->snd_ssthresh = initial_ssthresh;
}

/*
 * Compute congestion window to use.
 */
static inline void bictcp_update(struct bictcp *ca, u32 cwnd)
{
        if (ca->last_cwnd == cwnd &&
            (s32)(tcp_time_stamp - ca->last_time) <= HZ / 32)
                return;

        ca->last_cwnd = cwnd;
        ca->last_time = tcp_time_stamp;

        if (ca->epoch_start == 0) /* record the beginning of an epoch */
                ca->epoch_start = tcp_time_stamp;

        /* start off normal */
        if (cwnd <= low_window) {
                ca->cnt = cwnd;
                return;
        }

        /* binary increase */
        if (cwnd < ca->last_max_cwnd) {
                __u32   dist = (ca->last_max_cwnd - cwnd)
                        / BICTCP_B;

                if (dist > max_increment)
                        /* linear increase */
                        ca->cnt = cwnd / max_increment;
                else if (dist <= 1U)
                        /* binary search increase */
                        ca->cnt = (cwnd * smooth_part) / BICTCP_B;
                else
                        /* binary search increase */
                        ca->cnt = cwnd / dist;
        } else {
                /* slow start AMD linear increase */
                if (cwnd < ca->last_max_cwnd + BICTCP_B)
                        /* slow start */
                        ca->cnt = (cwnd * smooth_part) / BICTCP_B;
                else if (cwnd < ca->last_max_cwnd + max_increment*(BICTCP_B-1))
                        /* slow start */
                        ca->cnt = (cwnd * (BICTCP_B-1))
                                / (cwnd - ca->last_max_cwnd);
                else
                        /* linear increase */
                        ca->cnt = cwnd / max_increment;
        }

        /* if in slow start or link utilization is very low */
        if (ca->last_max_cwnd == 0) {
                if (ca->cnt > 20) /* increase cwnd 5% per RTT */
                        ca->cnt = 20;
        }

        ca->cnt = (ca->cnt << ACK_RATIO_SHIFT) / ca->delayed_ack;
        if (ca->cnt == 0)                       /* cannot be zero */
                ca->cnt = 1;
}

static void bictcp_cong_avoid(struct sock *sk, u32 ack, u32 acked)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct bictcp *ca = inet_csk_ca(sk);

        if (!tcp_is_cwnd_limited(sk))
                return;

        if (tp->snd_cwnd <= tp->snd_ssthresh)
                tcp_slow_start(tp, acked);
        else {
                bictcp_update(ca, tp->snd_cwnd);
                tcp_cong_avoid_ai(tp, ca->cnt);
        }

}

/*
 *      behave like Reno until low_window is reached,
 *      then increase congestion window slowly
 */
static u32 bictcp_recalc_ssthresh(struct sock *sk)
{
        const struct tcp_sock *tp = tcp_sk(sk);
        struct bictcp *ca = inet_csk_ca(sk);

        ca->epoch_start = 0;    /* end of epoch */

        /* Wmax and fast convergence */
        if (tp->snd_cwnd < ca->last_max_cwnd && fast_convergence)
                ca->last_max_cwnd = (tp->snd_cwnd * (BICTCP_BETA_SCALE + beta))
                        / (2 * BICTCP_BETA_SCALE);
        else
                ca->last_max_cwnd = tp->snd_cwnd;

        ca->loss_cwnd = tp->snd_cwnd;


        if (tp->snd_cwnd <= low_window)
                return max(tp->snd_cwnd >> 1U, 2U);
        else
                return max((tp->snd_cwnd * beta) / BICTCP_BETA_SCALE, 2U);
```

```
}

static u32 bictcp_undo_cwnd(struct sock *sk)
{
        const struct tcp_sock *tp = tcp_sk(sk);
        const struct bictcp *ca = inet_csk_ca(sk);
        return max(tp->snd_cwnd, ca->loss_cwnd);
}

static void bictcp_state(struct sock *sk, u8 new_state)
{
        if (new_state == TCP_CA_Loss)
                bictcp_reset(inet_csk_ca(sk));
}

/* Track delayed acknowledgment ratio using sliding window
 * ratio = (15*ratio + sample) / 16
 */
static void bictcp_acked(struct sock *sk, u32 cnt, s32 rtt)
{
        const struct inet_connection_sock *icsk = inet_csk(sk);

        if (icsk->icsk_ca_state == TCP_CA_Open) {
                struct bictcp *ca = inet_csk_ca(sk);
                cnt -= ca->delayed_ack >> ACK_RATIO_SHIFT;
                ca->delayed_ack += cnt;
        }
}


static struct tcp_congestion_ops bictcp __read_mostly = {
        .init           = bictcp_init,
        .ssthresh       = bictcp_recalc_ssthresh,
        .cong_avoid     = bictcp_cong_avoid,
        .set_state      = bictcp_state,
        .undo_cwnd      = bictcp_undo_cwnd,
        .pkts_acked     = bictcp_acked,
        .owner          = THIS_MODULE,
        .name           = "bic",
};

static int __init bictcp_register(void)
{
        BUILD_BUG_ON(sizeof(struct bictcp) > ICSK_CA_PRIV_SIZE);
        return tcp_register_congestion_control(&bictcp);
}

static void __exit bictcp_unregister(void)
{
        tcp_unregister_congestion_control(&bictcp);
}

module_init(bictcp_register);
module_exit(bictcp_unregister);

MODULE_AUTHOR("Stephen Hemminger");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("BIC TCP");

```

- Home
- Development
- Services
- Training

- Docs
- Community
- Company
- Blog