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_lp.c

```
/*
 * TCP Low Priority (TCP-LP)
 *
 * TCP Low Priority is a distributed algorithm whose goal is to utilize only
 *   the excess network bandwidth as compared to the ``fair share`` of
 *   bandwidth as targeted by TCP.
 *
 * As of 2.6.13, Linux supports pluggable congestion control algorithms.
 * Due to the limitation of the API, we take the following changes from
 * the original TCP-LP implementation:
 *   o We use newReno in most core CA handling. Only add some checking
 *     within cong_avoid.
 *   o Error correcting in remote HZ, therefore remote HZ will be keeped
 *     on checking and updating.
 *   o Handling calculation of One-Way-Delay (OWD) within rtt_sample, since
 *     OWD have a similar meaning as RTT. Also correct the buggy formular.
 *   o Handle reaction for Early Congestion Indication (ECI) within
 *     pkts_acked, as mentioned within pseudo code.
 *   o OWD is handled in relative format, where local time stamp will in
 *     tcp_time_stamp format.
 *
 * Original Author:
 *   Aleksandar Kuzmanovic <akuzma@northwestern.edu>
 * Available from:
 *   http://www.ece.rice.edu/~akuzma/Doc/akuzma/TCP-LP.pdf
 * Original implementation for 2.4.19:
 *   http://www-ece.rice.edu/networks/TCP-LP/
 *
 * 2.6.x module Authors:
 *   Wong Hoi Sing, Edison <hswong3i@gmail.com>
 *   Hung Hing Lun, Mike <hlhung3i@gmail.com>
 * SourceForge project page:
 *   http://tcp-lp-mod.sourceforge.net/
 */

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

/* resolution of owd */
#define LP_RESOL       1000

/**
 * enum tcp_lp_state
 * @LP_VALID_RHZ: is remote HZ valid?
 * @LP_VALID_OWD: is OWD valid?
 * @LP_WITHIN_THR: are we within threshold?
 * @LP_WITHIN_INF: are we within inference?
 *
 * TCP-LP's state flags.
 * We create this set of state flag mainly for debugging.
 */
enum tcp_lp_state {
        LP_VALID_RHZ = (1 << 0),
        LP_VALID_OWD = (1 << 1),
        LP_WITHIN_THR = (1 << 3),
        LP_WITHIN_INF = (1 << 4),
};

/**
 * struct lp
 * @flag: TCP-LP state flag
 * @sowd: smoothed OWD << 3
 * @owd_min: min OWD
 * @owd_max: max OWD
 * @owd_max_rsv: resrved max owd
 * @remote_hz: estimated remote HZ
 * @remote_ref_time: remote reference time
 * @local_ref_time: local reference time
 * @last_drop: time for last active drop
 * @inference: current inference
 *
 * TCP-LP's private struct.
 * We get the idea from original TCP-LP implementation where only left those we
 * found are really useful.
 */
struct lp {
        u32 flag;
        u32 sowd;
        u32 owd_min;
        u32 owd_max;
        u32 owd_max_rsv;
        u32 remote_hz;
        u32 remote_ref_time;
        u32 local_ref_time;
        u32 last_drop;
        u32 inference;
};

/**
 * tcp_lp_init
 *
 * Init all required variables.
 * Clone the handling from Vegas module implementation.
 */
static void tcp_lp_init(struct sock *sk)
{
        struct lp *lp = inet_csk_ca(sk);

        lp->flag = 0;
        lp->sowd = 0;
        lp->owd_min = 0xffffffff;
        lp->owd_max = 0;
        lp->owd_max_rsv = 0;
        lp->remote_hz = 0;
        lp->remote_ref_time = 0;
        lp->local_ref_time = 0;
        lp->last_drop = 0;
        lp->inference = 0;
```

```c
}

/**
 * tcp_lp_cong_avoid
 *
 * Implementation of cong_avoid.
 * Will only call newReno CA when away from inference.
 * From TCP-LP's paper, this will be handled in additive increasement.
 */
static void tcp_lp_cong_avoid(struct sock *sk, u32 ack, u32 acked)
{
        struct lp *lp = inet_csk_ca(sk);

        if (!(lp->flag & LP_WITHIN_INF))
                tcp_reno_cong_avoid(sk, ack, acked);
}

/**
 * tcp_lp_remote_hz_estimator
 *
 * Estimate remote HZ.
 * We keep on updating the estimated value, where original TCP-LP
 * implementation only guest it for once and use forever.
 */
static u32 tcp_lp_remote_hz_estimator(struct sock *sk)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct lp *lp = inet_csk_ca(sk);
        s64 rhz = lp->remote_hz << 6;   /* remote HZ << 6 */
        s64 m = 0;

        /* not yet record reference time
         * go away!! record it before come back!! */
        if (lp->remote_ref_time == 0 || lp->local_ref_time == 0)
                goto out;

        /* we can't calc remote HZ with no different!! */
        if (tp->rx_opt.rcv_tsval == lp->remote_ref_time ||
            tp->rx_opt.rcv_tsecr == lp->local_ref_time)
                goto out;

        m = HZ * (tp->rx_opt.rcv_tsval -
             lp->remote_ref_time) / (tp->rx_opt.rcv_tsecr -
                                     lp->local_ref_time);
        if (m < 0)
                m = -m;

        if (rhz > 0) {
                m -= rhz >> 6;  /* m is now error in remote HZ est */
                rhz += m;       /* 63/64 old + 1/64 new */
        } else
                rhz = m << 6;

 out:
        /* record time for successful remote HZ calc */
        if ((rhz >> 6) > 0)
                lp->flag |= LP_VALID_RHZ;
        else
                lp->flag &= ~LP_VALID_RHZ;

        /* record reference time stamp */
        lp->remote_ref_time = tp->rx_opt.rcv_tsval;
        lp->local_ref_time = tp->rx_opt.rcv_tsecr;

        return rhz >> 6;
```

```
}

/**
 * tcp_lp_owd_calculator
 *
 * Calculate one way delay (in relative format).
 * Original implement OWD as minus of remote time difference to local time
 * difference directly. As this time difference just simply equal to RTT, when
 * the network status is stable, remote RTT will equal to local RTT, and result
 * OWD into zero.
 * It seems to be a bug and so we fixed it.
 */
static u32 tcp_lp_owd_calculator(struct sock *sk)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct lp *lp = inet_csk_ca(sk);
        s64 owd = 0;

        lp->remote_hz = tcp_lp_remote_hz_estimator(sk);

        if (lp->flag & LP_VALID_RHZ) {
                owd =
                    tp->rx_opt.rcv_tsval * (LP_RESOL / lp->remote_hz) -
                    tp->rx_opt.rcv_tsecr * (LP_RESOL / HZ);
                if (owd < 0)
                        owd = -owd;
        }

        if (owd > 0)
                lp->flag |= LP_VALID_OWD;
        else
                lp->flag &= ~LP_VALID_OWD;

        return owd;
}

/**
 * tcp_lp_rtt_sample
 *
 * Implementation or rtt_sample.
 * Will take the following action,
 *   1. calc OWD,
 *   2. record the min/max OWD,
 *   3. calc smoothed OWD (SOWD).
 * Most ideas come from the original TCP-LP implementation.
 */
static void tcp_lp_rtt_sample(struct sock *sk, u32 rtt)
{
        struct lp *lp = inet_csk_ca(sk);
        s64 mowd = tcp_lp_owd_calculator(sk);

        /* sorry that we don't have valid data */
        if (!(lp->flag & LP_VALID_RHZ) || !(lp->flag & LP_VALID_OWD))
                return;

        /* record the next min owd */
        if (mowd < lp->owd_min)
                lp->owd_min = mowd;

        /* always forget the max of the max
         * we just set owd_max as one below it */
        if (mowd > lp->owd_max) {
                if (mowd > lp->owd_max_rsv) {
                        if (lp->owd_max_rsv == 0)
                                lp->owd_max = mowd;
                        else
                                lp->owd_max = lp->owd_max_rsv;
                        lp->owd_max_rsv = mowd;
                } else
                        lp->owd_max = mowd;
        }

        /* calc for smoothed owd */
        if (lp->sowd != 0) {
                mowd -= lp->sowd >> 3;  /* m is now error in owd est */
                lp->sowd += mowd;       /* owd = 7/8 owd + 1/8 new */
        } else
                lp->sowd = mowd << 3;   /* take the measured time be owd */
}

/**
 * tcp_lp_pkts_acked
 *
 * Implementation of pkts_acked.
 * Deal with active drop under Early Congestion Indication.
 * Only drop to half and 1 will be handle, because we hope to use back
 * newReno in increase case.
 * We work it out by following the idea from TCP-LP's paper directly
 */
static void tcp_lp_pkts_acked(struct sock *sk, u32 num_acked, s32 rtt_us)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct lp *lp = inet_csk_ca(sk);

        if (rtt_us > 0)
                tcp_lp_rtt_sample(sk, rtt_us);

        /* calc inference */
        if (tcp_time_stamp > tp->rx_opt.rcv_tsecr)
                lp->inference = 3 * (tcp_time_stamp - tp->rx_opt.rcv_tsecr);

        /* test if within inference */
        if (lp->last_drop && (tcp_time_stamp - lp->last_drop < lp->inference))
                lp->flag |= LP_WITHIN_INF;
        else
                lp->flag &= ~LP_WITHIN_INF;

        /* test if within threshold */
        if (lp->sowd >> 3 <
            lp->owd_min + 15 * (lp->owd_max - lp->owd_min) / 100)
                lp->flag |= LP_WITHIN_THR;
        else
                lp->flag &= ~LP_WITHIN_THR;

        pr_debug("TCP-LP: %05o|%5u|%5u|%15u|%15u|%15u\n", lp->flag,
                 tp->snd_cwnd, lp->remote_hz, lp->owd_min, lp->owd_max,
                 lp->sowd >> 3);

        if (lp->flag & LP_WITHIN_THR)
                return;

        /* FIXME: try to reset owd_min and owd_max here
         * so decrease the chance the min/max is no longer suitable
         * and will usually within threshold when whithin inference */
        lp->owd_min = lp->sowd >> 3;
        lp->owd_max = lp->sowd >> 2;
        lp->owd_max_rsv = lp->sowd >> 2;

        /* happened within inference
         * drop snd_cwnd into 1 */
        if (lp->flag & LP_WITHIN_INF)
                tp->snd_cwnd = 1U;

        /* happened after inference
         * cut snd_cwnd into half */
        else
                tp->snd_cwnd = max(tp->snd_cwnd >> 1U, 1U);

        /* record this drop time */
        lp->last_drop = tcp_time_stamp;
}

static struct tcp_congestion_ops tcp_lp __read_mostly = {
        .init = tcp_lp_init,
        .ssthresh = tcp_reno_ssthresh,
        .cong_avoid = tcp_lp_cong_avoid,
        .pkts_acked = tcp_lp_pkts_acked,

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

static int __init tcp_lp_register(void)
{
        BUILD_BUG_ON(sizeof(struct lp) > ICSK_CA_PRIV_SIZE);
        return tcp_register_congestion_control(&tcp_lp);
}

static void __exit tcp_lp_unregister(void)
{
        tcp_unregister_congestion_control(&tcp_lp);
}

module_init(tcp_lp_register);
module_exit(tcp_lp_unregister);

MODULE_AUTHOR("Wong Hoi Sing Edison, Hung Hing Lun Mike");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("TCP Low Priority");

```

- Home
- Development
- Services
- Training
- Docs
- Community
- Company
- Blog