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_metrics.c

```
#include <linux/rcupdate.h>
#include <linux/spinlock.h>
#include <linux/jiffies.h>
#include <linux/module.h>
#include <linux/cache.h>
#include <linux/slab.h>
#include <linux/init.h>
#include <linux/tcp.h>
#include <linux/hash.h>
#include <linux/tcp_metrics.h>
#include <linux/vmalloc.h>

#include <net/inet_connection_sock.h>
#include <net/net_namespace.h>
#include <net/request_sock.h>
#include <net/inetpeer.h>
#include <net/sock.h>
#include <net/ipv6.h>
#include <net/dst.h>
#include <net/tcp.h>
#include <net/genetlink.h>

int sysctl_tcp_nometrics_save __read_mostly;

static struct tcp_metrics_block *__tcp_get_metrics(const struct inetpeer_addr *saddr,
                                                   const struct inetpeer_addr *daddr,
                                                   struct net *net, unsigned int hash);

struct tcp_fastopen_metrics {
        u16     mss;
        u16     syn_loss:10;            /* Recurring Fast Open SYN losses */
        unsigned long   last_syn_loss;  /* Last Fast Open SYN loss */
        struct  tcp_fastopen_cookie     cookie;
};

/* TCP_METRIC_MAX includes 2 extra fields for userspace compatibility
 * Kernel only stores RTT and RTTVAR in usec resolution
 */
#define TCP_METRIC_MAX_KERNEL (TCP_METRIC_MAX - 2)

struct tcp_metrics_block {
        struct tcp_metrics_block __rcu  *tcpm_next;
        struct inetpeer_addr            tcpm_saddr;
```

```
44          struct inetpeer_addr            tcpm_daddr;
45          unsigned long                   tcpm_stamp;
46          u32                             tcpm_ts;
47          u32                             tcpm_ts_stamp;
48          u32                             tcpm_lock;
49          u32                             tcpm_vals[TCP_METRIC_MAX_KERNEL + 1];
50          struct tcp_fastopen_metrics     tcpm_fastopen;
51
52          struct rcu_head                 rcu_head;
53  };
54
55  static bool tcp_metric_locked(struct tcp_metrics_block *tm,
56                                enum tcp_metric_index idx)
57  {
58          return tm->tcpm_lock & (1 << idx);
59  }
60
61  static u32 tcp_metric_get(struct tcp_metrics_block *tm,
62                            enum tcp_metric_index idx)
63  {
64          return tm->tcpm_vals[idx];
65  }
66
67  static void tcp_metric_set(struct tcp_metrics_block *tm,
68                             enum tcp_metric_index idx,
69                             u32 val)
70  {
71          tm->tcpm_vals[idx] = val;
72  }
73
74  static bool addr_same(const struct inetpeer_addr *a,
75                        const struct inetpeer_addr *b)
76  {
77          const struct in6_addr *a6, *b6;
78
79          if (a->family != b->family)
80                  return false;
81          if (a->family == AF_INET)
82                  return a->addr.a4 == b->addr.a4;
83
84          a6 = (const struct in6_addr *) &a->addr.a6[0];
85          b6 = (const struct in6_addr *) &b->addr.a6[0];
86
87          return ipv6_addr_equal(a6, b6);
88  }
89
90  struct tcpm_hash_bucket {
91          struct tcp_metrics_block __rcu  *chain;
92  };
93
94  static DEFINE_SPINLOCK(tcp_metrics_lock);
95
96  static void tcpm_suck_dst(struct tcp_metrics_block *tm,
97                            const struct dst_entry *dst,
98                            bool fastopen_clear)
99  {
100         u32 msval;
101         u32 val;
102
103         tm->tcpm_stamp = jiffies;
104
105         val = 0;
106         if (dst_metric_locked(dst, RTAX_RTT))
107                 val |= 1 << TCP_METRIC_RTT;
108         if (dst_metric_locked(dst, RTAX_RTTVAR))
```

```
109                 val |= 1 << TCP_METRIC_RTTVAR;
110         if (dst_metric_locked(dst, RTAX_SSTHRESH))
111                 val |= 1 << TCP_METRIC_SSTHRESH;
112         if (dst_metric_locked(dst, RTAX_CWND))
113                 val |= 1 << TCP_METRIC_CWND;
114         if (dst_metric_locked(dst, RTAX_REORDERING))
115                 val |= 1 << TCP_METRIC_REORDERING;
116         tm->tcpm_lock = val;
117
118         msval = dst_metric_raw(dst, RTAX_RTT);
119         tm->tcpm_vals[TCP_METRIC_RTT] = msval * USEC_PER_MSEC;
120
121         msval = dst_metric_raw(dst, RTAX_RTTVAR);
122         tm->tcpm_vals[TCP_METRIC_RTTVAR] = msval * USEC_PER_MSEC;
123         tm->tcpm_vals[TCP_METRIC_SSTHRESH] = dst_metric_raw(dst, RTAX_SSTHRESH);
124         tm->tcpm_vals[TCP_METRIC_CWND] = dst_metric_raw(dst, RTAX_CWND);
125         tm->tcpm_vals[TCP_METRIC_REORDERING] = dst_metric_raw(dst, RTAX_REORDERING);
126         tm->tcpm_ts = 0;
127         tm->tcpm_ts_stamp = 0;
128         if (fastopen_clear) {
129                 tm->tcpm_fastopen.mss = 0;
130                 tm->tcpm_fastopen.syn_loss = 0;
131                 tm->tcpm_fastopen.cookie.len = 0;
132         }
133 }
134
135 #define TCP_METRICS_TIMEOUT             (60 * 60 * HZ)
136
137 static void tcpm_check_stamp(struct tcp_metrics_block *tm, struct dst_entry *dst)
138 {
139         if (tm && unlikely(time_after(jiffies, tm->tcpm_stamp + TCP_METRICS_TIMEOUT)))
140                 tcpm_suck_dst(tm, dst, false);
141 }
142
143 #define TCP_METRICS_RECLAIM_DEPTH       5
144 #define TCP_METRICS_RECLAIM_PTR         (struct tcp_metrics_block *) 0x1UL
145
146 static struct tcp_metrics_block *tcpm_new(struct dst_entry *dst,
147                                           struct inetpeer_addr *saddr,
148                                           struct inetpeer_addr *daddr,
149                                           unsigned int hash)
150 {
151         struct tcp_metrics_block *tm;
152         struct net *net;
153         bool reclaim = false;
154
155         spin_lock_bh(&tcp_metrics_lock);
156         net = dev_net(dst->dev);
157
158         /* While waiting for the spin-lock the cache might have been populated
159          * with this entry and so we have to check again.
160          */
161         tm = __tcp_get_metrics(saddr, daddr, net, hash);
162         if (tm == TCP_METRICS_RECLAIM_PTR) {
163                 reclaim = true;
164                 tm = NULL;
165         }
166         if (tm) {
167                 tcpm_check_stamp(tm, dst);
168                 goto out_unlock;
169         }
170
171         if (unlikely(reclaim)) {
172                 struct tcp_metrics_block *oldest;
173
```

```
174                 oldest = rcu_dereference(net->ipv4.tcp_metrics_hash[hash].chain);
175                 for (tm = rcu_dereference(oldest->tcpm_next); tm;
176                      tm = rcu_dereference(tm->tcpm_next)) {
177                         if (time_before(tm->tcpm_stamp, oldest->tcpm_stamp))
178                                 oldest = tm;
179                 }
180                 tm = oldest;
181         } else {
182                 tm = kmalloc(sizeof(*tm), GFP_ATOMIC);
183                 if (!tm)
184                         goto out_unlock;
185         }
186         tm->tcpm_saddr = *saddr;
187         tm->tcpm_daddr = *daddr;
188
189         tcpm_suck_dst(tm, dst, true);
190
191         if (likely(!reclaim)) {
192                 tm->tcpm_next = net->ipv4.tcp_metrics_hash[hash].chain;
193                 rcu_assign_pointer(net->ipv4.tcp_metrics_hash[hash].chain, tm);
194         }
195
196 out_unlock:
197         spin_unlock_bh(&tcp_metrics_lock);
198         return tm;
199 }
200
201 static struct tcp_metrics_block *tcp_get_encode(struct tcp_metrics_block *tm, int depth)
202 {
203         if (tm)
204                 return tm;
205         if (depth > TCP_METRICS_RECLAIM_DEPTH)
206                 return TCP_METRICS_RECLAIM_PTR;
207         return NULL;
208 }
209
210 static struct tcp_metrics_block *__tcp_get_metrics(const struct inetpeer_addr *saddr,
211                                                     const struct inetpeer_addr *daddr,
212                                                     struct net *net, unsigned int hash)
213 {
214         struct tcp_metrics_block *tm;
215         int depth = 0;
216
217         for (tm = rcu_dereference(net->ipv4.tcp_metrics_hash[hash].chain); tm;
218              tm = rcu_dereference(tm->tcpm_next)) {
219                 if (addr_same(&tm->tcpm_saddr, saddr) &&
220                     addr_same(&tm->tcpm_daddr, daddr))
221                         break;
222                 depth++;
223         }
224         return tcp_get_encode(tm, depth);
225 }
226
227 static struct tcp_metrics_block *__tcp_get_metrics_req(struct request_sock *req,
228                                                         struct dst_entry *dst)
229 {
230         struct tcp_metrics_block *tm;
231         struct inetpeer_addr saddr, daddr;
232         unsigned int hash;
233         struct net *net;
234
235         saddr.family = req->rsk_ops->family;
236         daddr.family = req->rsk_ops->family;
237         switch (daddr.family) {
238         case AF_INET:
```

```
239                 saddr.addr.a4 = inet_rsk(req)->ir_loc_addr;
240                 daddr.addr.a4 = inet_rsk(req)->ir_rmt_addr;
241                 hash = (__force unsigned int) daddr.addr.a4;
242                 break;
243 #if IS_ENABLED(CONFIG_IPV6)
244         case AF_INET6:
245                 *(struct in6_addr *)saddr.addr.a6 = inet_rsk(req)->ir_v6_loc_addr;
246                 *(struct in6_addr *)daddr.addr.a6 = inet_rsk(req)->ir_v6_rmt_addr;
247                 hash = ipv6_addr_hash(&inet_rsk(req)->ir_v6_rmt_addr);
248                 break;
249 #endif
250         default:
251                 return NULL;
252         }
253
254         net = dev_net(dst->dev);
255         hash = hash_32(hash, net->ipv4.tcp_metrics_hash_log);
256
257         for (tm = rcu_dereference(net->ipv4.tcp_metrics_hash[hash].chain); tm;
258              tm = rcu_dereference(tm->tcpm_next)) {
259                 if (addr_same(&tm->tcpm_saddr, &saddr) &&
260                     addr_same(&tm->tcpm_daddr, &daddr))
261                         break;
262         }
263         tcpm_check_stamp(tm, dst);
264         return tm;
265 }
266
267 static struct tcp_metrics_block *__tcp_get_metrics_tw(struct inet_timewait_sock *tw)
268 {
269         struct tcp_metrics_block *tm;
270         struct inetpeer_addr saddr, daddr;
271         unsigned int hash;
272         struct net *net;
273
274         if (tw->tw_family == AF_INET) {
275                 saddr.family = AF_INET;
276                 saddr.addr.a4 = tw->tw_rcv_saddr;
277                 daddr.family = AF_INET;
278                 daddr.addr.a4 = tw->tw_daddr;
279                 hash = (__force unsigned int) daddr.addr.a4;
280         }
281 #if IS_ENABLED(CONFIG_IPV6)
282         else if (tw->tw_family == AF_INET6) {
283                 if (ipv6_addr_v4mapped(&tw->tw_v6_daddr)) {
284                         saddr.family = AF_INET;
285                         saddr.addr.a4 = tw->tw_rcv_saddr;
286                         daddr.family = AF_INET;
287                         daddr.addr.a4 = tw->tw_daddr;
288                         hash = (__force unsigned int) daddr.addr.a4;
289                 } else {
290                         saddr.family = AF_INET6;
291                         *(struct in6_addr *)saddr.addr.a6 = tw->tw_v6_rcv_saddr;
292                         daddr.family = AF_INET6;
293                         *(struct in6_addr *)daddr.addr.a6 = tw->tw_v6_daddr;
294                         hash = ipv6_addr_hash(&tw->tw_v6_daddr);
295                 }
296         }
297 #endif
298         else
299                 return NULL;
300
301         net = twsk_net(tw);
302         hash = hash_32(hash, net->ipv4.tcp_metrics_hash_log);
303
```

```
304         for (tm = rcu_dereference(net->ipv4.tcp_metrics_hash[hash].chain); tm;
305              tm = rcu_dereference(tm->tcpm_next)) {
306                 if (addr_same(&tm->tcpm_saddr, &saddr) &&
307                     addr_same(&tm->tcpm_daddr, &daddr))
308                         break;
309         }
310         return tm;
311 }
312
313 static struct tcp_metrics_block *tcp_get_metrics(struct sock *sk,
314                                                   struct dst_entry *dst,
315                                                   bool create)
316 {
317         struct tcp_metrics_block *tm;
318         struct inetpeer_addr saddr, daddr;
319         unsigned int hash;
320         struct net *net;
321
322         if (sk->sk_family == AF_INET) {
323                 saddr.family = AF_INET;
324                 saddr.addr.a4 = inet_sk(sk)->inet_saddr;
325                 daddr.family = AF_INET;
326                 daddr.addr.a4 = inet_sk(sk)->inet_daddr;
327                 hash = (__force unsigned int) daddr.addr.a4;
328         }
329 #if IS_ENABLED(CONFIG_IPV6)
330         else if (sk->sk_family == AF_INET6) {
331                 if (ipv6_addr_v4mapped(&sk->sk_v6_daddr)) {
332                         saddr.family = AF_INET;
333                         saddr.addr.a4 = inet_sk(sk)->inet_saddr;
334                         daddr.family = AF_INET;
335                         daddr.addr.a4 = inet_sk(sk)->inet_daddr;
336                         hash = (__force unsigned int) daddr.addr.a4;
337                 } else {
338                         saddr.family = AF_INET6;
339                         *(struct in6_addr *)saddr.addr.a6 = sk->sk_v6_rcv_saddr;
340                         daddr.family = AF_INET6;
341                         *(struct in6_addr *)daddr.addr.a6 = sk->sk_v6_daddr;
342                         hash = ipv6_addr_hash(&sk->sk_v6_daddr);
343                 }
344         }
345 #endif
346         else
347                 return NULL;
348
349         net = dev_net(dst->dev);
350         hash = hash_32(hash, net->ipv4.tcp_metrics_hash_log);
351
352         tm = __tcp_get_metrics(&saddr, &daddr, net, hash);
353         if (tm == TCP_METRICS_RECLAIM_PTR)
354                 tm = NULL;
355         if (!tm && create)
356                 tm = tcpm_new(dst, &saddr, &daddr, hash);
357         else
358                 tcpm_check_stamp(tm, dst);
359
360         return tm;
361 }
362
363 /* Save metrics learned by this TCP session.  This function is called
364  * only, when TCP finishes successfully i.e. when it enters TIME-WAIT
365  * or goes from LAST-ACK to CLOSE.
366  */
367 void tcp_update_metrics(struct sock *sk)
368 {
```

```
369         const struct inet_connection_sock *icsk = inet_csk(sk);
370         struct dst_entry *dst = __sk_dst_get(sk);
371         struct tcp_sock *tp = tcp_sk(sk);
372         struct tcp_metrics_block *tm;
373         unsigned long rtt;
374         u32 val;
375         int m;
376
377         if (sysctl_tcp_nometrics_save || !dst)
378                 return;
379
380         if (dst->flags & DST_HOST)
381                 dst_confirm(dst);
382
383         rcu_read_lock();
384         if (icsk->icsk_backoff || !tp->srtt_us) {
385                 /* This session failed to estimate rtt. Why?
386                  * Probably, no packets returned in time.  Reset our
387                  * results.
388                  */
389                 tm = tcp_get_metrics(sk, dst, false);
390                 if (tm && !tcp_metric_locked(tm, TCP_METRIC_RTT))
391                         tcp_metric_set(tm, TCP_METRIC_RTT, 0);
392                 goto out_unlock;
393         } else
394                 tm = tcp_get_metrics(sk, dst, true);
395
396         if (!tm)
397                 goto out_unlock;
398
399         rtt = tcp_metric_get(tm, TCP_METRIC_RTT);
400         m = rtt - tp->srtt_us;
401
402         /* If newly calculated rtt larger than stored one, store new
403          * one. Otherwise, use EWMA. Remember, rtt overestimation is
404          * always better than underestimation.
405          */
406         if (!tcp_metric_locked(tm, TCP_METRIC_RTT)) {
407                 if (m <= 0)
408                         rtt = tp->srtt_us;
409                 else
410                         rtt -= (m >> 3);
411                 tcp_metric_set(tm, TCP_METRIC_RTT, rtt);
412         }
413
414         if (!tcp_metric_locked(tm, TCP_METRIC_RTTVAR)) {
415                 unsigned long var;
416
417                 if (m < 0)
418                         m = -m;
419
420                 /* Scale deviation to rttvar fixed point */
421                 m >>= 1;
422                 if (m < tp->mdev_us)
423                         m = tp->mdev_us;
424
425                 var = tcp_metric_get(tm, TCP_METRIC_RTTVAR);
426                 if (m >= var)
427                         var = m;
428                 else
429                         var -= (var - m) >> 2;
430
431                 tcp_metric_set(tm, TCP_METRIC_RTTVAR, var);
432         }
433
```

```
434         if (tcp_in_initial_slowstart(tp)) {
435                 /* Slow start still did not finish. */
436                 if (!tcp_metric_locked(tm, TCP_METRIC_SSTHRESH)) {
437                         val = tcp_metric_get(tm, TCP_METRIC_SSTHRESH);
438                         if (val && (tp->snd_cwnd >> 1) > val)
439                                 tcp_metric_set(tm, TCP_METRIC_SSTHRESH,
440                                                tp->snd_cwnd >> 1);
441                 }
442                 if (!tcp_metric_locked(tm, TCP_METRIC_CWND)) {
443                         val = tcp_metric_get(tm, TCP_METRIC_CWND);
444                         if (tp->snd_cwnd > val)
445                                 tcp_metric_set(tm, TCP_METRIC_CWND,
446                                                tp->snd_cwnd);
447                 }
448         } else if (tp->snd_cwnd > tp->snd_ssthresh &&
449                    icsk->icsk_ca_state == TCP_CA_Open) {
450                 /* Cong. avoidance phase, cwnd is reliable. */
451                 if (!tcp_metric_locked(tm, TCP_METRIC_SSTHRESH))
452                         tcp_metric_set(tm, TCP_METRIC_SSTHRESH,
453                                        max(tp->snd_cwnd >> 1, tp->snd_ssthresh));
454                 if (!tcp_metric_locked(tm, TCP_METRIC_CWND)) {
455                         val = tcp_metric_get(tm, TCP_METRIC_CWND);
456                         tcp_metric_set(tm, TCP_METRIC_CWND, (val + tp->snd_cwnd) >> 1);
457                 }
458         } else {
459                 /* Else slow start did not finish, cwnd is non-sense,
460                  * ssthresh may be also invalid.
461                  */
462                 if (!tcp_metric_locked(tm, TCP_METRIC_CWND)) {
463                         val = tcp_metric_get(tm, TCP_METRIC_CWND);
464                         tcp_metric_set(tm, TCP_METRIC_CWND,
465                                        (val + tp->snd_ssthresh) >> 1);
466                 }
467                 if (!tcp_metric_locked(tm, TCP_METRIC_SSTHRESH)) {
468                         val = tcp_metric_get(tm, TCP_METRIC_SSTHRESH);
469                         if (val && tp->snd_ssthresh > val)
470                                 tcp_metric_set(tm, TCP_METRIC_SSTHRESH,
471                                                tp->snd_ssthresh);
472                 }
473                 if (!tcp_metric_locked(tm, TCP_METRIC_REORDERING)) {
474                         val = tcp_metric_get(tm, TCP_METRIC_REORDERING);
475                         if (val < tp->reordering &&
476                             tp->reordering != sysctl_tcp_reordering)
477                                 tcp_metric_set(tm, TCP_METRIC_REORDERING,
478                                                tp->reordering);
479                 }
480         }
481         tm->tcpm_stamp = jiffies;
482 out_unlock:
483         rcu_read_unlock();
484 }
485
486 /* Initialize metrics on socket. */
487
488 void tcp_init_metrics(struct sock *sk)
489 {
490         struct dst_entry *dst = __sk_dst_get(sk);
491         struct tcp_sock *tp = tcp_sk(sk);
492         struct tcp_metrics_block *tm;
493         u32 val, crtt = 0; /* cached RTT scaled by 8 */
494
495         if (dst == NULL)
496                 goto reset;
497
498         dst_confirm(dst);
```

```
499
500         rcu_read_lock();
501         tm = tcp_get_metrics(sk, dst, true);
502         if (!tm) {
503                 rcu_read_unlock();
504                 goto reset;
505         }
506
507         if (tcp_metric_locked(tm, TCP_METRIC_CWND))
508                 tp->snd_cwnd_clamp = tcp_metric_get(tm, TCP_METRIC_CWND);
509
510         val = tcp_metric_get(tm, TCP_METRIC_SSTHRESH);
511         if (val) {
512                 tp->snd_ssthresh = val;
513                 if (tp->snd_ssthresh > tp->snd_cwnd_clamp)
514                         tp->snd_ssthresh = tp->snd_cwnd_clamp;
515         } else {
516                 /* ssthresh may have been reduced unnecessarily during.
517                  * 3WHS. Restore it back to its initial default.
518                  */
519                 tp->snd_ssthresh = TCP_INFINITE_SSTHRESH;
520         }
521         val = tcp_metric_get(tm, TCP_METRIC_REORDERING);
522         if (val && tp->reordering != val) {
523                 tcp_disable_fack(tp);
524                 tcp_disable_early_retrans(tp);
525                 tp->reordering = val;
526         }
527
528         crtt = tcp_metric_get(tm, TCP_METRIC_RTT);
529         rcu_read_unlock();
530 reset:
531         /* The initial RTT measurement from the SYN/SYN-ACK is not ideal
532          * to seed the RTO for later data packets because SYN packets are
533          * small. Use the per-dst cached values to seed the RTO but keep
534          * the RTT estimator variables intact (e.g., srtt, mdev, rttvar).
535          * Later the RTO will be updated immediately upon obtaining the first
536          * data RTT sample (tcp_rtt_estimator()). Hence the cached RTT only
537          * influences the first RTO but not later RTT estimation.
538          *
539          * But if RTT is not available from the SYN (due to retransmits or
540          * syn cookies) or the cache, force a conservative 3secs timeout.
541          *
542          * A bit of theory. RTT is time passed after "normal" sized packet
543          * is sent until it is ACKed. In normal circumstances sending small
544          * packets force peer to delay ACKs and calculation is correct too.
545          * The algorithm is adaptive and, provided we follow specs, it
546          * NEVER underestimate RTT. BUT! If peer tries to make some clever
547          * tricks sort of "quick acks" for time long enough to decrease RTT
548          * to low value, and then abruptly stops to do it and starts to delay
549          * ACKs, wait for troubles.
550          */
551         if (crtt > tp->srtt_us) {
552                 /* Set RTO like tcp_rtt_estimator(), but from cached RTT. */
553                 crtt /= 8 * USEC_PER_MSEC;
554                 inet_csk(sk)->icsk_rto = crtt + max(2 * crtt, tcp_rto_min(sk));
555         } else if (tp->srtt_us == 0) {
556                 /* RFC6298: 5.7 We've failed to get a valid RTT sample from
557                  * 3WHS. This is most likely due to retransmission,
558                  * including spurious one. Reset the RTO back to 3secs
559                  * from the more aggressive 1sec to avoid more spurious
560                  * retransmission.
561                  */
562                 tp->rttvar_us = jiffies_to_usecs(TCP_TIMEOUT_FALLBACK);
563                 tp->mdev_us = tp->mdev_max_us = tp->rttvar_us;
```

```
564
565                 inet_csk(sk)->icsk_rto = TCP_TIMEOUT_FALLBACK;
566         }
567         /* Cut cwnd down to 1 per RFC5681 if SYN or SYN-ACK has been
568          * retransmitted. In light of RFC6298 more aggressive 1sec
569          * initRTO, we only reset cwnd when more than 1 SYN/SYN-ACK
570          * retransmission has occurred.
571          */
572         if (tp->total_retrans > 1)
573                 tp->snd_cwnd = 1;
574         else
575                 tp->snd_cwnd = tcp_init_cwnd(tp, dst);
576         tp->snd_cwnd_stamp = tcp_time_stamp;
577 }
578
579 bool tcp_peer_is_proven(struct request_sock *req, struct dst_entry *dst,
580                         bool paws_check, bool timestamps)
581 {
582         struct tcp_metrics_block *tm;
583         bool ret;
584
585         if (!dst)
586                 return false;
587
588         rcu_read_lock();
589         tm = __tcp_get_metrics_req(req, dst);
590         if (paws_check) {
591                 if (tm &&
592                     (u32)get_seconds() - tm->tcpm_ts_stamp < TCP_PAWS_MSL &&
593                     ((s32)(tm->tcpm_ts - req->ts_recent) > TCP_PAWS_WINDOW ||
594                      !timestamps))
595                         ret = false;
596                 else
597                         ret = true;
598         } else {
599                 if (tm && tcp_metric_get(tm, TCP_METRIC_RTT) && tm->tcpm_ts_stamp)
600                         ret = true;
601                 else
602                         ret = false;
603         }
604         rcu_read_unlock();
605
606         return ret;
607 }
608 EXPORT_SYMBOL_GPL(tcp_peer_is_proven);
609
610 void tcp_fetch_timewait_stamp(struct sock *sk, struct dst_entry *dst)
611 {
612         struct tcp_metrics_block *tm;
613
614         rcu_read_lock();
615         tm = tcp_get_metrics(sk, dst, true);
616         if (tm) {
617                 struct tcp_sock *tp = tcp_sk(sk);
618
619                 if ((u32)get_seconds() - tm->tcpm_ts_stamp <= TCP_PAWS_MSL) {
620                         tp->rx_opt.ts_recent_stamp = tm->tcpm_ts_stamp;
621                         tp->rx_opt.ts_recent = tm->tcpm_ts;
622                 }
623         }
624         rcu_read_unlock();
625 }
626 EXPORT_SYMBOL_GPL(tcp_fetch_timewait_stamp);
627
628 /* VJ's idea. Save last timestamp seen from this destination and hold
```

```
 * it at least for normal timewait interval to use for duplicate
 * segment detection in subsequent connections, before they enter
 * synchronized state.
 */
bool tcp_remember_stamp(struct sock *sk)
{
        struct dst_entry *dst = __sk_dst_get(sk);
        bool ret = false;

        if (dst) {
                struct tcp_metrics_block *tm;

                rcu_read_lock();
                tm = tcp_get_metrics(sk, dst, true);
                if (tm) {
                        struct tcp_sock *tp = tcp_sk(sk);

                        if ((s32)(tm->tcpm_ts - tp->rx_opt.ts_recent) <= 0 ||
                            ((u32)get_seconds() - tm->tcpm_ts_stamp > TCP_PAWS_MSL &&
                             tm->tcpm_ts_stamp <= (u32)tp->rx_opt.ts_recent_stamp)) {
                                tm->tcpm_ts_stamp = (u32)tp->rx_opt.ts_recent_stamp;
                                tm->tcpm_ts = tp->rx_opt.ts_recent;
                        }
                        ret = true;
                }
                rcu_read_unlock();
        }
        return ret;
}

bool tcp_tw_remember_stamp(struct inet_timewait_sock *tw)
{
        struct tcp_metrics_block *tm;
        bool ret = false;

        rcu_read_lock();
        tm = __tcp_get_metrics_tw(tw);
        if (tm) {
                const struct tcp_timewait_sock *tcptw;
                struct sock *sk = (struct sock *) tw;

                tcptw = tcp_twsk(sk);
                if ((s32)(tm->tcpm_ts - tcptw->tw_ts_recent) <= 0 ||
                    ((u32)get_seconds() - tm->tcpm_ts_stamp > TCP_PAWS_MSL &&
                     tm->tcpm_ts_stamp <= (u32)tcptw->tw_ts_recent_stamp)) {
                        tm->tcpm_ts_stamp = (u32)tcptw->tw_ts_recent_stamp;
                        tm->tcpm_ts       = tcptw->tw_ts_recent;
                }
                ret = true;
        }
        rcu_read_unlock();

        return ret;
}

static DEFINE_SEQLOCK(fastopen_seqlock);

void tcp_fastopen_cache_get(struct sock *sk, u16 *mss,
                            struct tcp_fastopen_cookie *cookie,
                            int *syn_loss, unsigned long *last_syn_loss)
{
        struct tcp_metrics_block *tm;

        rcu_read_lock();
        tm = tcp_get_metrics(sk, __sk_dst_get(sk), false);
```

```
694         if (tm) {
695                 struct tcp_fastopen_metrics *tfom = &tm->tcpm_fastopen;
696                 unsigned int seq;
697
698                 do {
699                         seq = read_seqbegin(&fastopen_seqlock);
700                         if (tfom->mss)
701                                 *mss = tfom->mss;
702                         *cookie = tfom->cookie;
703                         *syn_loss = tfom->syn_loss;
704                         *last_syn_loss = *syn_loss ? tfom->last_syn_loss : 0;
705                 } while (read_seqretry(&fastopen_seqlock, seq));
706         }
707         rcu_read_unlock();
708 }
709
710 void tcp_fastopen_cache_set(struct sock *sk, u16 mss,
711                             struct tcp_fastopen_cookie *cookie, bool syn_lost)
712 {
713         struct dst_entry *dst = __sk_dst_get(sk);
714         struct tcp_metrics_block *tm;
715
716         if (!dst)
717                 return;
718         rcu_read_lock();
719         tm = tcp_get_metrics(sk, dst, true);
720         if (tm) {
721                 struct tcp_fastopen_metrics *tfom = &tm->tcpm_fastopen;
722
723                 write_seqlock_bh(&fastopen_seqlock);
724                 if (mss)
725                         tfom->mss = mss;
726                 if (cookie && cookie->len > 0)
727                         tfom->cookie = *cookie;
728                 if (syn_lost) {
729                         ++tfom->syn_loss;
730                         tfom->last_syn_loss = jiffies;
731                 } else
732                         tfom->syn_loss = 0;
733                 write_sequnlock_bh(&fastopen_seqlock);
734         }
735         rcu_read_unlock();
736 }
737
738 static struct genl_family tcp_metrics_nl_family = {
739         .id             = GENL_ID_GENERATE,
740         .hdrsize        = 0,
741         .name           = TCP_METRICS_GENL_NAME,
742         .version        = TCP_METRICS_GENL_VERSION,
743         .maxattr        = TCP_METRICS_ATTR_MAX,
744         .netnsok        = true,
745 };
746
747 static struct nla_policy tcp_metrics_nl_policy[TCP_METRICS_ATTR_MAX + 1] = {
748         [TCP_METRICS_ATTR_ADDR_IPV4]    = { .type = NLA_U32, },
749         [TCP_METRICS_ATTR_ADDR_IPV6]    = { .type = NLA_BINARY,
750                                             .len = sizeof(struct in6_addr), },
751         /* Following attributes are not received for GET/DEL,
752          * we keep them for reference
753          */
754 #if 0
755         [TCP_METRICS_ATTR_AGE]          = { .type = NLA_MSECS, },
756         [TCP_METRICS_ATTR_TW_TSVAL]     = { .type = NLA_U32, },
757         [TCP_METRICS_ATTR_TW_TS_STAMP]  = { .type = NLA_S32, },
758         [TCP_METRICS_ATTR_VALS]         = { .type = NLA_NESTED, },
```

```
759         [TCP_METRICS_ATTR_FOPEN_MSS]    = { .type = NLA_U16, },
760         [TCP_METRICS_ATTR_FOPEN_SYN_DROPS]      = { .type = NLA_U16, },
761         [TCP_METRICS_ATTR_FOPEN_SYN_DROP_TS]    = { .type = NLA_MSECS, },
762         [TCP_METRICS_ATTR_FOPEN_COOKIE] = { .type = NLA_BINARY,
763                                             .len = TCP_FASTOPEN_COOKIE_MAX, },
764 #endif
765 };
766
767 /* Add attributes, caller cancels its header on failure */
768 static int tcp_metrics_fill_info(struct sk_buff *msg,
769                                  struct tcp_metrics_block *tm)
770 {
771         struct nlattr *nest;
772         int i;
773
774         switch (tm->tcpm_daddr.family) {
775         case AF_INET:
776                 if (nla_put_be32(msg, TCP_METRICS_ATTR_ADDR_IPV4,
777                                  tm->tcpm_daddr.addr.a4) < 0)
778                         goto nla_put_failure;
779                 if (nla_put_be32(msg, TCP_METRICS_ATTR_SADDR_IPV4,
780                                  tm->tcpm_saddr.addr.a4) < 0)
781                         goto nla_put_failure;
782                 break;
783         case AF_INET6:
784                 if (nla_put(msg, TCP_METRICS_ATTR_ADDR_IPV6, 16,
785                             tm->tcpm_daddr.addr.a6) < 0)
786                         goto nla_put_failure;
787                 if (nla_put(msg, TCP_METRICS_ATTR_SADDR_IPV6, 16,
788                             tm->tcpm_saddr.addr.a6) < 0)
789                         goto nla_put_failure;
790                 break;
791         default:
792                 return -EAFNOSUPPORT;
793         }
794
795         if (nla_put_msecs(msg, TCP_METRICS_ATTR_AGE,
796                           jiffies - tm->tcpm_stamp) < 0)
797                 goto nla_put_failure;
798         if (tm->tcpm_ts_stamp) {
799                 if (nla_put_s32(msg, TCP_METRICS_ATTR_TW_TS_STAMP,
800                                 (s32) (get_seconds() - tm->tcpm_ts_stamp)) < 0)
801                         goto nla_put_failure;
802                 if (nla_put_u32(msg, TCP_METRICS_ATTR_TW_TSVAL,
803                                 tm->tcpm_ts) < 0)
804                         goto nla_put_failure;
805         }
806
807         {
808                 int n = 0;
809
810                 nest = nla_nest_start(msg, TCP_METRICS_ATTR_VALS);
811                 if (!nest)
812                         goto nla_put_failure;
813                 for (i = 0; i < TCP_METRIC_MAX_KERNEL + 1; i++) {
814                         u32 val = tm->tcpm_vals[i];
815
816                         if (!val)
817                                 continue;
818                         if (i == TCP_METRIC_RTT) {
819                                 if (nla_put_u32(msg, TCP_METRIC_RTT_US + 1,
820                                                 val) < 0)
821                                         goto nla_put_failure;
822                                 n++;
823                                 val = max(val / 1000, 1U);
```

```
824                         }
825                         if (i == TCP_METRIC_RTTVAR) {
826                                 if (nla_put_u32(msg, TCP_METRIC_RTTVAR_US + 1,
827                                                 val) < 0)
828                                         goto nla_put_failure;
829                                 n++;
830                                 val = max(val / 1000, 1U);
831                         }
832                         if (nla_put_u32(msg, i + 1, val) < 0)
833                                 goto nla_put_failure;
834                         n++;
835                 }
836                 if (n)
837                         nla_nest_end(msg, nest);
838                 else
839                         nla_nest_cancel(msg, nest);
840         }
841
842         {
843                 struct tcp_fastopen_metrics tfom_copy[1], *tfom;
844                 unsigned int seq;
845
846                 do {
847                         seq = read_seqbegin(&fastopen_seqlock);
848                         tfom_copy[0] = tm->tcpm_fastopen;
849                 } while (read_seqretry(&fastopen_seqlock, seq));
850
851                 tfom = tfom_copy;
852                 if (tfom->mss &&
853                     nla_put_u16(msg, TCP_METRICS_ATTR_FOPEN_MSS,
854                                 tfom->mss) < 0)
855                         goto nla_put_failure;
856                 if (tfom->syn_loss &&
857                     (nla_put_u16(msg, TCP_METRICS_ATTR_FOPEN_SYN_DROPS,
858                                 tfom->syn_loss) < 0 ||
859                      nla_put_msecs(msg, TCP_METRICS_ATTR_FOPEN_SYN_DROP_TS,
860                                 jiffies - tfom->last_syn_loss) < 0))
861                         goto nla_put_failure;
862                 if (tfom->cookie.len > 0 &&
863                     nla_put(msg, TCP_METRICS_ATTR_FOPEN_COOKIE,
864                             tfom->cookie.len, tfom->cookie.val) < 0)
865                         goto nla_put_failure;
866         }
867
868         return 0;
869
870 nla_put_failure:
871         return -EMSGSIZE;
872 }
873
874 static int tcp_metrics_dump_info(struct sk_buff *skb,
875                                  struct netlink_callback *cb,
876                                  struct tcp_metrics_block *tm)
877 {
878         void *hdr;
879
880         hdr = genlmsg_put(skb, NETLINK_CB(cb->skb).portid, cb->nlh->nlmsg_seq,
881                           &tcp_metrics_nl_family, NLM_F_MULTI,
882                           TCP_METRICS_CMD_GET);
883         if (!hdr)
884                 return -EMSGSIZE;
885
886         if (tcp_metrics_fill_info(skb, tm) < 0)
887                 goto nla_put_failure;
888
```

```
889         return genlmsg_end(skb, hdr);
890
891 nla_put_failure:
892         genlmsg_cancel(skb, hdr);
893         return -EMSGSIZE;
894 }
895
896 static int tcp_metrics_nl_dump(struct sk_buff *skb,
897                                 struct netlink_callback *cb)
898 {
899         struct net *net = sock_net(skb->sk);
900         unsigned int max_rows = 1U << net->ipv4.tcp_metrics_hash_log;
901         unsigned int row, s_row = cb->args[0];
902         int s_col = cb->args[1], col = s_col;
903
904         for (row = s_row; row < max_rows; row++, s_col = 0) {
905                 struct tcp_metrics_block *tm;
906                 struct tcpm_hash_bucket *hb = net->ipv4.tcp_metrics_hash + row;
907
908                 rcu_read_lock();
909                 for (col = 0, tm = rcu_dereference(hb->chain); tm;
910                      tm = rcu_dereference(tm->tcpm_next), col++) {
911                         if (col < s_col)
912                                 continue;
913                         if (tcp_metrics_dump_info(skb, cb, tm) < 0) {
914                                 rcu_read_unlock();
915                                 goto done;
916                         }
917                 }
918                 rcu_read_unlock();
919         }
920
921 done:
922         cb->args[0] = row;
923         cb->args[1] = col;
924         return skb->len;
925 }
926
927 static int __parse_nl_addr(struct genl_info *info, struct inetpeer_addr *addr,
928                            unsigned int *hash, int optional, int v4, int v6)
929 {
930         struct nlattr *a;
931
932         a = info->attrs[v4];
933         if (a) {
934                 addr->family = AF_INET;
935                 addr->addr.a4 = nla_get_be32(a);
936                 if (hash)
937                         *hash = (__force unsigned int) addr->addr.a4;
938                 return 0;
939         }
940         a = info->attrs[v6];
941         if (a) {
942                 if (nla_len(a) != sizeof(struct in6_addr))
943                         return -EINVAL;
944                 addr->family = AF_INET6;
945                 memcpy(addr->addr.a6, nla_data(a), sizeof(addr->addr.a6));
946                 if (hash)
947                         *hash = ipv6_addr_hash((struct in6_addr *) addr->addr.a6);
948                 return 0;
949         }
950         return optional ? 1 : -EAFNOSUPPORT;
951 }
952
953 static int parse_nl_addr(struct genl_info *info, struct inetpeer_addr *addr,
```

```
954                         unsigned int *hash, int optional)
955 {
956         return __parse_nl_addr(info, addr, hash, optional,
957                                TCP_METRICS_ATTR_ADDR_IPV4,
958                                TCP_METRICS_ATTR_ADDR_IPV6);
959 }
960
961 static int parse_nl_saddr(struct genl_info *info, struct inetpeer_addr *addr)
962 {
963         return __parse_nl_addr(info, addr, NULL, 0,
964                                TCP_METRICS_ATTR_SADDR_IPV4,
965                                TCP_METRICS_ATTR_SADDR_IPV6);
966 }
967
968 static int tcp_metrics_nl_cmd_get(struct sk_buff *skb, struct genl_info *info)
969 {
970         struct tcp_metrics_block *tm;
971         struct inetpeer_addr saddr, daddr;
972         unsigned int hash;
973         struct sk_buff *msg;
974         struct net *net = genl_info_net(info);
975         void *reply;
976         int ret;
977         bool src = true;
978
979         ret = parse_nl_addr(info, &daddr, &hash, 0);
980         if (ret < 0)
981                 return ret;
982
983         ret = parse_nl_saddr(info, &saddr);
984         if (ret < 0)
985                 src = false;
986
987         msg = nlmsg_new(NLMSG_DEFAULT_SIZE, GFP_KERNEL);
988         if (!msg)
989                 return -ENOMEM;
990
991         reply = genlmsg_put_reply(msg, info, &tcp_metrics_nl_family, 0,
992                                   info->genlhdr->cmd);
993         if (!reply)
994                 goto nla_put_failure;
995
996         hash = hash_32(hash, net->ipv4.tcp_metrics_hash_log);
997         ret = -ESRCH;
998         rcu_read_lock();
999         for (tm = rcu_dereference(net->ipv4.tcp_metrics_hash[hash].chain); tm;
1000              tm = rcu_dereference(tm->tcpm_next)) {
1001                 if (addr_same(&tm->tcpm_daddr, &daddr) &&
1002                     (!src || addr_same(&tm->tcpm_saddr, &saddr))) {
1003                         ret = tcp_metrics_fill_info(msg, tm);
1004                         break;
1005                 }
1006         }
1007         rcu_read_unlock();
1008         if (ret < 0)
1009                 goto out_free;
1010
1011         genlmsg_end(msg, reply);
1012         return genlmsg_reply(msg, info);
1013
1014 nla_put_failure:
1015         ret = -EMSGSIZE;
1016
1017 out_free:
1018         nlmsg_free(msg);
```

```
1019         return ret;
1020 }
1021
1022 #define deref_locked_genl(p)    \
1023         rcu_dereference_protected(p, lockdep_genl_is_held() && \
1024                                   lockdep_is_held(&tcp_metrics_lock))
1025
1026 #define deref_genl(p)   rcu_dereference_protected(p, lockdep_genl_is_held())
1027
1028 static int tcp_metrics_flush_all(struct net *net)
1029 {
1030         unsigned int max_rows = 1U << net->ipv4.tcp_metrics_hash_log;
1031         struct tcpm_hash_bucket *hb = net->ipv4.tcp_metrics_hash;
1032         struct tcp_metrics_block *tm;
1033         unsigned int row;
1034
1035         for (row = 0; row < max_rows; row++, hb++) {
1036                 spin_lock_bh(&tcp_metrics_lock);
1037                 tm = deref_locked_genl(hb->chain);
1038                 if (tm)
1039                         hb->chain = NULL;
1040                 spin_unlock_bh(&tcp_metrics_lock);
1041                 while (tm) {
1042                         struct tcp_metrics_block *next;
1043
1044                         next = deref_genl(tm->tcpm_next);
1045                         kfree_rcu(tm, rcu_head);
1046                         tm = next;
1047                 }
1048         }
1049         return 0;
1050 }
1051
1052 static int tcp_metrics_nl_cmd_del(struct sk_buff *skb, struct genl_info *info)
1053 {
1054         struct tcpm_hash_bucket *hb;
1055         struct tcp_metrics_block *tm;
1056         struct tcp_metrics_block __rcu **pp;
1057         struct inetpeer_addr saddr, daddr;
1058         unsigned int hash;
1059         struct net *net = genl_info_net(info);
1060         int ret;
1061         bool src = true, found = false;
1062
1063         ret = parse_nl_addr(info, &daddr, &hash, 1);
1064         if (ret < 0)
1065                 return ret;
1066         if (ret > 0)
1067                 return tcp_metrics_flush_all(net);
1068         ret = parse_nl_saddr(info, &saddr);
1069         if (ret < 0)
1070                 src = false;
1071
1072         hash = hash_32(hash, net->ipv4.tcp_metrics_hash_log);
1073         hb = net->ipv4.tcp_metrics_hash + hash;
1074         pp = &hb->chain;
1075         spin_lock_bh(&tcp_metrics_lock);
1076         for (tm = deref_locked_genl(*pp); tm; tm = deref_locked_genl(*pp)) {
1077                 if (addr_same(&tm->tcpm_daddr, &daddr) &&
1078                     (!src || addr_same(&tm->tcpm_saddr, &saddr))) {
1079                         *pp = tm->tcpm_next;
1080                         kfree_rcu(tm, rcu_head);
1081                         found = true;
1082                 } else {
1083                         pp = &tm->tcpm_next;
```

```
1084                    }
1085            }
1086            spin_unlock_bh(&tcp_metrics_lock);
1087            if (!found)
1088                    return -ESRCH;
1089            return 0;
1090 }
1091
1092 static const struct genl_ops tcp_metrics_nl_ops[] = {
1093            {
1094                    .cmd = TCP_METRICS_CMD_GET,
1095                    .doit = tcp_metrics_nl_cmd_get,
1096                    .dumpit = tcp_metrics_nl_dump,
1097                    .policy = tcp_metrics_nl_policy,
1098            },
1099            {
1100                    .cmd = TCP_METRICS_CMD_DEL,
1101                    .doit = tcp_metrics_nl_cmd_del,
1102                    .policy = tcp_metrics_nl_policy,
1103                    .flags = GENL_ADMIN_PERM,
1104            },
1105 };
1106
1107 static unsigned int tcpmhash_entries;
1108 static int __init set_tcpmhash_entries(char *str)
1109 {
1110            ssize_t ret;
1111
1112            if (!str)
1113                    return 0;
1114
1115            ret = kstrtouint(str, 0, &tcpmhash_entries);
1116            if (ret)
1117                    return 0;
1118
1119            return 1;
1120 }
1121 __setup("tcpmhash_entries=", set_tcpmhash_entries);
1122
1123 static int __net_init tcp_net_metrics_init(struct net *net)
1124 {
1125            size_t size;
1126            unsigned int slots;
1127
1128            slots = tcpmhash_entries;
1129            if (!slots) {
1130                    if (totalram_pages >= 128 * 1024)
1131                            slots = 16 * 1024;
1132                    else
1133                            slots = 8 * 1024;
1134            }
1135
1136            net->ipv4.tcp_metrics_hash_log = order_base_2(slots);
1137            size = sizeof(struct tcpm_hash_bucket) << net->ipv4.tcp_metrics_hash_log;
1138
1139            net->ipv4.tcp_metrics_hash = kzalloc(size, GFP_KERNEL | __GFP_NOWARN);
1140            if (!net->ipv4.tcp_metrics_hash)
1141                    net->ipv4.tcp_metrics_hash = vzalloc(size);
1142
1143            if (!net->ipv4.tcp_metrics_hash)
1144                    return -ENOMEM;
1145
1146            return 0;
1147 }
1148
```

```
1149 static void __net_exit tcp_net_metrics_exit(struct net *net)
1150 {
1151         unsigned int i;
1152
1153         for (i = 0; i < (1U << net->ipv4.tcp_metrics_hash_log) ; i++) {
1154                 struct tcp_metrics_block *tm, *next;
1155
1156                 tm = rcu_dereference_protected(net->ipv4.tcp_metrics_hash[i].chain, 1);
1157                 while (tm) {
1158                         next = rcu_dereference_protected(tm->tcpm_next, 1);
1159                         kfree(tm);
1160                         tm = next;
1161                 }
1162         }
1163         kvfree(net->ipv4.tcp_metrics_hash);
1164 }
1165
1166 static __net_initdata struct pernet_operations tcp_net_metrics_ops = {
1167         .init   =       tcp_net_metrics_init,
1168         .exit   =       tcp_net_metrics_exit,
1169 };
1170
1171 void __init tcp_metrics_init(void)
1172 {
1173         int ret;
1174
1175         ret = register_pernet_subsys(&tcp_net_metrics_ops);
1176         if (ret < 0)
1177                 goto cleanup;
1178         ret = genl_register_family_with_ops(&tcp_metrics_nl_family,
1179                                             tcp_metrics_nl_ops);
1180         if (ret < 0)
1181                 goto cleanup_subsys;
1182         return;
1183
1184 cleanup_subsys:
1185         unregister_pernet_subsys(&tcp_net_metrics_ops);
1186
1187 cleanup:
1188         return;
1189 }
1190
```