

简介

主要目的是为了检测由 RTO 偏小引起的伪 RTO。主要算法是：

当重传定时器超时后，记这时候的 `snd.nxt` 为 `highmark`。然后看收到的两个确认是否符合这个要求：`ack > snd.una && ack < highmark`，如果都符合，那么就宣布是个伪 RTO。

FRTTO 先重传第一个未被确认的包，并且在收到第一个确认后做如下判断：a. 如果这个 `ack` 确认了新的数据但是序列号小于 `highmark`，那么再发 2 个新的数据包，等收到第二个确认再来判断。b. 这个 `ack` 是个重复确认或者大于等于 `highmark`，说明原来的包是真丢了，那么说明不是一个伪 FRTTO，还是进入到传统的 RTO 恢复当中去。对于收到的第二个确认，也是做和第一个确认相同的判断，只不过对符合要求的情况不需要在发包了，直接宣布是伪 RTO。

实现

F-RTO is implemented (mainly) in four functions:

- * - `tcp_use_frto()` is used to determine if TCP is can use F-RTO
 - * - `tcp_enter_frto()` prepares TCP state on RTO if F-RTO is used, it is called when `tcp_use_frto()` showed green light
 - * - `tcp_process_frto()` handles incoming ACKs during F-RTO algorithm
 - * - `tcp_enter_frto_loss()` is called if there is not enough evidence to prove that the * RTO is indeed spurious. It transfers the control from F-RTO to the conventional RTO recovery
- */

1. `tcp_enter_frto()`进入 FRTTO 状态

```
void tcp_enter_frto(struct sock *sk)
{
    const struct inet_connection_sock *icsk = inet_csk(sk);
    struct tcp_sock *tp = tcp_sk(sk);
    struct sk_buff *skb;

    tp->frto_counter = 1; //表示刚刚进入 FRTTO 阶段

    if (icsk->icsk_ca_state <= TCP_CA_Disorder ||
        tp->snd_una == tp->high_seq ||
        (icsk->icsk_ca_state == TCP_CA_Loss && !icsk->icsk_retransmits)) {
        /*进入 FRTTO 的处理，如果当前网络状况还可以，比如 ca_state 是 OPEN 或者 Disorder;
        snd_una == high_seq, 只差一个包没有被确认； 虽然是 LOSS 状态，但是之前没有发生过
        超时重传， 那么就记录下现在的 ssthresh，并告知拥塞控制算法出现了 EVENT_FRTO 事件
        */
        tp->prior_ssthresh = tcp_current_ssthresh(sk);
        tp->snd_ssthresh = icsk->icsk_ca_ops->ssthresh(sk);
        tcp_ca_event(sk, CA_EVENT_FRTO);
    }
}
```

```

}

/* Have to clear retransmission markers here to keep the bookkeeping
 * in shape, even though we are not yet in Loss state.
 * If something was really lost, it is eventually caught up
 * in tcp_enter_frto_loss.
 */
/* 为进入 LOSS 状态做点准备工作*/
tp->retrans_out = 0;
tp->undo_marker = tp->snd_una;
tp->undo_retrans = 0;
/* 把发送队列 write_queue 上所有已经发送过的包都置为未重传*/
sk_stream_for_retrans_queue(skb, sk) {
    TCP_SKB_CB(skb)->sacked &= ~TCPCB_RETRANS;
}
tcp_sync_left_out(tp);
/* FRTO 状态实际上还是 OPEN 状态啦 */
tcp_set_ca_state(sk, TCP_CA_Open);
/* 记录下进入 FRTO 时的 snd_nxt */
tp->frto_highmark = tp->snd_nxt;
}

```

2. 处理 FRTO 状态

当接收方收到 ACK 后，在 `tcp_ack()` 中会检测当前是否处于 FRTO 状态，如果是的话 (`tp->frto_counter != 0`)，就会调用 `tcp_process_frto` 来处理。

```

static void tcp_process_frto(struct sock *sk, u32 prior_snd_una)
{
    struct tcp_sock *tp = tcp_sk(sk);

    tcp_sync_left_out(tp);
    /* 如果收到当前的确认后，snd_una 并没有前进，说明又是一个 DACK；或者
    snd_una >= 进入 FRTO 时的 snd_nxt，所有发出去的包都收到了，那么很有可能是重传的
    包填了 hole，也就是说第一次重传的数据是真丢了。这两种情况都可以判断出之前的包是
    真丢了，所以还是进入 LOSS 状态吧 */
    if (tp->snd_una == prior_snd_una ||
        !before(tp->snd_una, tp->frto_highmark)) {
        /* RTO was caused by loss, start retransmitting in
        * go-back-N slow start
        */
        tcp_enter_frto_loss(sk);
        return;
    }
    /* 如果收到的确认是大于 snd.una 且小于 frto_highmark 的，那就符合 spurious RTO 的预期了*/
}

```

```

if (tp->frto_counter == 1) {
/* 这是进入 FRTTO 后收到的第一个 ACK */
/* First ACK after RTO advances the window: allow two new
   * segments out.
   */
    tp->snd_cwnd = tcp_packets_in_flight(tp) + 2;
} else {
/* 这是进入 FRTTO 后收到的第 2 个 ACK */
/* Also the second ACK after RTO advances the window.
   * The RTO was likely spurious. Reduce cwnd and continue
   * in congestion avoidance
   */
    tp->snd_cwnd = min(tp->snd_cwnd, tp->snd_ssthresh);
    tcp_moderate_cwnd(tp);
}

/* F-RTO affects on two new ACKs following RTO.
   * At latest on third ACK the TCP behavior is back to normal.
   */
/* frto_counter 的取值为 0,1,2,1 表示刚刚进入 FRTTO，还没有收到 ACK； 2 表示已经收到了一个 ACK，这是第二个 ACK； 0 表示已经收到了 2 个 ACK，FRTTO 阶段已经结束了，在 tcp_ack 里也不会再调用 tcp_process_ftro 了。*/
tp->frto_counter = (tp->frto_counter + 1) % 3;
}

```

3. 没有足够的证据表明是个伪 RTO，那么就进入到 Loss 状态去吧

```

static void tcp_enter_frto_loss(struct sock *sk)
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct sk_buff *skb;
    int cnt = 0;

    tp->sacked_out = 0;
    tp->lost_out = 0;
    tp->fackets_out = 0;

    sk_stream_for_retrans_queue(skb, sk) { //遍历 write_queue 里面所有已经发了的包
        cnt += tcp_skb_pcount(skb);
        TCP_SKB_CB(skb)->sacked &= ~TCPCB_LOST; //打上丢包的标记
        if (!(TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED)) {
//如果之前没有被 SACK 确认并且序列号小于 frto_highmark 的包，都标记为丢了

            /* Do not mark those segments lost that were
               * forward transmitted after RTO
               */

```

```

        if (!after(TCP_SKB_CB(skb)->end_seq,
                  tp->frto_highmark))
            TCP_SKB_CB(skb)->sacked |= TCPCB_LOST;
            tp->lost_out += tcp_skb_pcount(skb);
        }
    } else { //被 SACK 确认过
        tp->sacked_out += tcp_skb_pcount(skb);
        tp->fackets_out = cnt;
    }
}

tcp_sync_left_out(tp);
/* 看看之前在 FRTO 期间发了几个包，如果 frto_counter = 1 的话，表示收到了一个确认，
但是没有发新包；如果 frto_counter = 2 的话，收到了 2 个确认，没有新包； 如果
frto_counter = 0 的话，表示收到了 2 个确认，发了 2 个新包。所以 + frto_counter 会使得
in_flight 的数目保持不变。*/
tp->snd_cwnd = tp->frto_counter + tcp_packets_in_flight(tp)+1;
tp->snd_cwnd_cnt = 0;
tp->snd_cwnd_stamp = tcp_time_stamp;
tp->undo_marker = 0; /* 不能撤销窗口调整*/
tp->frto_counter = 0;

tp->reordering = min_t(unsigned int, tp->reordering,
                        sysctl_tcp_reordering);
tcp_set_ca_state(sk, TCP_CA_Loss);
tp->high_seq = tp->frto_highmark; //进入 FRTO 是的 snd_nxt
TCP_ECN_queue_cwr(tp);

clear_all_retrans_hints(tp); //清空所有的重传提示，重传的时候从 write_queue 的起点
开始遍历
}

```