

SNMP counter

This document explains the meaning of SNMP counters.

General IPv4 counters

All layer 4 packets and ICMP packets will change these counters, but these counters won't be changed by layer 2 packets (such as STP) or ARP packets.

- [IpInReceives](#)

Defined in [RFC1213 ipInReceives](#)

The number of packets received by the IP layer. It gets increasing at the beginning of `ip_rcv` function, always be updated together with `IpExtInOctets`. It will be increased even if the packet is dropped later (e.g. due to the IP header is invalid or the checksum is wrong and so on). It indicates the number of aggregated segments after GRO/LRO.

- [IpInDelivers](#)

Defined in [RFC1213 ipInDelivers](#)

The number of packets delivers to the upper layer protocols. E.g. TCP, UDP, ICMP and so on. If no one listens on a raw socket, only kernel supported protocols will be delivered, if someone listens on the raw socket, all valid IP packets will be delivered.

- [IpOutRequests](#)

Defined in [RFC1213 ipOutRequests](#)

The number of packets sent via IP layer, for both single cast and multicast packets, and would always be updated together with `IpExtOutOctets`.

- [IpExtInOctets](#) and [IpExtOutOctets](#)

They are Linux kernel extensions, no RFC definitions. Please note, RFC1213 indeed defines `ifInOctets` and `ifOutOctets`, but they are different things. The `ifInOctets` and `ifOutOctets` include the MAC layer header size but `IpExtInOctets` and `IpExtOutOctets` don't, they only include the IP layer header and the IP layer data.

- [IpExtInNoECTPkts](#), [IpExtInECT1Pkts](#), [IpExtInECT0Pkts](#), [IpExtInCEPkts](#)

They indicate the number of four kinds of ECN IP packets, please refer [Explicit Congestion Notification](#) for more details.

These 4 counters calculate how many packets received per ECN status. They count the real frame number regardless the LRO/GRO. So for the same packet, you might find that `IpInReceives` count 1, but `IpExtInNoECTPkts` counts 2 or more.

- [IpInHdrErrors](#)

Defined in [RFC1213 ipInHdrErrors](#). It indicates the packet is dropped due to the IP header error. It might happen in both IP input and IP forward paths.

- [IpInAddrErrors](#)

Defined in [RFC1213 ipInAddrErrors](#). It will be increased in two scenarios: (1) The IP address is invalid. (2) The destination IP address is not a local address and IP forwarding is not enabled

- [IpExtInNoRoutes](#)

This counter means the packet is dropped when the IP stack receives a packet and can't find a route for it from the route table. It might happen when IP forwarding is enabled and the destination IP address is not a local address and there is no route for the destination IP address.

- [IpInUnknownProtos](#)

Defined in [RFC1213 ipInUnknownProtos](#). It will be increased if the layer 4 protocol is unsupported by kernel. If an application is using raw socket, kernel will always deliver the packet to the raw socket and this counter won't be increased.

- [IpExtInTruncatedPkts](#)

For IPv4 packet, it means the actual data size is smaller than the "Total Length" field in the IPv4 header.

- [IpInDiscards](#)

Defined in [RFC1213 ipInDiscards](#). It indicates the packet is dropped in the IP receiving path and due to kernel internal reasons (e.g. no enough memory).

- [IpOutDiscards](#)

Defined in [RFC1213 ipOutDiscards](#). It indicates the packet is dropped in the IP sending path and due to kernel internal reasons.

- [IpOutNoRoutes](#)

Defined in [RFC1213 ipOutNoRoutes](#). It indicates the packet is dropped in the IP sending path and no route is found for it.

ICMP counters

- [IcmpInMsgs](#) and [IcmpOutMsgs](#)

Defined by [RFC1213 icmpInMsgs](#) and [RFC1213 icmpOutMsgs](#)

As mentioned in the RFC1213, these two counters include errors, they would be increased even if the ICMP packet has an invalid type. The ICMP output path will check the header of a raw socket, so the `IcmpOutMsgs` would still be updated if the IP header is constructed by a userspace program

- ICMP named types

These counters include most of common ICMP types, they are:

`IcmpInDestUnreachs`: [RFC1213 icmpInDestUnreachs](#)

`IcmpInTimeExcds`: [RFC1213 icmpInTimeExcds](#)

`IcmpInPamProbs`: [RFC1213 icmpInPamProbs](#)

`IcmpInSrcQuenchs`: [RFC1213 icmpInSrcQuenchs](#)

`IcmpInRedirects`: [RFC1213 icmpInRedirects](#)

`IcmpInEchos`: [RFC1213 icmpInEchos](#)

`IcmpInEchoReps`: [RFC1213 icmpInEchoReps](#)

`IcmpInTimestamps`: [RFC1213 icmpInTimestamps](#)

`IcmpInTimestampReps`: [RFC1213 icmpInTimestampReps](#)

`IcmpInAddrMasks`: [RFC1213 icmpInAddrMasks](#)

`IcmpInAddrMaskReps`: [RFC1213 icmpInAddrMaskReps](#)

`IcmpOutDestUnreachs`: [RFC1213 icmpOutDestUnreachs](#)

`IcmpOutTimeExcds`: [RFC1213 icmpOutTimeExcds](#)

`IcmpOutPamProbs`: [RFC1213 icmpOutPamProbs](#)

`IcmpOutSrcQuenchs`: [RFC1213 icmpOutSrcQuenchs](#)

`IcmpOutRedirects`: [RFC1213 icmpOutRedirects](#)

`IcmpOutEchos`: [RFC1213 icmpOutEchos](#)

`IcmpOutEchoReps`: [RFC1213 icmpOutEchoReps](#)

`IcmpOutTimestamps`: [RFC1213 icmpOutTimestamps](#)

`IcmpOutTimestampReps`: [RFC1213 icmpOutTimestampReps](#)

`IcmpOutAddrMasks`: [RFC1213 icmpOutAddrMasks](#)

`IcmpOutAddrMaskReps`: [RFC1213 icmpOutAddrMaskReps](#)

Every ICMP type has two counters: 'In' and 'Out'. E.g., for the ICMP Echo packet, they are `IcmpInEchos` and `IcmpOutEchos`. Their meanings are straightforward. The 'In' counter means kernel receives such a packet and the 'Out' counter means kernel sends such a packet.

- ICMP numeric types

They are `lcmplnMsgInType[N]` and `lcmplnMsgOutType[N]`, the `[N]` indicates the ICMP type number. These counters track all kinds of ICMP packets. The ICMP type number definition could be found in the [ICMP parameters](#) document.

For example, if the Linux kernel sends an ICMP Echo packet, the `lcmplnMsgOutType8` would increase 1. And if kernel gets an ICMP Echo Reply packet, `lcmplnMsgInType0` would increase 1.

- `lcmplnCsumErrors`

This counter indicates the checksum of the ICMP packet is wrong. Kernel verifies the checksum after updating the `lcmplnMsgs` and before updating `lcmplnMsgInType[N]`. If a packet has bad checksum, the `lcmplnMsgs` would be updated but none of `lcmplnMsgInType[N]` would be updated.

- `lcmplnErrors` and `lcmplnOutErrors`

Defined by [RFC1213](#) `lcmplnErrors` and [RFC1213](#) `lcmplnOutErrors`

When an error occurs in the ICMP packet handler path, these two counters would be updated. The receiving packet path use `lcmplnErrors` and the sending packet path use `lcmplnOutErrors`. When `lcmplnCsumErrors` is increased, `lcmplnErrors` would always be increased too.

relationship of the ICMP counters

The sum of `lcmplnMsgOutType[N]` is always equal to `lcmplnOutMsgs`, as they are updated at the same time. The sum of `lcmplnMsgInType[N]` plus `lcmplnErrors` should be equal or larger than `lcmplnMsgs`. When kernel receives an ICMP packet, kernel follows below logic:

1. increase `lcmplnMsgs`
2. if has any error, update `lcmplnErrors` and finish the process
3. update `lcmplnMsgOutType[N]`
4. handle the packet depending on the type, if has any error, update `lcmplnErrors` and finish the process

So if all errors occur in step (2), `lcmplnMsgs` should be equal to the sum of `lcmplnMsgOutType[N]` plus `lcmplnErrors`. If all errors occur in step (4), `lcmplnMsgs` should be equal to the sum of `lcmplnMsgOutType[N]`. If the errors occur in both step (2) and step (4), `lcmplnMsgs` should be less than the sum of `lcmplnMsgOutType[N]` plus `lcmplnErrors`.

General TCP counters

- `TcpInSegs`

Defined in [RFC1213](#) `tcpInSegs`

The number of packets received by the TCP layer. As mentioned in RFC1213, it includes the packets received in error, such as checksum error, invalid TCP header and so on. Only one error won't be included: if the layer 2 destination address is not the NIC's layer 2 address. It might happen if the packet is a multicast or broadcast packet, or the NIC is in promiscuous mode. In these situations, the packets would be delivered to the TCP layer, but the TCP layer will discard these packets before increasing `TcpInSegs`. The `TcpInSegs` counter isn't aware of GRO. So if two packets are merged by GRO, the `TcpInSegs` counter would only increase 1.

- `TcpOutSegs`

Defined in [RFC1213](#) `tcpOutSegs`

The number of packets sent by the TCP layer. As mentioned in RFC1213, it excludes the retransmitted packets. But it includes the SYN, ACK and RST packets. Doesn't like `TcpInSegs`, the `TcpOutSegs` is aware of GSO, so if a packet would be split to 2 by GSO, `TcpOutSegs` will increase 2.

- `TcpActiveOpens`

Defined in [RFC1213](#) `tcpActiveOpens`

It means the TCP layer sends a SYN, and come into the SYN-SENT state. Every time `TcpActiveOpens` increases 1, `TcpOutSegs` should always increase 1.

- `TcpPassiveOpens`

Defined in [RFC1213](#) `tcpPassiveOpens`

It means the TCP layer receives a SYN, replies a SYN+ACK, come into the SYN-RCVD state.

- `TcpExtTCPRcvCoalesce`

When packets are received by the TCP layer and are not be read by the application, the TCP layer will try to merge them. This counter indicate how many packets are merged in such situation. If GRO is enabled, lots of packets would be merged by GRO, these packets wouldn't be counted to `TcpExtTCPRcvCoalesce`.

- `TcpExtTCPAutoCorking`

When sending packets, the TCP layer will try to merge small packets to a bigger one. This counter increase 1 for every packet merged in such situation. Please refer to the LWN article for more details: <https://lwn.net/Articles/576263/>

- `TcpExtTCPOrigDataSent`

This counter is explained by [kernel commit f19c29e3e391](#), I pasted the explanation below:

```
TCPOrigDataSent: number of outgoing packets with original data (excluding
retransmission but including data-in-SYN). This counter is different from
TcpOutSegs because TcpOutSegs also tracks pure ACKs. TCPOrigDataSent is
more useful to track the TCP retransmission rate.
```

- `TCPSynRetrans`

This counter is explained by [kernel commit f19c29e3e391](#), I pasted the explanation below:

```
TCPSynRetrans: number of SYN and SYN/ACK retransmits to break down
retransmissions into SYN, fast-retransmits, timeout retransmits, etc.
```

- `TCPFastOpenActiveFail`

This counter is explained by [kernel commit f19c29e3e391](#), I pasted the explanation below:

```
TCPFastOpenActiveFail: Fast Open attempts (SYN/data) failed because
the remote does not accept it or the attempts timed out.
```

- `TcpExtListenOverflows` and `TcpExtListenDrops`

When kernel receives a SYN from a client, and if the TCP accept queue is full, kernel will drop the SYN and add 1 to `TcpExtListenOverflows`. At the same time kernel will also add 1 to `TcpExtListenDrops`. When a TCP socket is in LISTEN state, and kernel need to drop a packet, kernel would always add 1 to `TcpExtListenDrops`. So increase `TcpExtListenOverflows` would let `TcpExtListenDrops` increasing at the same time, but `TcpExtListenDrops` would also increase without `TcpExtListenOverflows` increasing, e.g. a memory allocation fail would also let `TcpExtListenDrops` increase.

Note: The above explanation is based on kernel 4.10 or above version, on an old kernel, the TCP stack has different behavior when TCP accept queue is full. On the old kernel, TCP stack won't drop the SYN, it would complete the 3-way handshake. As the accept queue is full, TCP stack will keep the socket in the TCP half-open queue. As it is in the half-open queue, TCP stack will send SYN+ACK on an exponential backoff timer, after client replies ACK, TCP stack checks whether the accept queue is still full, if it is not full, moves the socket to the accept queue, if it is full, keeps the socket in the half-open queue, at next time client replies ACK, this socket will get another chance to move to the accept queue.

TCP Fast Open

- `TcpEstabResets`

Defined in [RFC1213](#) `tcpEstabResets`.

- `TcpAttemptFails`

Defined in [RFC1213](#) `tcpAttemptFails`.

- `TcpOutRsts`

Defined in [RFC1213 tcpOutRsts](#). The RFC says this counter indicates the 'segments sent containing the RST flag', but in linux kernel, this counter indicates the segments kernel tried to send. The sending process might be failed due to some errors (e.g. memory alloc failed).

- `TcpExtTCPSpuriousRtxHostQueues`

When the TCP stack wants to retransmit a packet, and finds that packet is not lost in the network, but the packet is not sent yet, the TCP stack would give up the retransmission and update this counter. It might happen if a packet stays too long time in a qdisc or driver queue.

- `TcpEstabResets`

The socket receives a RST packet in Establish or CloseWait state.

- `TcpExtTCPKeepAlive`

This counter indicates many keepalive packets were sent. The keepalive won't be enabled by default. A userspace program could enable it by setting the `SO_KEEPALIVE` socket option.

- `TcpExtTCPSpuriousRTOs`

The spurious retransmission timeout detected by the [F-RTO](#) algorithm.

TCP Fast Path

When kernel receives a TCP packet, it has two paths to handler the packet, one is fast path, another is slow path. The comment in kernel code provides a good explanation of them, I pasted them below:

```
It is split into a fast path and a slow path. The fast path is
disabled when:

- A zero window was announced from us
- zero window probing
  is only handled properly on the slow path.
- Out of order segments arrived.
- Urgent data is expected.
- There is no buffer space left
- Unexpected TCP flags/window values/header lengths are received
  (detected by checking the TCP header against pred flags)
- Data is sent in both directions. The fast path only supports pure senders
  or pure receivers (this means either the sequence number or the ack
  value must stay constant)
- Unexpected TCP option.
```

Kernel will try to use fast path unless any of the above conditions are satisfied. If the packets are out of order, kernel will handle them in slow path, which means the performance might be not very good. Kernel would also come into slow path if the "Delayed ack" is used, because when using "Delayed ack", the data is sent in both directions. When the TCP window scale option is not used, kernel will try to enable fast path immediately when the connection comes into the established state, but if the TCP window scale option is used, kernel will disable the fast path at first, and try to enable it after kernel receives packets.

- `TcpExtTCPPureAcks` and `TcpExtTCPHPAcks`

If a packet set ACK flag and has no data, it is a pure ACK packet, if kernel handles it in the fast path, `TcpExtTCPHPAcks` will increase 1, if kernel handles it in the slow path, `TcpExtTCPPureAcks` will increase 1.

- `TcpExtTCPHPHits`

If a TCP packet has data (which means it is not a pure ACK packet), and this packet is handled in the fast path, `TcpExtTCPHPHits` will increase 1.

TCP abort

- `TcpExtTCPAbortOnData`

It means TCP layer has data in flight, but need to close the connection. So TCP layer sends a RST to the other side, indicate the connection is not closed very graceful. An easy way to increase this counter is using the `SO_LINGER` option. Please refer to the `SO_LINGER` section of the [socket man page](#):

By default, when an application closes a connection, the close function will return immediately and kernel will try to send the in-flight data async. If you use the `SO_LINGER` option, set `l_onoff` to 1, and `l_linger` to a positive number, the close function won't return immediately, but wait for the in-flight data are acked by the other side, the max wait time is `l_linger` seconds. If set `l_onoff` to 1 and set `l_linger` to 0, when the application closes a connection, kernel will send a RST immediately and increase the `TcpExtTCPAbortOnData` counter.

- `TcpExtTCPAbortOnClose`

This counter means the application has unread data in the TCP layer when the application wants to close the TCP connection. In such a situation, kernel will send a RST to the other side of the TCP connection.

- `TcpExtTCPAbortOnMemory`

When an application closes a TCP connection, kernel still need to track the connection, let it complete the TCP disconnect process. E.g. an app calls the close method of a socket, kernel sends fin to the other side of the connection, then the app has no relationship with the socket any more, but kernel need to keep the socket, this socket becomes an orphan socket, kernel waits for the reply of the other side, and would come to the `TIME_WAIT` state finally. When kernel has no enough memory to keep the orphan socket, kernel would send an RST to the other side, and delete the socket, in such situation, kernel will increase 1 to the `TcpExtTCPAbortOnMemory`. Two conditions would trigger `TcpExtTCPAbortOnMemory`:

1. the memory used by the TCP protocol is higher than the third value of the `tcp_mem`. Please refer the `tcp_mem` section in the [TCP man page](#):

- the orphan socket count is higher than `net.ipv4.tcp_max_orphans`

- `TcpExtTCPAbortOnTimeout`

This counter will increase when any of the TCP timers expire. In such situation, kernel won't send RST, just give up the connection.

- `TcpExtTCPAbortOnLinger`

When a TCP connection comes into `FIN_WAIT_2` state, instead of waiting for the fin packet from the other side, kernel could send a RST and delete the socket immediately. This is not the default behavior of Linux kernel TCP stack. By configuring the `TCP_LINGER2` socket option, you could let kernel follow this behavior.

- `TcpExtTCPAbortFailed`

The kernel TCP layer will send RST if the [RFC2525 2.17 section](#) is satisfied. If an internal error occurs during this process, `TcpExtTCPAbortFailed` will be increased.

TCP Hybrid Slow Start

The Hybrid Slow Start algorithm is an enhancement of the traditional TCP congestion window Slow Start algorithm. It uses two pieces of information to detect whether the max bandwidth of the TCP path is approached. The two pieces of information are ACK train length and increase in packet delay. For detail information, please refer the [Hybrid Slow Start paper](#). Either ACK train length or packet delay hits a specific threshold, the congestion control algorithm will come into the Congestion Avoidance state. Until v4.20, two congestion control algorithms are using Hybrid Slow Start, they are cubic (the default congestion control algorithm) and cdp. Four snmp counters relate with the Hybrid Slow Start algorithm

- `TcpExtTCPHystartTrainDetect`

How many times the ACK train length threshold is detected

- `TcpExtTCPHystartTrainCwnd`

The sum of CWND detected by ACK train length. Dividing this value by `TcpExtTCPHystartTrainDetect` is the average CWND

which detected by the ACK train length.

- `TcpExtTCPhystartDelayDetect`

How many times the packet delay threshold is detected.

- `TcpExtTCPhystartDelayCwnd`

The sum of CWND detected by packet delay. Dividing this value by `TcpExtTCPhystartDelayDetect` is the average CWND which detected by the packet delay.

TCP retransmission and congestion control

The TCP protocol has two retransmission mechanisms: SACK and fast recovery. They are exclusive with each other. When SACK is enabled, the kernel TCP stack would use SACK, or kernel would use fast recovery. The SACK is a TCP option, which is defined in [RFC2018](#), the fast recovery is defined in [RFC6582](#), which is also called 'Reno'.

The TCP congestion control is a big and complex topic. To understand the related snmp counter, we need to know the states of the congestion control state machine. There are 5 states: Open, Disorder, CWR, Recovery and Loss. For details about these states, please refer page 5 and page 6 of this document:

<https://pdfs.semanticscholar.org/0e9c/968d09ab2e53e24c4dca5b2d67c7f7140f8e.pdf>

- `TcpExtTCPRenoRecovery` and `TcpExtTCPSackRecovery`

When the congestion control comes into Recovery state, if sack is used, `TcpExtTCPSackRecovery` increases 1, if sack is not used, `TcpExtTCPRenoRecovery` increases 1. These two counters mean the TCP stack begins to retransmit the lost packets.

- `TcpExtTCPSACKReneging`

A packet was acknowledged by SACK, but the receiver has dropped this packet, so the sender needs to retransmit this packet. In this situation, the sender adds 1 to `TcpExtTCPSACKReneging`. A receiver could drop a packet which has been acknowledged by SACK, although it is unusual, it is allowed by the TCP protocol. The sender doesn't really know what happened on the receiver side. The sender just waits until the RTO expires for this packet, then the sender assumes this packet has been dropped by the receiver.

- `TcpExtTCPRenoReorder`

The reorder packet is detected by fast recovery. It would only be used if SACK is disabled. The fast recovery algorithm detects reordering by the duplicate ACK number. E.g., if retransmission is triggered, and the original retransmitted packet is not lost, it is just out of order, the receiver would acknowledge multiple times, one for the retransmitted packet, another for the arriving of the original out of order packet. Thus the sender would find more ACKs than its expectation, and the sender knows out of order occurs.

- `TcpExtTCPTSReorder`

The reorder packet is detected when a hole is filled. E.g., assume the sender sends packet 1,2,3,4,5, and the receiving order is 1,2,4,5,3. When the sender receives the ACK of packet 3 (which will fill the hole), two conditions will let `TcpExtTCPTSReorder` increase 1: (1) if the packet 3 is not re-transmitted yet. (2) if the packet 3 is retransmitted but the timestamp of the packet 3's ACK is earlier than the retransmission timestamp.

- `TcpExtTCPSACKReorder`

The reorder packet detected by SACK. The SACK has two methods to detect reorder: (1) DSACK is received by the sender. It means the sender sends the same packet more than one times. And the only reason is the sender believes an out of order packet is lost so it sends the packet again. (2) Assume packet 1,2,3,4,5 are sent by the sender, and the sender has received SACKs for packet 2 and 5, now the sender receives SACK for packet 4 and the sender doesn't retransmit the packet yet, the sender would know packet 4 is out of order. The TCP stack of kernel will increase `TcpExtTCPSACKReorder` for both of the above scenarios.

- `TcpExtTCPSlowStartRetrans`

The TCP stack wants to retransmit a packet and the congestion control state is 'Loss'.

- `TcpExtTCPFastRetrans`

The TCP stack wants to retransmit a packet and the congestion control state is not 'Loss'.

- `TcpExtTCPLostRetransmit`

A SACK points out that a retransmission packet is lost again.

- `TcpExtTCPRetransFail`

The TCP stack tries to deliver a retransmission packet to lower layers but the lower layers return an error.

- `TcpExtTCPSynRetrans`

The TCP stack retransmits a SYN packet.

DSACK

The DSACK is defined in [RFC2883](#). The receiver uses DSACK to report duplicate packets to the sender. There are two kinds of duplications: (1) a packet which has been acknowledged is duplicate. (2) an out of order packet is duplicate. The TCP stack counts these two kinds of duplications on both receiver side and sender side.

- `TcpExtTCPDSACKOldSent`

The TCP stack receives a duplicate packet which has been acked, so it sends a DSACK to the sender.

- `TcpExtTCPDSACKOfoSent`

The TCP stack receives an out of order duplicate packet, so it sends a DSACK to the sender.

- `TcpExtTCPDSACKRecv`

The TCP stack receives a DSACK, which indicates an acknowledged duplicate packet is received.

- `TcpExtTCPDSACKOfoRecv`

The TCP stack receives a DSACK, which indicate an out of order duplicate packet is received.

invalid SACK and DSACK

When a SACK (or DSACK) block is invalid, a corresponding counter would be updated. The validation method is base on the start/end sequence number of the SACK block. For more details, please refer the comment of the function `tcp_is_sackblock_valid` in the kernel source code. A SACK option could have up to 4 blocks, they are checked individually. E.g., if 3 blocks of a SACK is invalid, the corresponding counter would be updated 3 times. The comment of the [Add counters for discarded SACK blocks](#) patch has additional explanation:

- `TcpExtTCPSACKDiscard`

This counter indicates how many SACK blocks are invalid. If the invalid SACK block is caused by ACK recording, the TCP stack will only ignore it and won't update this counter.

- `TcpExtTCPDSACKIgnoredOld` and `TcpExtTCPDSACKIgnoredNoUndo`

When a DSACK block is invalid, one of these two counters would be updated. Which counter will be updated depends on the `undo_marker` flag of the TCP socket. If the `undo_marker` is not set, the TCP stack isn't likely to re-transmit any packets, and we still receive an invalid DSACK block, the reason might be that the packet is duplicated in the middle of the network. In such scenario, `TcpExtTCPDSACKIgnoredNoUndo` will be updated. If the `undo_marker` is set, `TcpExtTCPDSACKIgnoredOld` will be updated. As implied in its name, it might be an old packet.

SACK shift

The linux networking stack stores data in `sk_buff` struct (skb for short). If a SACK block acrosses multiple skb, the TCP stack will try to re-arrange data in these skb. E.g. if a SACK block acknowledges seq 10 to 15, skb1 has seq 10 to 13, skb2 has seq 14 to 20. The seq 14 and 15 in skb2 would be moved to skb1. This operation is 'shift'. If a SACK block acknowledges seq 10 to 20, skb1 has seq 10 to 13, skb2 has seq 14 to 20. All data in skb2 will be moved to skb1, and skb2 will be discard, this operation is 'merge'.

- `TcpExtTCPSackShifted`

A skb is shifted

- `TcpExtTCPSackMerged`

A skb is merged

- `TcpExtTCPSackShiftFallback`

A skb should be shifted or merged, but the TCP stack doesn't do it for some reasons.

TCP out of order

- `TcpExtTCPOFOQueue`

The TCP layer receives an out of order packet and has enough memory to queue it.

- `TcpExtTCPOFODrop`

The TCP layer receives an out of order packet but doesn't have enough memory, so drops it. Such packets won't be counted into `TcpExtTCPOFOQueue`.

- `TcpExtTCPOFOMerge`

The received out of order packet has an overlap with the previous packet. the overlap part will be dropped. All of `TcpExtTCPOFOMerge` packets will also be counted into `TcpExtTCPOFOQueue`.

TCP PAWS

PAWS (Protection Against Wrapped Sequence numbers) is an algorithm which is used to drop old packets. It depends on the TCP timestamps. For detail information, please refer the [timestamp wiki](#) and the [RFC of PAWS](#).

- `TcpExtPAWSActive`

Packets are dropped by PAWS in Syn-Sent status.

- `TcpExtPAWSEstab`

Packets are dropped by PAWS in any status other than Syn-Sent.

TCP ACK skip

In some scenarios, kernel would avoid sending duplicate ACKs too frequently. Please find more details in the `tcp_invalid_ratelimit` section of the [sysctl document](#). When kernel decides to skip an ACK due to `tcp_invalid_ratelimit`, kernel would update one of below counters to indicate the ACK is skipped in which scenario. The ACK would only be skipped if the received packet is either a SYN packet or it has no data.

- `TcpExtTCPACKSkippedSynRecv`

The ACK is skipped in Syn-Recv status. The Syn-Recv status means the TCP stack receives a SYN and replies SYN+ACK. Now the TCP stack is waiting for an ACK. Generally, the TCP stack doesn't need to send ACK in the Syn-Recv status. But in several scenarios, the TCP stack need to send an ACK. E.g., the TCP stack receives the same SYN packet repeatedly, the received packet does not pass the PAWS check, or the received packet sequence number is out of window. In these scenarios, the TCP stack needs to send ACK. If the ACK sending frequency is higher than `tcp_invalid_ratelimit` allows, the TCP stack will skip sending ACK and increase `TcpExtTCPACKSkippedSynRecv`.

- `TcpExtTCPACKSkippedPAWS`

The ACK is skipped due to PAWS (Protect Against Wrapped Sequence numbers) check fails. If the PAWS check fails in Syn-Recv, Fin-Wait-2 or Time-Wait statuses, the skipped ACK would be counted to `TcpExtTCPACKSkippedSynRecv`, `TcpExtTCPACKSkippedFinWait2` or `TcpExtTCPACKSkippedTimeWait`. In all other statuses, the skipped ACK would be counted to `TcpExtTCPACKSkippedPAWS`.

- `TcpExtTCPACKSkippedSeq`

The sequence number is out of window and the timestamp passes the PAWS check and the TCP status is not Syn-Recv, Fin-Wait-2, and Time-Wait.

- `TcpExtTCPACKSkippedFinWait2`

The ACK is skipped in Fin-Wait-2 status, the reason would be either PAWS check fails or the received sequence number is out of window.

- `TcpExtTCPACKSkippedTimeWait`

The ACK is skipped in Time-Wait status, the reason would be either PAWS check failed or the received sequence number is out of window.

- `TcpExtTCPACKSkippedChallenge`

The ACK is skipped if the ACK is a challenge ACK. The RFC 5961 defines 3 kind of challenge ACK, please refer [RFC 5961 section 3.2](#), [RFC 5961 section 4.2](#) and [RFC 5961 section 5.2](#). Besides these three scenarios, In some TCP status, the linux TCP stack would also send challenge ACKs if the ACK number is before the first unacknowledged number (more strict than [RFC 5961 section 5.2](#)).

TCP receive window

- `TcpExtTCPWantZeroWindowAdv`

Depending on current memory usage, the TCP stack tries to set receive window to zero. But the receive window might still be a no-zero value. For example, if the previous window size is 10, and the TCP stack receives 3 bytes, the current window size would be 7 even if the window size calculated by the memory usage is zero.

- `TcpExtTCPToZeroWindowAdv`

The TCP receive window is set to zero from a no-zero value.

- `TcpExtTCPFromZeroWindowAdv`

The TCP receive window is set to no-zero value from zero.

Delayed ACK

The TCP Delayed ACK is a technique which is used for reducing the packet count in the network. For more details, please refer the [Delayed ACK wiki](#)

- `TcpExtDelayedACKs`

A delayed ACK timer expires. The TCP stack will send a pure ACK packet and exit the delayed ACK mode.

- `TcpExtDelayedACKLocked`

A delayed ACK timer expires, but the TCP stack can't send an ACK immediately due to the socket is locked by a userspace program. The TCP stack will send a pure ACK later (after the userspace program unlock the socket). When the TCP stack sends the pure ACK later, the TCP stack will also update `TcpExtDelayedACKs` and exit the delayed ACK mode.

- `TcpExtDelayedACKLost`

It will be updated when the TCP stack receives a packet which has been ACKed. A Delayed ACK loss might cause this issue, but it would also be triggered by other reasons, such as a packet is duplicated in the network.

Tail Loss Probe (TLP)

TLP is an algorithm which is used to detect TCP packet loss. For more details, please refer the [TLP paper](#).

- TcpExtTCPLossProbes

A TLP probe packet is sent.

- TcpExtTCPLossProbeRecovery

A packet loss is detected and recovered by TLP.

TCP Fast Open description

TCP Fast Open is a technology which allows data transfer before the 3-way handshake complete. Please refer the [TCP Fast Open wiki](#) for a general description.

- TcpExtTCPFastOpenActive

When the TCP stack receives an ACK packet in the SYN-SENT status, and the ACK packet acknowledges the data in the SYN packet, the TCP stack understand the TFO cookie is accepted by the other side, then it updates this counter.

- TcpExtTCPFastOpenActiveFail

This counter indicates that the TCP stack initiated a TCP Fast Open, but it failed. This counter would be updated in three scenarios: (1) the other side doesn't acknowledge the data in the SYN packet. (2) The SYN packet which has the TFO cookie is timeout at least once. (3) after the 3-way handshake, the retransmission timeout happens net.ipv4.tcp_retries1 times, because some middle-boxes may black-hole fast open after the handshake.

- TcpExtTCPFastOpenPassive

This counter indicates how many times the TCP stack accepts the fast open request.

- TcpExtTCPFastOpenPassiveFail

This counter indicates how many times the TCP stack rejects the fast open request. It is caused by either the TFO cookie is invalid or the TCP stack finds an error during the socket creating process.

- TcpExtTCPFastOpenListenOverflow

When the pending fast open request number is larger than fastopenq->max_qlen, the TCP stack will reject the fast open request and update this counter. When this counter is updated, the TCP stack won't update TcpExtTCPFastOpenPassive or TcpExtTCPFastOpenPassiveFail. The fastopenq->max_qlen is set by the TCP_FASTOPEN socket operation and it could not be larger than net.core.somaxconn. For example:

```
setsockopt(sockfd, SOL_TCP, TCP_FASTOPEN, &qlen, sizeof(qlen));
```

- TcpExtTCPFastOpenCookieReqd

This counter indicates how many times a client wants to request a TFO cookie.

SYN cookies

SYN cookies are used to mitigate SYN flood, for details, please refer the [SYN cookies wiki](#).

- TcpExtSyncookiesSent

It indicates how many SYN cookies are sent.

- TcpExtSyncookiesRecv

How many reply packets of the SYN cookies the TCP stack receives.

- TcpExtSyncookiesFailed

The MSS decoded from the SYN cookie is invalid. When this counter is updated, the received packet won't be treated as a SYN cookie and the TcpExtSyncookiesRecv counter won't be updated.

Challenge ACK

For details of challenge ACK, please refer the explanation of TcpExtTCPACKSkippedChallenge.

- TcpExtTCPChallengeACK

The number of challenge acks sent.

- TcpExtTCPSYNChallenge

The number of challenge acks sent in response to SYN packets. After updates this counter, the TCP stack might send a challenge ACK and update the TcpExtTCPChallengeACK counter, or it might also skip to send the challenge and update the TcpExtTCPACKSkippedChallenge.

prune

When a socket is under memory pressure, the TCP stack will try to reclaim memory from the receiving queue and out of order queue. One of the reclaiming method is 'collapse', which means allocate a big skb, copy the contiguous skbs to the single big skb, and free these contiguous skbs.

- TcpExtPruneCalled

The TCP stack tries to reclaim memory for a socket. After updates this counter, the TCP stack will try to collapse the out of order queue and the receiving queue. If the memory is still not enough, the TCP stack will try to discard packets from the out of order queue (and update the TcpExtOfoPruned counter)

- TcpExtOfoPruned

The TCP stack tries to discard packet on the out of order queue.

- TcpExtRcvPruned

After 'collapse' and discard packets from the out of order queue, if the actually used memory is still larger than the max allowed memory, this counter will be updated. It means the 'prune' fails.

- TcpExtTCPRecvCollapsed

This counter indicates how many skbs are freed during 'collapse'.

examples

ping test

Run the ping command against the public dns server 8.8.8.8:

```
nstatuser@nstat-a:~$ ping 8.8.8.8 -c 1
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=119 time=17.8 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 17.875/17.875/17.875/0.000 ms
```

The nstat result:

```
nstatuser@nstat-a:~$ nstat
#kernel
IpInReceives          1          0.0
IpInDelivers          1          0.0
IpOutRequests         1          0.0
IcmpInMsgs            1          0.0
IcmpInEchoReps        1          0.0
IcmpOutMsgs           1          0.0
IcmpOutEchos          1          0.0
IcmpMsgInType0        1          0.0
```

IcmpMsgOutType8	1	0.0
IpExtInOctets	84	0.0
IpExtOutOctets	84	0.0
IpExtInNoECTPkts	1	0.0

The Linux server sent an ICMP Echo packet, so IpOutRequests, IcmpOutMsgs, IcmpOutEchos and IcmpMsgOutType8 were increased 1. The server got ICMP Echo Reply from 8.8.8.8, so IplnReceives, IcmpInMsgs, IcmpInEchoReps and IcmpMsgInType0 were increased 1. The ICMP Echo Reply was passed to the ICMP layer via IP layer, so IplnDelivers was increased 1. The default ping data size is 48, so an ICMP Echo packet and its corresponding Echo Reply packet are constructed by:

- 14 bytes MAC header
- 20 bytes IP header
- 16 bytes ICMP header
- 48 bytes data (default value of the ping command)

So the IpExtInOctets and IpExtOutOctets are 20+16+48=84.

tcp 3-way handshake

On server side, we run:

```
nstatuser@nstat-b:~$ nc -lknv 0.0.0.0 9000
Listening on [0.0.0.0] (family 0, port 9000)
```

On client side, we run:

```
nstatuser@nstat-a:~$ nc -nv 192.168.122.251 9000
Connection to 192.168.122.251 9000 port [tcp/*] succeeded!
```

The server listened on tcp 9000 port, the client connected to it, they completed the 3-way handshake.

On server side, we can find below nstat output:

```
nstatuser@nstat-b:~$ nstat | grep -i tcp
TcpPassiveOpens      1          0.0
TcpInSegs             2          0.0
TcpOutSegs            1          0.0
TcpExtTCPPureAcks     1          0.0
```

On client side, we can find below nstat output:

```
nstatuser@nstat-a:~$ nstat | grep -i tcp
TcpActiveOpens        1          0.0
TcpInSegs              1          0.0
TcpOutSegs             2          0.0
```

When the server received the first SYN, it replied a SYN+ACK, and came into SYN-RCVD state, so TcpPassiveOpens increased 1. The server received SYN, sent SYN+ACK, received ACK, so server sent 1 packet, received 2 packets, TcpInSegs increased 2, TcpOutSegs increased 1. The last ACK of the 3-way handshake is a pure ACK without data, so TcpExtTCPPureAcks increased 1.

When the client sent SYN, the client came into the SYN-SENT state, so TcpActiveOpens increased 1, the client sent SYN, received SYN+ACK, sent ACK, so client sent 2 packets, received 1 packet, TcpInSegs increased 1, TcpOutSegs increased 2.

TCP normal traffic

Run nc on server:

```
nstatuser@nstat-b:~$ nc -lkv 0.0.0.0 9000
Listening on [0.0.0.0] (family 0, port 9000)
```

Run nc on client:

```
nstatuser@nstat-a:~$ nc -v nstat-b 9000
Connection to nstat-b 9000 port [tcp/*] succeeded!
```

Input a string in the nc client ('hello' in our example):

```
nstatuser@nstat-a:~$ nc -v nstat-b 9000
Connection to nstat-b 9000 port [tcp/*] succeeded!
hello
```

The client side nstat output:

```
nstatuser@nstat-a:~$ nstat
#kernel
IpInReceives          1          0.0
IpInDelivers           1          0.0
IpOutRequests          1          0.0
TcpInSegs              1          0.0
TcpOutSegs             1          0.0
TcpExtTCPPureAcks      1          0.0
TcpExtTCPOrigDataSent  1          0.0
IpExtInOctets          52          0.0
IpExtOutOctets         58          0.0
IpExtInNoECTPkts       1          0.0
```

The server side nstat output:

```
nstatuser@nstat-b:~$ nstat
#kernel
IpInReceives          1          0.0
IpInDelivers           1          0.0
IpOutRequests          1          0.0
TcpInSegs              1          0.0
TcpOutSegs             1          0.0
IpExtInOctets          58          0.0
IpExtOutOctets         52          0.0
IpExtInNoECTPkts       1          0.0
```

Input a string in nc client side again ('world' in our example):

```
nstatuser@nstat-a:~$ nc -v nstat-b 9000
Connection to nstat-b 9000 port [tcp/*] succeeded!
hello
world
```

Client side nstat output:

```
nstatuser@nstat-a:~$ nstat
#kernel
IpInReceives          1          0.0
IpInDelivers           1          0.0
IpOutRequests          1          0.0
TcpInSegs              1          0.0
TcpOutSegs             1          0.0
TcpExtTCPHPacks        1          0.0
TcpExtTCPOrigDataSent  1          0.0
IpExtInOctets          52          0.0
IpExtOutOctets         58          0.0
IpExtInNoECTPkts       1          0.0
```

Server side nstat output:

```
nstatuser@nstat-b:~$ nstat
#kernel
IpInReceives          1          0.0
IpInDelivers           1          0.0
IpOutRequests          1          0.0
TcpInSegs              1          0.0
TcpOutSegs             1          0.0
```

TcpExtTCPPHPHits	1	0.0
IpExtInOctets	58	0.0
IpExtOutOctets	52	0.0
IpExtInNoECTPkts	1	0.0

Compare the first client-side nstat and the second client-side nstat, we could find one difference: the first one had a 'TcpExtTCPPureAcks', but the second one had a 'TcpExtTCPPHPAcks'. The first server-side nstat and the second server-side nstat had a difference too: the second server-side nstat had a TcpExtTCPPHPHits, but the first server-side nstat didn't have it. The network traffic patterns were exactly the same: the client sent a packet to the server, the server replied an ACK. But kernel handled them in different ways. When the TCP window scale option is not used, kernel will try to enable fast path immediately when the connection comes into the established state, but if the TCP window scale option is used, kernel will disable the fast path at first, and try to enable it after kernel receives packets. We could use the 'ss' command to verify whether the window scale option is used. e.g. run below command on either server or client:

```
nstatuser@nstat-a:~$ ss -o state established -i '( dport = :9000 or sport = :9000 )
Netid Recv-Q Send-Q Local Address:Port Peer Address:Port
tcp    0      0          192.168.122.250:40654 192.168.122.251:9000
      ts sack cubic wscale:7,7 rto:204 rtt:0.98/0.49 mss:1448 pmu:1500 rcvmss:536 advmss:1448 cwnd:10 bytes_acked:1 segs_out:2 segs_in:0
```

The 'wscale:7,7' means both server and client set the window scale option to 7. Now we could explain the nstat output in our test:

In the first nstat output of client side, the client sent a packet, server reply an ACK, when kernel handled this ACK, the fast path was not enabled, so the ACK was counted into 'TcpExtTCPPureAcks'.

In the second nstat output of client side, the client sent a packet again, and received another ACK from the server, in this time, the fast path is enabled, and the ACK was qualified for fast path, so it was handled by the fast path, so this ACK was counted into TcpExtTCPPHPAcks.

In the first nstat output of server side, fast path was not enabled, so there was no 'TcpExtTCPPHPHits'.

In the second nstat output of server side, the fast path was enabled, and the packet received from client qualified for fast path, so it was counted into 'TcpExtTCPPHPHits'.

TcpExtTCPAbortOnClose

On the server side, we run below python script:

```
import socket
import time

port = 9000

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('0.0.0.0', port))
s.listen(1)
sock, addr = s.accept()
while True:
    time.sleep(9999999)
```

This python script listen on 9000 port, but doesn't read anything from the connection.

On the client side, we send the string "hello" by nc:

```
nstatuser@nstat-a:~$ echo "hello" | nc nstat-b 9000
```

Then, we come back to the server side, the server has received the "hello" packet, and the TCP layer has acked this packet, but the application didn't read it yet. We type Ctrl-C to terminate the server script. Then we could find TcpExtTCPAbortOnClose increased 1 on the server side:

```
nstatuser@nstat-b:~$ nstat | grep -i abort
TcpExtTCPAbortOnClose      1          0.0
```

If we run tcpdump on the server side, we could find the server sent a RST after we type Ctrl-C.

TcpExtTCPAbortOnMemory and TcpExtTCPAbortOnTimeout

Below is an example which let the orphan socket count be higher than net.ipv4.tcp_max_orphans. Change tcp_max_orphans to a smaller value on client:

```
sudo bash -c "echo 10 > /proc/sys/net/ipv4/tcp_max_orphans"
```

Client code (create 64 connection to server):

```
nstatuser@nstat-a:~$ cat client_orphan.py
import socket
import time

server = 'nstat-b' # server address
port = 9000

count = 64

connection_list = []

for i in range(64):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((server, port))
    connection_list.append(s)
    print("connection_count: %d" % len(connection_list))

while True:
    time.sleep(99999)
```

Server code (accept 64 connection from client):

```
nstatuser@nstat-b:~$ cat server_orphan.py
import socket
import time

port = 9000
count = 64

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('0.0.0.0', port))
s.listen(count)
connection_list = []
while True:
    sock, addr = s.accept()
    connection_list.append((sock, addr))
    print("connection_count: %d" % len(connection_list))
```

Run the python scripts on server and client.

On server:

```
python3 server_orphan.py
```

On client:

```
python3 client_orphan.py
```

Run iptables on server:

```
sudo iptables -A INPUT -i ens3 -p tcp --destination-port 9000 -j DROP
```

Type Ctrl-C on client, stop client_orphan.py.

Check TcpExtTCPAbortOnMemory on client:


```
nstatuser@nstat-a:~$ nstat | grep -i abort
TcpExtTCPAbortOnMemory      54          0.0
```

Check orphaned socket count on client:

```
nstatuser@nstat-a:~$ ss -s
Total: 131 (kernel 0)
TCP:    14 (estab 1, closed 0, orphaned 10, synrecv 0, timewait 0/0), ports 0
```

Transport	Total	IP	IPv6
* *	0	-	-
RAW	1	0	1
UDP	1	1	0
TCP	14	13	1
INET	16	14	2
FRAG	0	0	0

The explanation of the test: after run `server_orphan.py` and `client_orphan.py`, we set up 64 connections between server and client. Run the `iptables` command, the server will drop all packets from the client, type `Ctrl-C` on `client_orphan.py`, the system of the client would try to close these connections, and before they are closed gracefully, these connections became orphan sockets. As the `iptables` of the server blocked packets from the client, the server won't receive `fin` from the client, so all connection on clients would be stuck on `FIN_WAIT_1` stage, so they will keep as orphan sockets until timeout. We have `echo 10 to /proc/sys/net/ipv4/tcp_max_orphans`, so the client system would only keep 10 orphan sockets, for all other orphan sockets, the client system sent `RST` for them and delete them. We have 64 connections, so the `'ss -s'` command shows the system has 10 orphan sockets, and the value of `TcpExtTCPAbortOnMemory` was 54.

An additional explanation about orphan socket count: You could find the exactly orphan socket count by the `'ss -s'` command, but when kernel decide whether increases `TcpExtTCPAbortOnMemory` and sends `RST`, kernel doesn't always check the exactly orphan socket count. For increasing performance, kernel checks an approximate count firstly, if the approximate count is more than `tcp_max_orphans`, kernel checks the exact count again. So if the approximate count is less than `tcp_max_orphans`, but exactly count is more than `tcp_max_orphans`, you would find `TcpExtTCPAbortOnMemory` is not increased at all. If `tcp_max_orphans` is large enough, it won't occur, but if you decrease `tcp_max_orphans` to a small value like our test, you might find this issue. So in our test, the client set up 64 connections although the `tcp_max_orphans` is 10. If the client only set up 11 connections, we can't find the change of `TcpExtTCPAbortOnMemory`.

Continue the previous test, we wait for several minutes. Because of the `iptables` on the server blocked the traffic, the server wouldn't receive `fin`, and all the client's orphan sockets would timeout on the `FIN_WAIT_1` state finally. So we wait for a few minutes, we could find 10 timeout on the client:

```
nstatuser@nstat-a:~$ nstat | grep -i abort
TcpExtTCPAbortOnTimeout      10          0.0
```

TcpExtTCPAbortOnLinger

The server side code:

```
nstatuser@nstat-b:~$ cat server_linger.py
import socket
import time

port = 9000

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('0.0.0.0', port))
s.listen(1)
sock, addr = s.accept()
while True:
    time.sleep(9999999)
```

The client side code:

```
nstatuser@nstat-a:~$ cat client_linger.py
import socket
import struct

server = 'nstat-b' # server address
port = 9000

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_LINGER, struct.pack('ii', 1, 10))
s.setsockopt(socket.SOL_TCP, socket.TCP_LINGER2, struct.pack('i', -1))
s.connect((server, port))
s.close()
```

Run `server_linger.py` on server:

```
nstatuser@nstat-b:~$ python3 server_linger.py
```

Run `client_linger.py` on client:

```
nstatuser@nstat-a:~$ python3 client_linger.py
```

After run `client_linger.py`, check the output of `nstat`:

```
nstatuser@nstat-a:~$ nstat | grep -i abort
TcpExtTCPAbortOnLinger       1          0.0
```

TcpExtTCPRcvCoalesce

On the server, we run a program which listen on TCP port 9000, but doesn't read any data:

```
import socket
import time
port = 9000
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('0.0.0.0', port))
s.listen(1)
sock, addr = s.accept()
while True:
    time.sleep(9999999)
```

Save the above code as `server_coalesce.py`, and run:

```
python3 server_coalesce.py
```

On the client, save below code as `client_coalesce.py`:

```
import socket
server = 'nstat-b'
port = 9000
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((server, port))
```

Run:

```
nstatuser@nstat-a:~$ python3 -i client_coalesce.py
```

We use `-i` to come into the interactive mode, then a packet:

```
>>> s.send(b'foo')
3
```

Send a packet again:

```
>>> s.send(b'bar')
3
```

On the server, run `nstat`:

```

ubuntu@nstat-b:~$ nstat
#kernel
IpInReceives          2          0.0
IpInDelivers          2          0.0
IpOutRequests          2          0.0
TcpInSegs             2          0.0
TcpOutSegs            2          0.0
TcpExtTCPRcvCoalesce  1          0.0
IpExtInOctets         110        0.0
IpExtOutOctets         104        0.0
IpExtInNoECTPkts      2          0.0

```

The client sent two packets, server didn't read any data. When the second packet arrived at server, the first packet was still in the receiving queue. So the TCP layer merged the two packets, and we could find the `TcpExtTCPRcvCoalesce` increased 1.

TcpExtListenOverflows and TcpExtListenDrops

On server, run the `nc` command, listen on port 9000:

```

nstatuser@nstat-b:~$ nc -lkv 0.0.0.0 9000
Listening on [0.0.0.0] (family 0, port 9000)

```

On client, run 3 `nc` commands in different terminals:

```

nstatuser@nstat-a:~$ nc -v nstat-b 9000
Connection to nstat-b 9000 port [tcp/*] succeeded!

```

The `nc` command only accepts 1 connection, and the accept queue length is 1. On current linux implementation, set queue length to `n` means the actual queue length is `n+1`. Now we create 3 connections, 1 is accepted by `nc`, 2 in accepted queue, so the accept queue is full.

Before running the 4th `nc`, we clean the `nstat` history on the server:

```

nstatuser@nstat-b:~$ nstat -n

```

Run the 4th `nc` on the client:

```

nstatuser@nstat-a:~$ nc -v nstat-b 9000

```

If the `nc` server is running on kernel 4.10 or higher version, you won't see the "Connection to ... succeeded!" string, because kernel will drop the SYN if the accept queue is full. If the `nc` client is running on an old kernel, you would see that the connection is succeeded, because kernel would complete the 3 way handshake and keep the socket on half open queue. I did the test on kernel 4.15. Below is the `nstat` on the server:

```

nstatuser@nstat-b:~$ nstat
#kernel
IpInReceives          4          0.0
IpInDelivers          4          0.0
TcpInSegs             4          0.0
TcpExtListenOverflows 4          0.0
TcpExtListenDrops     4          0.0
IpExtInOctets         240        0.0
IpExtInNoECTPkts      4          0.0

```

Both `TcpExtListenOverflows` and `TcpExtListenDrops` were 4. If the time between the 4th `nc` and the `nstat` was longer, the value of `TcpExtListenOverflows` and `TcpExtListenDrops` would be larger, because the SYN of the 4th `nc` was dropped, the client was retrying.

IpInAddrErrors, IpExtInNoRoutes and IpOutNoRoutes

server A IP address: 192.168.122.250 server B IP address: 192.168.122.251 Prepare on server A, add a route to server B:

```

$ sudo ip route add 8.8.8.8/32 via 192.168.122.251

```

Prepare on server B, disable `send_redirects` for all interfaces:

```

$ sudo sysctl -w net.ipv4.conf.all.send_redirects=0
$ sudo sysctl -w net.ipv4.conf.ens3.send_redirects=0
$ sudo sysctl -w net.ipv4.conf.lo.send_redirects=0
$ sudo sysctl -w net.ipv4.conf.default.send_redirects=0

```

We want to let sever A send a packet to 8.8.8.8, and route the packet to server B. When server B receives such packet, it might send a ICMP Redirect message to server A, set `send_redirects` to 0 will disable this behavior.

First, generate `InAddrErrors`. On server B, we disable IP forwarding:

```

$ sudo sysctl -w net.ipv4.conf.all.forwarding=0

```

On server A, we send packets to 8.8.8.8:

```

$ nc -v 8.8.8.8 53

```

On server B, we check the output of `nstat`:

```

$ nstat
#kernel
IpInReceives          3          0.0
IpInAddrErrors         3          0.0
IpExtInOctets         180        0.0
IpExtInNoECTPkts      3          0.0

```

As we have let server A route 8.8.8.8 to server B, and we disabled IP forwarding on server B, Server A sent packets to server B, then server B dropped packets and increased `IpInAddrErrors`. As the `nc` command would re-send the SYN packet if it didn't receive a SYN+ACK, we could find multiple `IpInAddrErrors`.

Second, generate `IpExtInNoRoutes`. On server B, we enable IP forwarding:

```

$ sudo sysctl -w net.ipv4.conf.all.forwarding=1

```

Check the route table of server B and remove the default route:

```

$ ip route show
default via 192.168.122.1 dev ens3 proto static
192.168.122.0/24 dev ens3 proto kernel scope link src 192.168.122.251
$ sudo ip route delete default via 192.168.122.1 dev ens3 proto static

```

On server A, we contact 8.8.8.8 again:

```

$ nc -v 8.8.8.8 53
nc: connect to 8.8.8.8 port 53 (tcp) failed: Network is unreachable

```

On server B, run `nstat`:

```

$ nstat
#kernel
IpInReceives          1          0.0
IpOutRequests          1          0.0
IcmpOutMsgs           1          0.0
IcmpOutDestUnreaches  1          0.0
IcmpMsgOutType3        1          0.0
IpExtInNoRoutes        1          0.0
IpExtInOctets          60          0.0
IpExtOutOctets         88          0.0
IpExtInNoECTPkts       1          0.0

```

We enabled IP forwarding on server B, when server B received a packet which destination IP address is 8.8.8.8, server B will try to forward this packet. We have deleted the default route, there was no route for 8.8.8.8, so server B increase `IpExtInNoRoutes` and sent the "ICMP Destination Unreachable" message to server A.

Third, generate IpOutNoRoutes. Run ping command on server B:

```
$ ping -c 1 8.8.8.8
connect: Network is unreachable
```

Run nstat on server B:

```
$ nstat
#kernel
IpOutNoRoutes          1          0.0
```

We have deleted the default route on server B. Server B couldn't find a route for the 8.8.8.8 IP address, so server B increased IpOutNoRoutes.

TcpExtTCPACKSkippedSynRecv

In this test, we send 3 same SYN packets from client to server. The first SYN will let server create a socket, set it to Syn-Recv status, and reply a SYN/ACK. The second SYN will let server reply the SYN/ACK again, and record the reply time (the duplicate ACK reply time). The third SYN will let server check the previous duplicate ACK reply time, and decide to skip the duplicate ACK, then increase the TcpExtTCPACKSkippedSynRecv counter.

Run tcpdump to capture a SYN packet:

```
nstatuser@nstat-a:~$ sudo tcpdump -c 1 -w /tmp/syn.pcap port 9000
tcpdump: listening on ens3, link-type EN10MB (Ethernet), capture size 262144 bytes
```

Open another terminal, run nc command:

```
nstatuser@nstat-a:~$ nc nstat-b 9000
```

As the nstat-b didn't listen on port 9000, it should reply a RST, and the nc command exited immediately. It was enough for the tcpdump command to capture a SYN packet. A linux server might use hardware offload for the TCP checksum, so the checksum in the /tmp/syn.pcap might be not correct. We call tcpwrite to fix it:

```
nstatuser@nstat-a:~$ tcpwrite --infile=/tmp/syn.pcap --outfile=/tmp/syn_fixcsum.pcap --fixcsum
```

On nstat-b, we run nc to listen on port 9000:

```
nstatuser@nstat-b:~$ nc -lkv 9000
Listening on [0.0.0.0] (family 0, port 9000)
```

On nstat-a, we blocked the packet from port 9000, or nstat-a would send RST to nstat-b:

```
nstatuser@nstat-a:~$ sudo iptables -A INPUT -p tcp --sport 9000 -j DROP
```

Send 3 SYN repeatedly to nstat-b:

```
nstatuser@nstat-a:~$ for i in {1..3}; do sudo tcpreplay -i ens3 /tmp/syn_fixcsum.pcap; done
```

Check snmp counter on nstat-b:

```
nstatuser@nstat-b:~$ nstat | grep -i skip
TcpExtTCPACKSkippedSynRecv      1          0.0
```

As we expected, TcpExtTCPACKSkippedSynRecv is 1.

TcpExtTCPACKSkippedPAWS

To trigger PAWS, we could send an old SYN.

On nstat-b, let nc listen on port 9000:

```
nstatuser@nstat-b:~$ nc -lkv 9000
Listening on [0.0.0.0] (family 0, port 9000)
```

On nstat-a, run tcpdump to capture a SYN:

```
nstatuser@nstat-a:~$ sudo tcpdump -w /tmp/paws_pre.pcap -c 1 port 9000
tcpdump: listening on ens3, link-type EN10MB (Ethernet), capture size 262144 bytes
```

On nstat-a, run nc as a client to connect nstat-b:

```
nstatuser@nstat-a:~$ nc -v nstat-b 9000
Connection to nstat-b 9000 port [tcp/*] succeeded!
```

Now the tcpdump has captured the SYN and exit. We should fix the checksum:

```
nstatuser@nstat-a:~$ tcpwrite --infile /tmp/paws_pre.pcap --outfile /tmp/paws.pcap --fixcsum
```

Send the SYN packet twice:

```
nstatuser@nstat-a:~$ for i in {1..2}; do sudo tcpreplay -i ens3 /tmp/paws.pcap; done
```

On nstat-b, check the snmp counter:

```
nstatuser@nstat-b:~$ nstat | grep -i skip
TcpExtTCPACKSkippedPAWS        1          0.0
```

We sent two SYN via tcpreplay, both of them would let PAWS check failed, the nstat-b replied an ACK for the first SYN, skipped the ACK for the second SYN, and updated TcpExtTCPACKSkippedPAWS.

TcpExtTCPACKSkippedSeq

To trigger TcpExtTCPACKSkippedSeq, we send packets which have valid timestamp (to pass PAWS check) but the sequence number is out of window. The linux TCP stack would avoid to skip if the packet has data, so we need a pure ACK packet. To generate such a packet, we could create two sockets: one on port 9000, another on port 9001. Then we capture an ACK on port 9001, change the source/destination port numbers to match the port 9000 socket. Then we could trigger TcpExtTCPACKSkippedSeq via this packet.

On nstat-b, open two terminals, run two nc commands to listen on both port 9000 and port 9001:

```
nstatuser@nstat-b:~$ nc -lkv 9000
Listening on [0.0.0.0] (family 0, port 9000)
```

```
nstatuser@nstat-b:~$ nc -lkv 9001
Listening on [0.0.0.0] (family 0, port 9001)
```

On nstat-a, run two nc clients:

```
nstatuser@nstat-a:~$ nc -v nstat-b 9000
Connection to nstat-b 9000 port [tcp/*] succeeded!
```

```
nstatuser@nstat-a:~$ nc -v nstat-b 9001
Connection to nstat-b 9001 port [tcp/*] succeeded!
```

On nstat-a, run tcpdump to capture an ACK:

```
nstatuser@nstat-a:~$ sudo tcpdump -w /tmp/seq_pre.pcap -c 1 dst port 9001
tcpdump: listening on ens3, link-type EN10MB (Ethernet), capture size 262144 bytes
```

On nstat-b, send a packet via the port 9001 socket. E.g. we sent a string 'foo' in our example:

```
nstatuser@nstat-b:~$ nc -lkv 9001
Listening on [0.0.0.0] (family 0, port 9001)
Connection from nstat-a 42132 received!
foo
```

On nstat-a, the tcpdump should have captured the ACK. We should check the source port numbers of the two nc clients:

```
nstatuser@nstat-a:~$ ss -ta '( dport = :9000 || dport = :9001 )' | tee
State Recv-Q Send-Q Local Address:Port Peer Address:Port
ESTAB 0 0 192.168.122.250:50208 192.168.122.251:9000
ESTAB 0 0 192.168.122.250:42132 192.168.122.251:9001
```

Run `tcprewrite`, change port 9001 to port 9000, change port 42132 to port 50208:

```
nstatuser@nstat-a:~$ tcprewrite --infile /tmp/seq_pre.pcap --outfile /tmp/seq.pcap -r 9001:9000 -r 42132:50208 --fixcsum
```

Now the `/tmp/seq.pcap` is the packet we need. Send it to `nstat-b`:

```
nstatuser@nstat-a:~$ for i in {1..2}; do sudo tcpreplay -i ens3 /tmp/seq.pcap; done
```

Check `TcpExtTCPACKSkippedSeq` on `nstat-b`:

```
nstatuser@nstat-b:~$ nstat | grep -i skip
TcpExtTCPACKSkippedSeq      1          0.0
```