Linux TCP Implementation Issues in High-Speed Networks

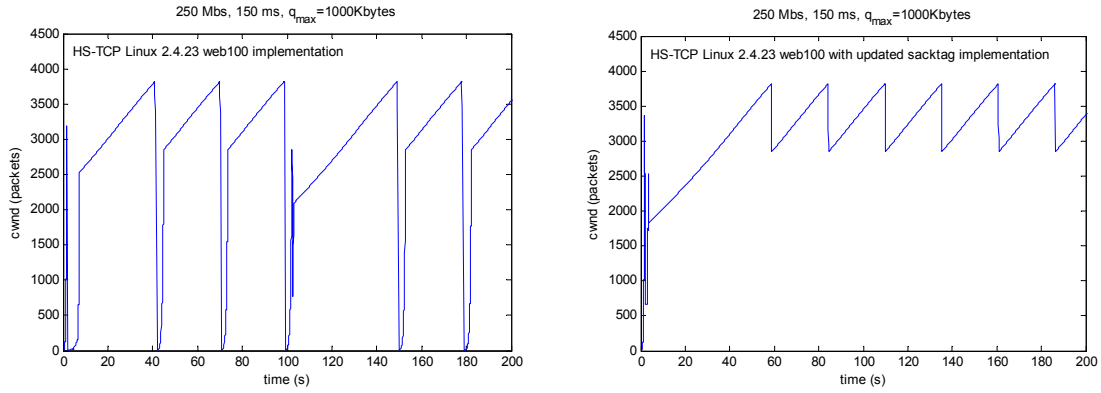
D.J.Leith

Hamilton Institute, Ireland
www.hamilton.ie

1. Implementation Issues

1.1. SACK algorithm inefficient

Packets in flight and not yet acknowledged are held on the writequeue linked list at the transmitter. On receipt of SACK information, the writequeue is walked and the relevant packets tagged as sacked or lost. In the original implementation writequeue is walked for each sack block (i.e. up to three times) and again to mark packets as lost. Lost packets are retransmitted via another walk of writequeue. For large delay-bandwidth products, the writequeue is very large (many thousands of packets) and these walks become too time consuming to complete between the arrival of ACKs.



(a) *(b)*

Figure 1 Example illustrating impact of SACK processing inefficiency. Transfer between servers (Dell 1650, 2.8GHz Xeon, PCI-X 133Mhz/64bit & Intel Pro/1000 NICs) running Linux 2.4.23 with web100 (sbufmode=0=rbufmode; WAD_IFQ=1, WAD_FloydAIMD=1) via dummynet router of same specification running FreeBSD 4.8.

Repeated timeouts during loss recovery are evident in Figure 1(a) (timeouts occur due to throttle action of softirq queue when rate at which incoming packets are serviced is persistently too low – see below). The corresponding performance with a more efficient SACK processing algorithm is shown in Figure 1(b). More test results are presented in a later section. This algorithm incorporates the following changes.

- Tagging algorithm modified to walk writequeue holes (i.e. packets not yet sacked), rather than walking entire writequeue. Walk therefore now scales with number of lost packets rather than delay-bandwidth product.
- Tagging algorithm modified to make a single walk rather than a walk per sack block (possibl;e by pre-sorting sack blocks by sequence number).
- We now cache pointers for the retransmission (using idea of Kelly) walk and also for walk on receipt of a dsack (for which a walk of the writequeue holes is insufficient).
- when number of holes in writequeue becomes large, sack blocks of incoming ACK's are cached and the walk of the writequeue performed less frequently – typically there is redundancy in the SACK information of ACK's whereby later sack information subsumes that in earlier ACK's and so caching leads to significant efficiency savings. Note that Kelly proposes an alternative approach based on specific redundancy in the sack blocks between ACK's – this fails when ACK's are being frequently lost (which is commonly the case on high-speed links due to pressure on the computational resources and associated dropping from softirq queue).

1.2. Throttle action excessively severe

On overflow, the netdev softirq queue enters a throttle state whereby all incoming packets are dropped until the queue empties. This invariably leads to a TCP timeout. The timeout is often particularly damaging as the throttle action breaks ACK clock (many incoming ACK's are dropped) so that recovery from the timeout can be very slow for large delay-bandwidth products.

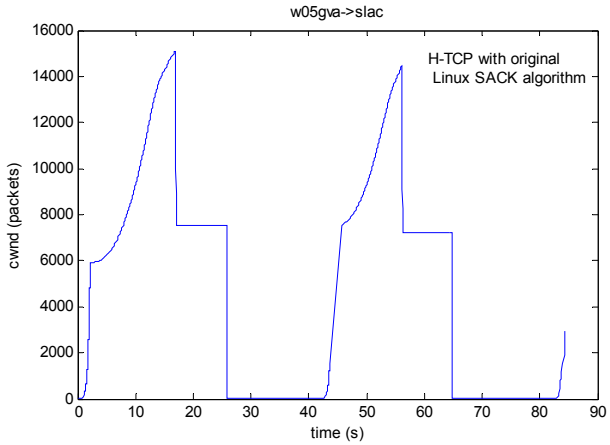


Figure 2 TCP transfer between Geneva, Switzerland and Stanford, California using Linux 2.4.23 with web100 and standard SACK processing algorithm. The slow recovery from the throttle-induced TCP timeouts is evident.

It is proposed that the throttle action be disabled as being unnecessarily severe. Instead, a receiver just drops incoming packets when the softirq queue overflows. Note that simply increasing the length of the netdev queue is often not a good alternative solution as the additional queueing delay can lead to frequent TCP timeouts. Disabling of the throttle action also does not remove the need for a more efficient SACK processing algorithm (see previous section). Disabling the throttle action while retaining an inefficient SACK processing algorithm leads to a heavy processor burden with large numbers of ACK's being dropped and an associated loss of SACK information and damage to the TCP ACK clock.

1.3. Prevalence of SACK reneging

Inspection of data flows on high speed links reveals that sack reneging by receivers (i.e. ACKing of a packet that has already been acknowledged via a SACK) is surprisingly common. Presumably this arises due to pressure on the receiver at high data rates. The standard Linux TCP implementation treats sack reneging in a similar way to a timeout i.e. it resets `cwnd` to one and enters loss state. This is unnecessarily severe. It is proposed that the implementation be modified to reset the SACK tagging of the packets in flight (i.e. the packets on writequeue) but otherwise leave `cwnd` unchanged (i.e. still enter loss state to take care of packet accounting etc, but do not reset `cwnd` and `snd_ssthresh`).

1.4. Prevalence of packet re-ordering

Re-ordering of packets is common on high-speed networks (e.g. due to router implementations) and can lead to spurious entry of the TCP recovery state. The undo mechanism in Linux will then back out of this state once reordering becomes evident.

- in original implementation, undo mechanism is switched off (`prior_ssthresh` reset to zero) after first entry into recovery state. Reordering during recovery or repeated reordering therefore cannot be undone. Implementation modified to retain undo mechanism after first entry to recovery state.
- original undo mechanism usually stores 0.75`cwnd` in `prior_ssthresh` and resets `snd_ssthresh` to `prior_ssthresh` on an undo; if `cwnd` is below `snd_ssthresh` then it slow start's up to `snd_ssthresh` before re-entering congestion avoidance. This is inaccurate, and performs poorly when reordering etc causes repeated undo actions, see for example Figure 3. Implementation is modified to save exact undo value of `snd_ssthresh`.

Use of accurate `snd_ssthresh` undo is activated when sysctl `net.ipv4.tcp_htcp_undo`>0, otherwise original algorithm is used.

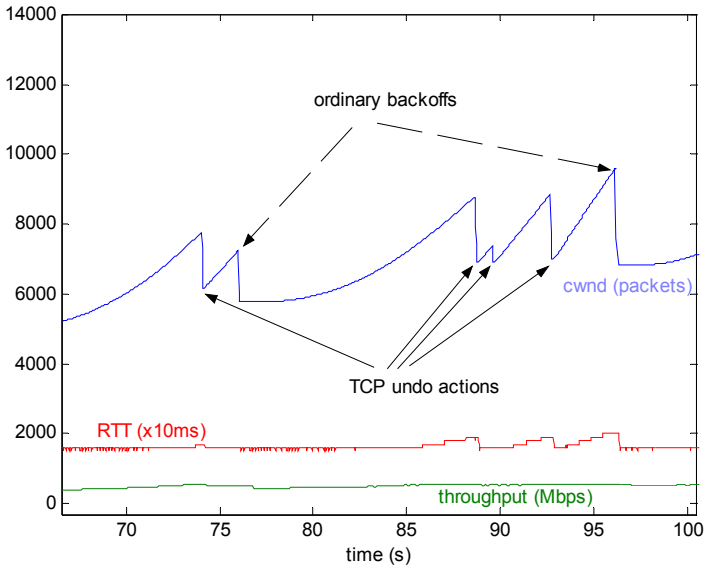


Figure 3 TCP transfer between Dublin and Stanford using modified Linux implementation with net.ipv4.tcp_htcp_undo=0 (original undo mechanism). The impact of inaccuracy in the undo mechanism is evident. Compare with the time history shown in Figure 8 which uses the same Linux implementation but with net.ipv4.tcp_htcp_undo=1 (accurate undo mechanism).

1.5.Burst moderation inefficient

Moderation to reduce packet bursts due to dropped ACK's etc is achieved in original linux implementation by capping `cwnd` based on the estimated number of packets in flight. In high-speed networks the accounting for packets in flight is often inaccurate during loss recovery.

- Packets in flight cap on `cwnd` in routine `cwnd_down()` removed. Change activated when sysctl `net.ipv4.tcp_htcp_cap`>0. The negative impact of the cap in `cwnd_down()` can be seen in Figures 4 and 5. Compare with the time history shown in Figure 8 which uses the same Linux implementation but with net.ipv4.tcp_htcp_cap=1 (cap removed).
- burst moderation is re-implemented using direct accounting of packets transmitted per ACK during loss recovery. Changes activated when sysctl `net.ipv4.tcp_htcp_moderation`>0, otherwise original algorithm is used. The negative impact of the original burst moderation mechanism can be seen in Figures 6 and 7.

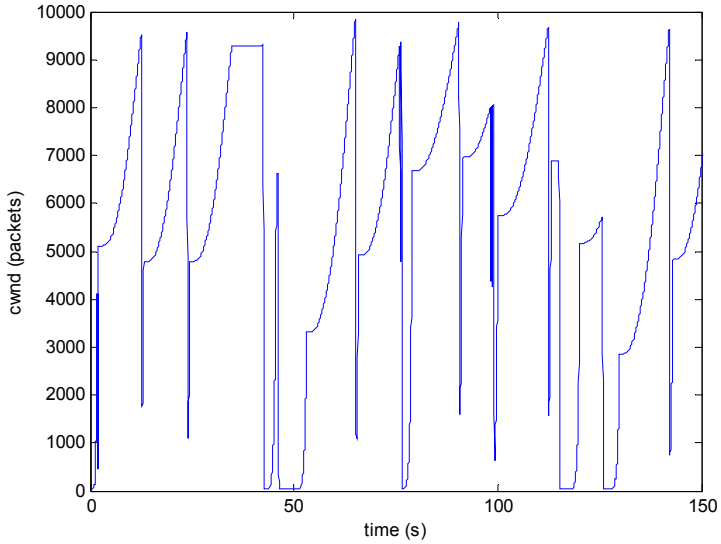
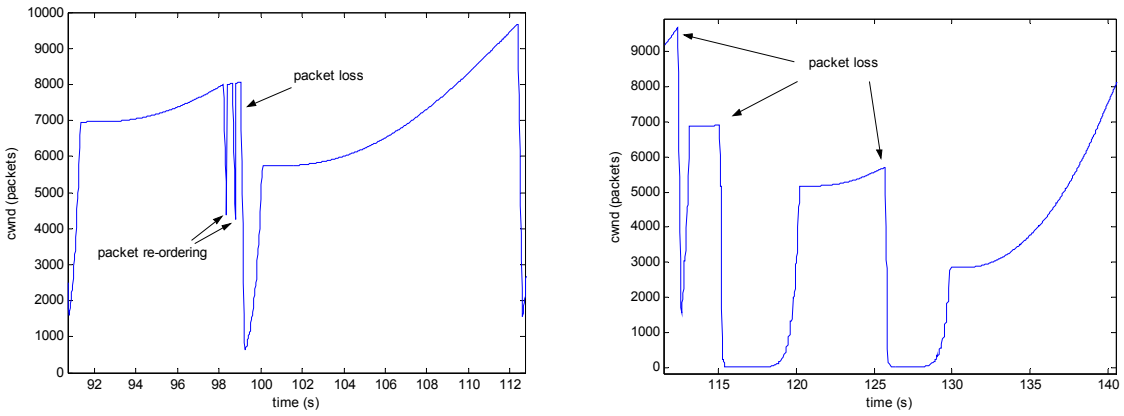


Figure 4 TCP transfer between Dublin and Stanford using modified Linux implementation with net.ipv4.tcp_htcp_cap=0 (`cwnd` restricted to `packets_in_flight+1` in `cwnd_down()`).



(a) *(b)*

Figure 5 Detail from Figure 4 showing packet reordering episodes with associated invocation of the undo mechanism. During packet re-ordering the impact of capping `cwnd` is evident (compare with Figure 3). Also shown are a number of backoff events due to packet loss. Notice that capping `cwnd` leads to a large reduction in `cwnd` during loss recovery, followed by the slow starting of `cwnd` back up to `snd_ssthresh`. It can be seen from Figure 5(b) that this reduction in `cwnd` (these events are regular loss recovery, *not* timeouts) can be sufficiently large that the ACK clock is damaged, leading to very slow loss recovery.

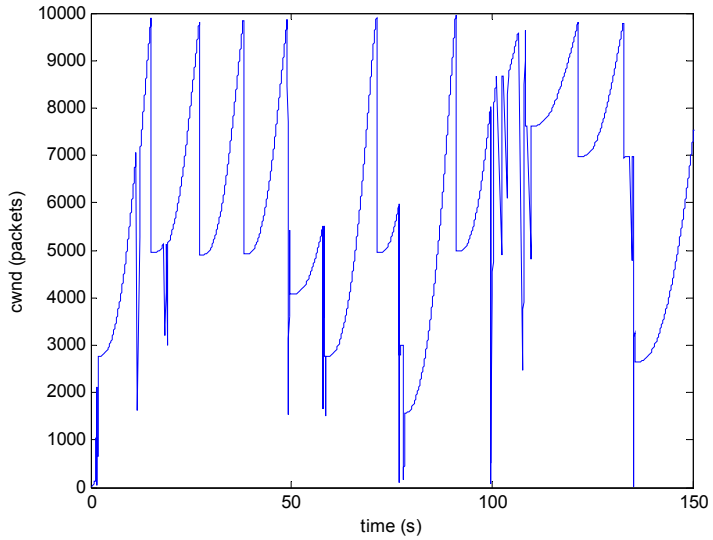
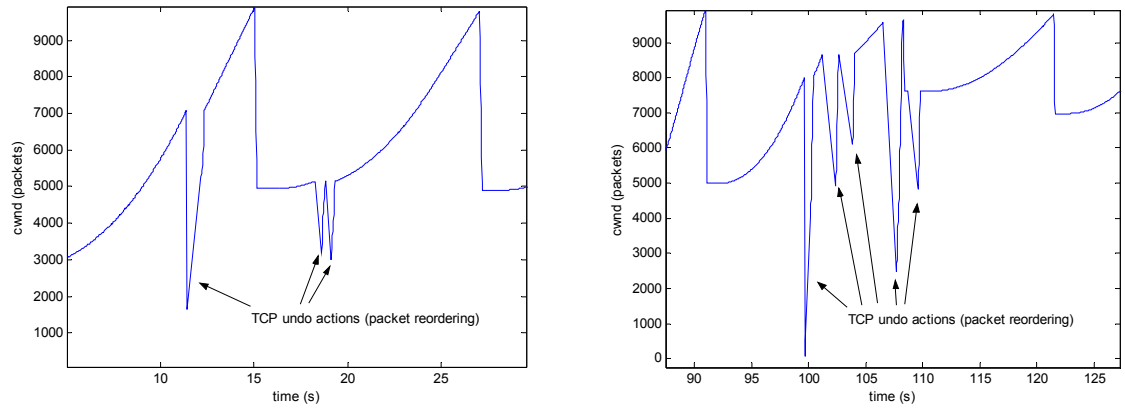


Figure 6 TCP transfer between Dublin and Stanford using modified Linux implementation with net.ipv4.tcp_htcp_moderation=0 (original burst moderation mechanism).



(a) *(b)*

Figure 7 Detail from Figure 6 showing episodes with associated invocation of the undo mechanism.

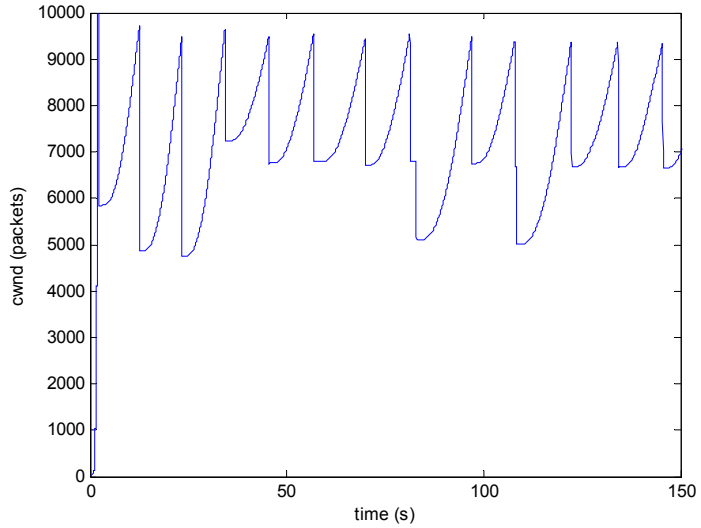
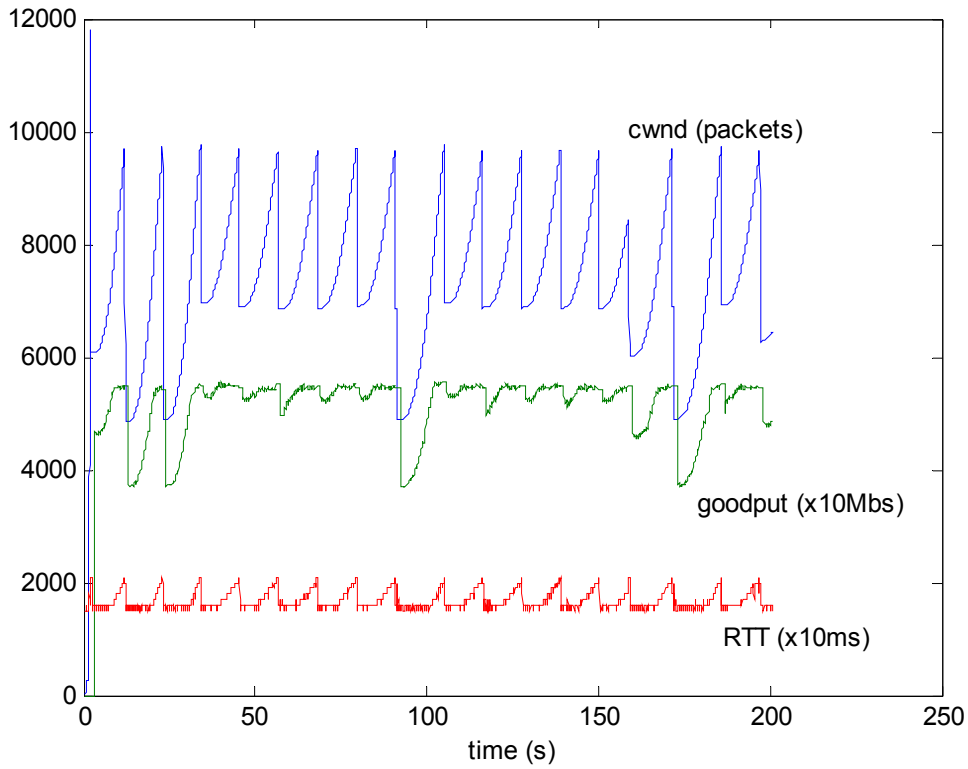


Figure 8 TCP transfer between Dublin and Stanford using full functionality of modified Linux implementation. The data shown in Figures 3-8 was collected on the evening of Wednesday March 24th 2004; all transfers were carried out within a single one hour period. Unless otherwise stated, all tests are made using the H-TCP variant of TCP.

2. Brief Test Results

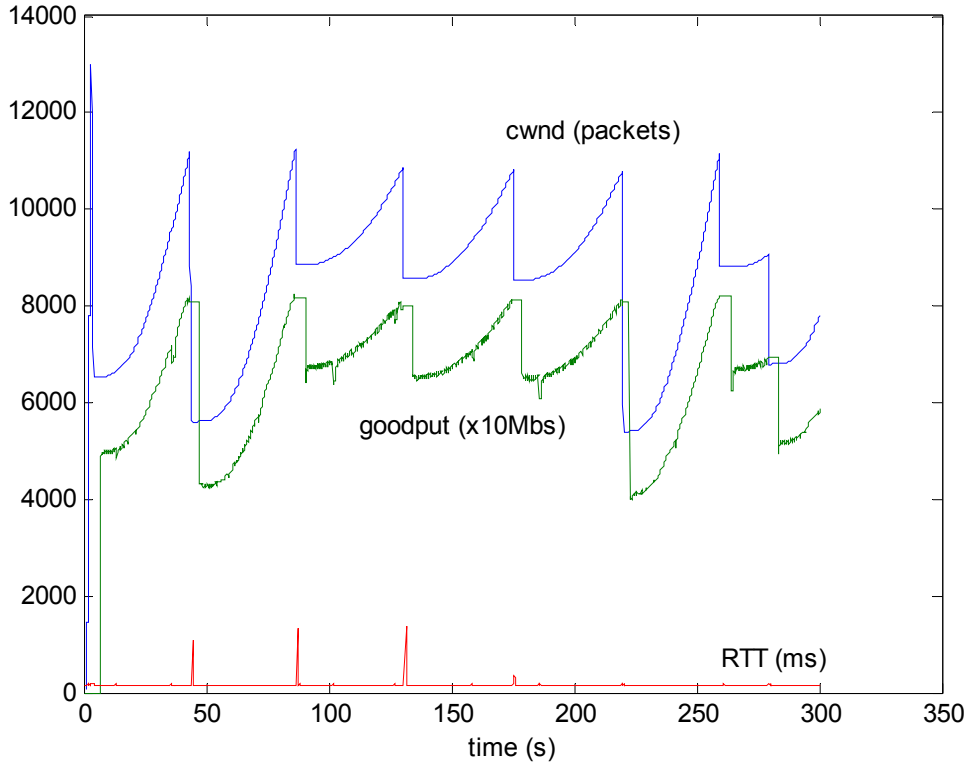
Dublin machine is a Dell PowerEdge 1750, 2.4GHz Xeon, PCI-X 133MHz/64bit & Intel Pro/1000 NIC.

Dublin, Ireland – SLAC, California



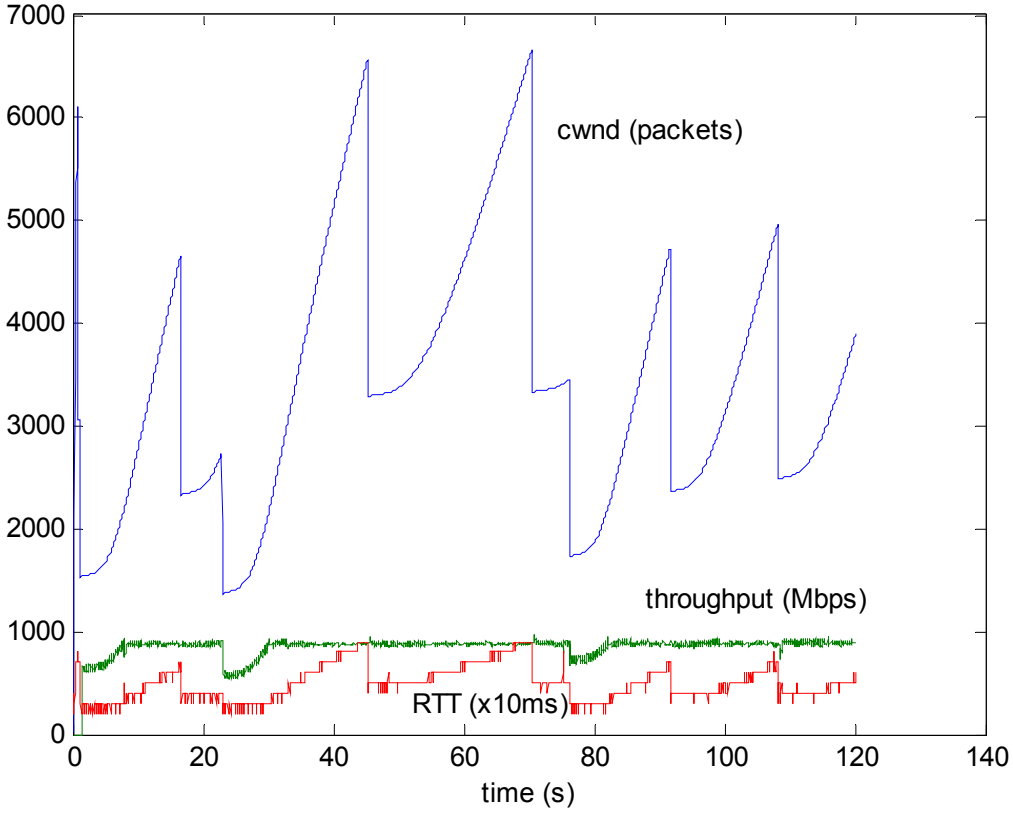
traceroute to iepm-resp.slac.stanford.edu (134.79.240.36), 30 hops max, 38 byte packets
1 calpurnia-vlan7.bh.access.hea.net (193.1.31.97) 0.818 ms 0.811 ms 0.867 ms
2 mantova-vlan3.bh.access.hea.net (193.1.198.245) 0.354 ms 0.222 ms 0.246 ms
3 hyperion-gige3-3-0.bh.core.hea.net (193.1.196.121) 0.479 ms 0.350 ms 0.372 ms
4 deimos-gige5-1.cwt.core.hea.net (193.1.195.129) 0.605 ms 0.599 ms 0.621 ms
5 heanet.ie1.ie.geant.net (62.40.103.229) 0.730 ms 0.723 ms 0.746 ms
6 ie.uk1.uk.geant.net (62.40.96.138) 13.221 ms 13.090 ms 13.112 ms
7 uk.ny1.ny.geant.net (62.40.96.169) 81.836 ms 81.738 ms 81.765 ms
8 esnet-gw.ny1.ny.geant.net (62.40.103.214) 81.748 ms 81.741 ms 89.963 ms
9 chicr1-oc192-aoacr1.es.net (134.55.209.57) 102.141 ms 102.106 ms 102.132 ms
10 snvcr1-oc192-chicr1.es.net (134.55.209.53) 150.311 ms 150.283 ms 150.110 ms
11 slac-pos-snv.es.net (134.55.209.2) 150.498 ms 150.484 ms 150.594 ms
12 * * *
13 * * *
14 iepm-resp.slac.stanford.edu (134.79.240.36) 151.768 ms 151.830 ms 151.775 ms

Dublin, Ireland – LBL, California



traceroute to net100.lbl.gov (131.243.2.93), 30 hops max, 38 byte packets
1 calpurnia-vlan7.bh.access.hea.net (193.1.31.97) 0.810 ms 1.434 ms 0.868 ms
2 mantova-vlan3.bh.access.hea.net (193.1.198.245) 0.230 ms 0.223 ms 0.247 ms
3 hyperion-gige3-0-0.bh.core.hea.net (193.1.196.173) 0.355 ms 0.349 ms 0.373 ms
4 deimos-gige5-1.cwt.core.hea.net (193.1.195.129) 0.605 ms 0.600 ms 0.621 ms
5 heanet.ie1.ie.geant.net (62.40.103.229) 0.730 ms 0.725 ms 0.747 ms
6 ie.uk1.uk.geant.net (62.40.96.138) 13.098 ms 13.090 ms 13.237 ms
7 uk.ny1.ny.geant.net (62.40.96.169) 81.803 ms 81.800 ms 81.820 ms
8 esnet-gw.ny1.ny.geant.net (62.40.103.214) 81.803 ms 81.798 ms 81.820 ms
9 chicr1-oc192-aoacr1.es.net (134.55.209.57) 102.041 ms 102.050 ms 102.154 ms
10 snvcr1-oc192-chicr1.es.net (134.55.209.53) 150.128 ms 150.095 ms 150.126 ms
11 lbl-snv-oc48.es.net (134.55.209.6) 151.485 ms 151.470 ms 151.508 ms
12 lbnl-ge-lbl2.es.net (198.129.224.1) 153.971 ms 153.207 ms 153.247 ms
13 ir1000gw.lbl.gov (131.243.128.210) 151.612 ms 151.588 ms 151.621 ms
14 net100.lbl.gov (131.243.2.93) 151.484 ms 151.597 ms 151.500 ms

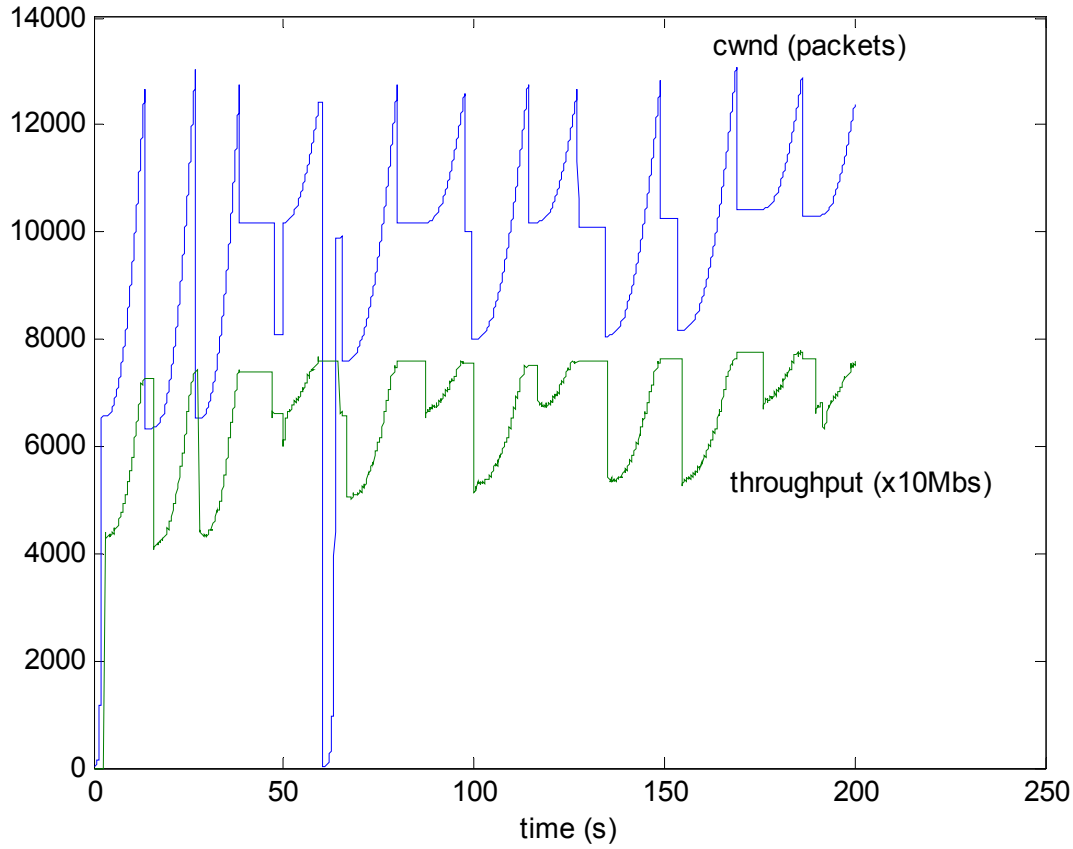
Dublin, Ireland – CERN, Switzerland




```
traceroute to 192.91.239.4 (192.91.239.4), 30 hops max, 38 byte packets
 1  calpurnia-vlan7.bh.access.hea.net (193.1.31.97)  0.885 ms  0.831 ms  1.023 ms
 2  mantova-vlan3.bh.access.hea.net (193.1.198.245)  0.370 ms  0.244 ms  0.228 ms
 3  hyperion-gige3-3-0.bh.core.hea.net (193.1.196.121)  0.505 ms  0.375 ms  0.362 ms
 4  deimos-gige5-1.cwt.core.hea.net (193.1.195.129)  0.639 ms  0.639 ms  0.633 ms
 5  heanet.ie1.ie.geant.net (62.40.103.229)  0.911 ms  0.917 ms  0.757 ms
 6  ie.uk1.uk.geant.net (62.40.96.138)  13.342 ms  13.205 ms  13.341 ms
 7  uk.fr1.fr.geant.net (62.40.96.89)  20.543 ms  28.688 ms  20.408 ms
 8  fr.ch1.ch.geant.net (62.40.96.29)  28.713 ms  28.563 ms  28.580 ms
 9  switch-bckp-gw.ch1.ch.geant.net (62.40.103.22)  28.813 ms  28.831 ms  28.792 ms
10  r04gva-v-7.cern.datatag.org (192.91.239.234)  28.847 ms  28.792 ms  28.767 ms
11  w04gva-ge-0.cern.datatag.org (192.91.239.4)  28.941 ms  28.787 ms  28.664 ms
```

Delay-bandwidth product for this path is thought to be around 2000 packets, with a 4096 packet queue located at the link in CERN.

CERN, Switzerland – SLAC, California



traceroute to iepm-resp.slac.stanford.edu (134.79.240.36), 30 hops max, 38 byte packets
1 r04chi-v-570.caltech.datatag.org (192.91.239.56) 116.555 ms 116.392 ms 116.368 ms
2 abilene-tge.cern.ch (192.91.236.197) 116.563 ms 116.539 ms 116.527 ms
3 iplsng-chinng.abilene.ucaid.edu (198.32.8.77) 124.311 ms 126.566 ms 120.273 ms
4 kscyng-iplsng.abilene.ucaid.edu (198.32.8.81) 129.591 ms 129.539 ms 129.494 ms
5 dnvrng-kscyng.abilene.ucaid.edu (198.32.8.13) 148.182 ms 140.079 ms 140.110 ms
6 snvang-dnvrng.abilene.ucaid.edu (198.32.8.1) 164.923 ms 164.923 ms 164.857 ms
7 losang-snvang.abilene.ucaid.edu (198.32.8.94) 172.375 ms 172.364 ms 172.363 ms
8 hpr-lax-gsr1--abilene-LA-10ge.cenic.net (137.164.25.2) 172.931 ms 173.353 ms 172.542 ms
9 dc-lax-dc1--lax-hpr1-ge.cenic.net (137.164.22.12) 172.726 ms 172.737 ms 172.650 ms
10 dc-sac-dc1--lax-dc1-pos.cenic.net (137.164.22.127) 181.870 ms 181.945 ms 181.971 ms
11 dc-oak-dc2--csac-dc1-ge.cenic.net (137.164.22.110) 184.068 ms 183.861 ms 183.716 ms
12 dc-oak-dc1--oak-dc2-ge.cenic.net (137.164.22.36) 184.009 ms 184.273 ms 183.627 ms
13 dc-stan--oak-dc1-ge.cenic.net (137.164.23.42) 184.681 ms 184.716 ms 184.623 ms
14 rtr-dmz1-vlan401.slac.stanford.edu (192.68.191.85) 175.147 ms 175.081 ms 175.092 ms
15 * * *
16 iepm-resp.slac.stanford.edu (134.79.240.36) 175.039 ms 175.049 ms 175.062 ms