

TCP Performance Analysis Based on Packet Capture

Stanislav Shalunov <shalunov@internet2.edu>

2003-02-05, E2E Performance Measurement Workshop, Miami

Packet Capture

- TCP connection runs; some performance is observed. Why?
- Web100: a look at TCP variables (potent but not the topic)
- Packet capture: you get almost all you could, right?
- Need a tool: the choice is not important in principle (since the job looks so simple) but using a bad tool can ruin your day.
- Use `tcpdump` (basic familiarity is pretty much assumed).
- Captures headers (or whole packets) on the wire matching certain criterion (e.g., belonging to a given TCP connection).
- Can decode and pretty-print or just store to a file.

Where to Capture?

- Packets lost in the network show up at the sender
- They do not show up at the receiver
- Congestion window algorithm runs at the sender
- Time correlation is straightforward at the sender
- At receiver, must take into account events that happened in the past
- **Always capture at the sender.**
- Capture the same flow at the receiver if you can so that you know what *actually* happened in the network

Hand-Examination of `tcpdump` Output

- Packet-by-packet printout is examined by a human
- An arduous task requiring expertise
- Need to understand TCP Reno algorithms
- Need to have a lot of patience
- Time estimates vary:
 - from ‘20 minutes for a few dozen packets’
 - to ‘easily hours for even a simple trace’
- Experts will disagree on interpretation
- Like root canal: you have to do what you have to do
- **Avoid hand-examination if possible.**

tcpdump invocation

- `tcpdump` *<options>* *<filter>*
- On most systems, need to be root, but on BSD can simply make `/dev/bpf[0-9]` world-readable
- `-w` and `-r` *<file>*: write to and read from *<file>*
- `-n` should be specified any time `-r` is not specified
- `-e` (print link-layer headers) should be used any time a LAN problem is investigated
- `-p` should be specified any time host's own traffic is captured
- `-c` is useful for quick looking
- `-s` needs to be used (at recording time!) to decode TCP with a lot of options or things like DNS

Don't run tcpdump -x without the supplement

```
#!/usr/bin/perl
# tcpdumpx - filter for tcpdump to show printable characters as is with '-x'.

open(TCPDUMP,"tcpdump -x -l @ARGV|");
while (<TCPDUMP>) {
    if (/^\s+(\S\S)+/) {
        $sav = $_; $asc = "";
        while (s/\s*(\S\S)\s*//) {
            $i = hex($1);
            if ($i < 32 || $i > 126) {$asc .= "."}
            else {$asc .= pack(C,hex($1))}
        }
        $foo = "." x length($asc);
        $_ = $sav;
        s/\t/ /g;
        s/^\$foo/\$asc/;
    }
    print;
}
```

Let's read a trace just for fun

- Part of early GigaTCP testing
- Back-to-back GE connection
- 4470-byte MTU to emulate the SONET default
- Large window size (2 MB in this example)
- Stalls for no apparent reason
- Captured at sender (902 packets) and receiver (500 packets)
- Problem eventually resolved
- A quirk of TCP implementation uncovered
- Inadequate setting discovered in the driver
- A bug in BSD TCP stack found

Sender's View of the Connection

```
23:54:23.817524 10.0.0.2.1072 > 10.0.0.1.5001: S 577076198:577076198(0) win 65535
    <mss 4430,nop,wscale 6,nop,nop,timestamp 2323869 0> (DF) (ttl 64, id 32041)
23:54:23.817674 10.0.0.1.5001 > 10.0.0.2.1072: S 168494072:168494072(0) ack
    577076199 win 65535 <mss 4430,nop,wscale 6,nop,nop,
    timestamp 2321727 2323869> (DF) (ttl 64, id 18233)
23:54:23.817715 10.0.0.2.1072 > 10.0.0.1.5001: . ack 1 win 32768
    <nop,nop,timestamp 2323869 2321727> (DF) (ttl 64, id 32042)
23:54:23.817864 10.0.0.1.5001 > 10.0.0.2.1072: . ack 1 win 32768
    <nop,nop,timestamp 2321727 2323869> (DF) (ttl 64, id 18234)
```

Connection initiated. Actual window advertized by both sides (taking window scaling into account) is $32768 \times 2^6 = 2097152 = 2^{21} = 2 \text{ MB}$.

```
23:54:23.818382 10.0.0.2.1072 > 10.0.0.1.5001: P 1:4097(4096) ack 1 win 32768
  <nop,nop,timestamp 2323869 2321727> (DF) (ttl 64, id 32043)
23:54:23.818403 10.0.0.2.1072 > 10.0.0.1.5001: P 4097:8193(4096) ack 1 win 32768
  <nop,nop,timestamp 2323869 2321727> (DF) (ttl 64, id 32044)
23:54:23.818475 10.0.0.2.1072 > 10.0.0.1.5001: P 8193:12289(4096) ack 1 win 32768
  <nop,nop,timestamp 2323869 2321727> (DF) (ttl 64, id 32046)
23:54:23.818493 10.0.0.2.1072 > 10.0.0.1.5001: P 12289:16385(4096) ack 1 win 32768
  <nop,nop,timestamp 2323869 2321727> (DF) (ttl 64, id 32047)
... We proceed to blast packets...
23:54:23.818613 10.0.0.1.5001 > 10.0.0.2.1072: . ack 8193 win 32640
  <nop,nop,timestamp 2321727 2323869> (DF) (ttl 64, id 18235)
23:54:23.818645 10.0.0.2.1072 > 10.0.0.1.5001: P 32769:36865(4096) ack 1 win 32768
  <nop,nop,timestamp 2323869 2321727> (DF) (ttl 64, id 32052)
23:54:23.818664 10.0.0.2.1072 > 10.0.0.1.5001: P 36865:40961(4096) ack 1 win 32768
  <nop,nop,timestamp 2323869 2321727> (DF) (ttl 64, id 32053)
23:54:23.818692 10.0.0.2.1072 > 10.0.0.1.5001: P 40961:45057(4096) ack 1 win 32768
  <nop,nop,timestamp 2323869 2321727> (DF) (ttl 64, id 32054)
23:54:23.818712 10.0.0.2.1072 > 10.0.0.1.5001: P 45057:49153(4096) ack 1 win 32768
  <nop,nop,timestamp 2323869 2321727> (DF) (ttl 64, id 32055)
23:54:23.818726 10.0.0.1.5001 > 10.0.0.2.1072: . ack 16385 win 32512
  <nop,nop,timestamp 2321727 2323869> (DF) (ttl 64, id 18236)
```

... We see some ACKs coming in, but at this point we really don't care as we can send 2 MB (512 packets) before we need to even bother to look at anything the remote side says...

```
23:54:23.822377 10.0.0.2.1072 > 10.0.0.1.5001: P 528385:532481(4096)
    ack 1 win 32768 <nop,nop,timestamp 2323870 2321728> (DF) (ttl 64, id 32173)
23:54:23.822391 10.0.0.1.5001 > 10.0.0.2.1072: . ack 466945 win 25600
    <nop,nop,timestamp 2321728 2323870> (DF) (ttl 64, id 18291)
23:54:23.822419 10.0.0.2.1072 > 10.0.0.1.5001: P 532481:536577(4096)
    ack 1 win 32768 <nop,nop,timestamp 2323870 2321728> (DF) (ttl 64, id 32174)
```

Everything goes well until this point. The preceding packet is the first packet lost in this connection. We can blast a lot more according to the window size. Since there were no losses up to this point (130 packets), our cwnd is the same as window size for a while now. We go on to send more stuff...

23:54:23.996387 10.0.0.2.1072 > 10.0.0.1.5001: . 2293761:2297857(4096)
ack 1 win 32768 <nop,nop,timestamp 2323887 2321745> (DF) (ttl 64, id 32604)
23:54:23.996407 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 32768
<nop,nop,timestamp 2321745 2323870> (DF) (ttl 64, id 18389)
23:54:23.996444 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 32768
<nop,nop,timestamp 2321745 2323870> (DF) (ttl 64, id 18390)
23:54:23.996495 10.0.0.2.1072 > 10.0.0.1.5001: . 2297857:2301953(4096)
ack 1 win 32768 <nop,nop,timestamp 2323887 2321745> (DF) (ttl 64, id 32605)
23:54:23.996579 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 32768
<nop,nop,timestamp 2321745 2323870> (DF) (ttl 64, id 18391)
23:54:23.996597 10.0.0.2.1072 > 10.0.0.1.5001: . 2301953:2306049(4096)
ack 1 win 32768 <nop,nop,timestamp 2323887 2321745> (DF) (ttl 64, id 32606)
23:54:23.996657 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 32768
<nop,nop,timestamp 2321745 2323870> (DF) (ttl 64, id 18392)
23:54:23.996698 10.0.0.2.1072 > 10.0.0.1.5001: . 2306049:2310145(4096)
ack 1 win 32768 <nop,nop,timestamp 2323887 2321745> (DF) (ttl 64, id 32607)
23:54:23.996753 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 32768
<nop,nop,timestamp 2321745 2323870> (DF) (ttl 64, id 18393)
23:54:23.996801 10.0.0.2.1072 > 10.0.0.1.5001: . 2310145:2314241(4096)
ack 1 win 32768 <nop,nop,timestamp 2323887 2321745> (DF) (ttl 64, id 32608)

```
23:54:23.996853 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 32768
    <nop,nop,timestamp 2321745 2323870> (DF) (ttl 64, id 18394)
23:54:23.996898 10.0.0.2.1072 > 10.0.0.1.5001: . 2314241:2318337(4096)
    ack 1 win 32768 <nop,nop,timestamp 2323887 2321745> (DF) (ttl 64, id 32609)
23:54:23.996967 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 32768
    <nop,nop,timestamp 2321745 2323870> (DF) (ttl 64, id 18395)
```

The other end has noticed the loss and is reporting it with duplicate ACKs. Fast Retransmit should kick in and we need to resend starting from offset 532481. Instead, we proceed to send the full window before we turn around to face the problem of loss. **Why?** We proceed to send...

```
23:54:24.005786 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 32768
    <nop,nop,timestamp 2321746 2323870> (DF) (ttl 64, id 18469)
23:54:24.005875 10.0.0.2.1072 > 10.0.0.1.5001: . 2621441:2625537(4096)
    ack 1 win 32768 <nop,nop,timestamp 2323888 2321745> (DF) (ttl 64, id 32684)
23:54:24.005899 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 32768
    <nop,nop,timestamp 2321746 2323870> (DF) (ttl 64, id 18470)
23:54:24.006007 10.0.0.2.1072 > 10.0.0.1.5001: P 2625537:2629633(4096)
    ack 1 win 32768 <nop,nop,timestamp 2323888 2321745> (DF) (ttl 64, id 32685)
23:54:24.006037 10.0.0.2.1072 > 10.0.0.1.5001: . 532481:536577(4096)
    ack 1 win 32768 <nop,nop,timestamp 2323888 2321745> (DF) (ttl 64, id 32686)
```

At this point, we have sent $2629633 - 532481 = 2097152$ bytes ahead, which is exactly our window size. Finally, we notice the duplicate ACKs are coming in and resend the segment starting from 532481.

```
23:54:24.006060 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 32768
    <nop,nop,timestamp 2321746 2323870> (DF) (ttl 64, id 18471)
23:54:24.006178 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 32768
    <nop,nop,timestamp 2321746 2323870> (DF) (ttl 64, id 18472)
```

Up to this point, everything was happening in rapid fire mode. Now's the first time there's any waiting. We just resent segment starting from 532481, and there were only two more duplicate ACKs for it since then: not enough to warrant another Fast Retransmit. So, we wait 1 second.

```
23:54:25.000538 10.0.0.2.1072 > 10.0.0.1.5001: . 532481:536577(4096)
    ack 1 win 32768 <nop,nop,timestamp 2323988 2321746> (DF) (ttl 64, id 32687)
```

We resend the unlucky packet again, without getting any ACK back.

```
23:54:27.000562 10.0.0.2.1072 > 10.0.0.1.5001: . 532481:536577(4096)
    ack 1 win 32768 <nop,nop,timestamp 2324188 2321746> (DF) (ttl 64, id 32688)
```

We wait 2 seconds now, and send again. No response.

```
23:54:31.000646 10.0.0.2.1072 > 10.0.0.1.5001: . 532481:536577(4096)
    ack 1 win 32768 <nop,nop,timestamp 2324588 2321746> (DF) (ttl 64, id 32689)
```

Wait 4 seconds, send again, no response.

```
23:54:39.000751 10.0.0.2.1072 > 10.0.0.1.5001: . 532481:536577(4096)
    ack 1 win 32768 <nop,nop,timestamp 2325388 2321746> (DF) (ttl 64, id 32690)
```

Wait 8 seconds, send again, no response.

```
23:54:55.001004 10.0.0.2.1072 > 10.0.0.1.5001: . 532481:536577(4096)
    ack 1 win 32768 <nop,nop,timestamp 2326988 2321746> (DF) (ttl 64, id 32691)
```

Wait 16 seconds, send again, no response.

```
23:55:27.001500 10.0.0.2.1072 > 10.0.0.1.5001: . 532481:536577(4096)
    ack 1 win 32768 <nop,nop,timestamp 2330188 2321746> (DF) (ttl 64, id 32692)
```

Wait 32 seconds, send again, no response...

```
00:00:47.006327 10.0.0.2.1072 > 10.0.0.1.5001: . 532481:536577(4096)
    ack 1 win 32768 <nop,nop,timestamp 2362188 2321746> (DF) (ttl 64, id 32697)
```

```
00:01:51.007294 10.0.0.2.1072 > 10.0.0.1.5001: . 532481:536577(4096)
    ack 1 win 32768 <nop,nop,timestamp 2368588 2321746> (DF) (ttl 64, id 32698)
```

A pause of 64 seconds between these last packets. It's 446 seconds since we have sent the first retransmission.

```
00:02:55.008237 10.0.0.2.1072 > 10.0.0.1.5001: R 2629633:2629633(0)
    ack 1 win 32768 (DF) (ttl 64, id 32699)
```

Our patience is exhausted. Reset the connection.

Sender's Behavior

- The sender never sends beyond the advertised window size (good).
- When it notices the loss, it starts to resend the first lost packet with exponentially increasing intervals between attempts (good), but never sees any ACKs coming back.
- Fast Retransmit doesn't happen right after the sender gets duplicate ACKs; instead, the sender blasts the remainder of the window and only then resends.
- A few weeks later, after talking to Matt Mathis: Are packets waiting in the host's IP queue? BPF is below IP, so we're capturing one queue down.

Receiver's View of the Same Connection

```
23:54:23.681963 10.0.0.2.1072 > 10.0.0.1.5001: S 577076198:577076198(0) win 65535
  <mss 4430,nop,wscale 6,nop,nop,timestamp 2323869 0> (DF) (ttl 64, id 32041)
23:54:23.682021 10.0.0.1.5001 > 10.0.0.2.1072: S 168494072:168494072(0)
  ack 577076199 win 65535 <mss 4430,nop,wscale 6,
  nop,nop,timestamp 2321727 2323869> (DF) (ttl 64, id 18233)
23:54:23.682161 10.0.0.2.1072 > 10.0.0.1.5001: .
  ack 1 win 32768 <nop,nop,timestamp 2323869 2321727> (DF) (ttl 64, id 32042)
23:54:23.682208 10.0.0.1.5001 > 10.0.0.2.1072: .
  ack 1 win 32768 <nop,nop,timestamp 2321727 2323869> (DF) (ttl 64, id 18234)
23:54:23.682875 10.0.0.2.1072 > 10.0.0.1.5001: P 1:4097(4096) ack 1 win 32768
  <nop,nop,timestamp 2323869 2321727> (DF) (ttl 64, id 32043)
23:54:23.682907 10.0.0.2.1072 > 10.0.0.1.5001: P 4097:8193(4096) ack 1 win 32768
  <nop,nop,timestamp 2323869 2321727> (DF) (ttl 64, id 32044)
23:54:23.682949 10.0.0.1.5001 > 10.0.0.2.1072: . ack 8193 win 32640
  <nop,nop,timestamp 2321727 2323869> (DF) (ttl 64, id 18235)
23:54:23.682969 10.0.0.2.1072 > 10.0.0.1.5001: P 8193:12289(4096)
  ack 1 win 32768 <nop,nop,timestamp 2323869 2321727> (DF) (ttl 64, id 32046)
```

Initial handshake, start of transmission, everything goes well for a while...

```
23:54:23.687188 10.0.0.2.1072 > 10.0.0.1.5001: P 524289:528385(4096)
    ack 1 win 32768 <nop,nop,timestamp 2323870 2321728> (DF) (ttl 64, id 32172)
23:54:23.687198 10.0.0.1.5001 > 10.0.0.2.1072: . ack 524289 win 24704
    <nop,nop,timestamp 2321728 2323870> (DF) (ttl 64, id 18298)
23:54:23.687222 10.0.0.2.1072 > 10.0.0.1.5001: P 528385:532481(4096)
    ack 1 win 32768 <nop,nop,timestamp 2323870 2321728> (DF) (ttl 64, id 32173)
23:54:23.799127 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 24576
    <nop,nop,timestamp 2321739 2323870> (DF) (ttl 64, id 18299)
23:54:23.855860 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 24704
    <nop,nop,timestamp 2321744 2323870> (DF) (ttl 64, id 18300)
23:54:23.855957 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 24832
    <nop,nop,timestamp 2321745 2323870> (DF) (ttl 64, id 18302)
```

Apparently, BPF or tcpdump has lost some incoming packets here, which the TCP stack has seen, and which caused the TCP stack to generate those duplicate ACKs.

```
23:54:23.856019 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 24960
    <nop,nop,timestamp 2321745 2323870> (DF) (ttl 64, id 18303)
23:54:23.856070 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 25088
    <nop,nop,timestamp 2321745 2323870> (DF) (ttl 64, id 18304)
23:54:23.856123 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 25216
    <nop,nop,timestamp 2321745 2323870> (DF) (ttl 64, id 18305)
23:54:23.856180 10.0.0.2.1072 > 10.0.0.1.5001: . 2105345:2109441(4096)
    ack 1 win 32768 <nop,nop,timestamp 2323887 2321744> (DF) (ttl 64, id 32558)
```

Good grief! That's what I call burst loss. Everything between 532481 and 2105345 (384 packets) was lost. (Well, almost everything: something must have caused us to generate those duplicate ACKs.)

```
23:54:23.856220 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 25280
    <nop,nop,timestamp 2321745 2323870> (DF) (ttl 64, id 18306)
23:54:23.856264 10.0.0.2.1072 > 10.0.0.1.5001: . 2109441:2113537(4096)
    ack 1 win 32768 <nop,nop,timestamp 2323887 2321744> (DF) (ttl 64, id 32559)
23:54:23.856288 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 25408
    <nop,nop,timestamp 2321745 2323870> (DF) (ttl 64, id 18307)
23:54:23.856345 10.0.0.2.1072 > 10.0.0.1.5001: . 2113537:2117633(4096)
    ack 1 win 32768 <nop,nop,timestamp 2323887 2321745> (DF) (ttl 64, id 32560)
```

They go on blasting, we go on sending ACK for 532481...

```
23:54:23.870256 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 32768
    <nop,nop,timestamp 2321746 2323870> (DF) (ttl 64, id 18470)
23:54:23.870374 10.0.0.2.1072 > 10.0.0.1.5001: . 2621441:2625537(4096)
    ack 1 win 32768 <nop,nop,timestamp 2323888 2321745> (DF) (ttl 64, id 32684)
23:54:23.870403 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 32768
    <nop,nop,timestamp 2321746 2323870> (DF) (ttl 64, id 18471)
23:54:23.870495 10.0.0.2.1072 > 10.0.0.1.5001: P 2625537:2629633(4096)
    ack 1 win 32768 <nop,nop,timestamp 2323888 2321745> (DF) (ttl 64, id 32685)
23:54:23.870526 10.0.0.1.5001 > 10.0.0.2.1072: . ack 532481 win 32768
    <nop,nop,timestamp 2321746 2323870> (DF) (ttl 64, id 18472)
```

They have almost exhausted the window now (there's one packet's worth left in in, and from the examination of the sender's view it's known that this packet was in fact sent, but we don't see it).

```
00:02:54.876791 10.0.0.2.1072 > 10.0.0.1.5001: R 2629633:2629633(0)
    ack 1 win 32768 (DF) (ttl 64, id 32699)
```

Out of the blue they just send us RST packet. We ignore it, without sending an RST back, even though it's within our window ($2629633 - 532481 = 2097152 = 2^{21} = \text{window size}$). Was confirmed to be an off-by-one error in TCP code ('<' instead of ' \leq ').

Receiver's Summary

- Huge burst loss occurs in the middle of the sender's window.
- Other losses occur later, but we get most packets starting from some point (but, notably, not the last packet in the succession).
- We reject a valid reset (a bug).

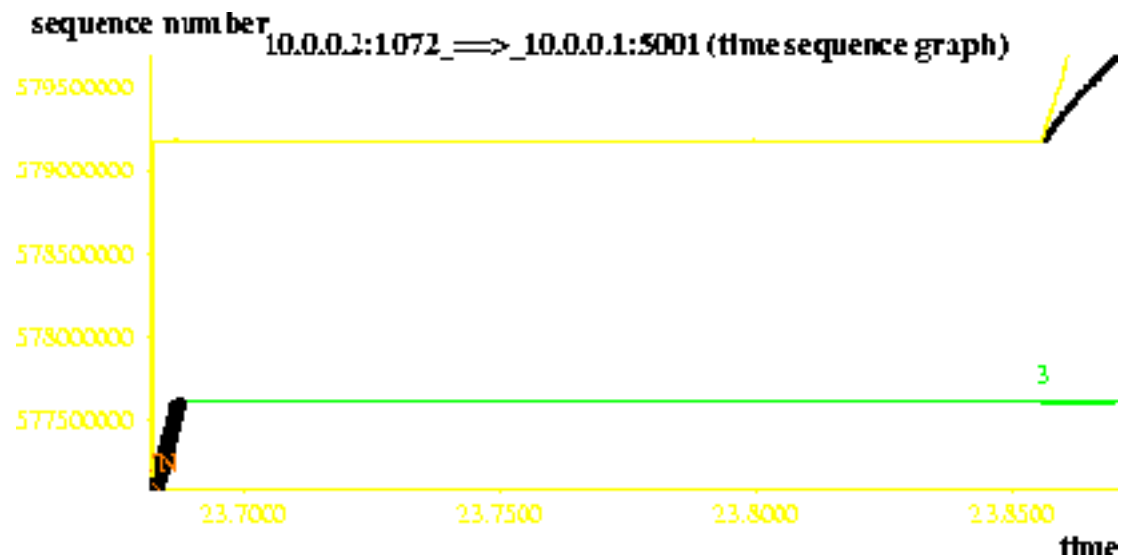
Additionally:

- While loss is happening, communication using small packet sizes works fine.
- The cause of the loss was in the `ti` driver on the receiver's side.
- Different buffer pools for packets of different sizes.
- Was solved by changing a `#define` in the driver source.

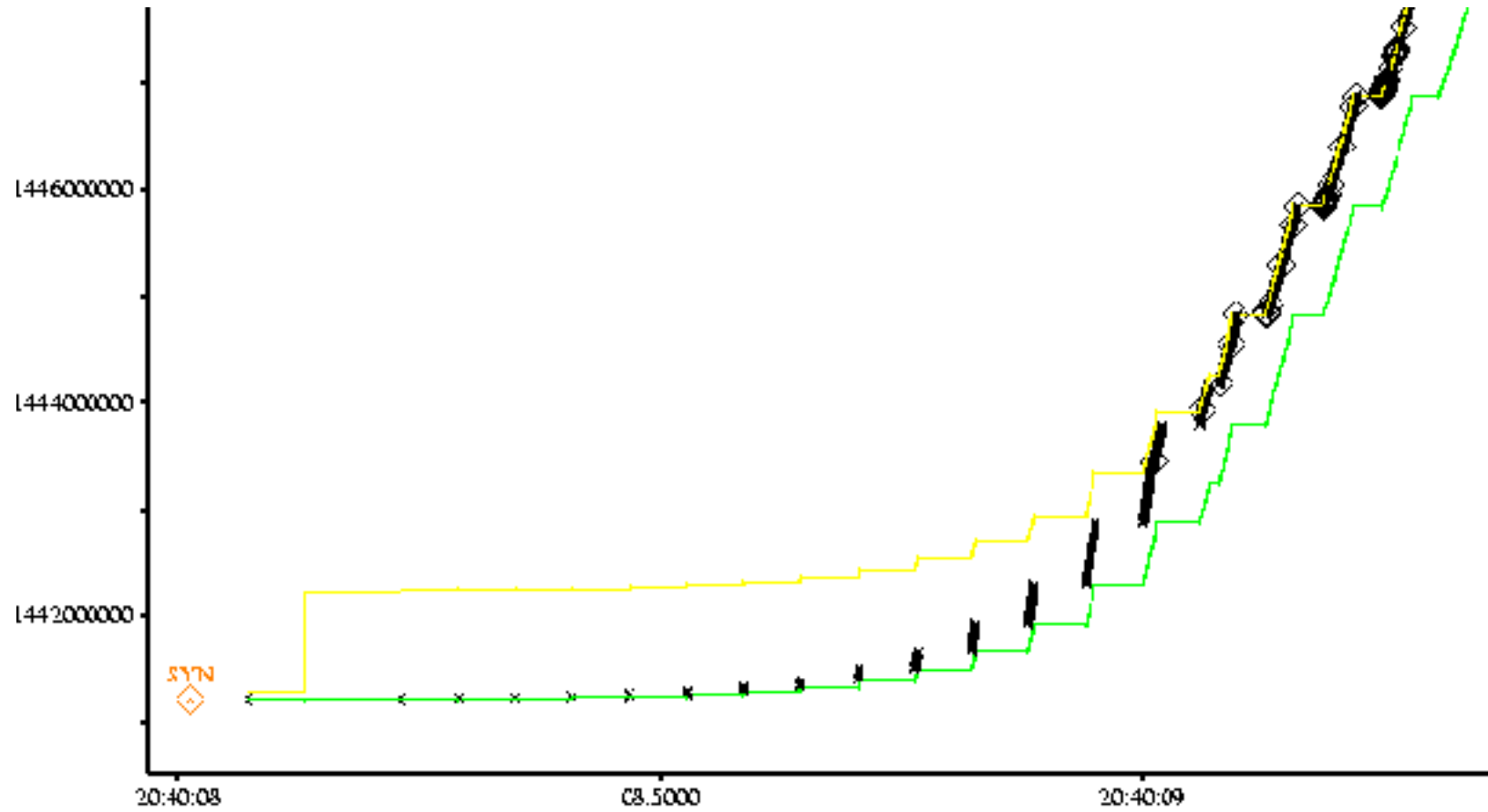
tcptrace: a Tool to Simplify Trace Examination

- Reads `tcpdump` files and does most of the dirty work for you
- Gives you one big picture instead of a lot of small packets
- In the default mode, doesn't do much
- Can produce textual summary of connections with `-l`
- Can produce plots; most useful is time sequence plot (`-S`)
- The time sequence plot mostly shows you The Truth (from 30000ft)
- Other plots suffer from inadequate averaging and processing
- Plots are examined with `xplot` (can zoom in and out)

xplot shows the same problem connection



xplot shows normal WAN slow start



Limitations of tcptrace and xplot

- The trace needs to be stored to file
- The plot is quite a bit larger than the trace
- The plot is loaded into memory
- Result: cannot examine fast connections
- (The kernel sent SIGKILL to xplot trying to load 1 GB plot of a 100 Mb/s-limited connection.)
- The bird's eye view is very useful, but it can't tell the details
- Correlation between sender and receiver is not easily possible
- With xplot, we would not have been able to diagnose the failed connection successfully analyzed before
- Throughput plots (-T) average over n packets, not over a period of time, so they are unnecessarily bursty
- Congestion window plots (-N) need to guess too much
- Segsize plots (-F) are not very useful when only full segments are sent and not very readable otherwise

Requirements for Packet Trace Reading Tools

- Process data on the fly without going to disk (use `libpcap` or pipe from `'tcpdump -w -'`)
- Do not require more than the TCP window size of memory
- Use as few CPU cycles as possible
- Present a picture even more high-level than `tcptrace -S`
- Fix `tcptrace -T`
- Even need hacks for post-processing decoded `tcpdump` output to exclude unwanted information, color-code, correlate several files, etc.