Data Center TCP (DCTCP)

Networking Services Team, Red Hat

Florian Westphal
1024D/F260502D fw@strlen.de
1C81 1AD5 EA8F 3047 7555
E8EE 5E2F DA6C F260 502D

February 6th 2015
TCP Congestion Control – History

Did not exist in the early days.

- RFC 793: “[..] segments may be lost due to [..] network congestion, TCP uses retransmission (after a timeout) to ensure delivery.”
- Sender transmits as much data it has to send & current rwnd allows
- Massive problems, ca. 1986: → congestion collapse: most packets in networks were retransmits
- RFC 2581 – TCP congestion control
  - Slow Start, Congestion avoidance: cwnd – sender imposed flow control
  - Fast Retransmit & Fast Recovery
Introduction to Congestion Control and ECN

Congestion Control (simplified, Reno)

![Graph showing the behavior of congestion control in Reno's simplified model. The graph illustrates the movement of cwnd over time with distinct phases of slow start and congestion avoidance.]

- Slow start
- Congestion avoidance

Time →
Congestion Control Issues

- Not trivial to decide when to grow/shrink cwnd value
  - Link might have large delay
  - Packet reordering does happen
  - Even if packets have been lost: not necessarily due to congestion
  - Available network capacity is not constant

- Active research topic, dozens of different algorithms
Linux TCP + Congestion Control: Architecture

- Many different congestion control algorithms
- Default: CUBIC (since 2006)
- Plugin-based congestion control framework
- Different algorithms give different weight to the available information (e.g. rtt, duplicate acks, rwin, ...):
  - Hybla: don’t penalize connections with high rtt
  - Veno: less aggressive cwnd decrease on loss
  - ...
Linux TCP + Congestion Control: Configuration

- sysctl net.ipv4.tcp_congestion_control=vegas
- ip route add $dst dev $dev congctl vegas\(^1\)
- setsockopt(., TCP_CONGESTION, "vegas", .);
- Every tcp connection has exactly one assigned algorithm
- But individual connection can each use different one

\(^1\)3.20, http://git.kernel.org/cgit/linux/kernel/git/davem/net-next.git/commit/?id=81164413ad096bafe8ad1068f3f095a7dd081d8b
Congestion Control – Queueing

ingress: median arrival rate in a given interval
ingress > Egress: queue starts to form
Issues for senders

- Bufferbloat: too big buffers take sender longer to realize loss when it occurs
- Too small buffers can cause needless loss during short bursts
- Simple ”drop when buffers are full” can affect many flows → global synchronization, incast
Wishlist

- Want to know about loss asap
- Active queue management in switches/routers
  - e.g. RED: drop with increasing probability once buffers start filling up
  - also: sfb, codel, fq_codel, choke, . . .

Ideally . . .
Allow detection of imminent congestion before loss occurs
Explicit Congestion Notification (ECN)

- Extension to IP to allow routers/switches to signal congestion before packets are dropped
- Uses two bits in the IPv4 header TOS octet, 3 states:
  1. ECN-unaware (00)
  2. Two ECN-Nonces, 01 and 10 – ECN-aware
  3. Congestion experienced (11)
- receiver can detect when congestion occurs
- but only sender could do something about it
ECN & TCP

- Two new TCP header flags: ECN-echo & Congestion Window Reduced
- ECE: Used by receiver to inform sender that it received CE-marked packet
- CWR: Used by sender to tell Receiver that congestion window was reduced
- Use is "negotiated" during three-way-handhake

To enable on Linux:

net.ipv4.tcpecn=1 or

ip route change 192.168.2.0/24 dev eth0 features ecn

---

3.20, http://git.kernel.org/cgit/linux/kernel/git/davem/net-next.git/commit/?id=f7b3bec6f5167efaf56b756abfadfb924cb1d3050
ECN issues

- bugs in middleboxes (e.g. firewalls, tcp ”accelerators“, etc):
  - ECN Blackholes: packets with SYN+CWR+ECE bits set are dropped
  - All packets might get CE marked (even more frequent with ipv6).
  - Even if signalling would work: proper marking (virtually) never happens
- Design:
  - Doesn't quantify the extent of the congestion, only presence

lots of pain for little gain
... And thus virtually no default-on
Summary

- Current tcp stacks are very good at detecting loss & loss recovery
- But loss still bad for latency
- ECN supported by all major OS and switch firmware
- Problematic due to myriad of implementation bugs / misconfigurations
- But if you have full control over all nodes involved (e.g. within datacenter ... )
Datacenter TCP

- Designed as improvement to TCP Congestion Control for DC traffic
  1. High burst tolerance (incast due to partition/aggregate)
  2. Low latency (short flows, queries)
  3. High throughput (large file transfers)

- ECN is used to estimate amount of bytes that experienced congestion (i.e., extent, not just presence)

```
P  P  P  M  M  M  ·  ·  ·  M  K
```

- Tailroom
  enough tolerance to absorb bursts

Suggested mark threshold \( k \) for 10Gbit Ethernet: 65 packets (\( \approx 100\text{KB} \))
DCTCP: congestion estimate

SND.UNA, SND.NXT used as 'observation window'
Add counters for marked and total bytes
for each acceptable ack:

1. Count the bytes acked
2. If ack has ECE set, also count those bytes as "marked"
3. If SND.UNA not yet reached, stop; else update alpha:
   1. Compute $F$: \( \frac{\text{marked}}{\text{total}} \)
   2. Compute $\alpha_{\text{new}} = (1 - g) \times \alpha_{\text{prev}} + g \times F$
   3. Start new observation window, valid until current SND.NXT acknowledged

- $F$ fraction of packets marked in last window
- $g$ is weight given to new samples (default: $\frac{1}{16}$)
**DCTCP: cwnd computation**

α represents fraction of marked packets

Congestion window is computed as follows:

\[
\text{cwnd}_{\text{new}} = \text{cwnd}_{\text{prev}} \times 1 - \frac{\alpha}{2}
\]

- α ≈ 0 little/no congestion, α ≈ 1: high/full congestion
- Everything depends on realistic estimate of the marked bytes
- How to infer when one of our data packets was marked?
- Simple and wrong solution: send ack for every single packet
DCTCP: ACK generation state machinery

- **CE=0**: normal ACKs
  - Send quick ACK

- **CE=1**: send quick ACK with ECE
  - (delayed) ECE ACK

Quickacks are only sent when state changes.
DCTCP: Implementation

- DCTCP congestion control module
- Stack was extended to provide a couple of more events to modules
- CC modules can now indicate (force) ECN
- Fallback to Reno CC if peer doesn’t support ECN
- Easiest way to enable:
  ip route change dev eth0 10.0.0.7/24 congctl dctcp
DCTCP: Operation

- Read Documentation/networking/dctcp.txt
- Suggest to only enable it for local network(s), not globally: `ip route ... congctl dctcp`
- Don’t need extra ecn-tuning on end-hosts, ecn will be used automatically

```bash
$ ss -nite
Send-Q Local Address:Port Peer Address:Port
12408 192.168.7.10:22 192.168.7.1:35274 [...] dctcp [...] ce_state 0 alpha 312 ab_ecn 2896 ab_tot 0
```
$ ss -nite
Send-Q  Local Address:Port Peer Address:Port
12408   192.168.7.10:22  192.168.7.1:35274 [..]
dctcp [..] ce_state 0 alpha 312 ab_ecn 2896 ab_tot 0

- dctcp-reno: fallback mode: other host using e.g. CUBIC with ecn off
- alpha: if large (max 1024): huge congestion or middlebox marking all packets
## DCTCP: results from data center deployment

<table>
<thead>
<tr>
<th>Latency (in ms)</th>
<th>CUBIC</th>
<th>DCTCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>4.0088</td>
<td>0.04219</td>
</tr>
<tr>
<td>Median</td>
<td>4.055</td>
<td>0.0395</td>
</tr>
<tr>
<td>Max</td>
<td>4.2</td>
<td>0.085</td>
</tr>
<tr>
<td>Min</td>
<td>3.32</td>
<td>0.028</td>
</tr>
<tr>
<td>Stddev</td>
<td>0.1666</td>
<td>0.01064</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Throughput, in Mbps</th>
<th>CUBIC</th>
<th>DCTCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>521.684</td>
<td>521.895</td>
</tr>
<tr>
<td>Median</td>
<td>464</td>
<td>523</td>
</tr>
<tr>
<td>Max</td>
<td>776</td>
<td>527</td>
</tr>
<tr>
<td>Min</td>
<td>403</td>
<td>519</td>
</tr>
<tr>
<td>Stddev</td>
<td>105.891</td>
<td>2.601</td>
</tr>
</tbody>
</table>

---

3 per flow, 19 senders in parallel

4 [http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=e3118e8359bb7](http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=e3118e8359bb7)
DCTCP: Issues

- Must configure all routers/switches to mark at $k$
- Must separate DCTCP and TCP traffic on switches (e.g. via DSCP marking) to maintain fairness
- Pure ACK loss breaks congestion estimate
- Both paper and ietf draft are not clear on a few details, e.g.
  - Should $\alpha$ be changed on loss?
  - ... only on timeout?

Questions?
Bibliography

M. Alizadeh, A. Greenberg, D.A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, M. Sridharan:
Data Center TCP (DCTCP), Data Center Networks session

S. Bensley, L. Eggert, D. Thaler
Microsoft’s Datacenter TCP (DCTCP): TCP Congestion Control for Datacenters