Restructuring Endpoint Congestion Control

Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, Hari Balakrishnan
Congestion control in UG networking

- **Congestion**: when traffic sources send too much data into a network

- **Congestion control** algorithms at sources react to network congestion and vary sending rate accordingly

- A fundamental problem with over 30 years of research
Traditional congestion control

Applications

TCP

Data

Operating System

Network interface

"datapath"
Trend #1: New datapaths

End of Moore’s law
Apps with strict perf reqs
Apps with unique needs

Kernel-bypass e.g., DPDK
Operating System TCP
Smart NICs (FPGAs & SoCs)
Libraries & libOSs (e.g., Quic)
Offload eng (RDMA, TOE)
Trend #2: New algorithms

- Increasing bandwidth
- Improving deployed algorithms
- Changing application requirements
- Different deployment scenarios
- Using new algorithmic tools
Algorithm complexity

Sprout (NSDI 2013): Bayesian forecasting
Remy (SIGCOMM 2013): Offline learning
PCC / PCC Vivace (NSDI 2015 / NSDI 2018): Online learning
Indigo (Usenix ATC 2018): Reinforcement learning
Nimbus (2018--): FFT/Signal processing

Floating point operations
User-level libraries
Extensive experimentation/tuning
Need cross-product of implementations!

H-TCP  Veno  Hybla
  TIMELY  XCP  Westwood
Compound Sprout  EBCC  BIC
Cubic  PRR  Binomial  Nimbus
DCQCN  Reno  Vegas  Indigo
  Tahoe  NewReno  Vivace
RCP  DCTCP  RC3  ABC
FAST  LEDBAT  NV  BBR
Illinois  Remy  PCC  Copa

OS TCP
USERSPACE
RDMA
SMARTNIC
FPGA
MTCP
DPDK
QUIC
DCCP
Need cross-product of implementations!
Trend #3: New datapath capabilities

- Independent CC
- Aggregate CC
- Software CC
- Offloaded CC
Congestion control today is too tightly coupled to datapath, hard to implement, and hard to evolve.
Congestion Control Plane (CCP)
Traditional datapath design

Application

TX  RX

Congestion control

Datapath state

Datapath

Network interface
Congestion Control Plane (CCP) design

Separate userspace process

- CCP agent
- CCP datapath component
- Datapath state
- Datapath

Application

TX  RX

Network interface

(for example)
Congestion control plane (CCP) design

Datapath

- Network interface
- Datapath state
- CCP datapath component
- CCP agent

Application

TX RX
Congestion control plane (CCP) design

Address space separation

1. User
2. Kernel

CCP agent

Datapath

Datapath state

CCP datapath component

Application

TX
RX

Network interface
Congestion control plane (CCP) design

(1) Address space separation
(2) Separate mechanism (datapath dependent) from policy (datapath independent)
(3) Make datapath mechanisms flexible through a DSL
Congestion control plane (CCP) design

Datapath state

CCP datapath component

Datapath

Application

TX

RX

Network interface

CCP agent

Write once & run anywhere

Sophisticated algorithms

New CC capabilities without datapath modifications
A possible usage of CCP: TCP’s AIMD

On every ACK:
\[ cwnd := cwnd + (# \text{ACK’d pkts}) / cwnd \]

On every loss:
\[ cwnd := \frac{cwnd}{2} \]
But what about performance?
Performance concerns

• Increased latency and decreased throughput
  • Should every packet cross address-space boundaries?

• Reacting to stale information
  • We don’t want to delay an algorithm reacting to congestion

• Loss of fidelity
  • Is behavior within CCP agent == behavior within the datapath?
(1) Throughput and Latency

Agent can’t process every packet as it comes.

Communicate info for batches of packets. Increases tput when agent receives packets.
(2) Reacting to stale network state

Agent will incur a delay before receiving info of last packet.

Split the CC implementation.
Reacting to stale network state

Agent will incur a delay before receiving info of last packet.

Split the CC implementation.
(2) Reacting to stale network state

**Sync measurement & control**
- Slow logic in the async CCP agent
- On every ACK:
  
  \[
  \text{cwnd} := \text{cwnd} + \left(\text{# ACK'd pkts}/\text{cwnd}\right)
  \]

**Fast** logic in the sync datapath component
- On every loss:
  
  \[
  \text{cwnd} := \text{cwnd} / 2
  \]
(3) Fidelity relative to datapath-only CC

Digital control theory: working with samples “as good” as regular signal if sampled at least every loop delay

Communicating once (or few times) per RTT is enough.

Natural time scale for CC: RTT.
What, when & how should CCP agent and datapath communicate?
Split congestion control

CCP datapath exposes standard primitives to agent.

Control: set cwnd, rate

Measurement: measure & summarize over time using a DSL

Standardized datapath interface
### CCP datapath DSL

<table>
<thead>
<tr>
<th>Measurement timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-order acked bytes</td>
</tr>
<tr>
<td>Out-of-order acked bytes</td>
</tr>
<tr>
<td>ECN-marked bytes</td>
</tr>
<tr>
<td>Lost bytes</td>
</tr>
<tr>
<td>Timeout occurred</td>
</tr>
<tr>
<td>RTT sample</td>
</tr>
<tr>
<td>Bytes in flight</td>
</tr>
<tr>
<td>Outgoing rate</td>
</tr>
<tr>
<td>Incoming rate</td>
</tr>
</tbody>
</table>

Summarize primitive measurements using **fold functions over packets** within the CCP datapath.

Write **custom report triggers** within the datapath to invoke CCP agent.

Implement **callback functions** within the CCP agent to respond to those triggers.
libccp implements a simple register-based machine and runs datapath CC programs on each packet. It is datapath-agnostic.

When true:
- \( := \text{minrtt} \ (\text{min minrtt Ack.rtt_sample_us}) \)
- \( := \text{curr_btl_est} \ (\text{max curr_btl_est Flow.rate_incoming}) \)
- (fallthrough)

Shim layer exposes datapath variables. It is datapath-specific.
CCP Performance
TCP CUBIC window dynamics

96 Mbit/s, 20 ms link RTT
Write once, run anywhere

Kernel

QUIC

mTCP

Copa

Link: 12 Mbit/s, 20 ms RTT

Cubic

Link: 24 Mbit/s, 20 ms RTT
CPU utilization while saturating 10 Gbit/s

Link: 10 Gbit/s, 100 µs RTT

5% on one core
Low-RTT: Incast w/ 50 senders (simulation)

Link: 10 Gbit/s, 10 µs
Slow start: Partitioning

Link: 48 Mbit/s, 100 ms RTT
TCP BBR: Partitioning

**CCP agent**
*asynchronous, on each report (one per 8 RTTs)*
- Calculate new rate based on measurements
- Handle switching between modes

**Datapath program**
*synchronous, per ACK*
- Pulse: rate = 1.25 x bottleneck rate
- After 1 RTT: rate = 0.75 x bottleneck rate
- After 2 RTT: rate = bottleneck rate
- After 8 RTT: repeat
Capabilities enabled

- Sophisticated algorithms
- Faster prototyping
- CC for flow aggregates
- Application-integrated CC
- Dynamic, path-specific CC
Looking forward

Next steps

- More algorithms
- Hardware datapaths
- Other new capabilities based on the CCP platform

Current status

- Datapaths:
  - Linux TCP
  - Quic
  - mTCP/DPDK
- CCP agent (Rust/Python)

https://ccp-project.github.io
ccp@csail.mit.edu
srinivas.narayana@rutgers.edu
eBPF

Front-End (Language)

▸ Event-driven semantics
▸ Explicit reporting model

(def (Report (acked 0)))
(when true
  (:= Report.acked (+ Report.acked Ack.bytes_acked))
  (:= Cwnd (+ Cwnd Report.acked))
  (fallthrough))
(when (> Flow.lost_pkts_sample 0)
  (report))

Back-End (Datapath)

▸ Congestion control enforcement
▸ Direct access to socket state
How to use CCP to build CC algos?

CCP agent: Asynchronous event-driven CC program.

Callbacks roughly every RTT or on program-specified conditions

```python
def onReport(self, report):
    if report['lost'] > 0:
        self.cwnd = self.cwnd / 2
    else:
        acked = report['acked']
        self.cwnd = self.cwnd + acked * MSS / self.cwnd
        self.update('cwnd', self.cwnd/MSS)
```
Computer Science @ Rutgers

- csrankings.org
  - #9 in algorithms & complexity
  - #15 in AI

- Growing presence in systems and networking. Last 5 years:
  - Best paper awards at PLDI, SIGCOMM, FAST, ICSE
  - IEEE MICRO top picks papers
  - ACM SIGPLAN doctoral dissertation award
  - Incoming faculty: best papers @ PLDI, Usenix Security, NDSS

- Apply to Rutgers CS for graduate school and postdocs