Cuback

draft-kazuho-ccwg-cuback-01

Kazuho Oku @ Fastly

IETF 127

Reno is simple

- driven by ACK
- increment cwnd by 1 SMSS each time the bytes newly acked reach a threshold

CUBIC is complex

- two state machines emitting window estimates; the sender takes the max:
  - the cubic curve, using the wall clock time
  - the Reno increase, driven by ACKs
- pausing and resuming the cubic clock
- smoothing: cwnd moves toward *W_cubic(t + RTT)* on each ACK
- fast convergence

Can we make CUBIC as simple as Reno?

- yes, if we drive the cubic increase function using ACKs
- Cuback = CUBic driven by the ACK clock

Introducing Cuback

- the design
- relationship to CUBIC
- ease of implementation
- simulation and production results
- conclusion and next steps

The increase function

```
  pending = pending + segments_acked
  while pending >= D(cwnd):
    pending = pending - D(cwnd)
    cwnd = cwnd + 1
```

Reno:

```
D(cwnd) = cwnd
```

Cuback:

```
D(cwnd) = ...
```

The Cuback increase function

```
D(w)        = max(2, A(w + 1) - A(w))

A(w)        = min(A_cubic(w), A_reno(w))

A_cubic(w) = bandwidth * (K(W_max, cwnd_epoch) + cbrt((w - W_max) / C))

A_reno(w)  = (w - cwnd_epoch) * (w + cwnd_epoch - 1)
               / (2 * alpha_cubic)                                  w <= W_max

           = (W_max - cwnd_epoch) * (W_max + cwnd_epoch - 1)
               / (2 * alpha_cubic)
             + (w - W_max) * (w + W_max - 1) / 2                    w >  W_max

K(W_max, cwnd_epoch) = cbrt((W_max - cwnd_epoch) / C)
```

The Cuback increase function: parameters

- per-epoch parameters set when entering recovery
  - *W_max, cwnd_epoch*
    - same as in CUBIC
  - *bandwidth*
    - estimate of the bottleneck flow rate, used to map the cubic curve from wall-clock time onto ACKs
    - calculated as `cwnd_prior / sRTT` when entering recovery
  - *K* may also be retained, as an optimization
- constants:
  - *alpha_cubic, beta_cubic, C*
    - same as in CUBIC

What no longer exists

- the wall clock
  - no *t_epoch*, no pausing / resuming the cubic clock when app-limited
- *W_est*, the separately evolving Reno-friendly window
  - replaced by A_reno, a closed form compared against *A_cubic*
- smoothing toward *W_cubic(t + RTT)*
  - CUBIC: requires smoothing because the increase is driven by time
  - Cuback: ACK-clocked as in Reno; needs no more smoothing than Reno does
- mutable states:
  - all values beyond Reno's are immutable during one congestion period
    - as in Reno, only 2 mutables exist, driven by ACKs (*cwnd and pending*)

What remains

- fast convergence
  - when congestion is observed and `cwnd < W_max`:
    - `W_max = cwnd * (1 + beta_cubic) / 2`
- response to spurious congestion events
  - restore variables (*W_max, cwnd_epoch, bandwidth*) to their original values

Application-limited senders

- Using draft-ietf-ccwg-ratelimited-increase (RLI), increase is permitted if:

```
(A(w + 1) − A(maxFS) <= maxFS and w + 1 <= 1.5 * maxFS) or
  w <= maxFS
```

- because RLI caps cwnd at *limit(maxFS)*: what one window of *maxFS* ACKs can yield
  - CUBIC: depends on per-ACK interplay of *cwnd, W_est*, and ACK timing
  - Cuback: determined by *A(w)* alone
- 1st clause: data needed to grow from *maxFS* to *w + 1* fits in one window;
  D(w) >= 2 caps it at *1.5 * maxFS*
- 2nd clause: as with Reno, cwnd may always reach *maxFS + 1*
- *A(w + 1)* is computed for *D(w)* anyway → no extra cube root

Relationship to CUBIC

- sensitivity to bandwidth estimate
- convergence under the ACK clock
- sensitivity to the timing of clock transitions
- the Reno-friendly estimate
- smoothing the cubic growth
- calculation overhead

Sensitivity to the bandwidth estimate

- bandwidth scales *A_cubic* linearly
  - underestimate → curve traversed faster; overestimate → slower
- error is bounded both ways
  - too large: *A_reno* (no bandwidth term) wins → Reno-friendly rate
  - too small: *D(w)* >= 2 → at most +50% per RTT, as in CUBIC
- only matters once the cubic curve governs, i.e. at large windows
  - many ACKs per RTT → accurate sRTT
    → accurate bandwidth *(cwnd_prior / sRTT)*

Convergence under the ACK clock

- **flows converge sooner than under CUBIC**
  - curve advances at *(current rate) / (rate recorded at last congestion event)*
  - incumbent gives up the most at a congestion event → its clock runs slow
  - newcomer achieves more than it recorded → its clock runs fast
  - advantage decays as achieved rate converges on recorded rate
    → fast convergence is still needed, as in CUBIC
- → short flows (e.g. HTTP object transfer) complete earlier

Sensitivity to the timing of clock transitions

- CUBIC must pause / resume its clock exactly at app-limited edges
  - miss the pause → W_cubic(t) runs ahead, cwnd grows unearned
  - miss the resume → elapsed time lost, flow grows too slowly
- edges are hard to observe (batched sends, in-stack pacers)
- Cuback has no clock: no ACKs → no growth, nothing to catch up
- sender below cwnd: growth bounded by delivered data, capped by RLI

The Reno-friendly estimate

- Reno-friendly estimate: the window Reno would have reached
  - CUBIC: in the same time
  - Cuback: after acking the same amount of data
- Therefore:
  - if the congestion period starts within the Reno-friendly region, the behavior is identical
  - else, Cuback enters the Reno-friendly region somewhat earlier
    (*cwnd* > *W_est* → more data acked per RTT than Reno would)

Smoothing the cubic growth

- CUBIC targets *W_cubic(t + RTT)*, increasing by *(target - cwnd) / cwnd* per ACK
  - otherwise a gap in ACKs becomes a step in cwnd → a burst
- Cuback: increase is proportional to acked data by construction
  - no smoothing needed

Calculation overhead

- *D(w)* needs 3 cube roots; CUBIC needs one only per epoch
  - store *bytes-until-next-increase* → evaluate per increase, not per ACK
  - retain *K*, carry *A(w + 1)* forward → 1 cube root per increase
- approximation in the draft: ~9 cycles (Zen 3)
  - vs ~900 cycles to AES-GCM-encrypt one 1460-byte segment
  - worst case ~3% of encryption cost, without the optimizations above
  - with the optimizations, ~0.5% at the worst case, typically far less

→ a non-issue for an encrypted transport

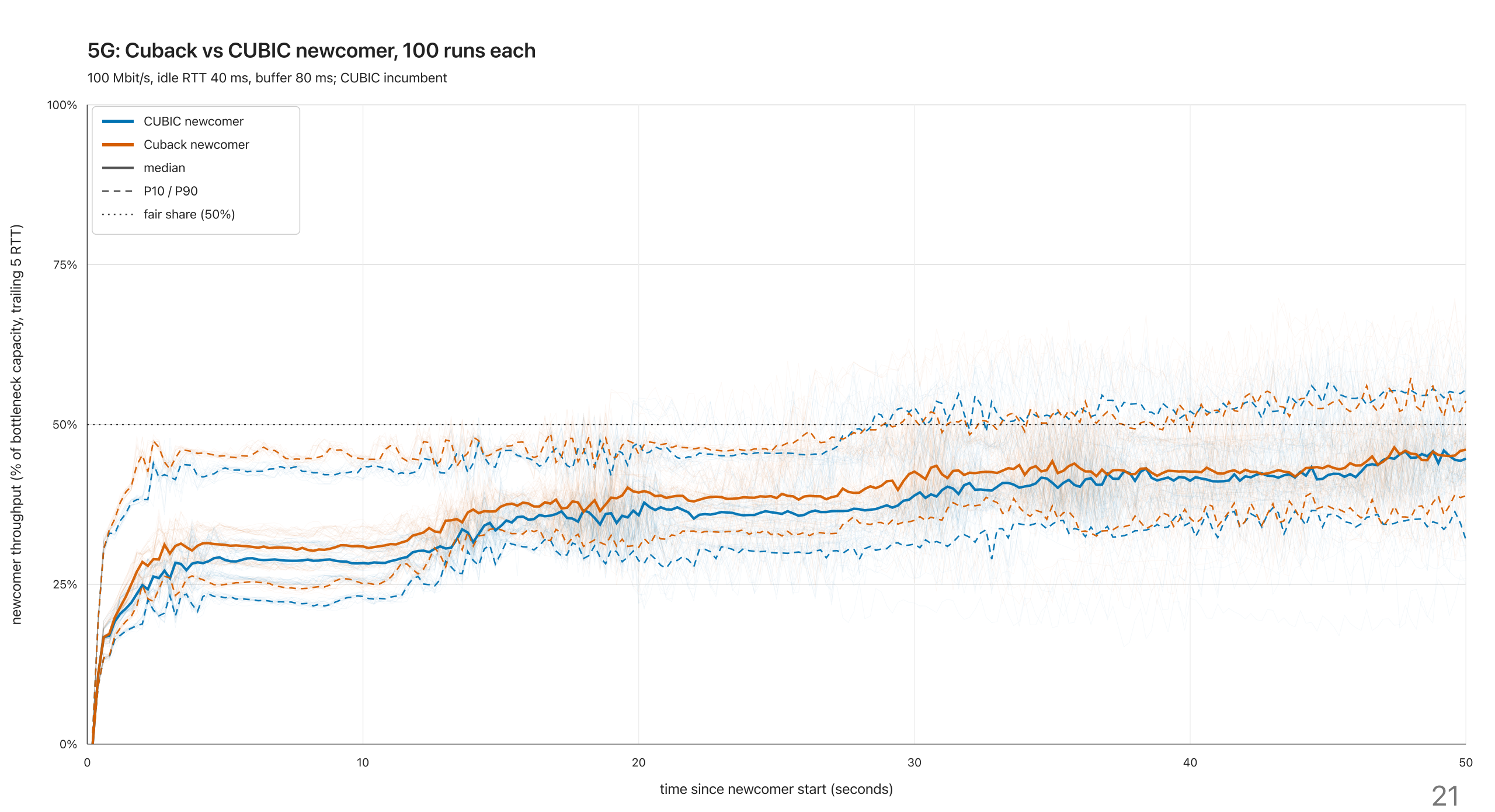
Ease of implementation

- CUBIC:
  - implement two state machines, then check their behavior against sequences of inputs: ACKs, elapsed time, pause / resume, ...
  - even with AI, verifying is hard: the correct value after each arbitrary sequence of inputs must be derived, while what is correct differs due to implementation details
- Cuback:
  - a pure function *D(w)* answers when cwnd is increased
  - test vectors provided for checking correctness

Simulation: setup

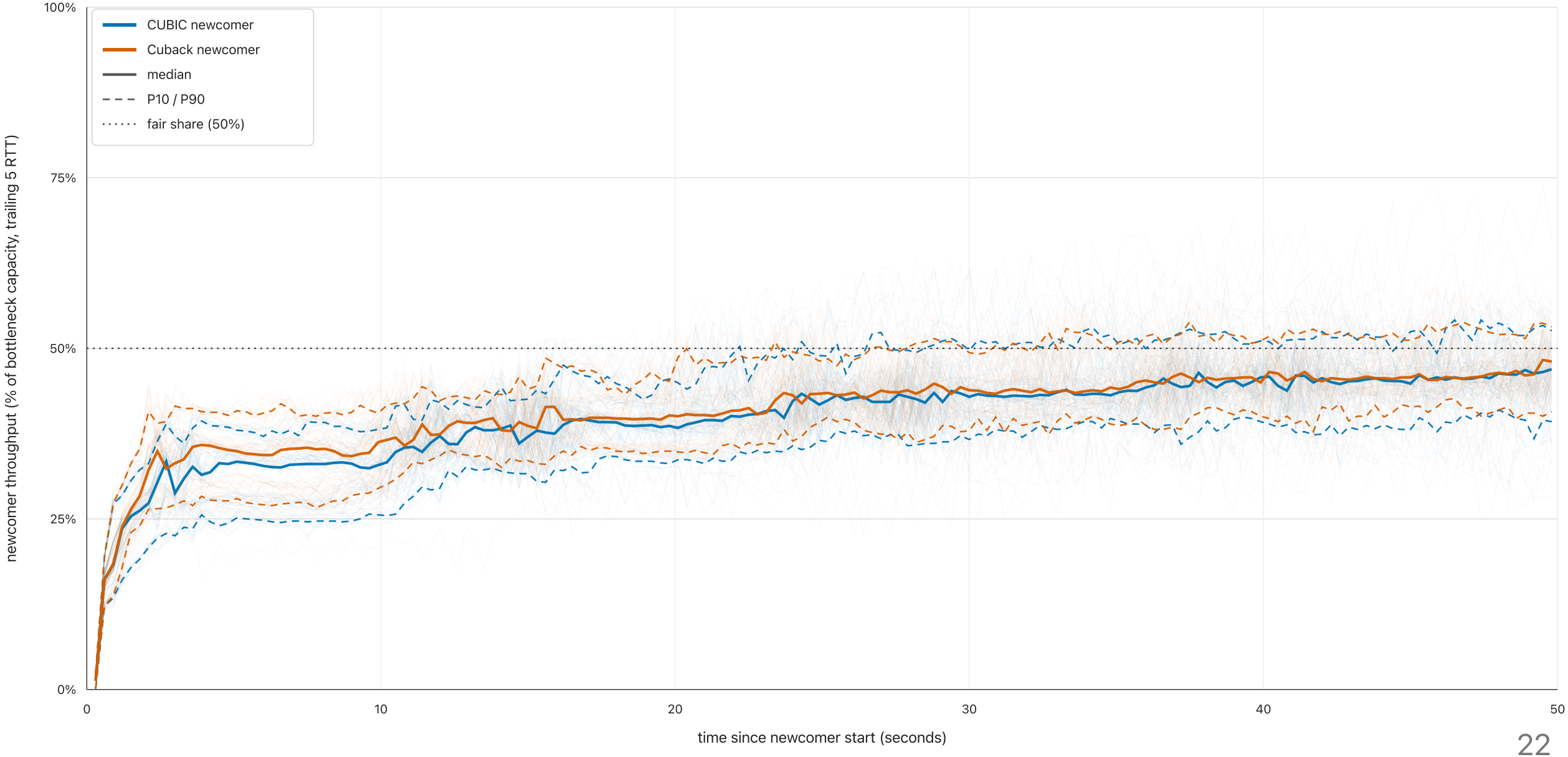
- CUBIC incumbent; newcomer (CUBIC or Cuback) joins at 20–40 s; 100 runs each

Profile	Bandwidth	Idle RTT	Buffer	BDP at full buffer
5G	100 Mbit/s	40 ms	80 ms	1.5 MB
LTE	30 Mbit/s	60 ms	120 ms	675 KB
DSL	30 Mbit/s	30 ms	50 ms	300 KB
Half DSL	15 Mbit/s	30 ms	50 ms	150 KB
Quarter DSL	7.5 Mbit/s	30 ms	50 ms	75 KB
Eighth DSL	3.75 Mbit/s	30 ms	50 ms	38 KB


5G: Cuback vs CUBIC newcomer, 100 runs each
100 Mbit/s, idle RTT 40 ms, buffer 80 ms; CUBIC incumbent
CUBIC newcomer
Cuback newcomer
median
P10 / P90
fair share (50%)
100%
75%
50%
25%
0%
newcomer throughput (% of bottleneck capacity, trailing 5 RTT)
0
10
20
30
40
50
time since newcomer start (seconds)

LTE: Cuback vs CUBIC newcomer, 100 runs each

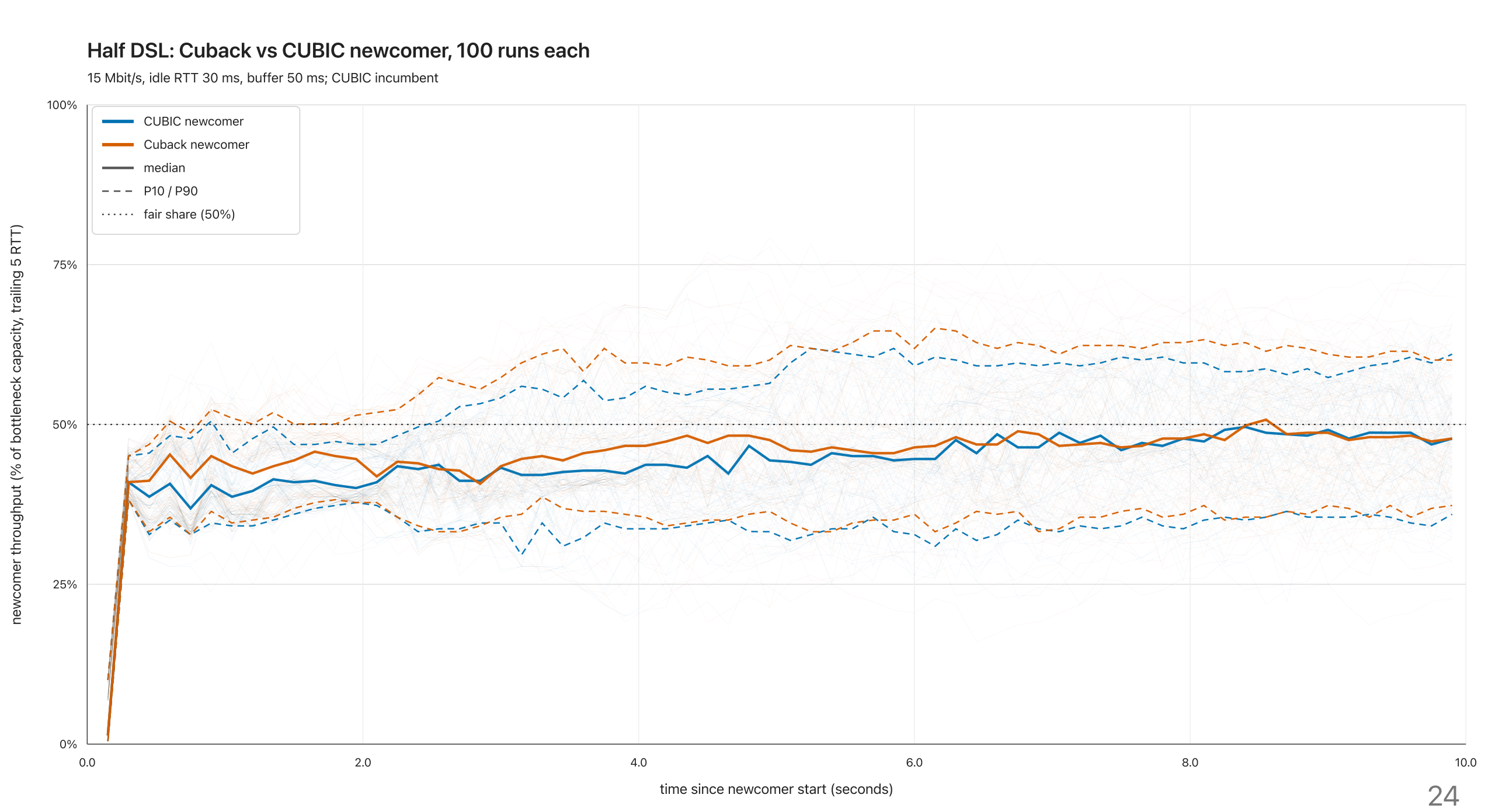
30 Mbit/s, idle RTT 60 ms, buffer 120 ms; CUBIC incumbent

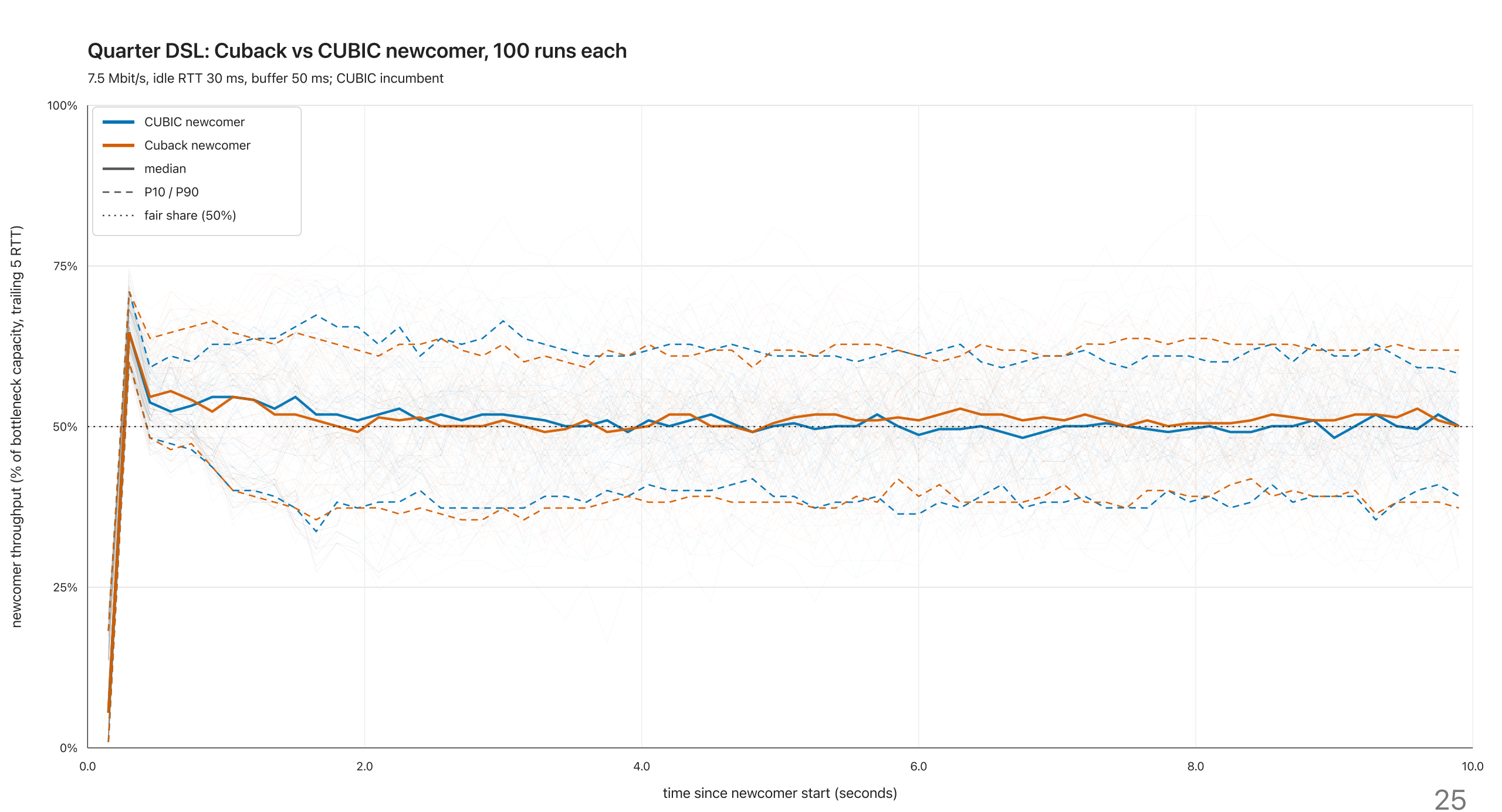


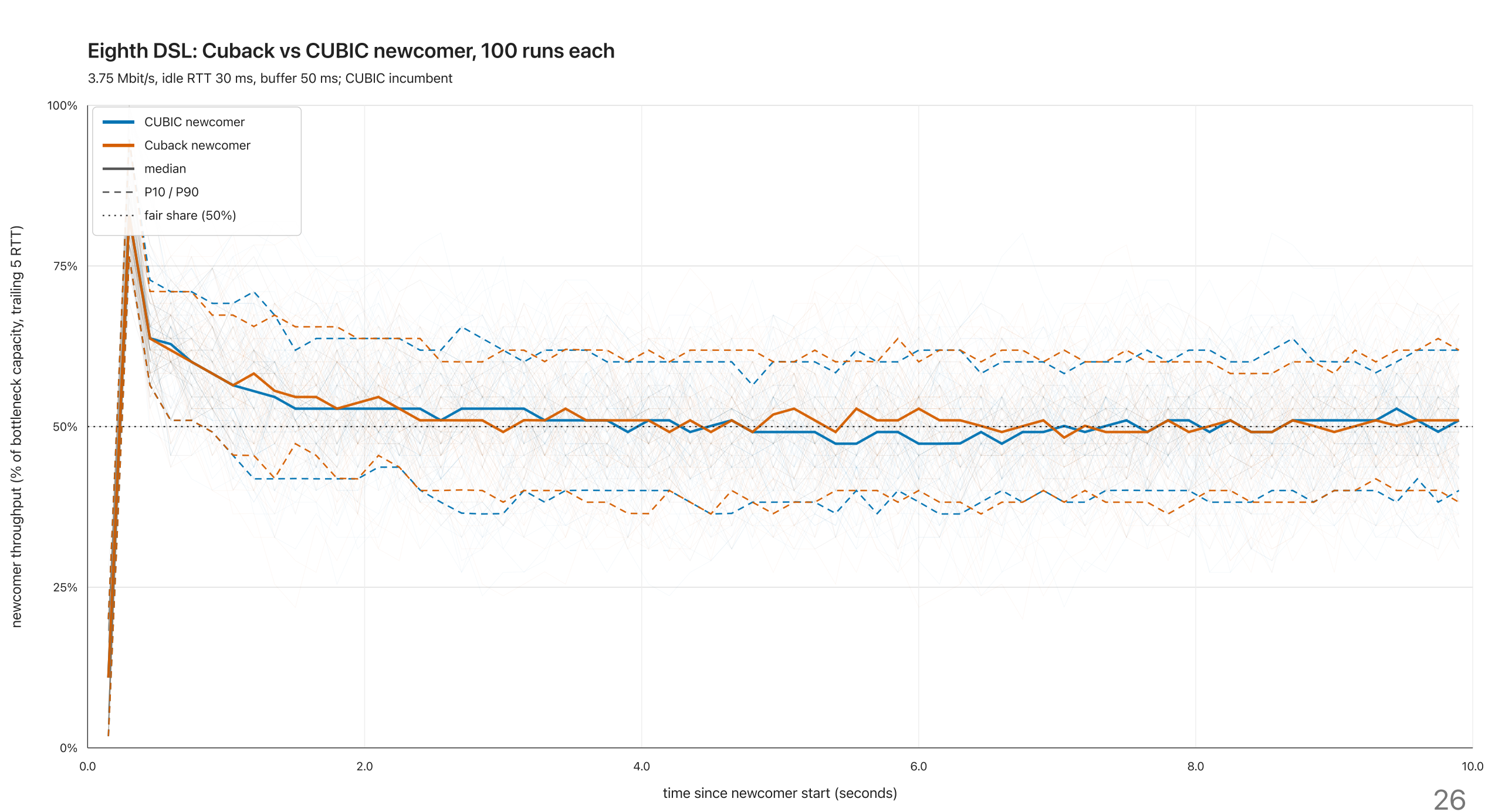
DSL: Cuback vs CUBIC newcomer, 100 runs each

30 Mbit/s, idle RTT 30 ms, buffer 50 ms; CUBIC incumbent




Half DSL: Cuback vs CUBIC newcomer, 100 runs each
15 Mbit/s, idle RTT 30 ms, buffer 50 ms; CUBIC incumbent
CUBIC newcomer
Cuback newcomer
median
P10 / P90
fair share (50%)
100%
75%
50%
25%
0%
newcomer throughput (% of bottleneck capacity, trailing 5 RTT)
0.0
2.0
4.0
6.0
8.0
10.0
time since newcomer start (seconds)


Quarter DSL: Cuback vs CUBIC newcomer, 100 runs each
7.5 Mbit/s, idle RTT 30 ms, buffer 50 ms; CUBIC incumbent
CUBIC newcomer
Cuback newcomer
median
P10 / P90
fair share (50%)
100%
75%
50%
25%
0%
newcomer throughput (% of bottleneck capacity, trailing 5 RTT)
0.0
2.0
4.0
6.0
8.0
10.0
time since newcomer start (seconds)

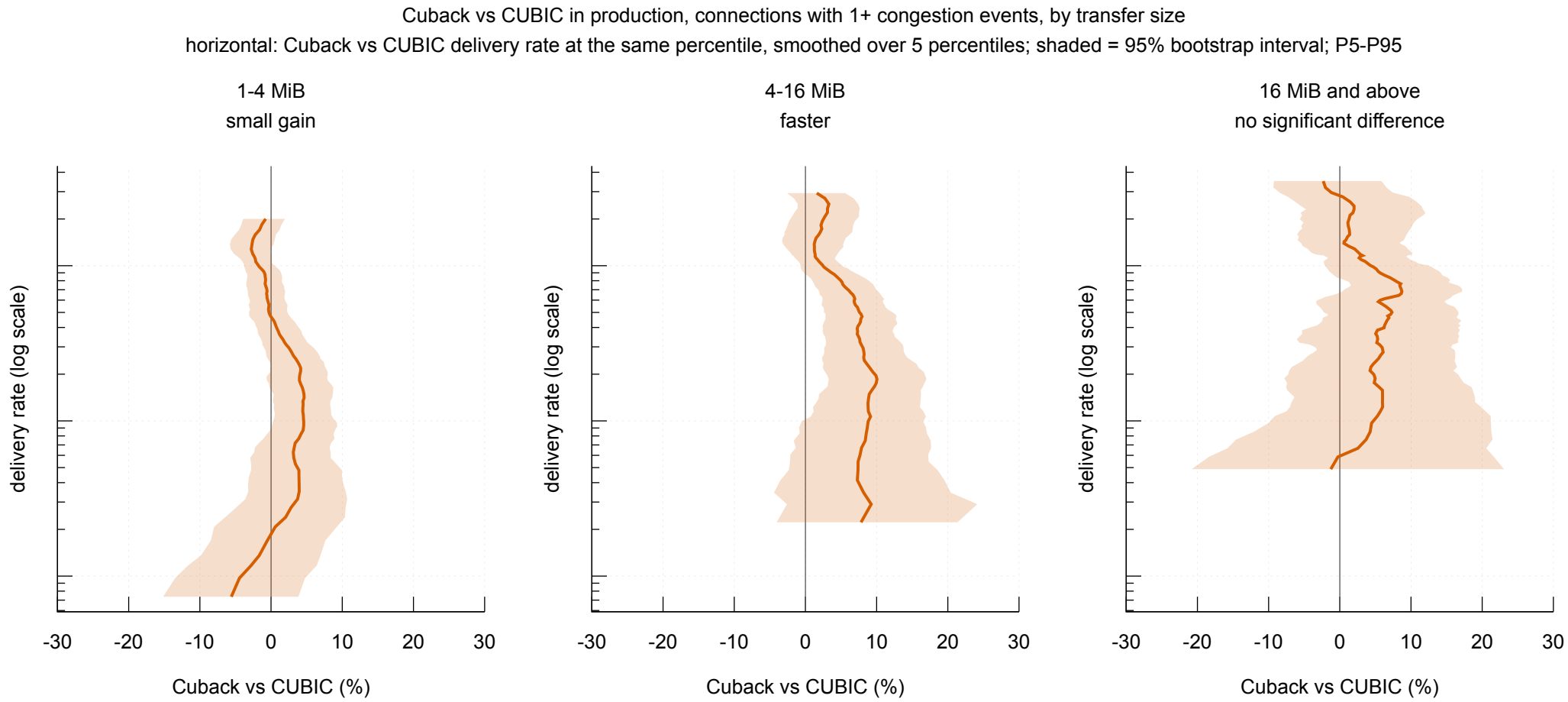

Eighth DSL: Cuback vs CUBIC newcomer, 100 runs each
3.75 Mbit/s, idle RTT 30 ms, buffer 50 ms; CUBIC incumbent
CUBIC newcomer
Cuback newcomer
median
P10 / P90
fair share (50%)
100%
75%
50%
25%
0%
newcomer throughput (% of bottleneck capacity, trailing 5 RTT)
0.0
2.0
4.0
6.0
8.0
10.0
time since newcomer start (seconds)

Simulation: results

- when cwnd is large (5G, LTE, DSL, Half DSL), a Cuback newcomer gains bandwidth faster than CUBIC
    - the gap closes as both approach the fair share
- when cwnd is small (Quarter, Eighth DSL), they behave identically (both in the Reno-friendly region)

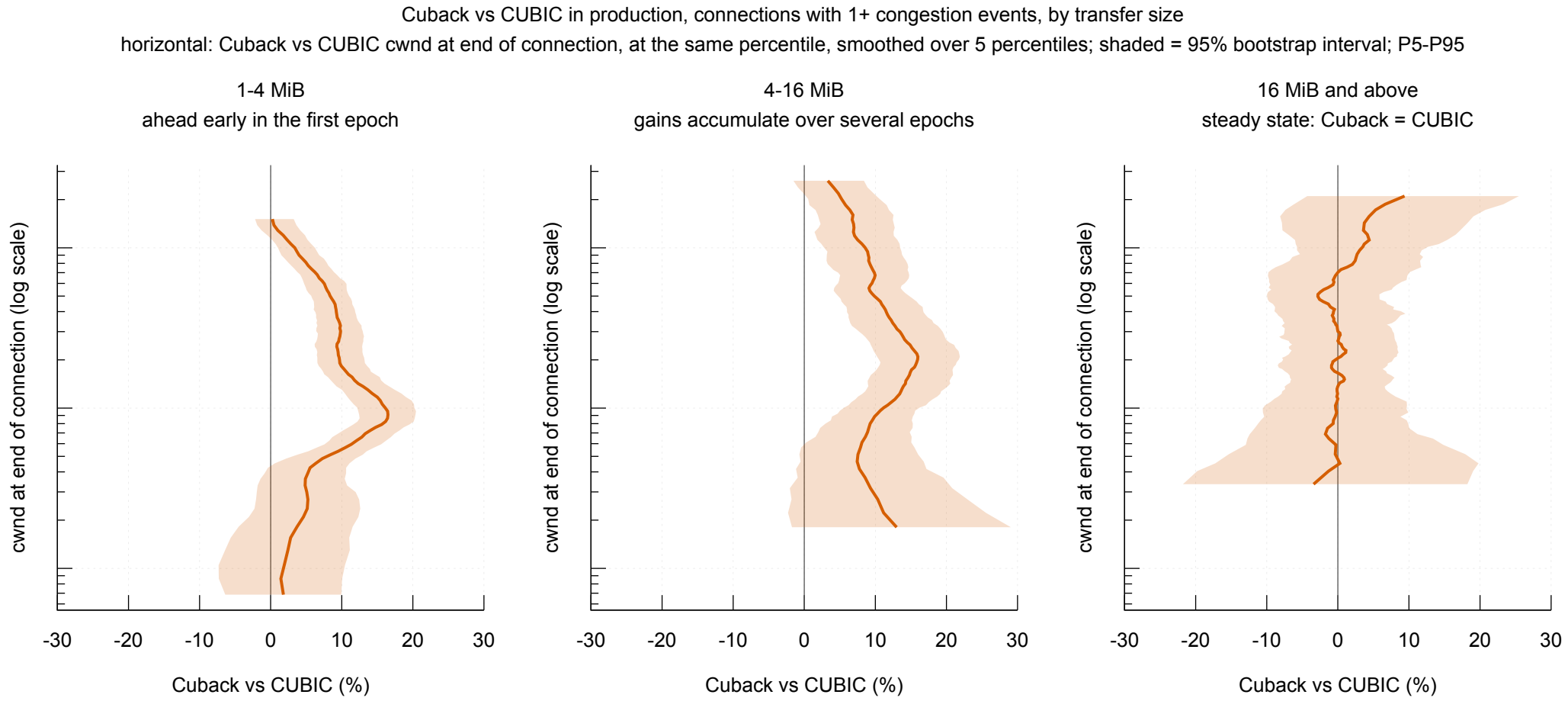
Production results

- A/B test on production traffic, 24 h; connections that sent > 1 MiB and were never flow-control blocked
- delivery rate: ACK rate over the last ~0.5 s during which the sender was cwnd-limited



Production results: why

Same connections, compared on cwnd at end of connection



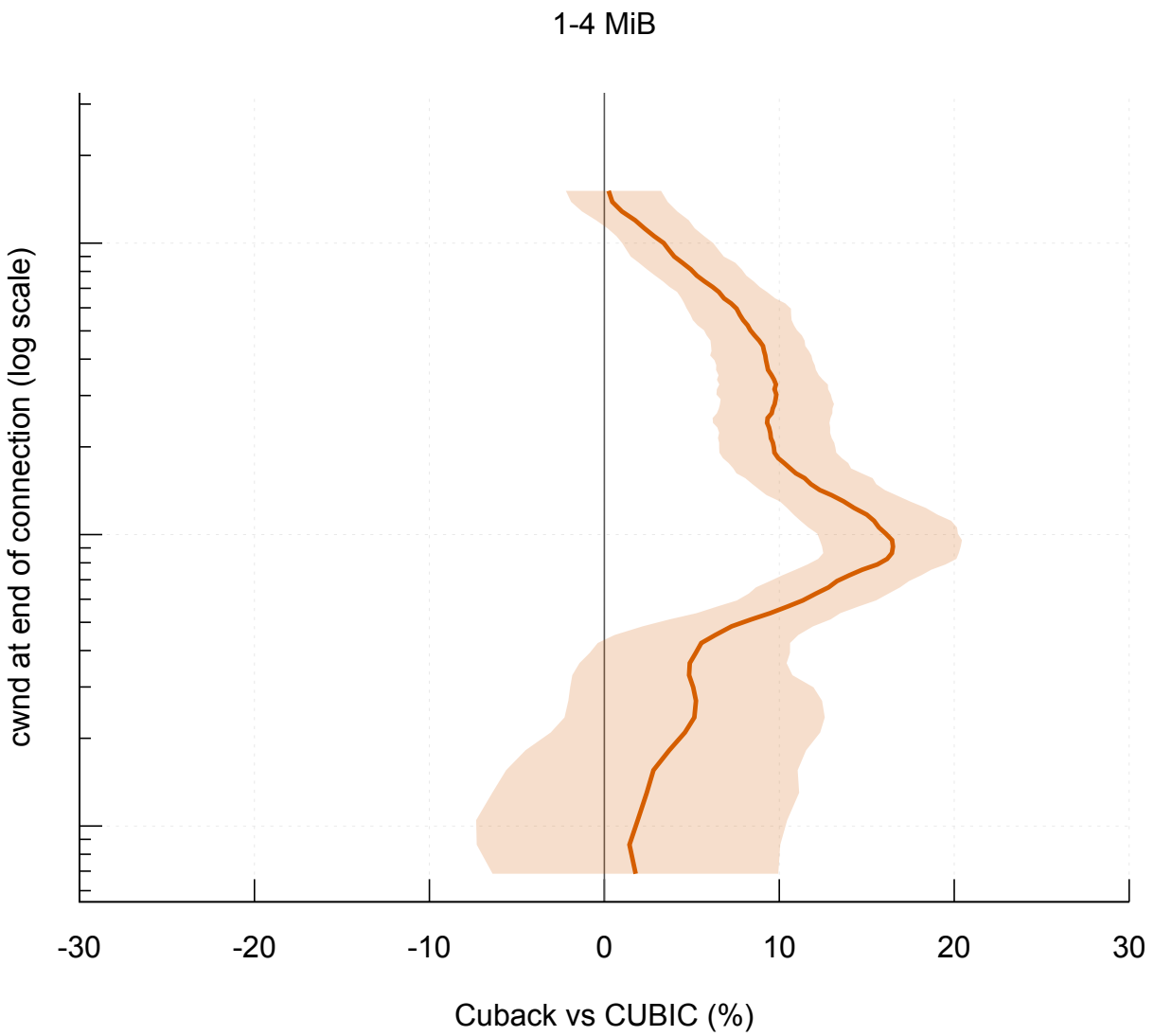
Conclusion: CUBIC vs. Cuback

- Multi-input state machines → a purely ACK-driven machine over values fixed for each congestion period
- Medium-sized transfers complete faster (HTTP TTLB; clearest at 4–16 MiB); no significant difference for long transfers

	CUBIC	**Cuback**
inputs	ACKs, wall clock, app-limited pause / resume	**ACKs only**
machines	two: cubic, Reno-friendly	**one**: Reno's loop, pure *D(cwnd)*
mutable state	cwnd, *W_est*, epoch clock (shifted on pause / resume)	**cwnd, pending** (as Reno)
per-period immutables	*W_max*, *cwnd_epoch*, *K*	*W_max*, *cwnd_epoch*, *bandwidth*, (*K*)
smoothing	per-ACK move toward *W_cubic(t + RTT)*	**none**
verification	drive sequences of transitions	**test vectors**

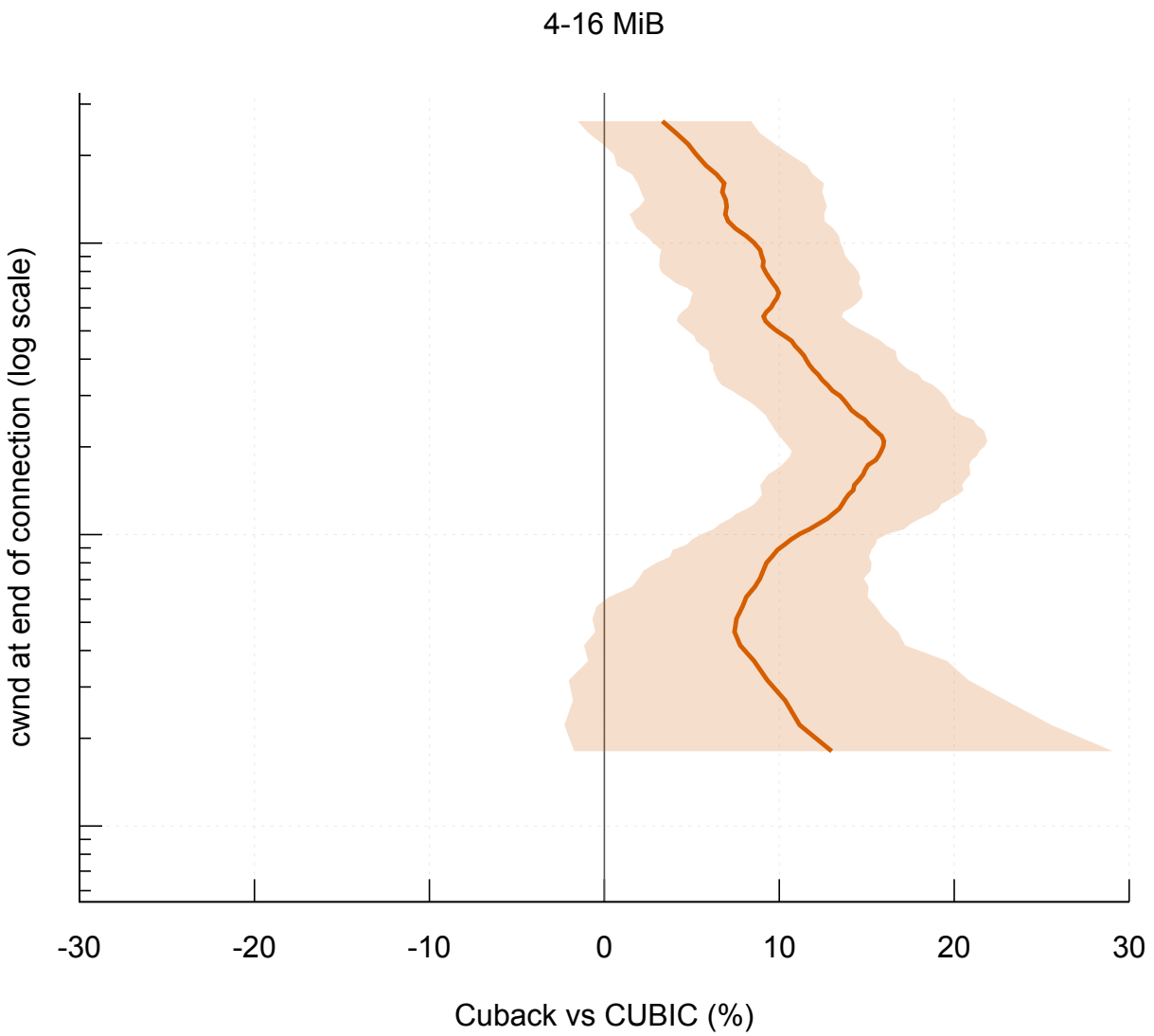
Next steps

- why spend effort on a variant of CUBIC?
  - CUBIC is the most widely used CC, yet
    - it is not easy to implement
    - it is not easy to verify that an implementation is correct
  - Cuback removes both difficulties, helping those building new transport stacks
  - and improves medium-sized object transfer time
- ideas and feedback welcome



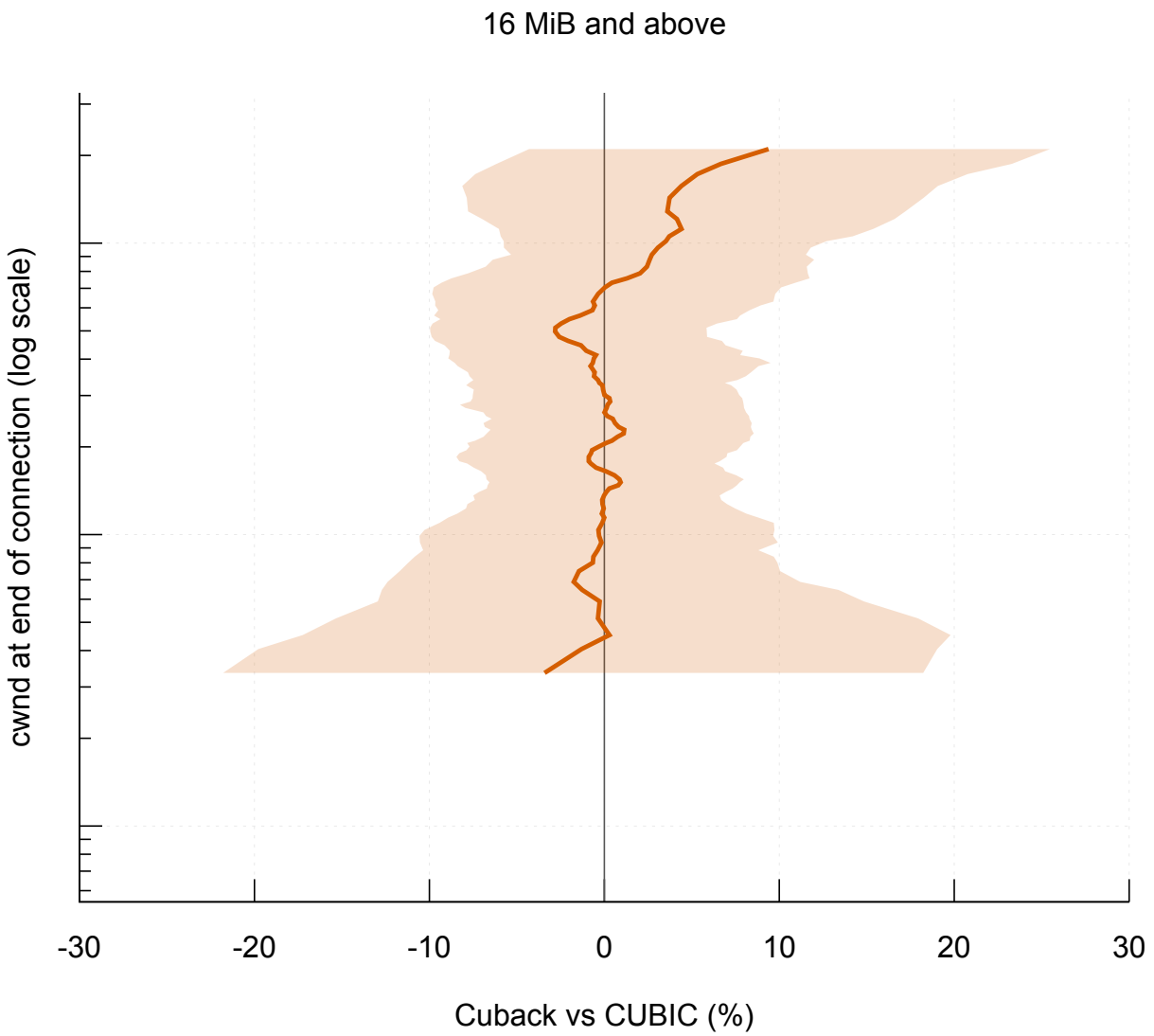
Backup: 1–4 MiB (likely explanation)

- median ~13–19 RTTs: transfers end early in the first epoch after slow-start exit
- moderate cwnd: the switch to the Reno-friendly region happens early in the epoch
  - Cuback's Reno-friendly estimate, counted in acked data, is ahead → higher cwnd
- smallest cwnd: both are Reno-friendly from the start → identical
- largest cwnd: the cubic curve governs; transfers end before the two diverge



Backup: 4–16 MiB (likely explanation)

- median ~34–62 RTTs: long enough to cover much of an epoch, often several
  - the per-epoch advantage repeats
- larger cwnd: the ACK clock runs fast while the flow gains share from other traffic



Backup: 16 MiB and above (likely explanation)

- median ~157 RTTs: mostly steady state
  - achieved rate ≈ recorded rate → Cuback = CUBIC by construction
- W_max is set by the path; both reach the same W_max each epoch
- 95% interval ≈ ±7–10%: no measurable difference