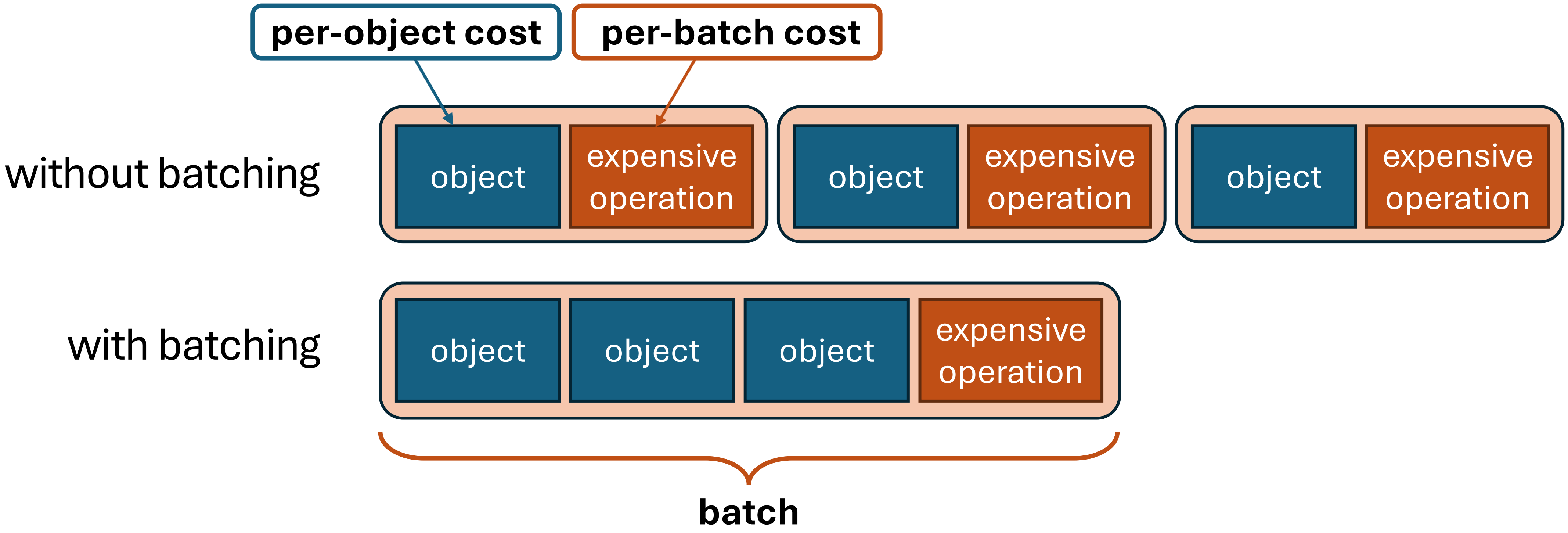
Batching with End-to-End Performance Estimation

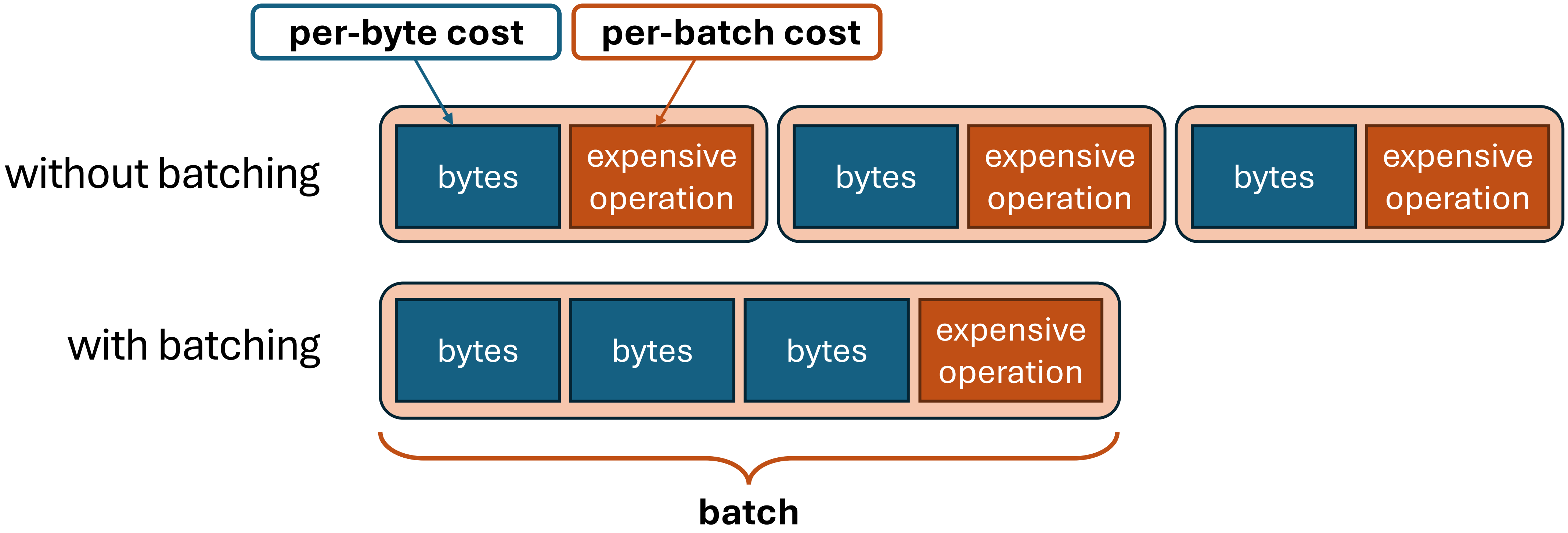
Avidan Borisov[†], Nadav Amit[†], Dan Tsafrir[†]

[†]Technion

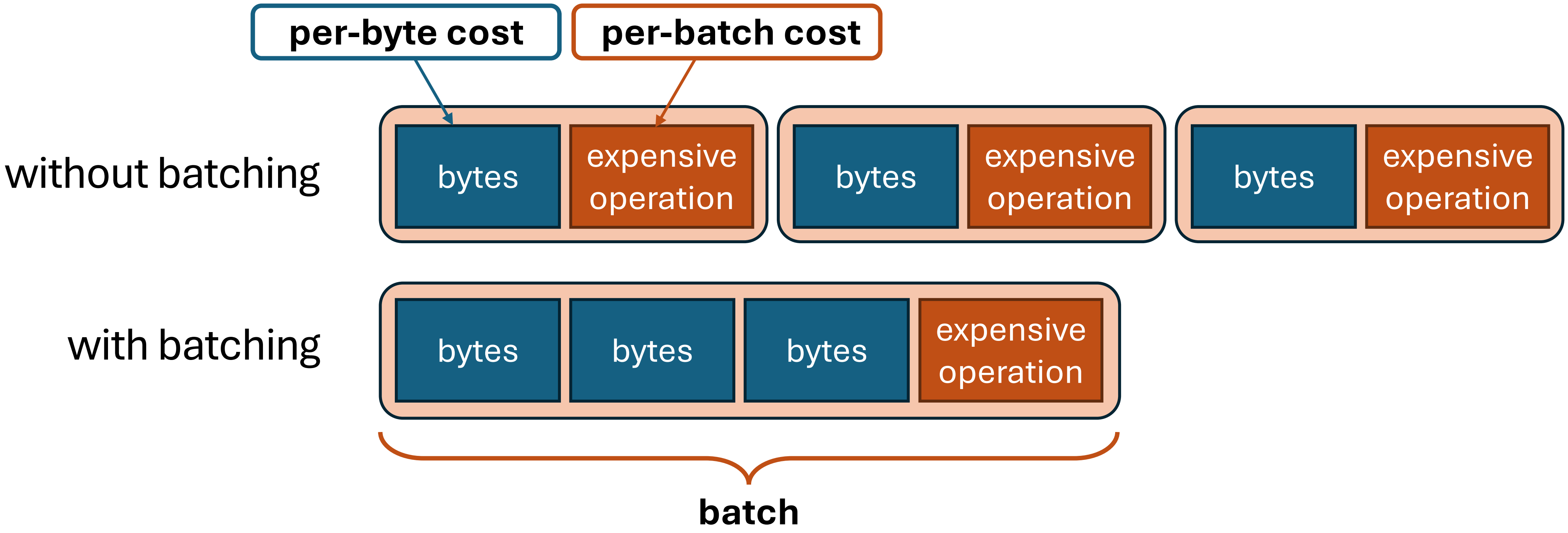
Background: batching



Background: batching in networking



Background: batching in networking



- “Batching trades-off latency for throughput”

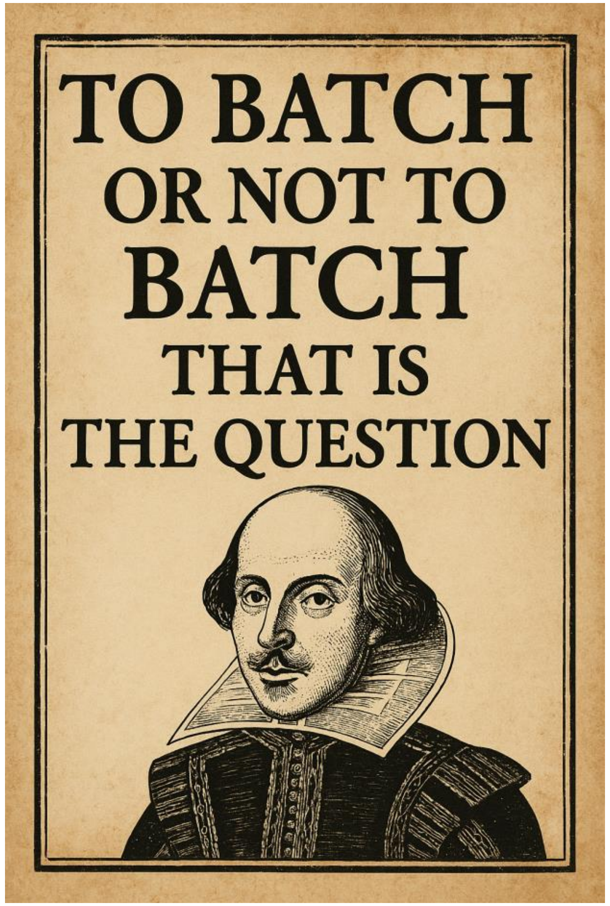
Seemingly “ideal” batching

- Assume batching never waits for more objects
  - Batching is only done in presence of congestion
  - IX [OSDI’14] calls this “adaptive batching”

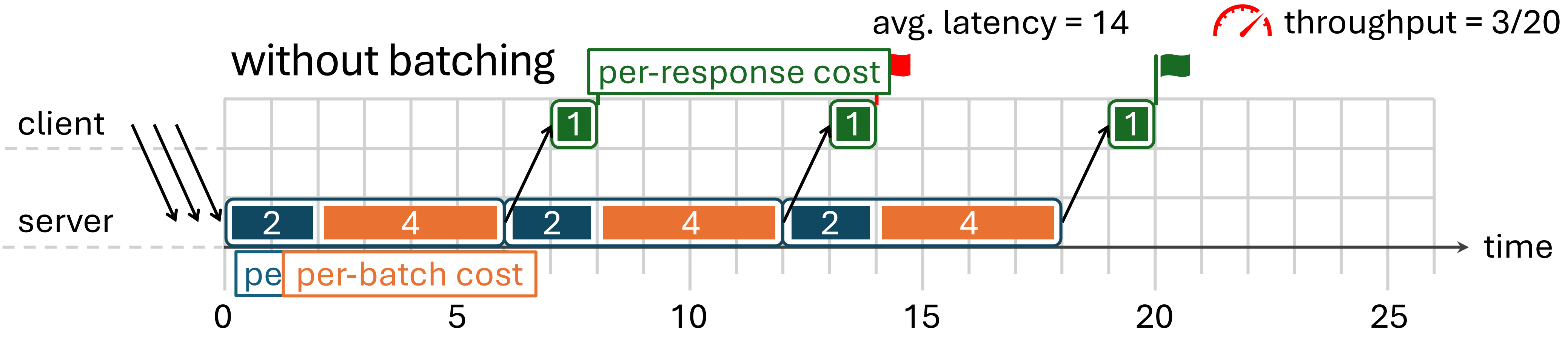
Seemingly “ideal” batching

- Assume batching never waits for more objects
  - Batching is only done in presence of congestion
  - IX [OSDI’14] calls this “adaptive batching”

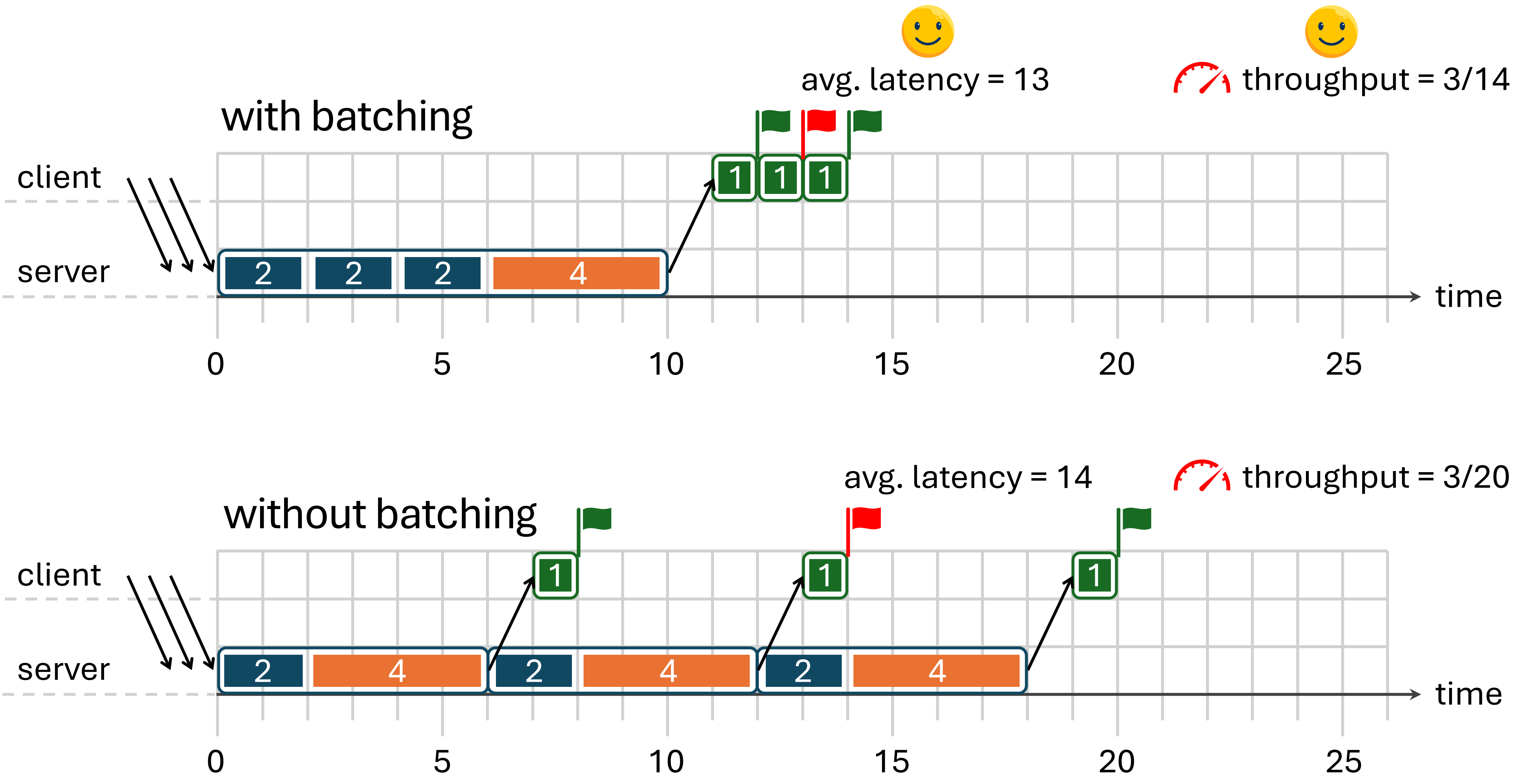
- 3 requests have arrived together to the server
  - Should the server process them as one batch?



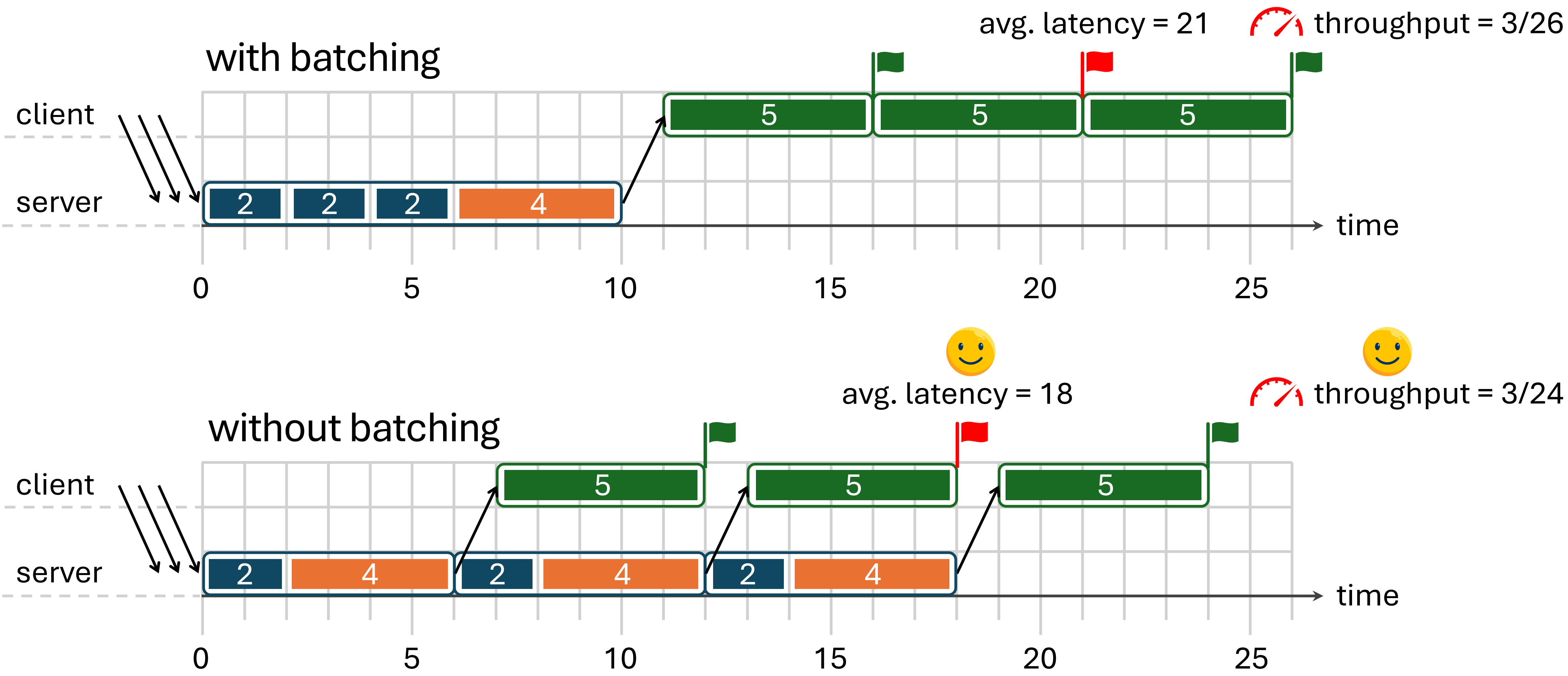
To batch or not to batch?



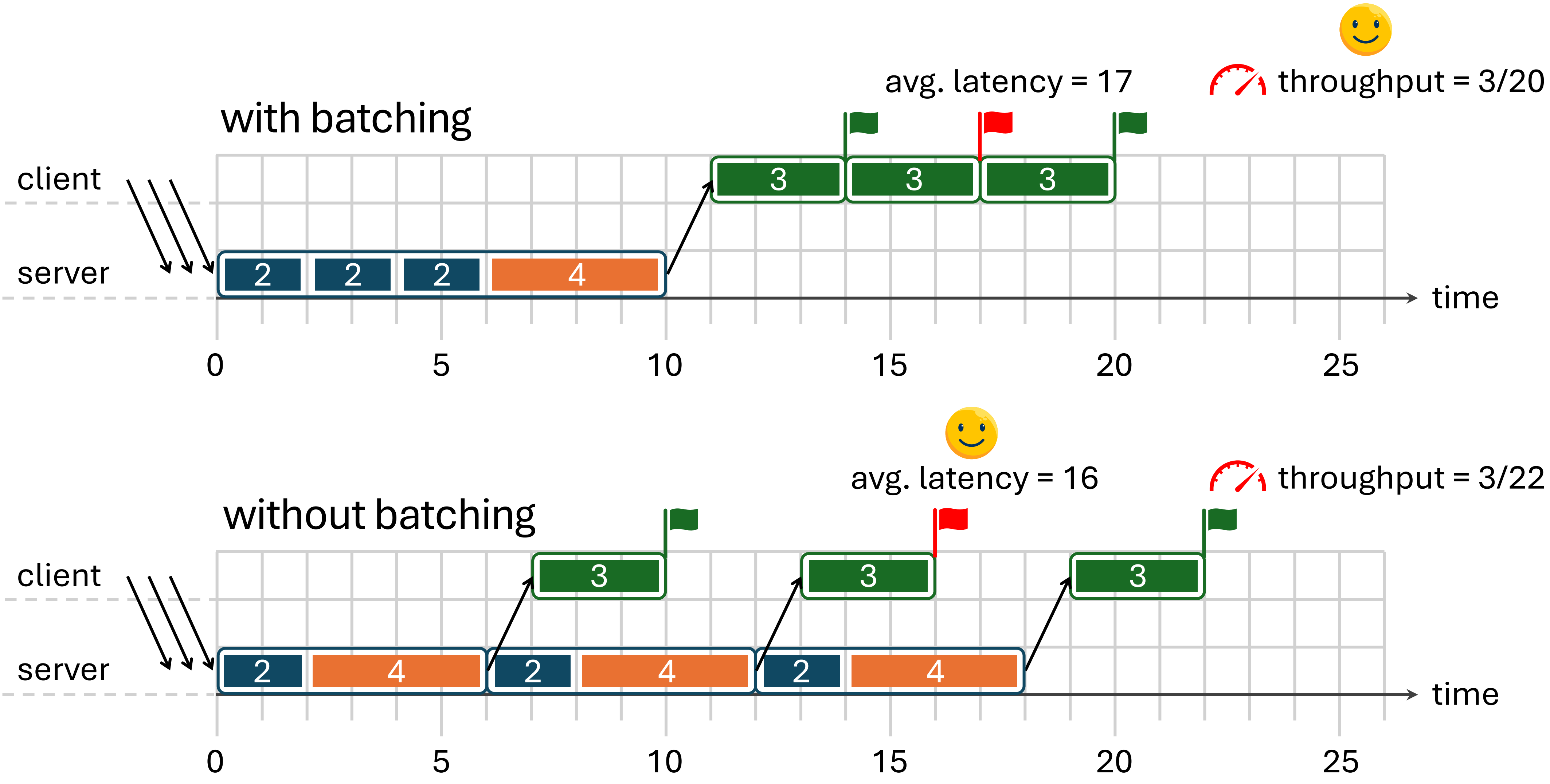
To batch or not to batch?



To batch or not to batch?



To batch or not to batch?

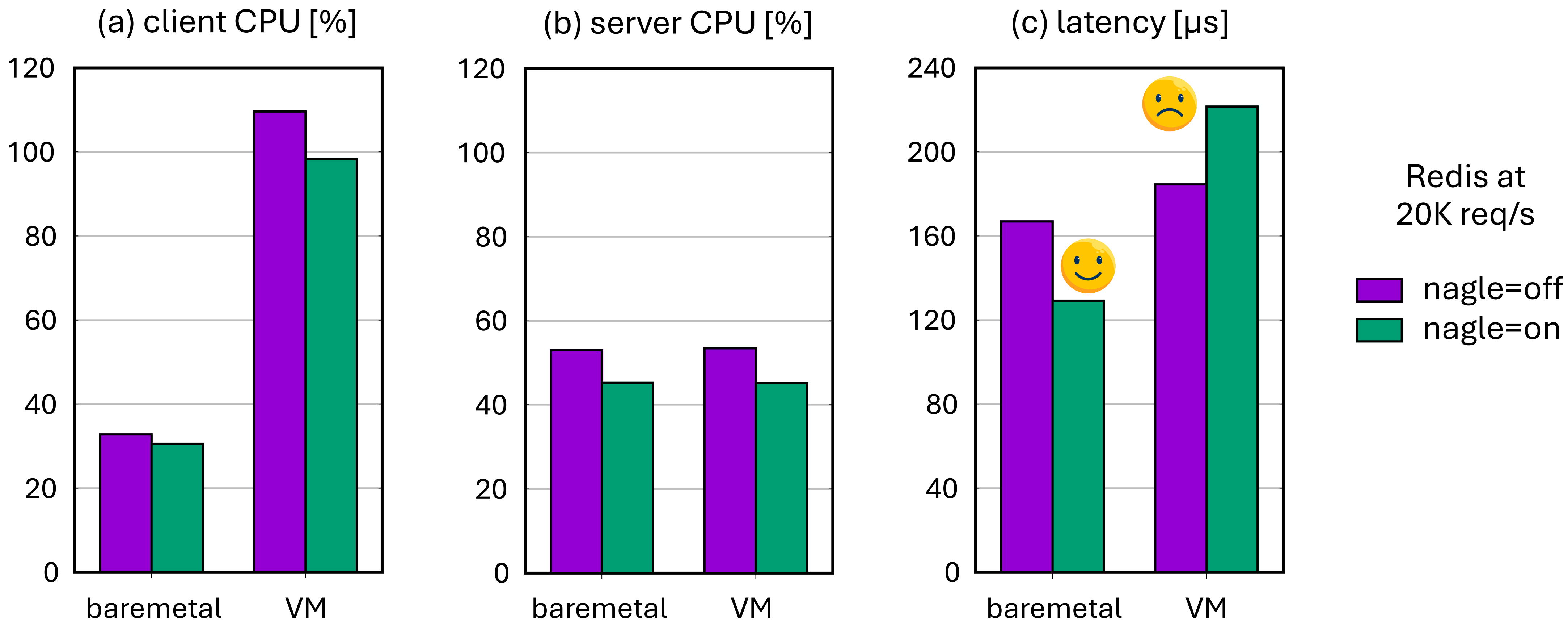


To batch or not to batch?

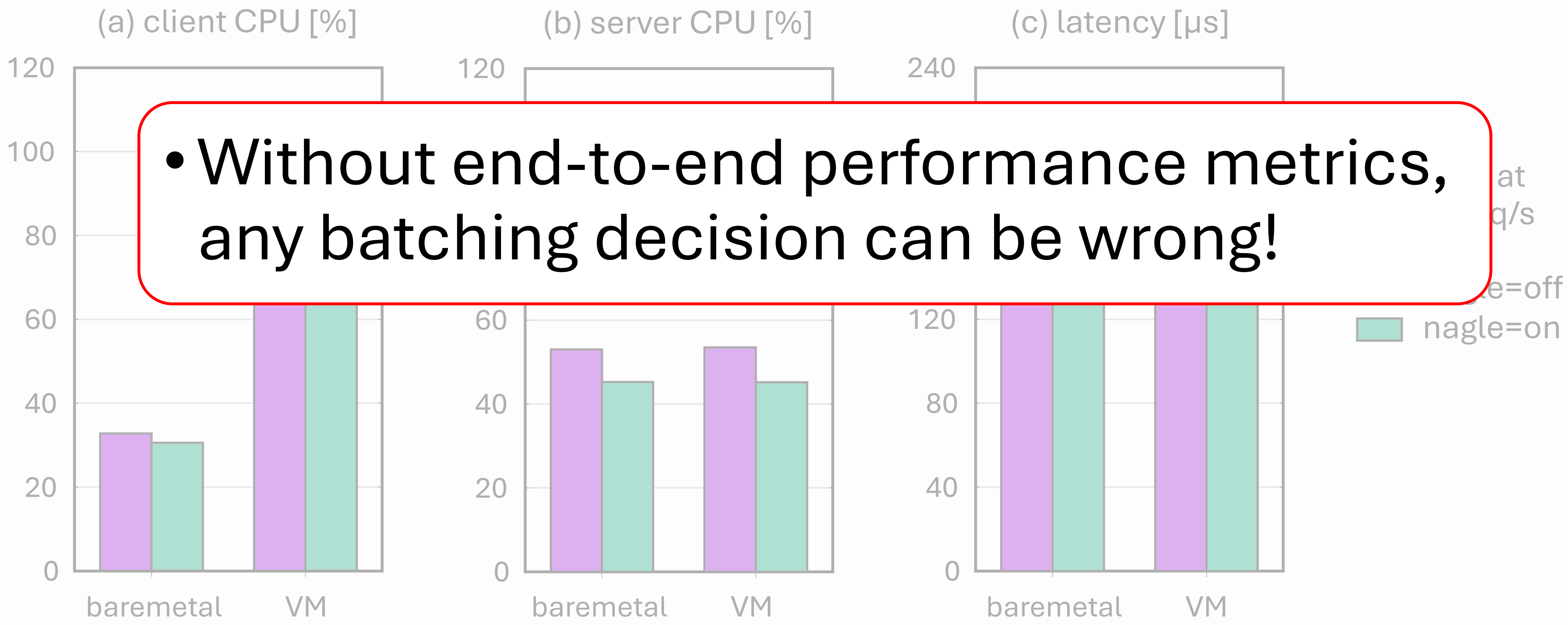
- Same server-side activity
- Completely different results

client-side cost	latency	throughput
1	better	better
5	worse	worse
3	worse	better

Not just in theory



Not just in theory



- Without end-to-end performance metrics, any batching decision can be wrong!

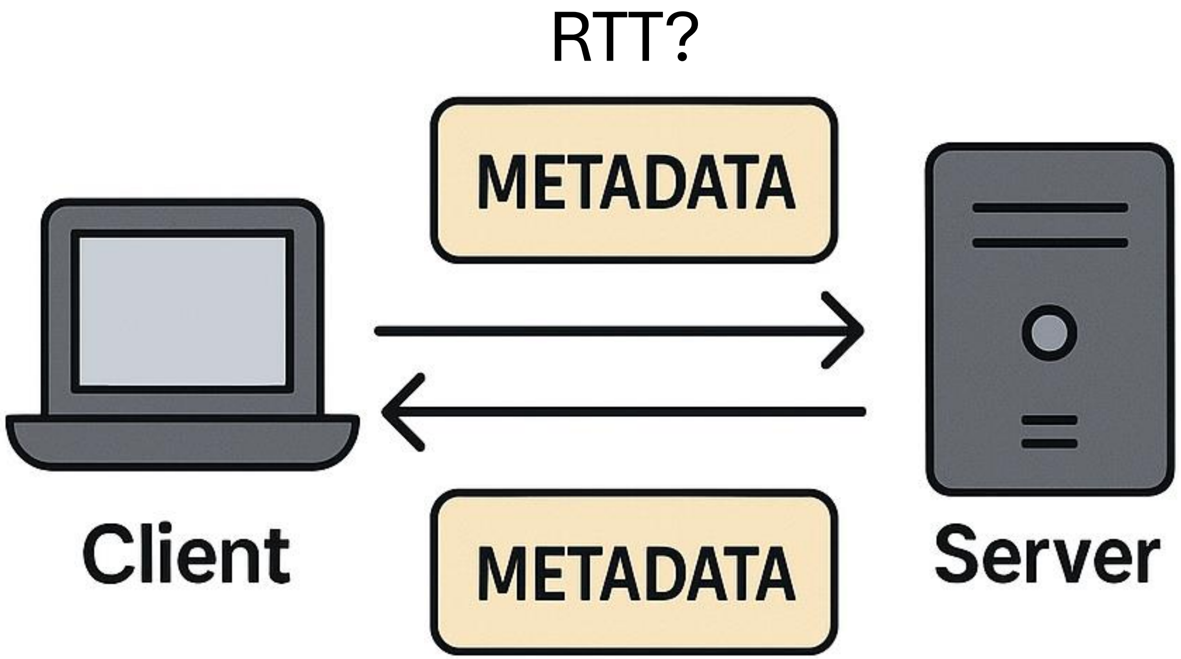
What does the literature say?

- State-of-the-practice (Linux)
  - Nagle
  - Auto-corking
  - TSO
  - xmit_more
  - …
- State-of-the-art
  - Netmap [ATC’12]
  - MegaPipe [OSDI’12]
  - IX [OSDI’14]
  - mTCP [NSDI’14]
  - Stackmap [ATC’16]
  - ZygOS [SOSP’17]
  - …
- None utilize end-to-end performance

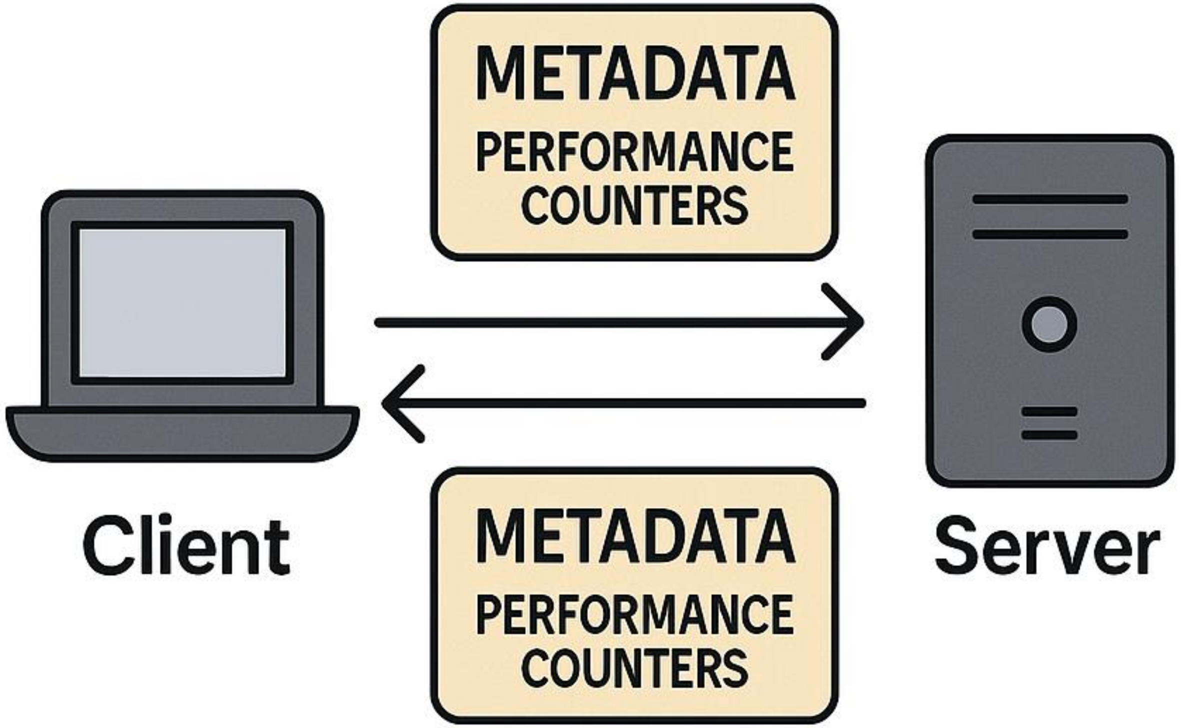
What does the literature say?

- State-of-the-practice (Linux)
  - Nagle
  - Auto-corking
  - TSO
  - xmit_more
  - …
- State-of-the-art
  - Netmap [ATC’12]
  - MegaPipe [OSDI’12]
  - IX [OSDI’14]
  - mTCP [NSDI’14]
  - Stackmap [ATC’16]
  - ZygOS [SOSP’17]
  - …
- None utilize end-to-end performance
  - Hypothesis: all may benefit

End-to-end latency is unknown



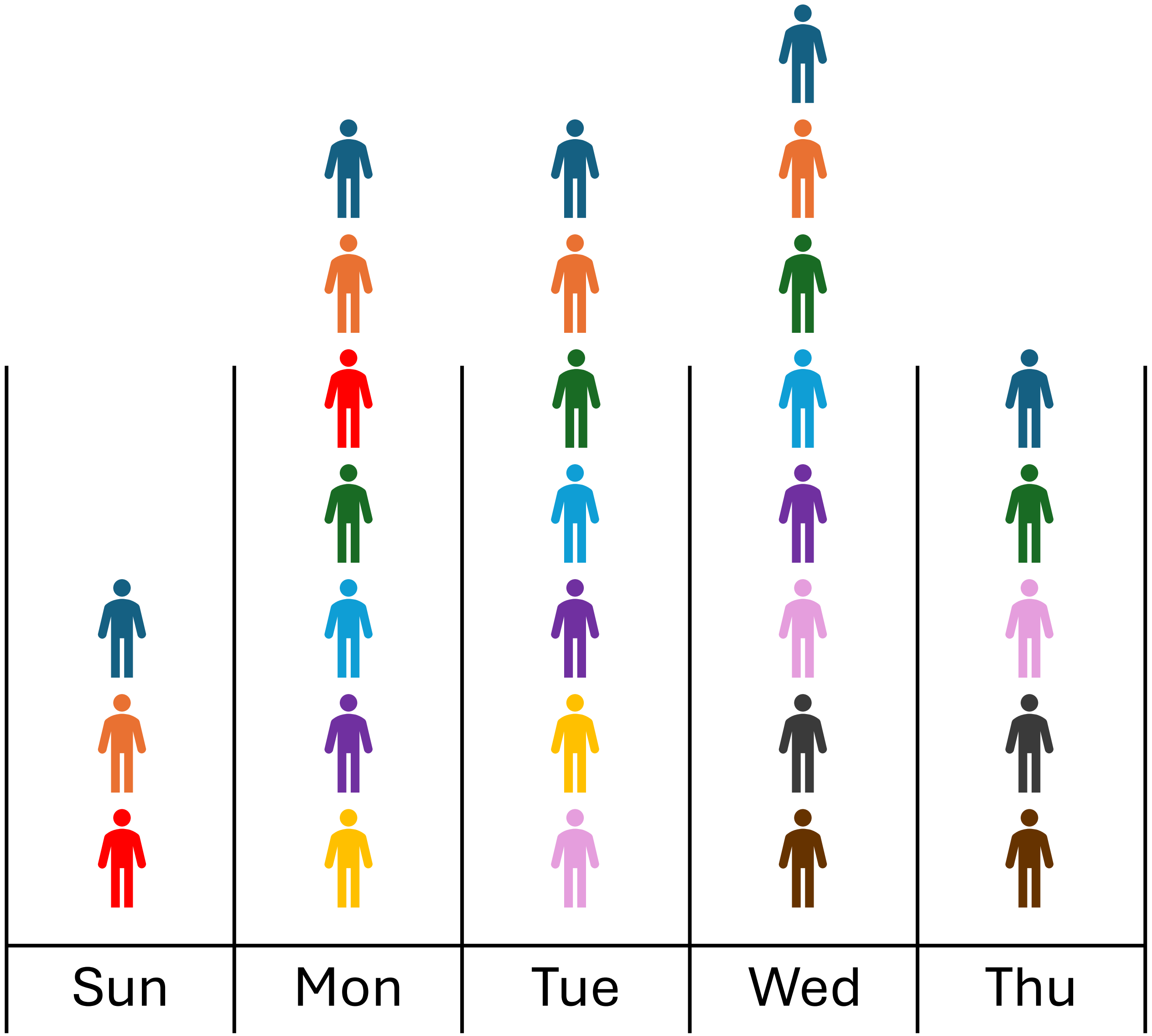
End-to-end latency is unknown



Little's law

$$Q = \lambda \times D$$

average occupancy | average rate | average delay

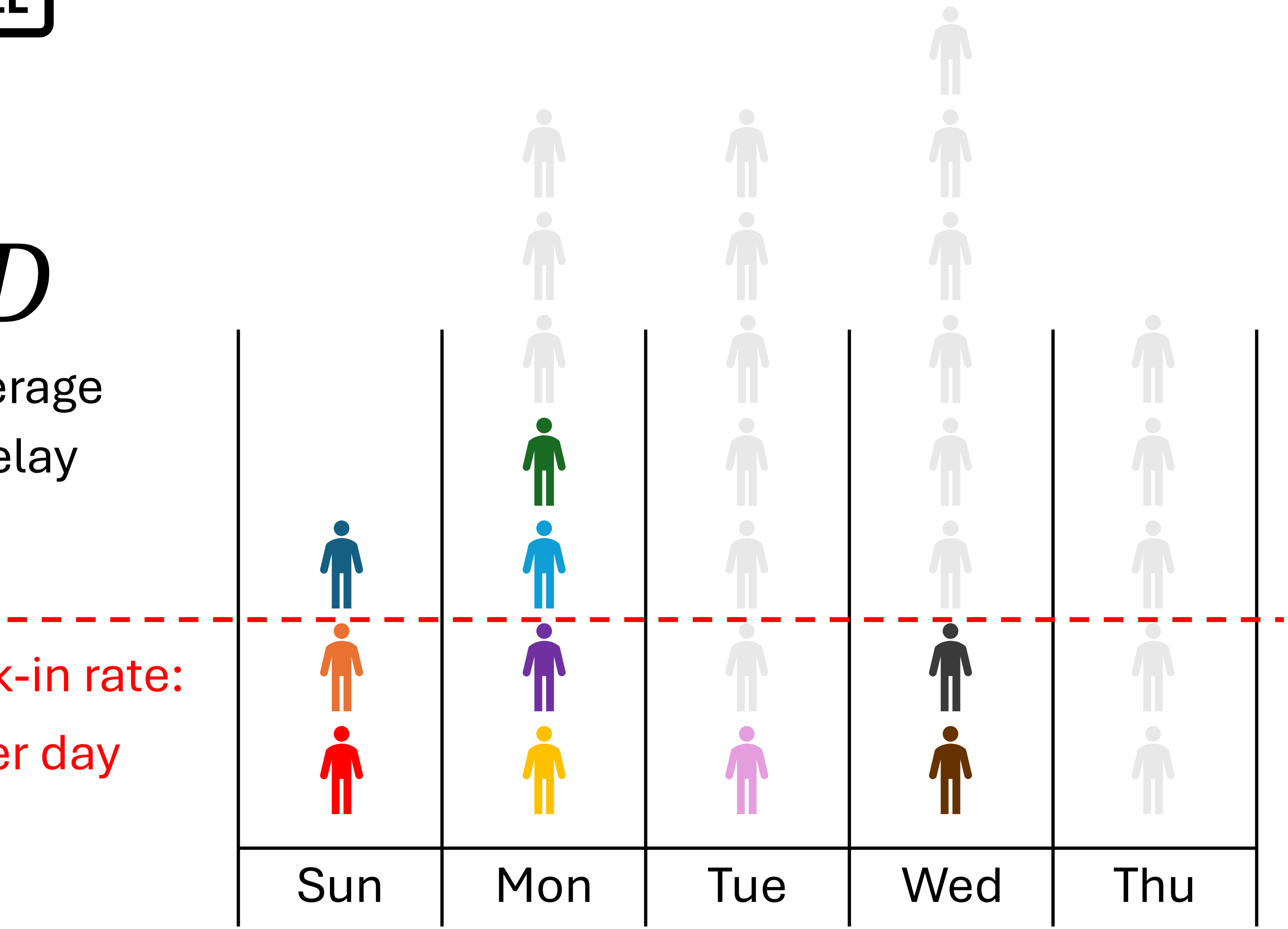


Little’s law

$$Q = \lambda \times D$$

average occupancy — average rate — average delay

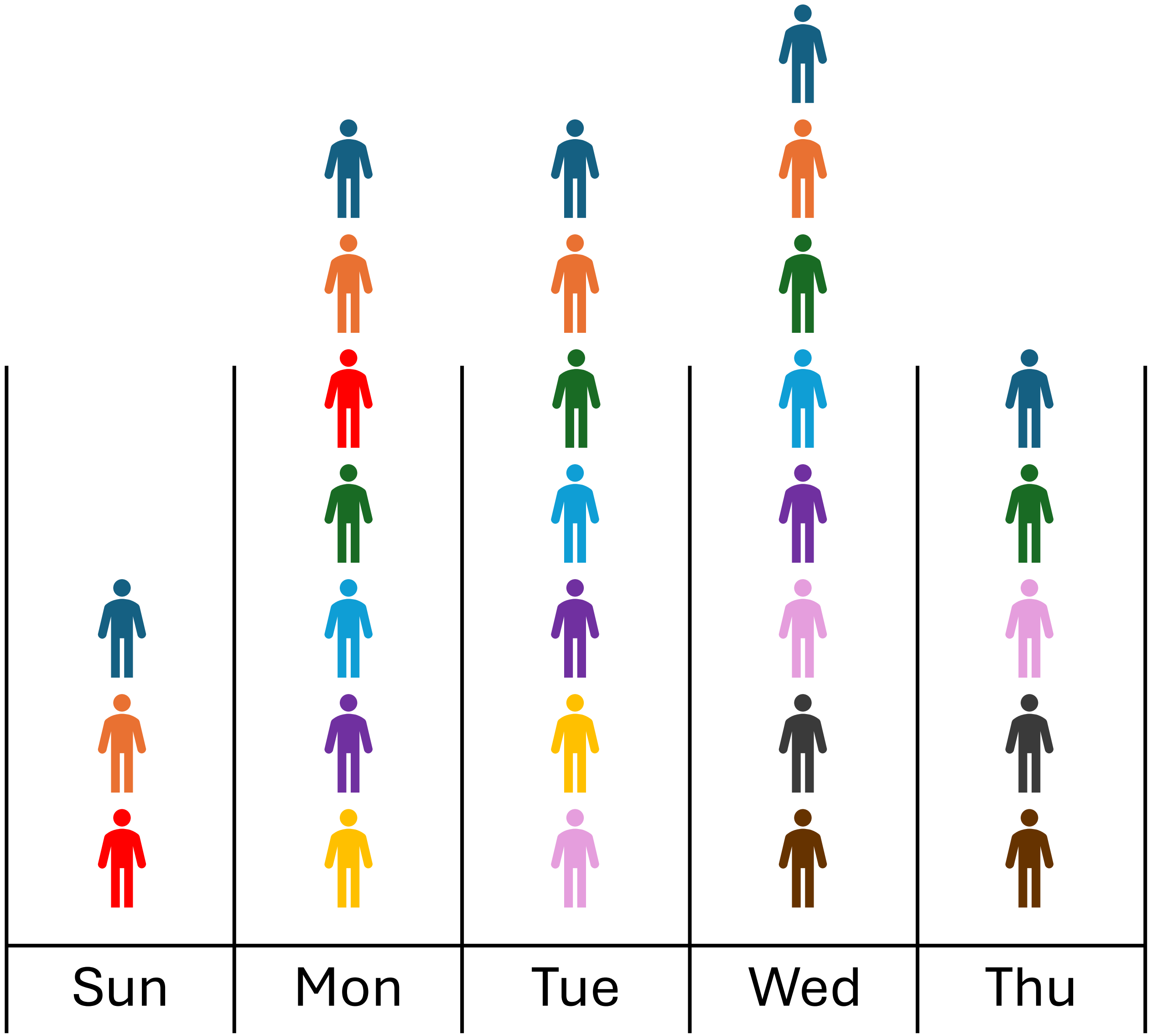
average check-in rate:
2 guests per day



Little's law

$$Q = \lambda \times D$$

average occupancy | average rate | average delay

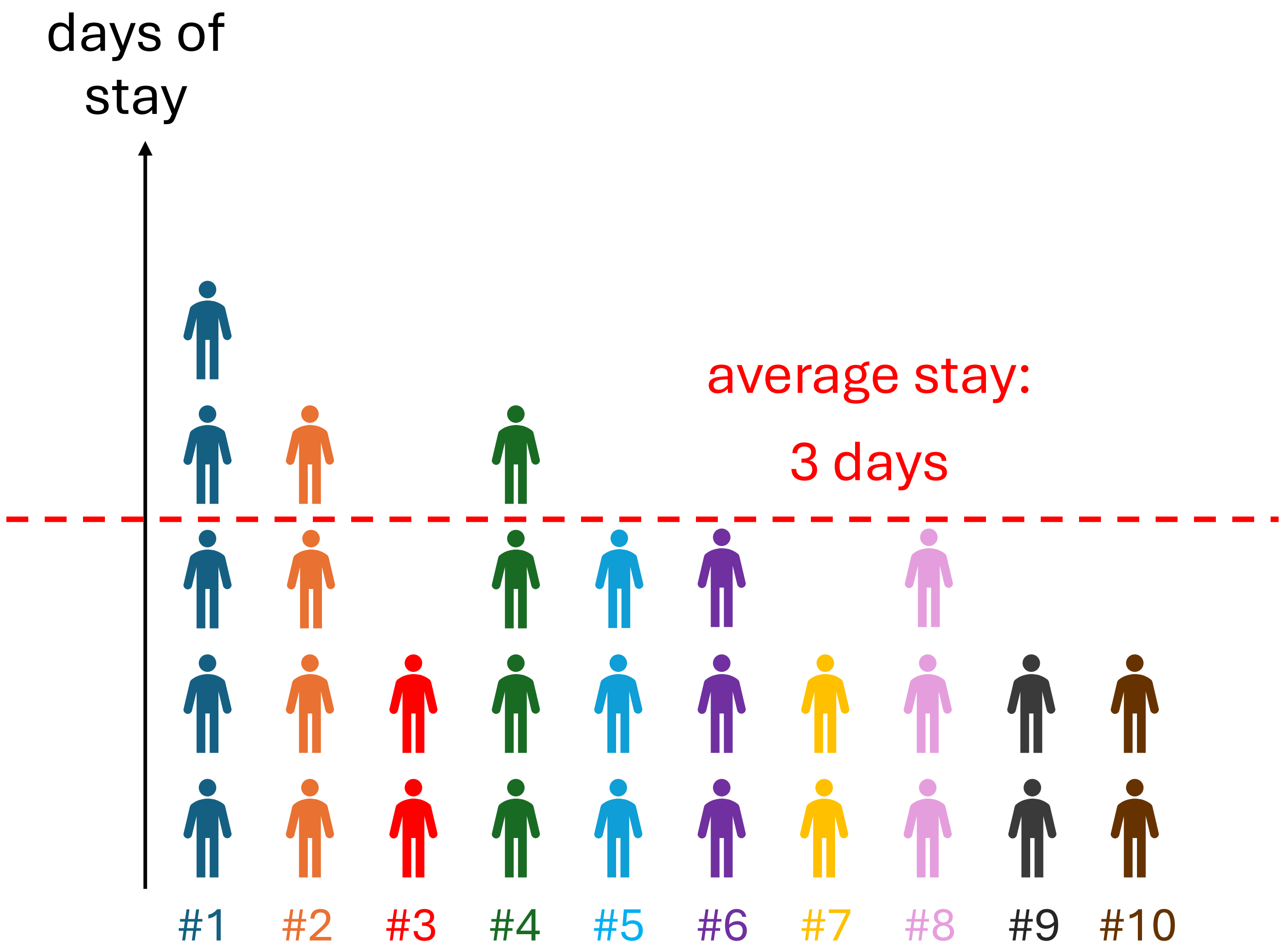


Little's law

HOTEL

$$Q = \lambda \times D$$

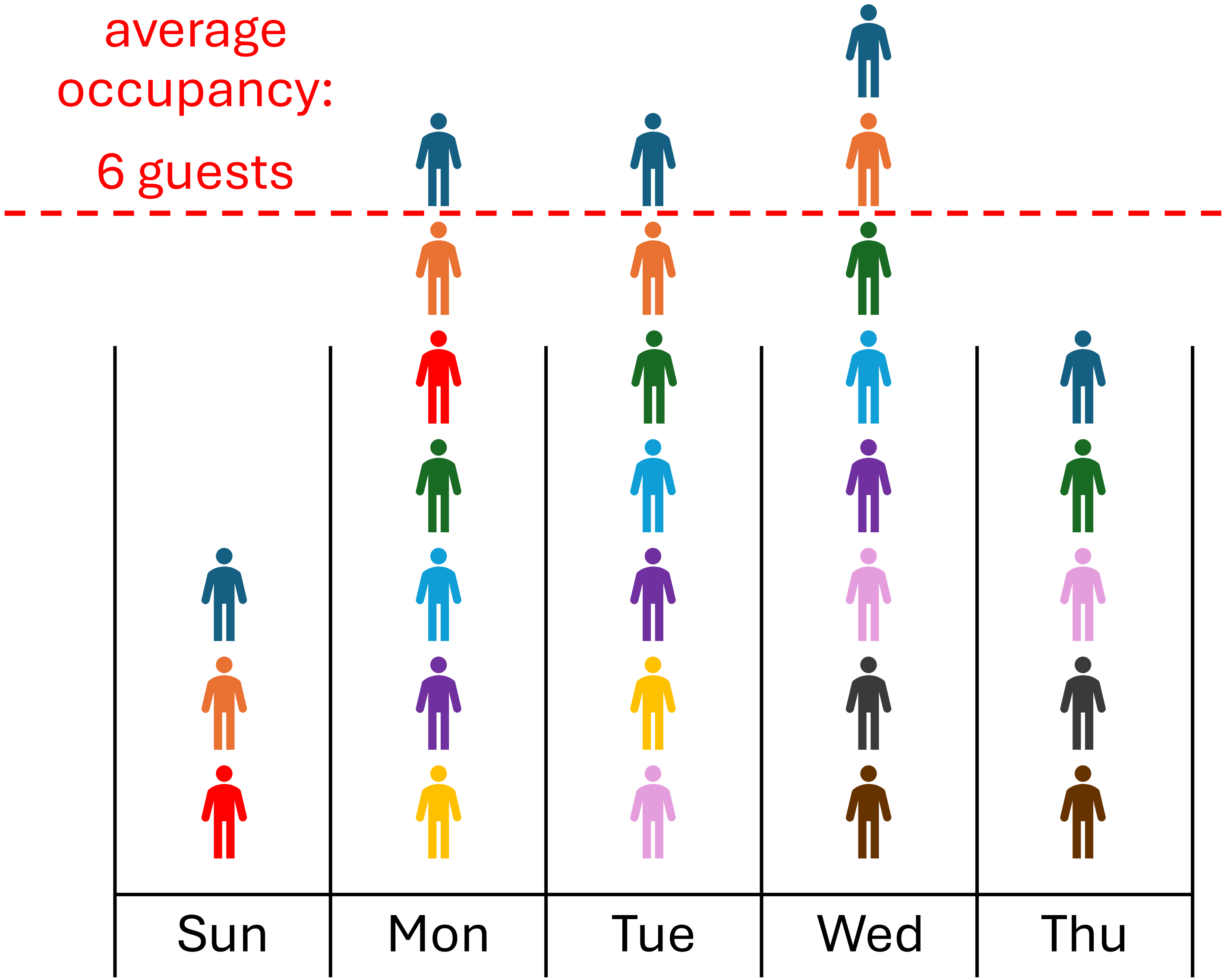
average occupancy | average rate | average delay



Little's law

$$Q = \lambda \times D$$

average occupancy | average rate | average delay



Little's law in network queues

On average:

easy easy hard

$$Q = \lambda \times D$$

queue size arrival rate queuing delay

Little’s law in network queues

On average:

easy easy

$$D = Q/\lambda$$

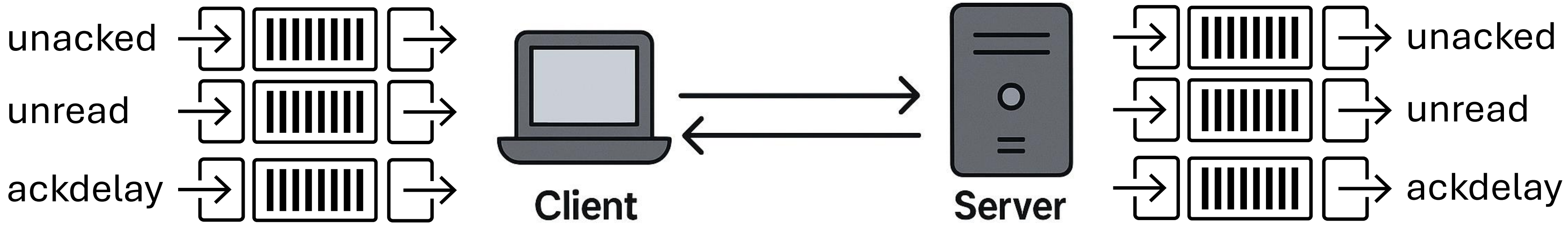
queuing delay queue size arrival rate

Combining delays into end-to-end latency

- Latency is a combination of queuing delays (*)

$$L_{unacked}^{local} + L_{unread}^{local} + L_{unread}^{remote} - L_{ackdelay}^{remote}$$

- These queues already exist in the Linux kernel
  - Just need to extract L from them



Queue performance tracking algorithm

- Simple 4-tuple structure pluggable to any queue:

$$q_s = (\text{time}, \text{size}, \text{total}, \text{integral})$$

timestamp of last state update

current #items in queue

total #items that left the queue until now

time-weighted accumulator of queue size

- Simple algorithms to maintain (`Track`) and extract metrics (`GetAvgs`)

- Passing this tuple to peers allows calculating remote performance

Bridging the semantic gap

- Previous approach needs no app assistance
  - Assumption – bytes/packets correlate to messages
  - Not always true

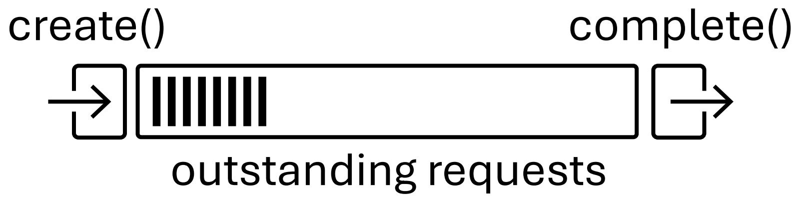
- Better correlation – count send() system calls
  - Deals with messages fragmented by network stack
  - Not always true

- Most accurate – let the app tell you. How?

Bridging the semantic gap

- Just one client-side user-space “request queue”
  - Same Little’s law-based algorithms in user-space
  - State shared via send() metadata

- Simple API with two functions
  - create(n)
  - complete(n)



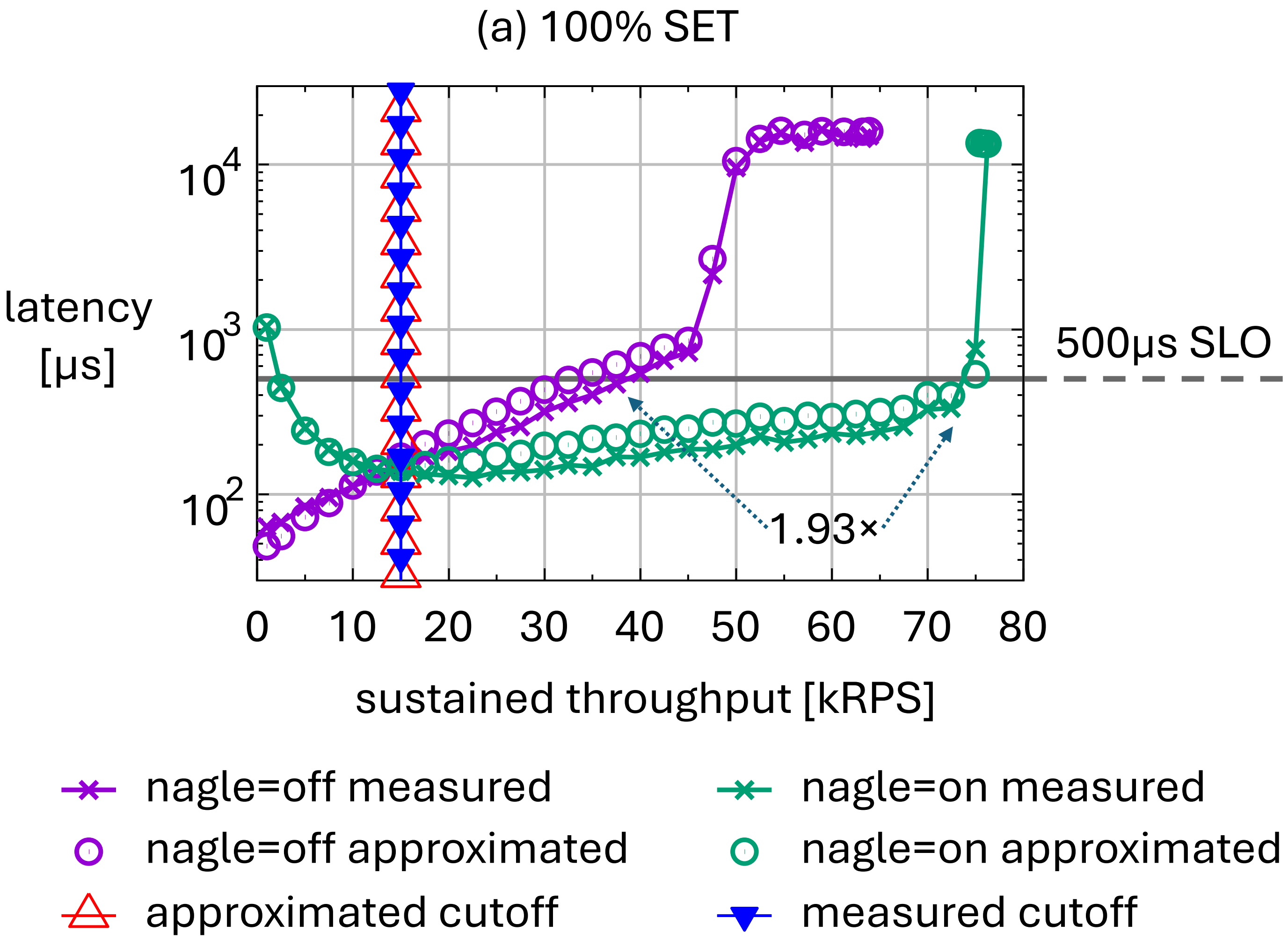
- Easy adoption into libc, gRPC, etc.

Evaluation

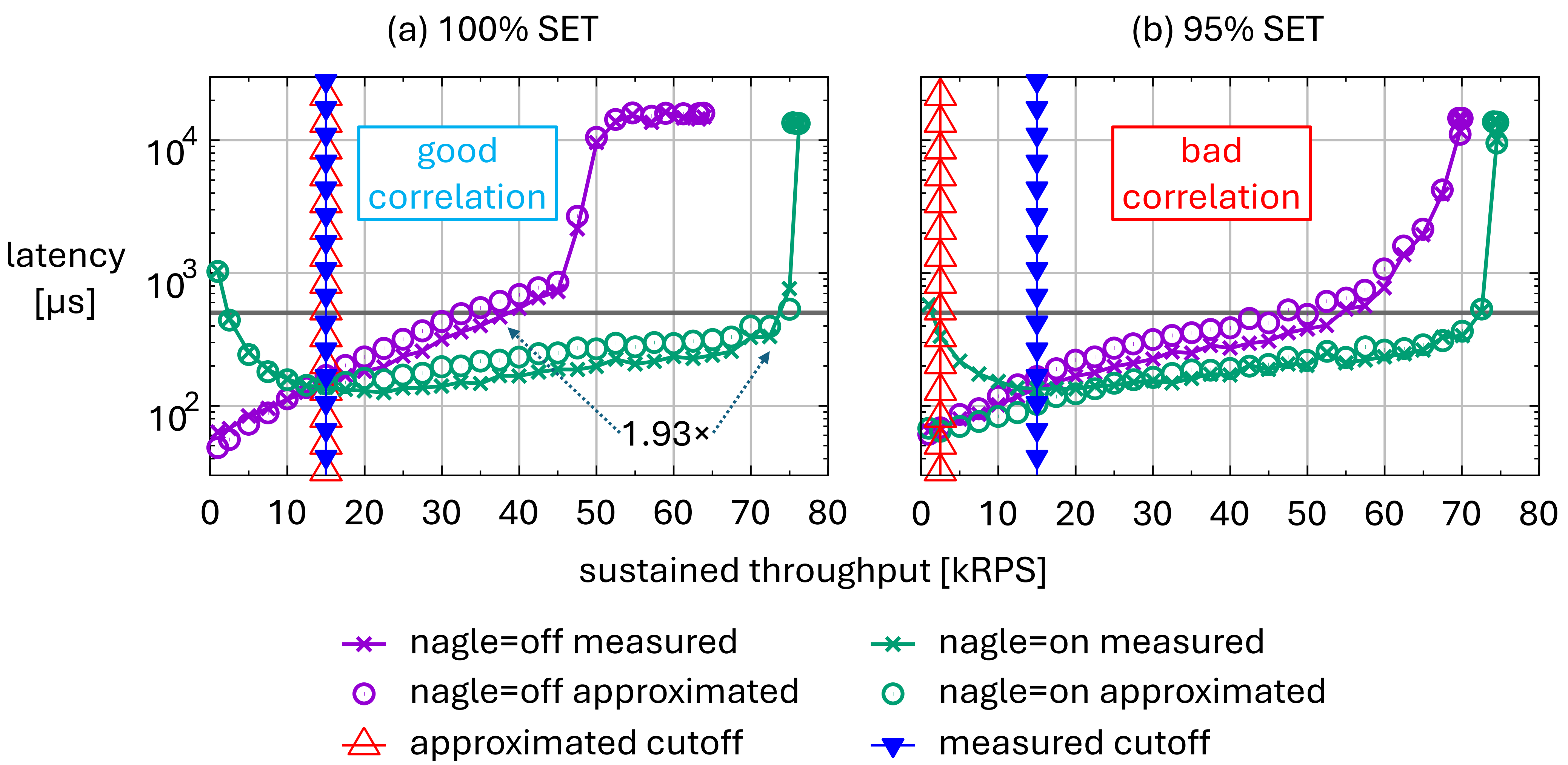
- *Offline prototype – shows potential*
  - Bytes as units
  - No exchange
  - No dynamic toggling – workload runs twice

- Nagle as batching algorithm
- Redis as server app
  - Hardcoded to disable Nagle
- Lancet as client app
  - Measures latency for varying load rates

Evaluation


(a) 100% SET
latency [μs]
10^4
10^3
10^2
500μs SLO
1.93×
0
10
20
30
40
50
60
70
80
sustained throughput [kRPS]
nagle=off measured
nagle=off approximated
approximated cutoff
nagle=on measured
nagle=on approximated
measured cutoff

Evaluation


(a) 100% SET
(b) 95% SET
latency [µs]
10^4
10^3
10^2
good correlation
bad correlation
1.93×
0 10 20 30 40 50 60 70 80
sustained throughput [kRPS]
nagle=off measured
nagle=off approximated
approximated cutoff
nagle=on measured
nagle=on approximated
measured cutoff

Future work and challenges

- Dynamic toggling
  - Granularity
  - Policy
- Metadata exchange
- Better batching heuristics

Conclusion

- The success of batching depends on end-to-end performance
- Approximating this information can improve batching