

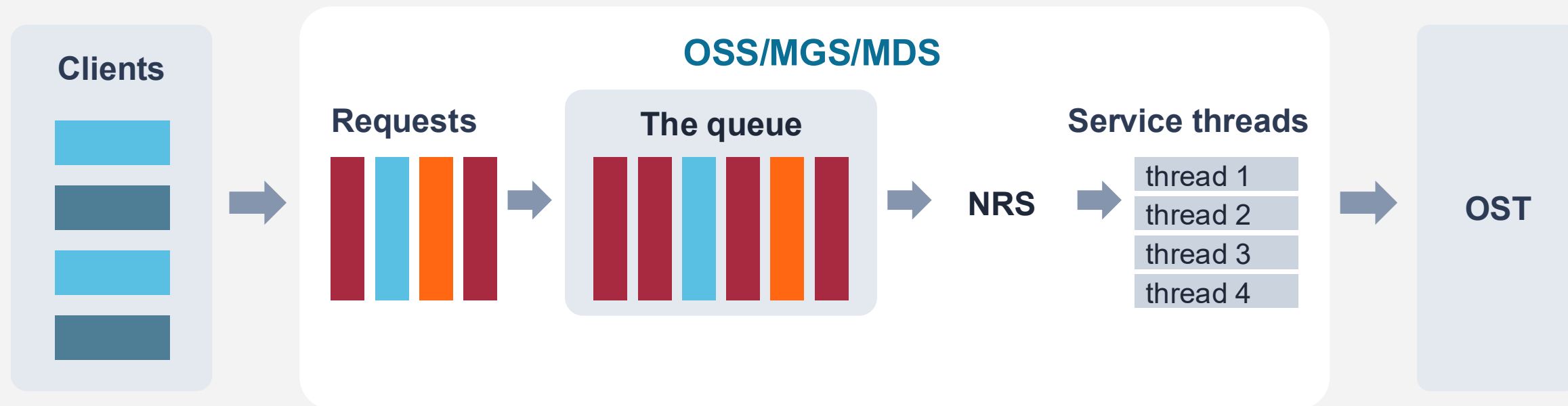
NRS updates: TBF improvements, mClock, and fair share

CHRIS HORN, QIAN YINGJIN

DDN · LAD 26

NRS Overview

What the Network Request Scheduler is



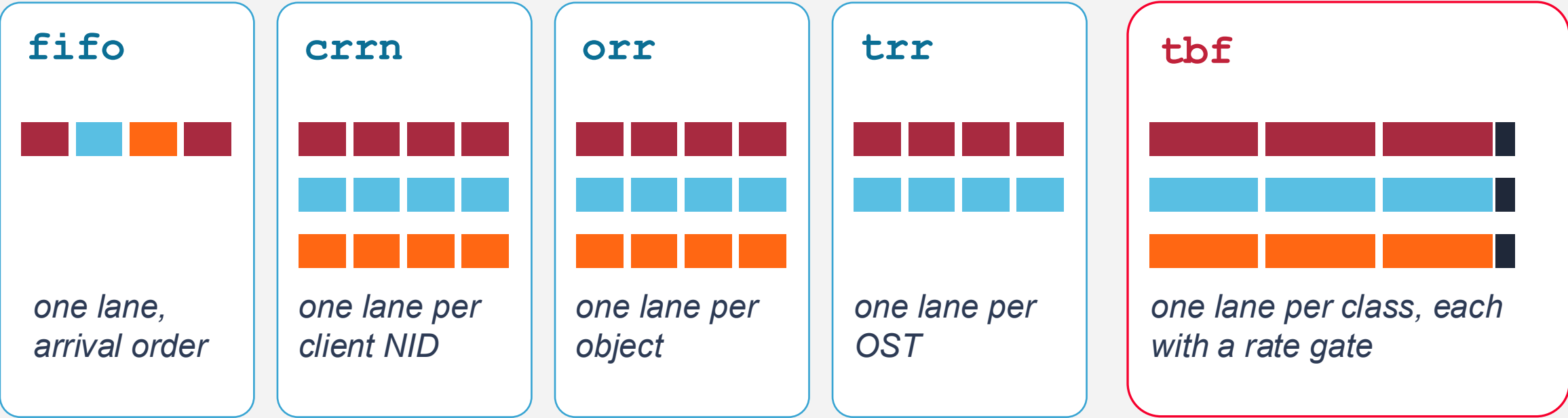
NRS picks which request a free thread takes next.

Server side, inside PTLRPC.
One scheduler per service and one per CPU partition.
Switchable at runtime with `lctl set_param`.

Five policies, two jobs

ORDER THE QUEUE

ALLOCATE THE CAPACITY



First in, first out. The default. It makes no ordering decision at all.

Round-robin between clients. No one client can monopolize the service.

Round-robin between objects and offset order inside each. Aimed at disk seeks.

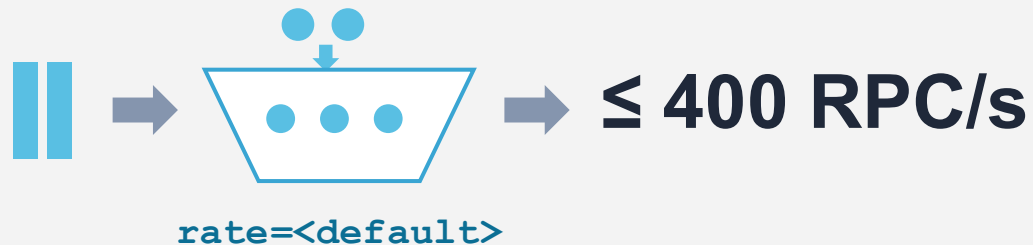
The same idea, but round-robin between OSTs rather than objects.

Rule-based rates per class: NID, job ID, opcode, uid, gid, project ID, nodemap. The only policy that allocates bandwidth.

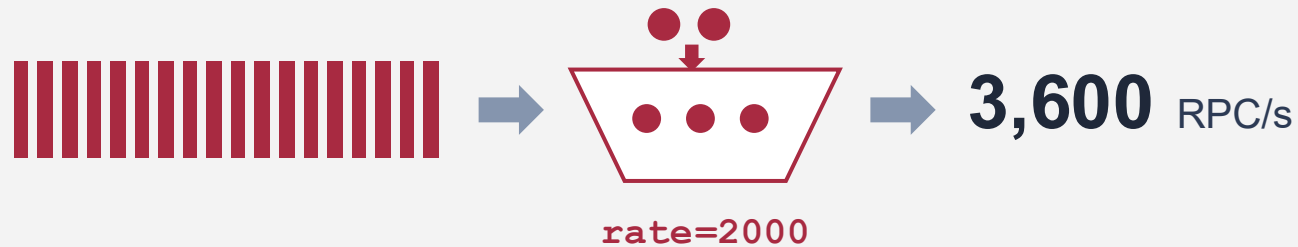
Token Bucket Filter (TBF) & The Noisy Neighbor

LOGIN NODE

```
$ cat secret_recipe.txt
```

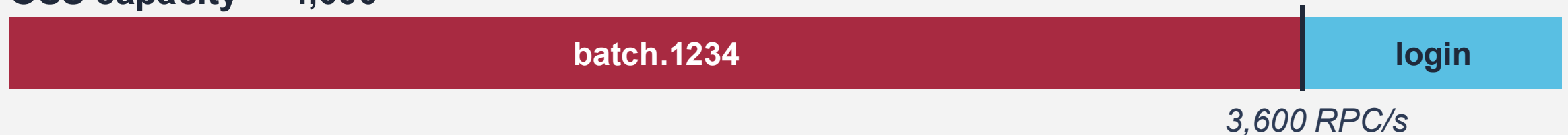


500-NODE JOB



TBF is Qian, Li Xi, Ihara, et al., “A configurable rule based classful token bucket filter network request scheduler for the Lustre file system”, SC’17.

OSS capacity ≈ 4,000



```
lctl set_param ost.OSS.ost_io.nrs_policies="tbf jobid"  
lctl set_param ost.OSS.ost_io.nrs_tbf_rule="start batch jobid={batch.1234} rate=3600"
```

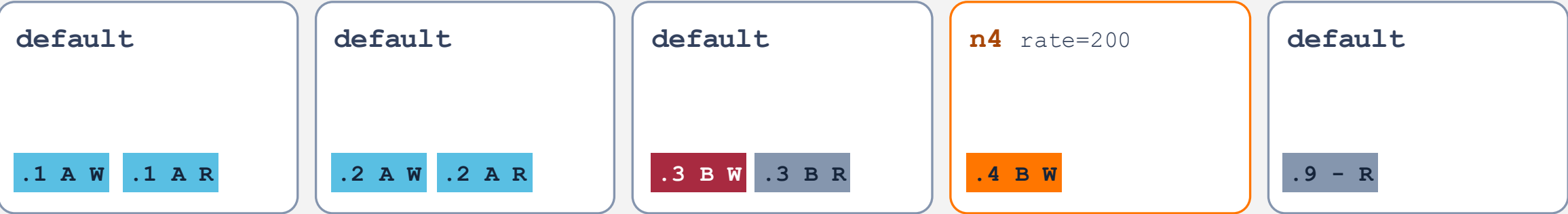
TBF Improvements

Problem: One classification type per scheduler

```
nrs_policies="tbf nid"
nrs_tbf_rule="start ta      nodemap={tenantA} rate=500"  ✗ EINVAL: nodemap not in key
nrs_tbf_rule="start b3w    nid={10.0.0.3@tcp}&opcode={ost_write} rate=50"  ✗ EINVAL
nrs_tbf_rule="start n4     nid={10.0.0.4@tcp} rate=200"  ✓ started
```

INCOMING RPCs

nid 10.0.0.1@tcp nm A op W	nid 10.0.0.1@tcp nm A op R	nid 10.0.0.3@tcp nm B op W
nid 10.0.0.2@tcp nm A op W	nid 10.0.0.2@tcp nm A op R	nid 10.0.0.4@tcp nm B op W
nid 10.0.0.3@tcp nm B op R	nid 10.0.0.9@tcp nm - op R	

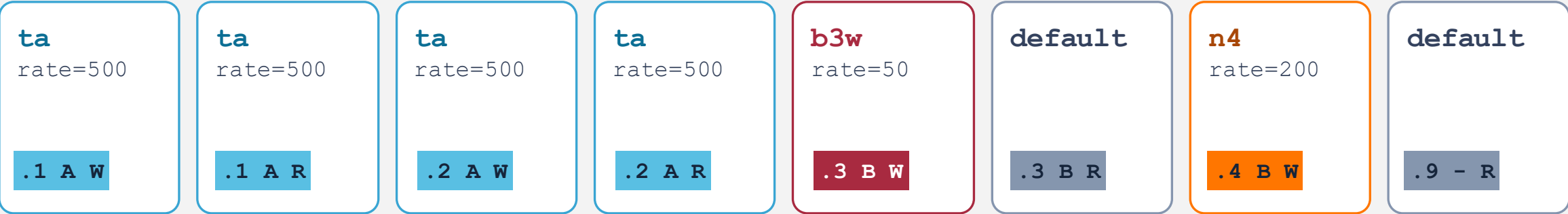


Problem: A wider key starts every rule and splits every bucket

```
nrs_policies="tbf nid+nodemap+opcode"
nrs_tbf_rule="start ta    nodemap={tenantA} rate=500" ✓
nrs_tbf_rule="start b3w  nid={10.0.0.3@tcp}&opcode={ost_write} rate=50" ✓
nrs_tbf_rule="start n4   nid={10.0.0.4@tcp} rate=200" ✓
```

INCOMING RPCs

nid 10.0.0.1@tcp nm A op W	nid 10.0.0.1@tcp nm A op R	nid 10.0.0.3@tcp nm B op W
nid 10.0.0.2@tcp nm A op W	nid 10.0.0.2@tcp nm A op R	nid 10.0.0.4@tcp nm B op W
nid 10.0.0.3@tcp nm B op R	nid 10.0.0.9@tcp nm - op R	



LU-20090: Per-Rule Scheduling Classes

```
nrs_policies="tbf nid classify=dynamic"  
nrs_tbf_rule="start ta    nodemap={tenantA} rate=500"  ✓ keys on nodemap  
nrs_tbf_rule="start b3w   nid={10.0.0.3@tcp}&opcode={ost_write} rate=50"  ✓ keys on nid + opcode  
nrs_tbf_rule="start n4    nid={10.0.0.4@tcp} rate=200"  ✓ keys on nid
```

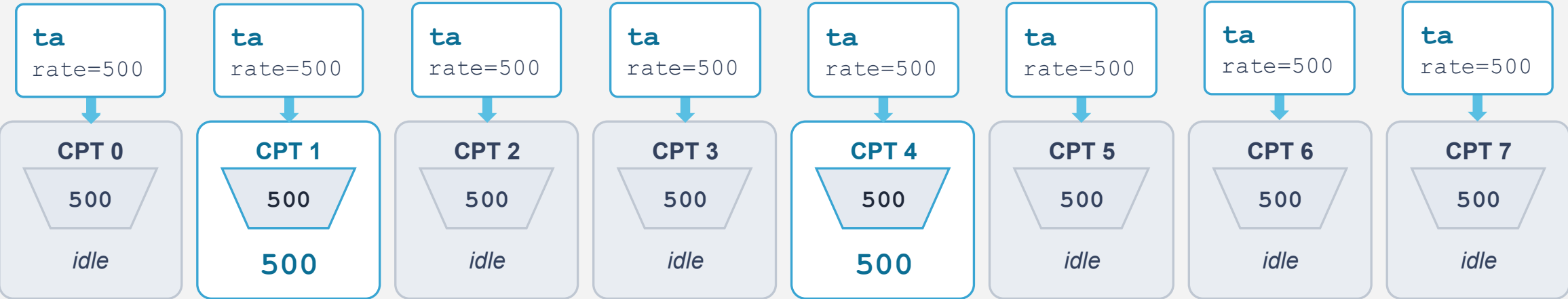
INCOMING RPCs

nid 10.0.0.1@tcp nm A op W	nid 10.0.0.1@tcp nm A op R	nid 10.0.0.3@tcp nm B op W
nid 10.0.0.2@tcp nm A op W	nid 10.0.0.2@tcp nm A op R	nid 10.0.0.4@tcp nm B op W
nid 10.0.0.3@tcp nm B op R	nid 10.0.0.9@tcp nm - op R	



Problem: One bucket for the tenant on every partition

`nrs_tbf_rule="start ta nodemap={tenantA} rate=500"` → **ta rate=500 is installed on every partition**



tenantA exceeds 500 RPC/s



LU-17158 (Etienne Aujames): a rule of 3,000 RPC/s let a job on 4 CPTs reach 12,000 RPC/s.

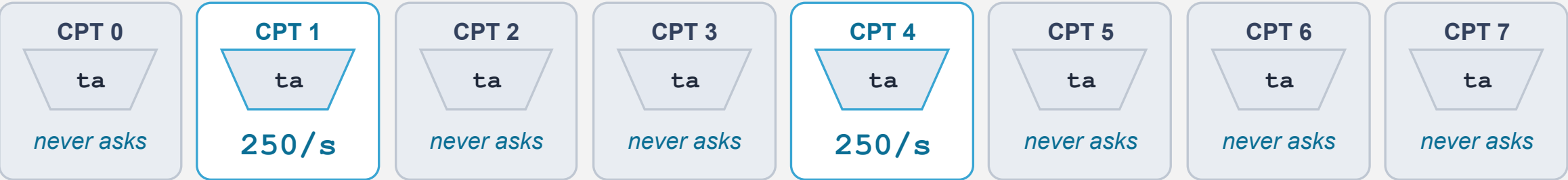
LU-20107: One node budget across every partition

```
nrs_policies="tbf nid classify=dynamic global=1"
nrs_hsch="tbf start ta nodemap={tenantA} rate=500"
```

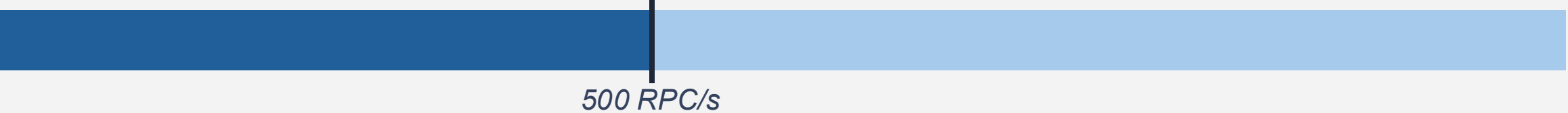
Node tier: one window per rule



Leaf tier: one rule copy per CPT scheduler



TENANTA ACTUALLY GETS



GLOBAL Global Policy Coordinator on MDT0 leases per-server rates (*design only*)

NODE one nrs_node_class per service per rule: the node rate and its token windows

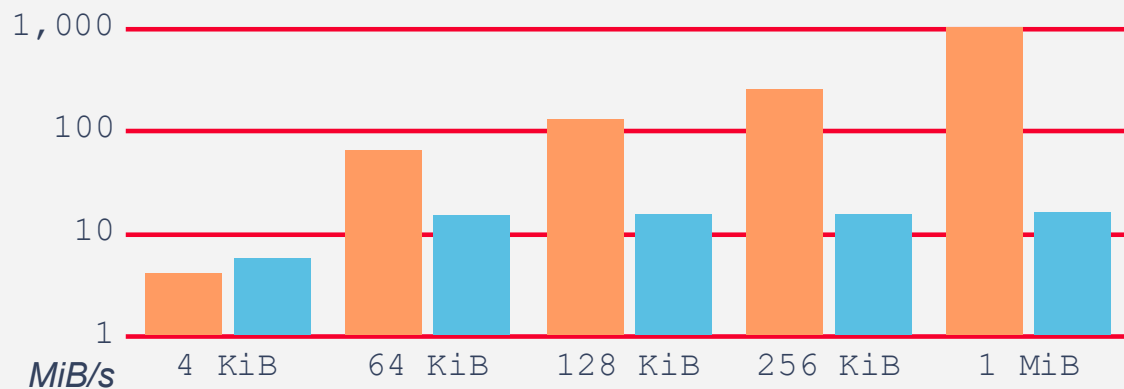
LEAF one rule copy and its class buckets on every CPT scheduler

LU-20184: Byte Rates & LU-20107: Per-rule Counters

LU-20184 An RPC count is not a byte count

A 4 KiB write and a 4 MiB write cost the same token. On an OSS the disk cost follows the bytes, so a `rate` in RPC/s is a proxy that drifts by 250×

■ **rate=1000** 4 → 1,000 ■ **bps=15** measured: flat



```
nrs_tbf_rule="start ... bps=5"  
nrs_tbf_rule="start ... bps=15"
```

Default unit MiB/s. `iops=1000` `bps=5MB/s` bounds both dimensions.

LU-20107 No rule showed what it did

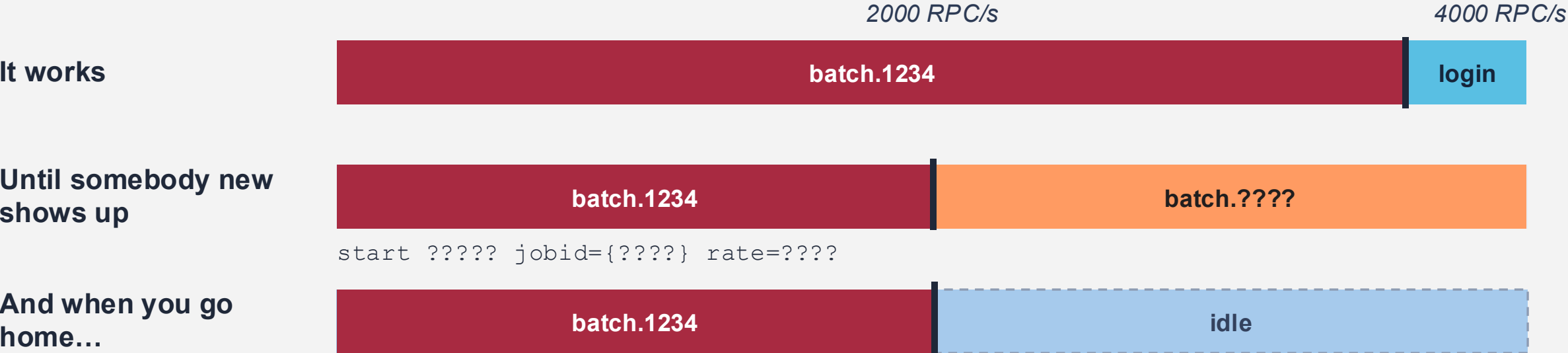
A dump showed the rate you set, never the rate the rule gave. To see the result you measured on the client. Counters now live on the rule.

```
lctl get_param *.*.*.nrs_tbf_rule_yaml  
- name: big  
  conditions: "jobid={big}"  
  limit_iops: 1000  
  enqueue: 48211  
  dispatch: 48190  
  exhausted: 3120  
  limit_dispatch: 48190  
  elapsed_ms: 48307
```

Old entry unchanged. `reset_stat` zeroes one rule, or all of them.

Problem: TBF is Limit-only and Capacity Unaware

```
nrs_tbf_rule="start batch jobid={batch.1234} rate=2000"
```



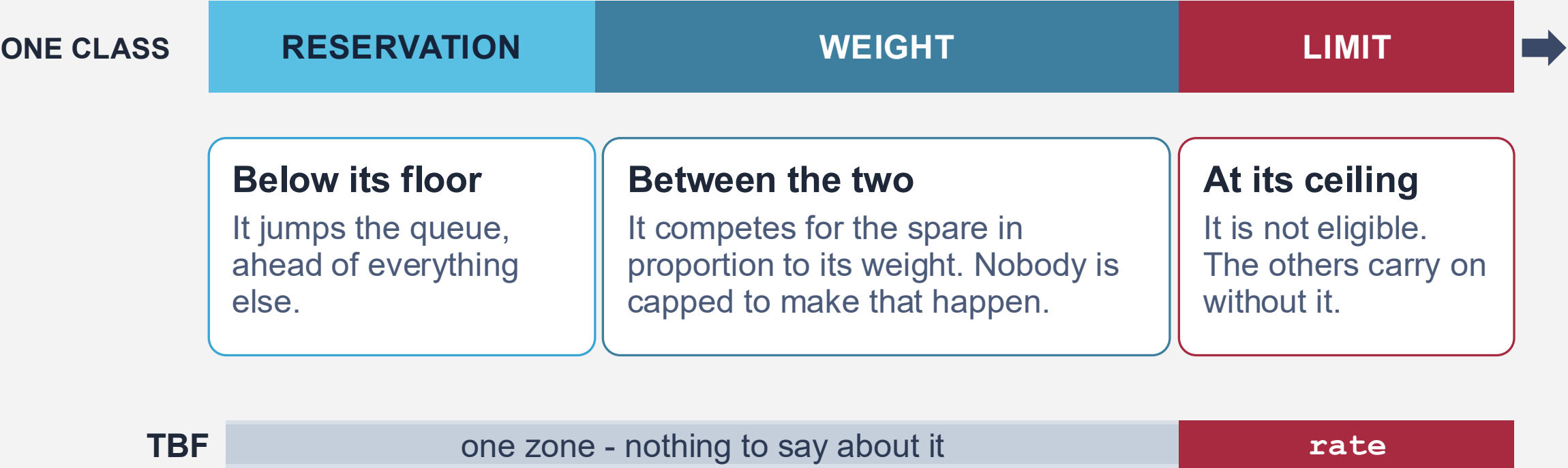
mClock



What is mClock?

Each class gets three numbers instead of one. Dispatch is decided from all three.

Gulati, A., Merchant, A., & Varman, P. (2010). mClock: Handling Throughput Variability for Hypervisor IO Scheduling. OSDI'10.



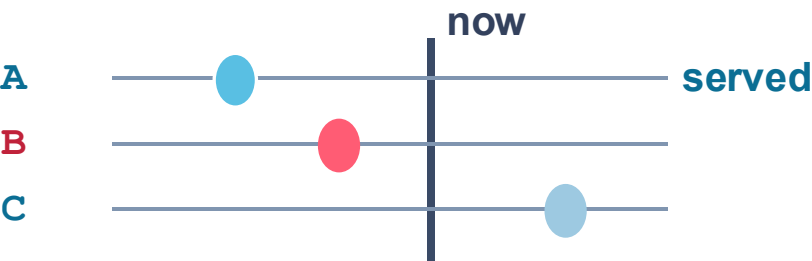
How mClock works

Every time a slot frees up: **is anybody behind their floor?**

YES ↓

Serve the smallest reservation tag

Eligible = its reservation tag has already passed.



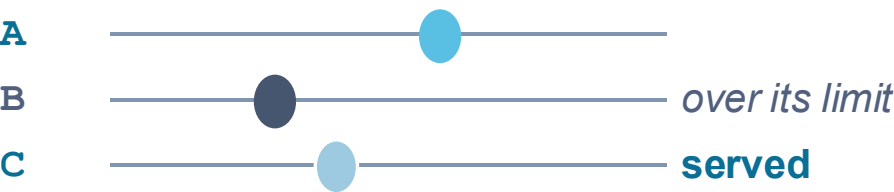
A and B are owed; A the longest.

Constraint Phase (the floor)

NO ↓

Serve the smallest weight tag

Every floor is met, so the spare goes to whoever is furthest behind their share.



B would have won, but its limit tag has not passed.

Weight Phase (the Share and Ceiling)

Reservations first, then the spare

LU-20099: The mClock NRS policy

tbf Fills one slot: the ceiling.

mclock Fills three: ceiling, reserve, weight.



ONE SHARED CLASSFUL SCHEDULER

The rule parser, the rule list and its precedence, and every classification key:
nid, jobid, nodemap, projid, uid, gid, and opcode

POLICY-START KEYS

shared classify= allow= global= limit=
mclock only **reserve=** **weight=** **locality=**

tbf_jobid_cache_size → nrs_class_cache_size
tbf_rate → nrs_class_def_rate
tbf_depth → nrs_class_def_depth
tbf_dyn_rule_thresh → nrs_class_dyn_rule_thresh (old names kept as aliases)

Three slots per rule (TBF fills one)

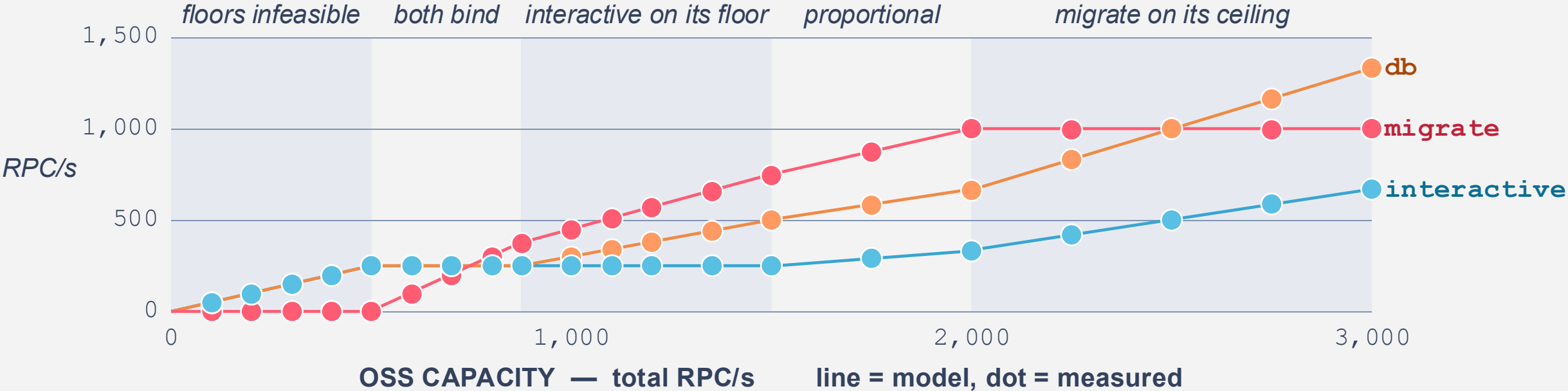
		tbf	mclock
LIMIT <i>the most it may have</i>	$\text{rate} \cdot \text{limit_rate} \cdot \text{limit_iops} \cdot \text{iops} \cdot L$ $\text{bps} \cdot \text{limit_bps}$	yes	yes
RESERVE <i>the least it will get</i>	$\text{resvd_rate} \cdot \text{resvd_iops} \cdot \text{resvd_bps} \cdot R$	—	yes
WEIGHT <i>its share of the spare</i>	$\text{weight} \cdot W$	—	yes
<code>"start db jobid={oltp.*} resvd_iops=250 weight=200 limit_iops=1000"</code> one rule, all three slots			

Every* TBF rule you have is already a valid mClock rule.

*except realtime, which has no mClock spelling.

The paper's tenants on a Lustre cluster

```
nrs_policies="mclock jobid classify=dynamic"  
nrs_mclock_rule="start interactive nodemap={login}      resvd_iops=250 weight=100"  
nrs_mclock_rule="start db                               jobid={oltp.*}    resvd_iops=250 weight=200"  
nrs_mclock_rule="start migrate                          jobid={lfs_migr.*} weight=300 limit_iops=1000"
```



Implementation matches the paper within 0.9%

Fairshare

What “fair” means and which ones we can do

FAIRNESS TYPE	ONE SHARE PER	THE mCLOCK POLICY	STATUS
size-fair	client node of the job	<code>mclock jobid+nid weight=1</code>	MEASURED
job-fair	job	<code>mclock jobid weight=1</code>	not yet tested
user-fair	user	<code>mclock uid weight=1</code>	not yet tested
priority-fair	job, times its band	<code>mclock jobid weight=1</code> + a rule per band, 1–1000	not yet tested
group-fair	group	<code>mclock gid weight=1</code> also <i>projid, nodemap</i>	not yet tested

OPPORTUNITY FAIRNESS - WE HAVE IT

The weight tree is work-conserving: an idle class cannot hold service back. A ceiling would destroy it.

COMPOSITE - WE DO NOT

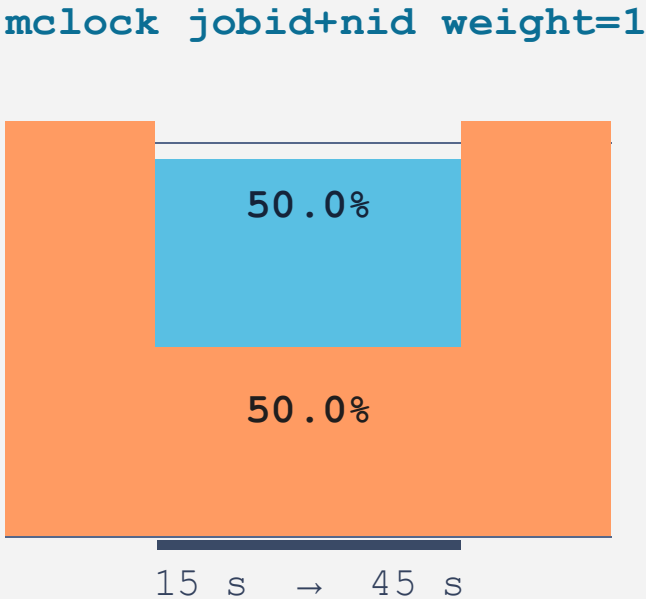
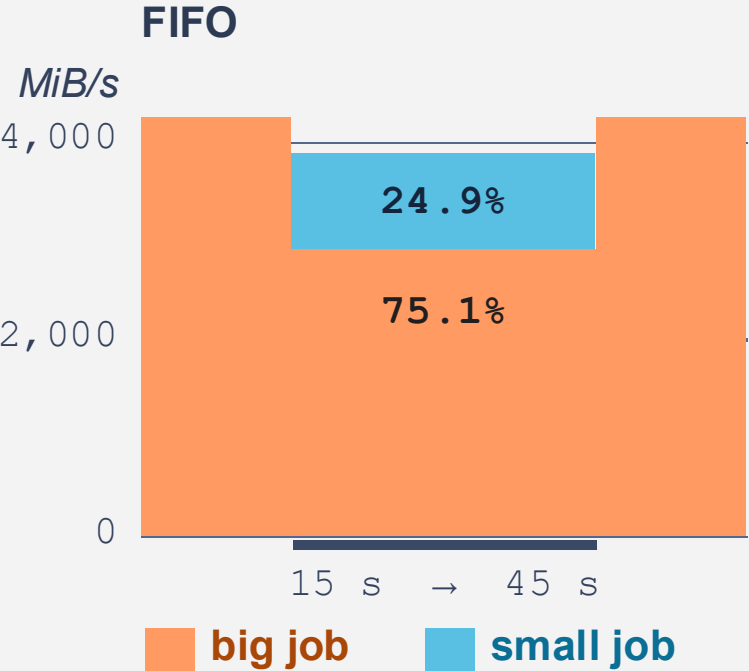
user-then-size-fair needs a weight controller.

Every primitive the paper names is already one start key.

Karrels, Huang, Zhang et al., “Fine-grained Policy-driven I/O Sharing for Burst Buffers”, SC’23.

Two jobs on every client

15 clients · big = 24 processes · small = 8 processes · the small job runs 15 s → 45 s · 16 CPTs



p99 WRITE LATENCY

FIFO → mCLOCK

small job 9.7 → **5.4 ms**

big job 9.8 → **14.1 ms**

The big job pays for the fair share in latency, not in wasted capacity. It is back at 4222 MiB/s within one sample of the small job stopping.

A quarter of the OST becomes half of it, with no rule written.

Where this leaves us

TBF Improvements

- One scheduler, many key types.
- A rate that means the server, not one CPU partition.
- Byte rates, and counters on every rule.

mClock

- A floor that holds while the device slows down.
- A share of the spare, capping nobody.
- Extension of TBF's own rule syntax; every rule but realtime.

Fair share

- Measured: 24.9% → 50.0%, for about 1% of throughput.
- ThemisIO's taxonomy maps to mClock, key for key.

STILL OPEN

Global ceiling precision

+3% over on 2 clients, +63% on 15.

The fast-path cost

reserve and limit cost ~8% with no queue.

Global Distributed QoS

Still a design, not code.

A limit you can set, a floor you can promise, and a fair share for the rest.

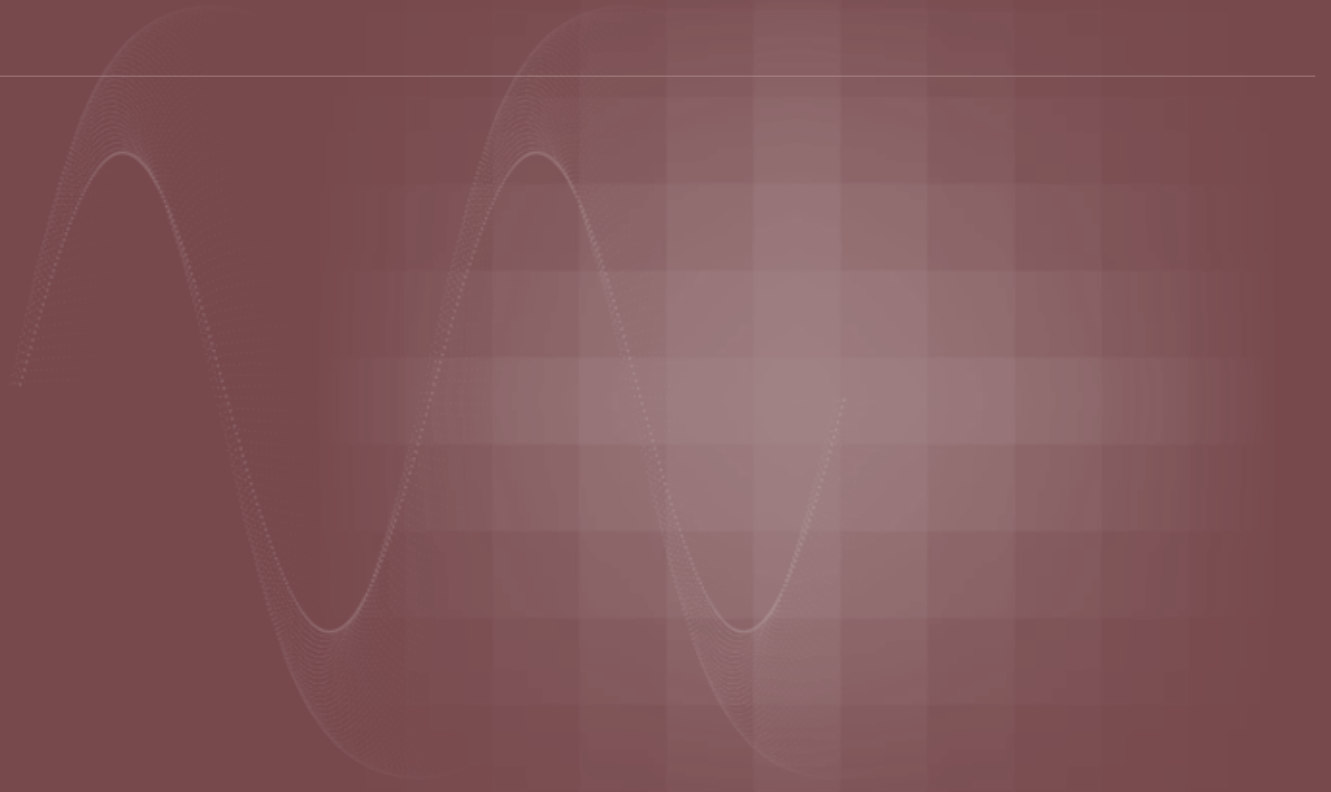
One more thing...

Yes, there is a man page.

Questions?

chorn@ddn.com

Backup

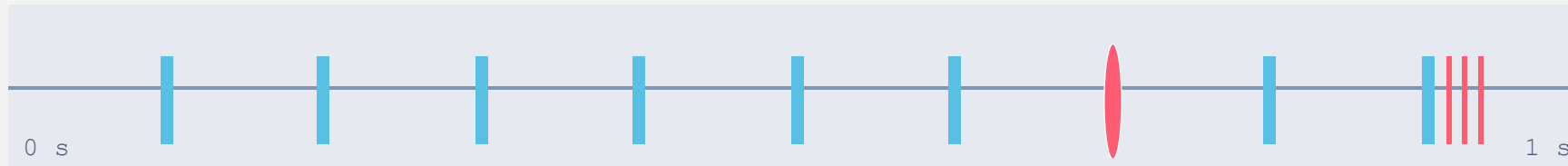


LU-20107: One window, paced tokens, bounded carry-over

ONE WINDOW 1 second · quota = the node rate · slides forward when a request lands after its end

paced grant one token per ask; each grant moves the next deadline forward by $1/\text{rate}$, so the quota spreads over the second

refused deadline = end of the window, staggered by the refusals so far



idle for a whole window → restart with the full quota

waiting CPTs retry across the next window, not all at once

carry-over $\leq \frac{1}{4}$ quota
unused quota → surplus
overrun → debt

no burst, no starved second

WHO ASKS

A class bucket asks for a token when a request lands in it empty, and again after each dequeue while it stays non-empty.

The first bucket to ask takes the next token. **An idle CPT never asks, so its share goes to whoever does.**

One busy client alone gets the whole node rate; two split it; the node sum never exceeds the rate plus one bounded carry-over.

TBF ON NODE TOKENS

A node rule's bucket holds at most one limit token.

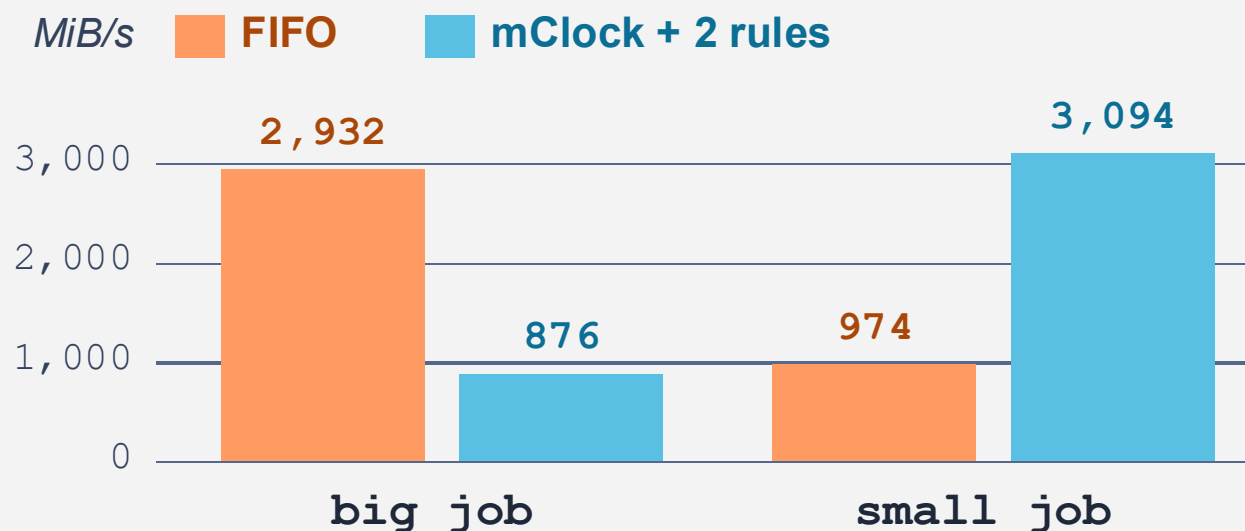
Dequeue dispatches when the bucket holds a token and its deadline has passed. Otherwise the bucket re-keys itself to the deadline in the CPT heap and the head throttles until then.

One spinlock per node class: taken under the CPT request lock, dropped before any queue or tree.

One window per rule: the busy CPTs take what the idle ones never ask for.

Adding a floor and a ceiling

```
nrs_policies="mclock jobid+nid weight=1 reserve=1 limit=1"
nrs_mclock_rule="start cap jobid={big.*} limit_iops=16000"
nrs_mclock_rule="start floor jobid={small.*} resvd_iops=8000"
```



THE FLOOR DID THIS, NOT THE CEILING

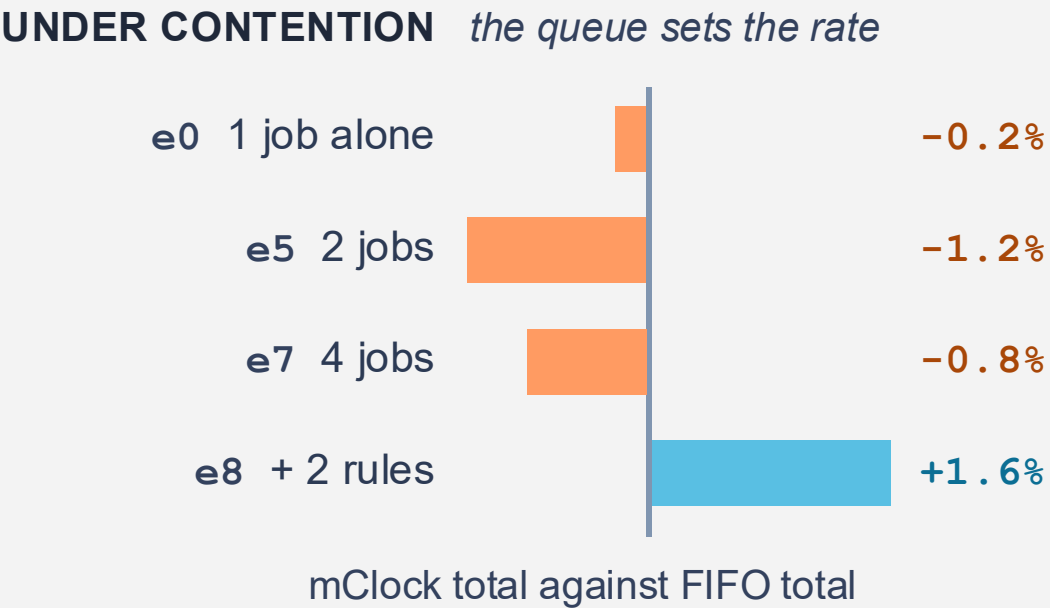
A rule holds its rate in **each** partition, so `limit_iops=16000` is 256,000 RPC/s across 16 of them. The big job ran at about 14,000. The ceiling never bound — the 8,000 RPC/s floor under the small job is the whole effect.

Small job p50 7.67 → 2.48 ms. Total 3906 → 3970 MiB/s. No throughput lost.

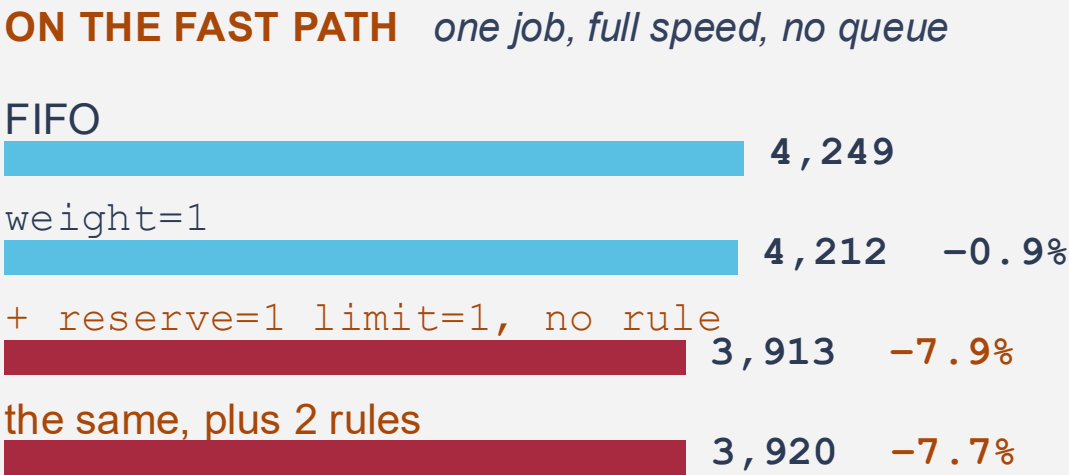
A floor turns a quarter of the OST into three quarters.

And the total went up, not down — the small job simply fills the threads the big job was holding.

What it costs



Four experiments, same workload under both policies. Once a queue forms, the queue sets the rate and the scheduler barely shows up.



THE DIMENSIONS COST, THE RULES DO NOT

e5d runs all three dimensions with no rule at all and still loses 7.9%. Adding two rules gives 3920 — rule matching costs nothing this test can measure.

With a queue, mClock is free. Without one, the dimensions are not.

Turn on only the dimensions you will use: on the fast path weight-only costs 0.9%, all three about 8%.

Where this does not hold

e4 A job needs requests in flight to claim its share

A 2-process job cannot fill half the threads. It takes the latency instead of the throughput.

share 1.6% → 4.5%

p50 3.57 → 1.19 ms

e2 Fairness is per job, per client node

A job on 100 nodes holds 100 shares. Against a 12-node job, a 1-node job is outvoted 12 to 1.

46.8% → 9.5%

one node against twelve

A queue has to form at all

A server that hands every request a thread makes all policies equal. The thread cap that forces a queue costs throughput on its own.

2235 → 2519 MiB/s

32 → 186 threads

No job ID means one share for the whole node. · 64 KiB direct writes on one OSS: no reads, no metadata, no mixed block sizes.

Fair between the things the server can see.

One share per job ID, per client node.

The noisy neighbor: First In, First Out (FIFO)

ONE LOGIN NODE



Glimpse RPC for `ls -l`.

ONE 500-NODE JOB



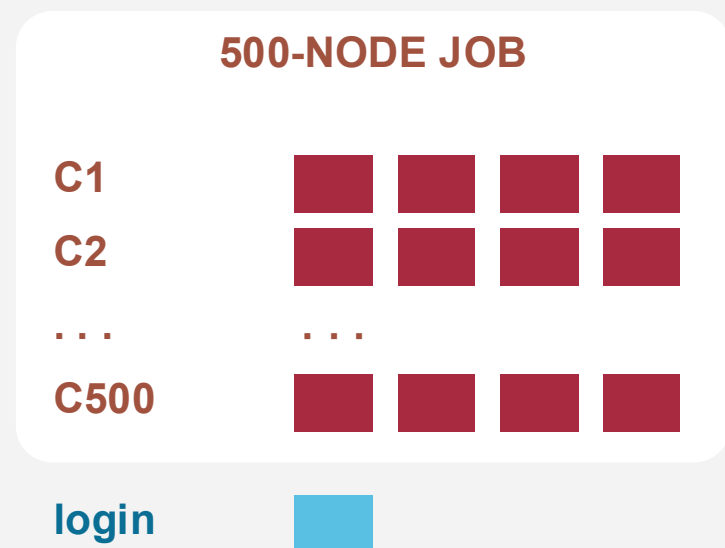
Dispatch order



`ls -l` request delayed by $\sim \text{\# clients} \times \text{\# RPCs/client}$

The noisy neighbor: Client Round-Robin over NIDs (CRR-N)

One queue per client NID



Dispatch order



`ls -l` request delayed by $\sim \# \text{ clients} \times \# \text{ RPCs/client}$

a batch of RPCs from each client, then yours

The noisy neighbor: Object and Target Round-Robin

Both order bulk read RPCs to cut disk seeks. The aim is throughput, not fairness.

ORR — a batch per object

500 objects on this OST means 500 batches before you.
delayed by **# objects × the quantum (256)**



TRR — one batch per OST

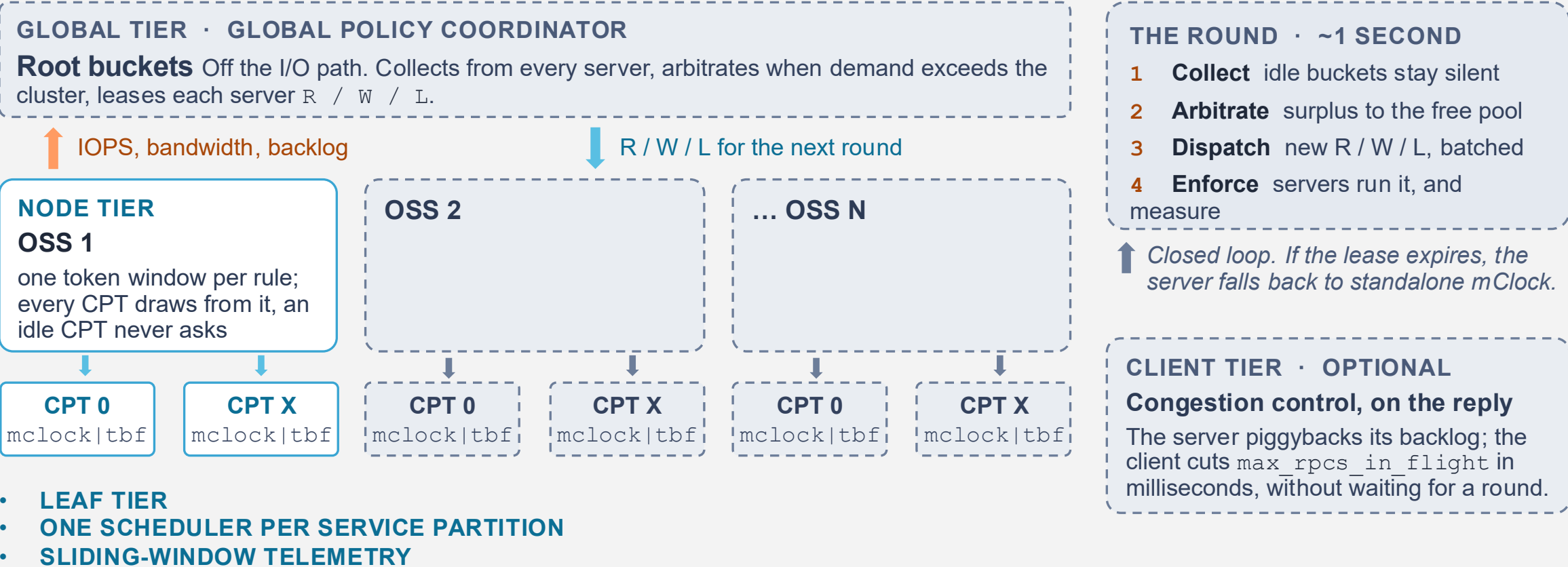
The quantum bounds the wait; your place in it is luck.
delayed by **the quantum (256) + your offset**



The noisy neighbour shows up here too — the trade for the throughput.
These policies solve a different problem, so the talk does not go further into them.

ORR handles reads only by default; a writing job bypasses it to the FIFO fallback.

LU-20107: Global Distributed QoS for Lustre



Design draws on GPFS distributed bandwidth quota management and mClock (Gulati et al., OSDI 2010).

How mClock works

Each tag is **the later of** `last tag + 1/rate` and **now**

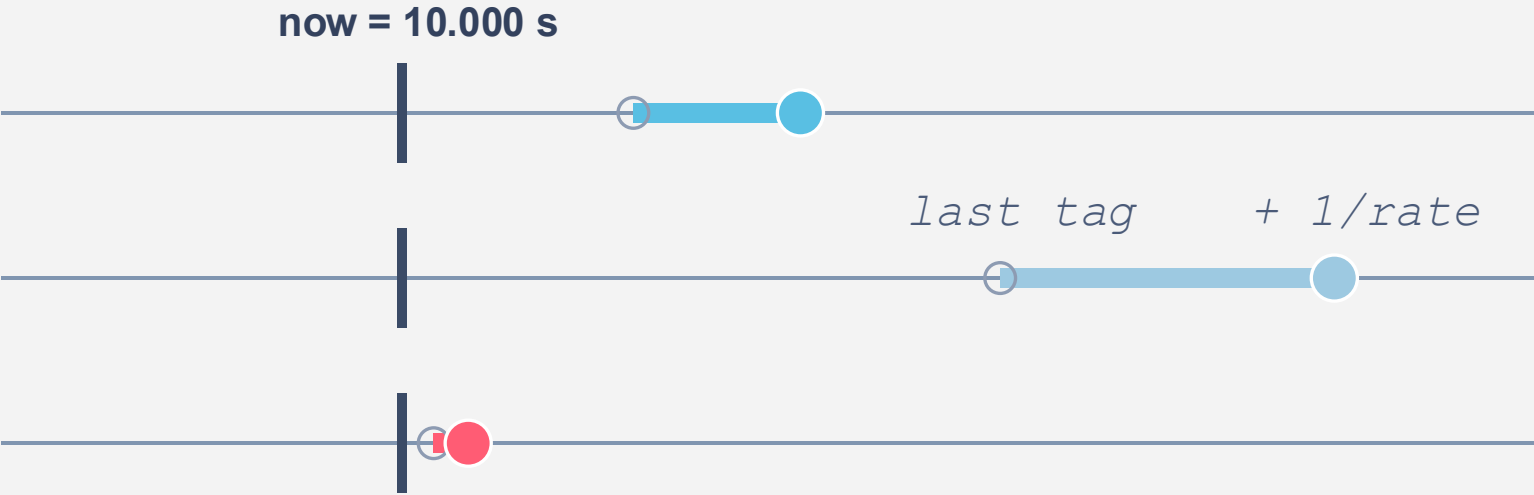
BUSY (IT HAS A BACKLOG)

RESERVATION 200 RPC/s
10.007 → **10.012** (+5 ms)

WEIGHT weight 100
10.018 → **10.028** (+10 ms)

LIMIT 1000 RPC/s
10.001 → **10.002** (+1 ms)

The bar is 1/rate, to scale. The ceiling clock ticks five times per tick of the floor.



THE SAME CLASS, AFTER GOING QUIET

RESERVATION 200 RPC/s
9.996 → **10.000 = now**



LU-20107: mClock on node tokens

For a node rule the R and L tags come from the node tokens instead of the per-CPT rate copy. Weight does not.

LIMIT
L tag = the limit token's deadline
Taken first, in the node lock.
Granted: the paced grant time. Refused: the retry time at the end of the window.

RESERVE
R tag = the reserved token's deadline
Taken only behind a granted limit token, and bounded below by the L tag, so reserved service still obeys the node ceiling.
Refused: the bucket stays out of the reserved tree until the next ask, as R=0 does.

WEIGHT
stays per CPT
A bucket that lacks its tokens re-keys only the shared tags and goes back to the limit tree.
The weight tag and virtual time do not move.
Fairness is local to each CPT.

BOTH TOKENS IN ONE CRITICAL SECTION
Separate takes could interleave with another CPT and leave the R deadline trailing an L deadline that CPT advanced. One section keeps both from the same point of the shared windows.

WHAT A DISPATCH DEBITS
Reserved service debits both tokens. Surplus service from the weight tree debits the limit token alone and keeps the reserved token for the next request.

share= which dimensions draw from the node budget; the other runs per CPT at 1/nheads of the rate

share=	node-wide	per-CPT	use case
RL	floor and ceiling	—	one class capped and guaranteed for the node (default)
L	ceiling	floor	cap the node sum, keep a fixed floor on every CPT
R	floor	ceiling	spread one reservation over the CPTs that have load
none	—	both	node tier bookkeeping only

Refused: share=R when the node reservation exceeds the per-CPT ceiling (it could all land on one CPT), and share= on a plain rule. R=0 or L=max turns a dimension off.

A node-wide floor on tokens — not a global weighted scheduler, no guarantee above server capacity.

mClock rule YAML: two rules, one dump

```
oss# lctl get_param -n *.*.*.nrs_mclock_rule_yaml
```

Two clients write 1 MiB direct RPCs at the same time: **gold** has an IOPS limit and a reserved floor, **bulk** an 8 MiB/s BPS limit.

```
regular_requests:
- cpt: 0
  rules:
  - name: bulk
    conditions: "jobid={ddbulk.0}"
    resvd_iops: 0
    resvd_bps: 1048576
    limit_iops: max
    limit_bps: 8388608
    weight: 20
    ref: 1
    enqueue: 200
    dispatch: 209715200
    exhausted: 513
    resvd_dispatch: 0
    limit_dispatch: 0
    weight_dispatch: 200
    forced: 0
    elapsed_ms: 31275
```

```
- name: gold
  conditions: "jobid={dd.0}"
  resvd_iops: 50
  resvd_bps: 0
  limit_iops: 200
  limit_bps: max
  weight: 80
  ref: 1
  enqueue: 400
  dispatch: 400
  exhausted: 0
  resvd_dispatch: 384
  limit_dispatch: 0
  weight_dispatch: 16
  forced: 0
  elapsed_ms: 31275
```

dispatch takes the rule's unit

bytes for a BPS limit, RPCs for an IOPS limit.
bulk's 209,715,200 is the 200 MiB dd wrote.

the ceiling held

the client measured 8.4 MB/s against bulk's 8 MiB/s cap; gold's 400 RPCs went out under limit_iops=200.

the tree fields say who paid

gold took 384 of 400 dispatches from the reserved tree and 16 from the weight tree; bulk's 200 all came from weight.