IN-MEMORY CACHING:

CURB TAIL LATENCY

WITH PELIKAN

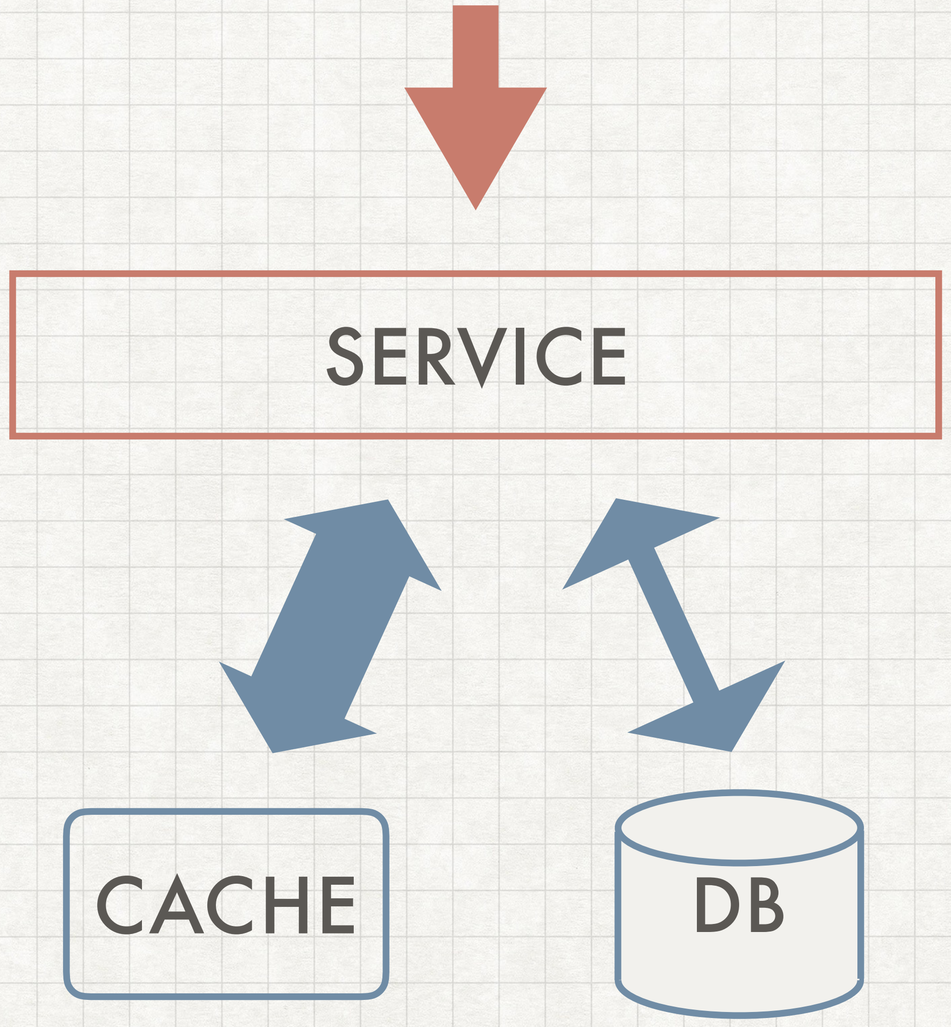
ABOUT ME

- 6 years at Twitter, on cache
- maintainer of Twemcache & Twitter's Redis fork
- operations of thousands of machines
- hundreds of (internal) customers

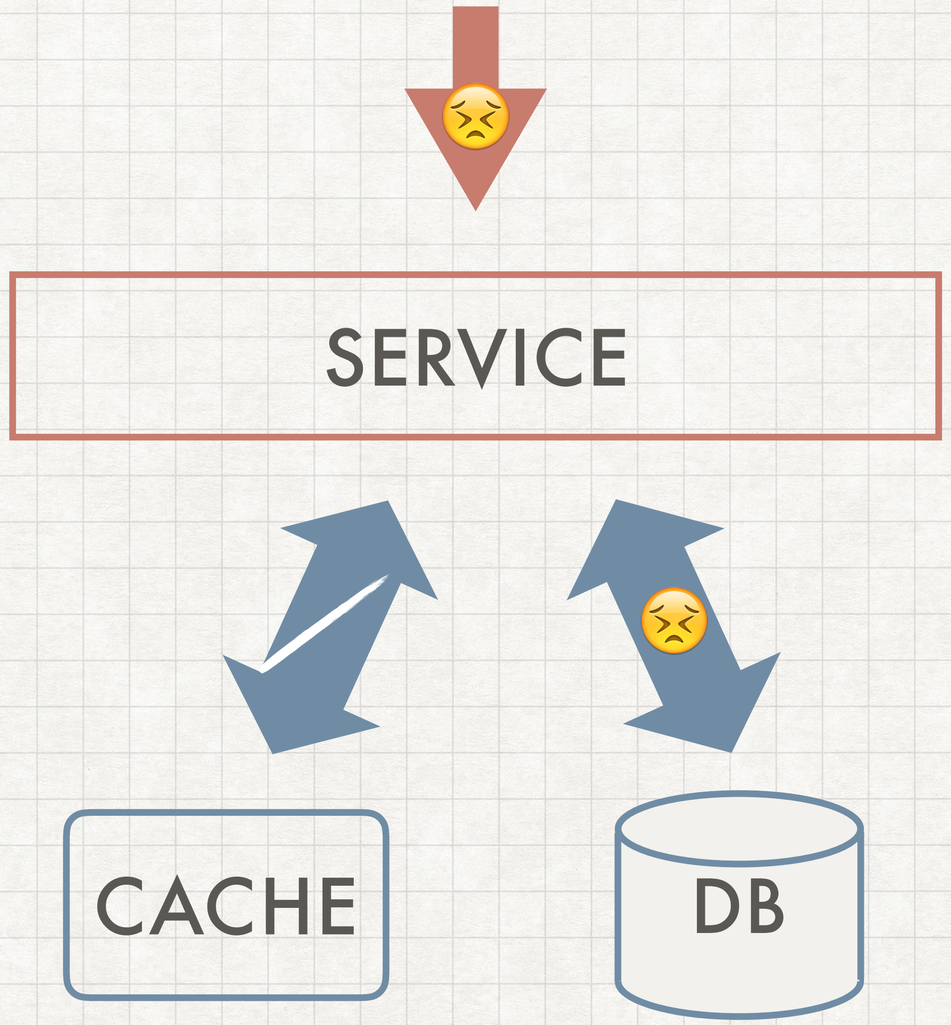
- Now working on Pelikan, a next-gen cache framework to replace the above @twitter
- Twitter: @thinkingfish

THE PROBLEM:
CACHE PERFORMANCE

CACHE
RULES
EVERYTHING
AROUND
ME



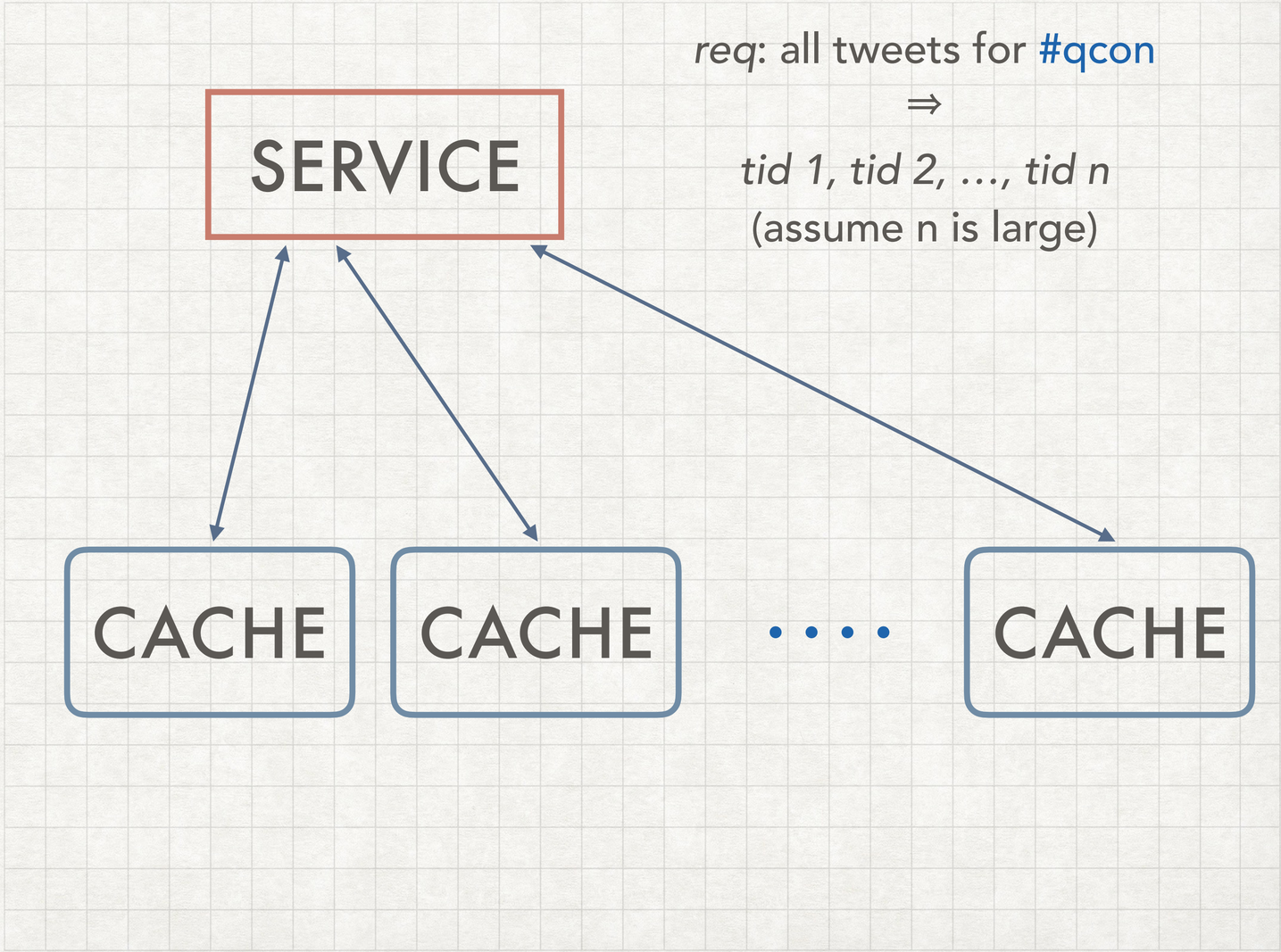
CACHE RUINS EVERYTHING AROUND ME



LATENCY & FANOUT

- what determines overall 99%-ile of *req*?

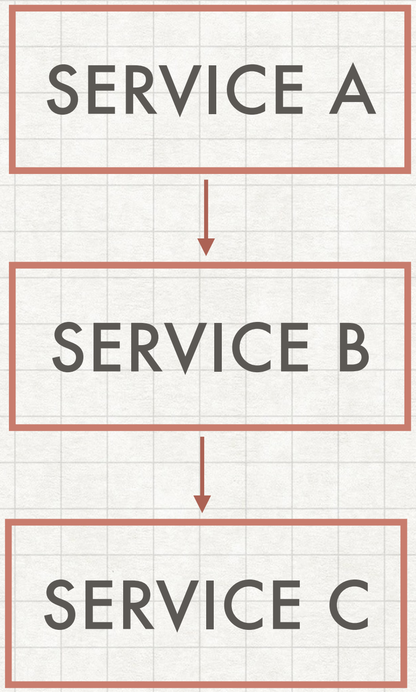
fanout	percentile
1	p99
10	p999
100	p9999
1000	p99999



LATENCY & DEPENDENCY

- what determines overall 99%-ile?

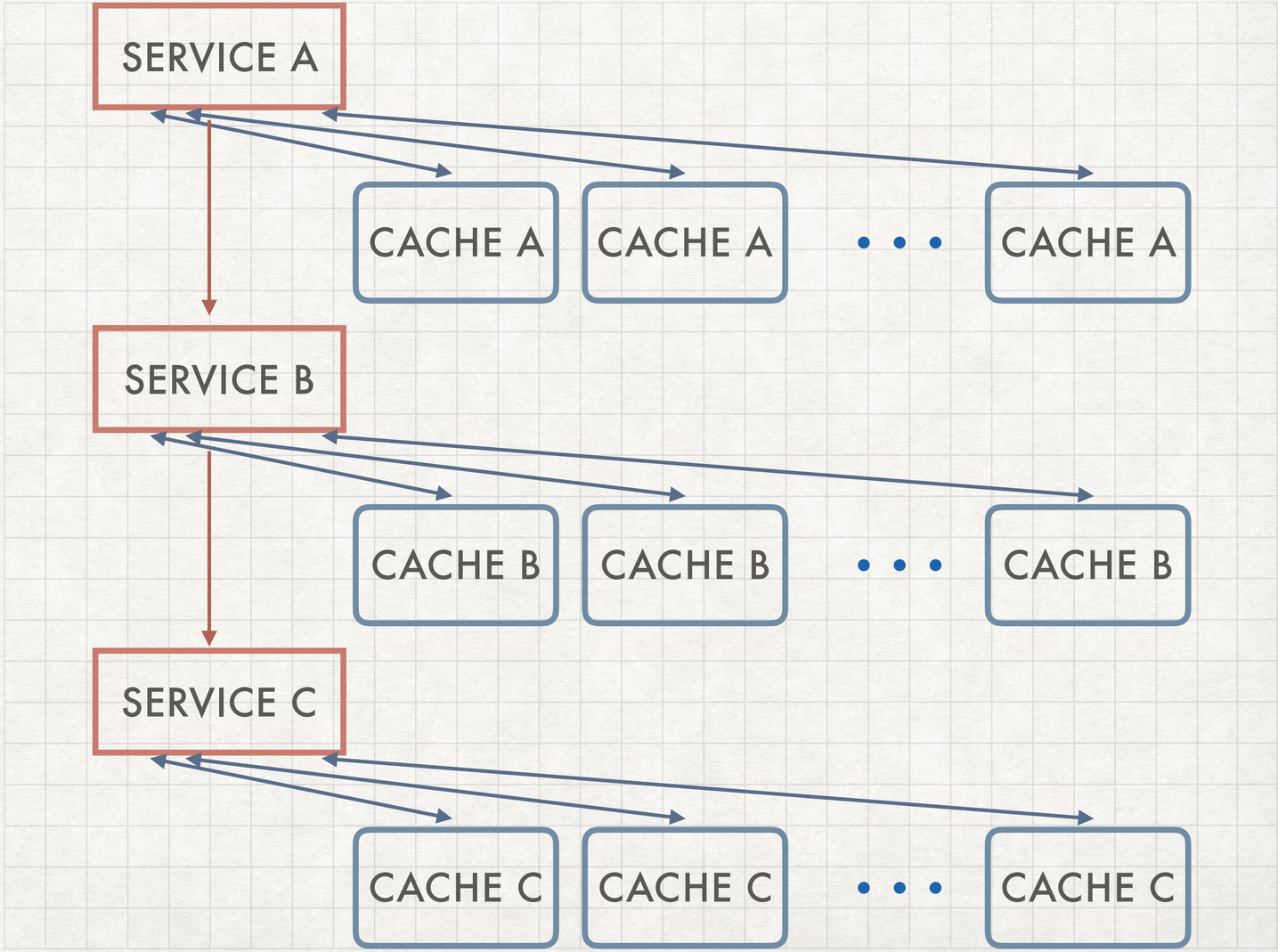
- *adding* all latencies together
- *N* steps ⇒ *N*x exposure to tail latency



get timeline
get tweets
get users for each tweet

CACHE IS UBIQUITOUS

- Exposure of cache tail latency increases with *both* **scale** and **dependency**!



GOOD CACHE PERFORMANCE

=

PREDICTABLE LATENCY

GOOD CACHE PERFORMANCE

=

PREDICTABLE TAIL LATENCY

KING OF PERFORMANCE

"MILLIONS OF QPS PER MACHINE"
"SUB-MILLISECOND LATENCIES"
"NEAR LINE-RATE THROUGHPUT"
...

GHOSTS OF PERFORMANCE

"*USUALLY* PRETTY FAST"

"HICCUPS *EVERY ONCE IN A WHILE*"

"TIMEOUT SPIKES *AT THE TOP OF THE HOUR*"

"SLOW *ONLY* WHEN MEMORY IS LOW"

…

I SPENT FIRST 3 MONTHS AT TWITTER LEARNING CACHE BASICS...

...AND THE NEXT 5 YEARS CHASING GHOSTS

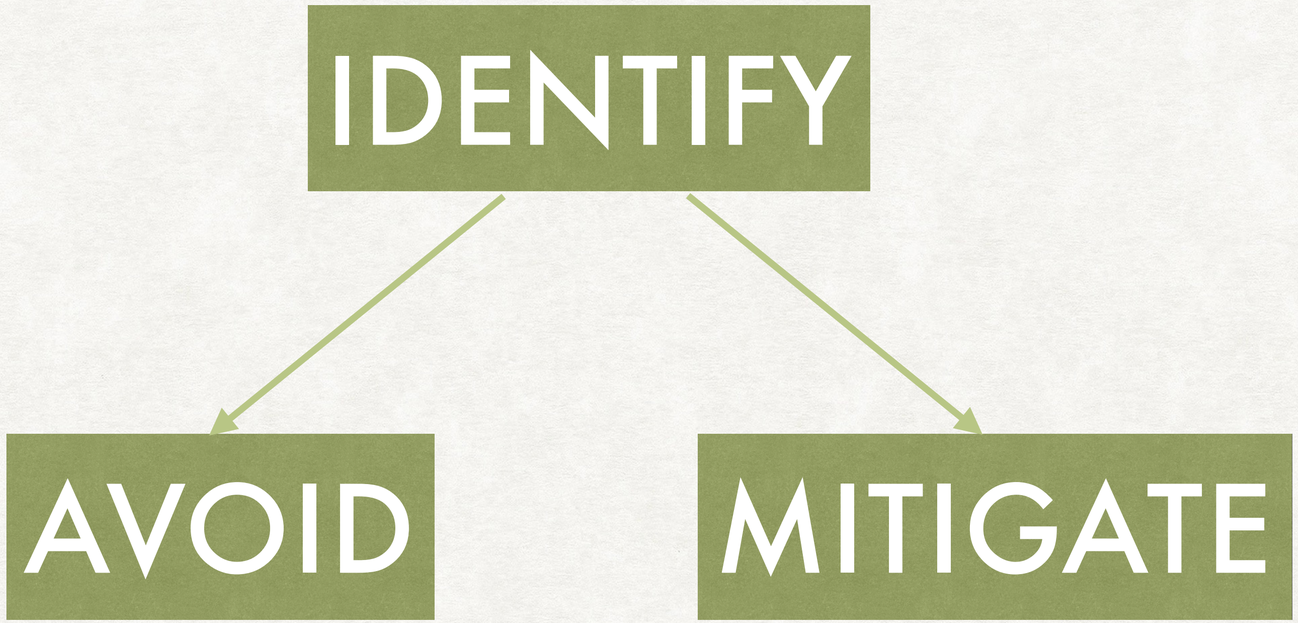
GHOSTBUSTERS™

CHAINING DOWN GHOSTS

=

MINIMIZE INDETERMINISTIC BEHAVIOR

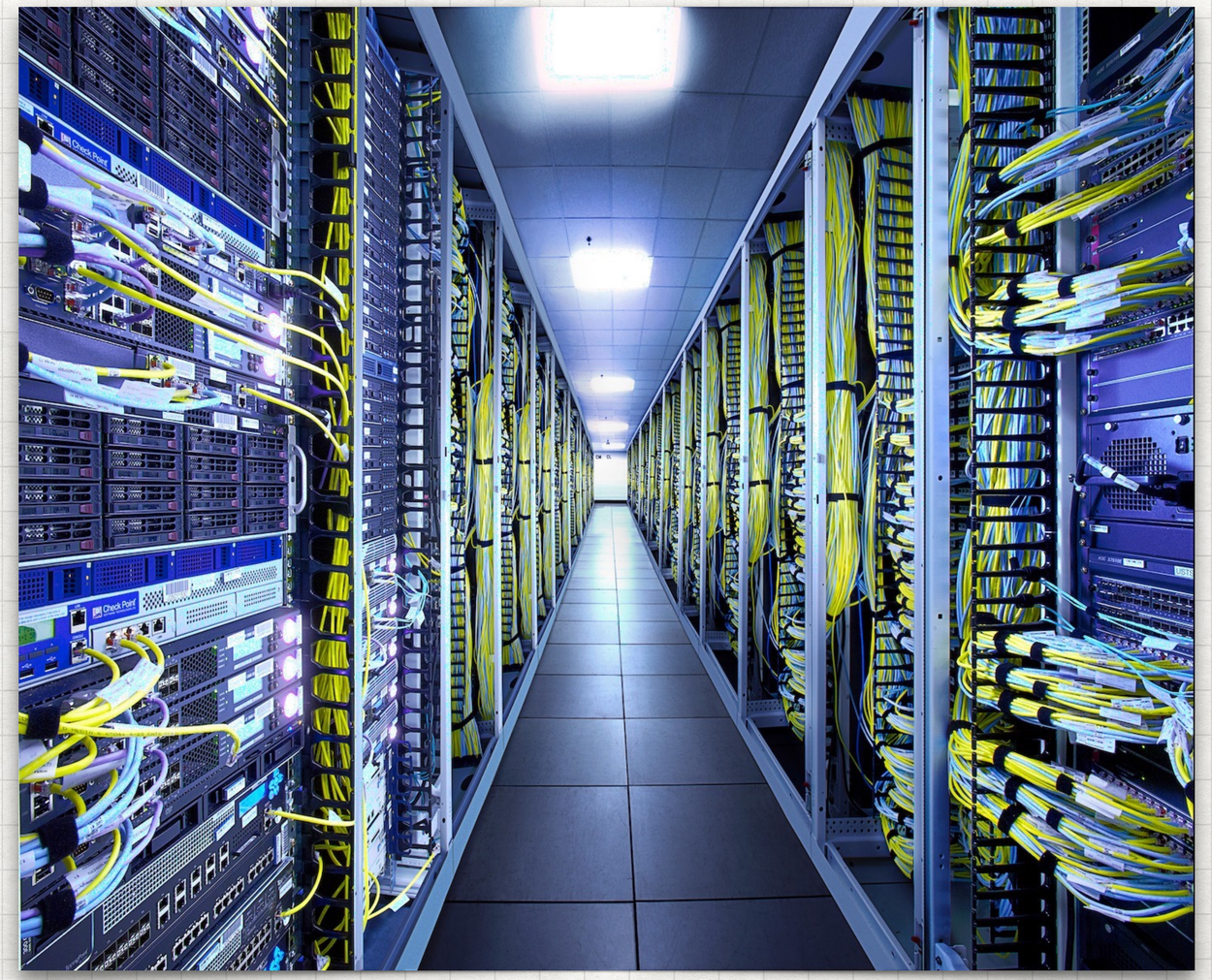
HOW?



A PRIMER:
CACHING IN DATACENTER

DATACENTER

- geographically centralized
- highly homogeneous network
- relatively reliable infrastructure



CACHING

MAINLY:

REQUEST → RESPONSE

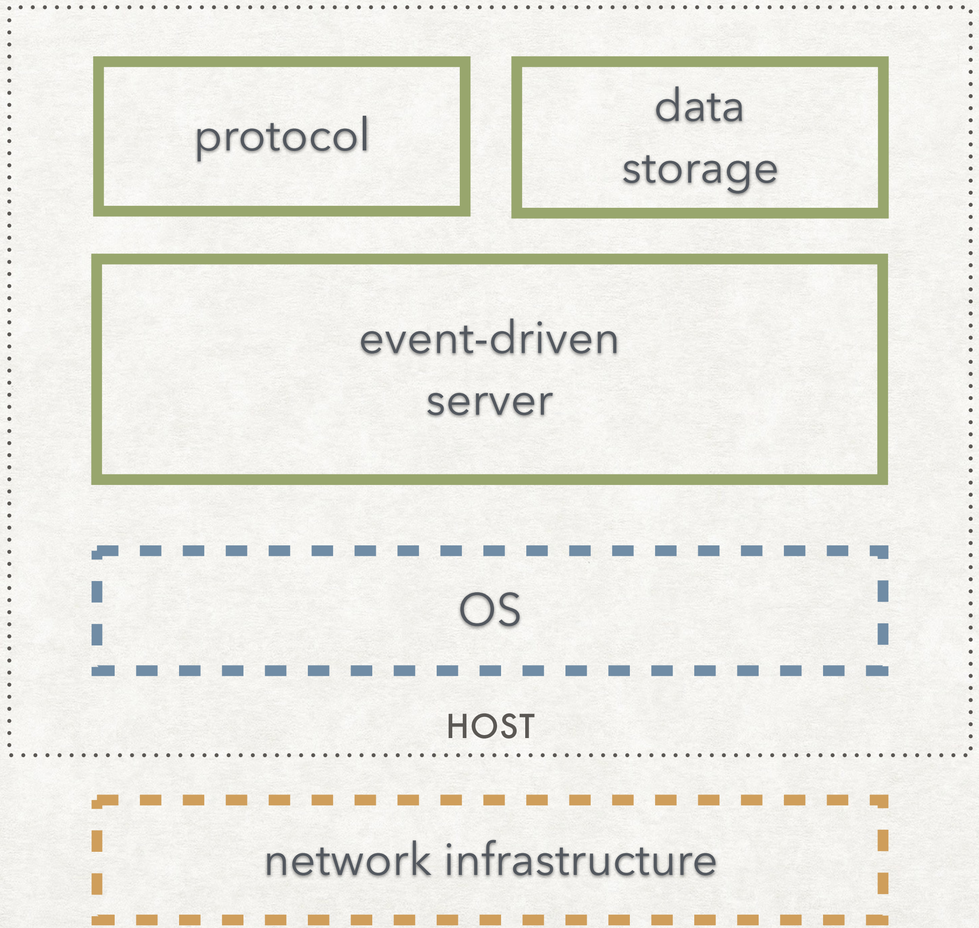
INITIALLY:

CONNECT

ALSO (BECAUSE WE ARE GROWN-UPS):

STATS, LOGGING, HEALTH CHECK...

CACHE SERVER: BIRD'S VIEW



HOW DID WE UNCOVER THE UNCERTAINTIES?

“

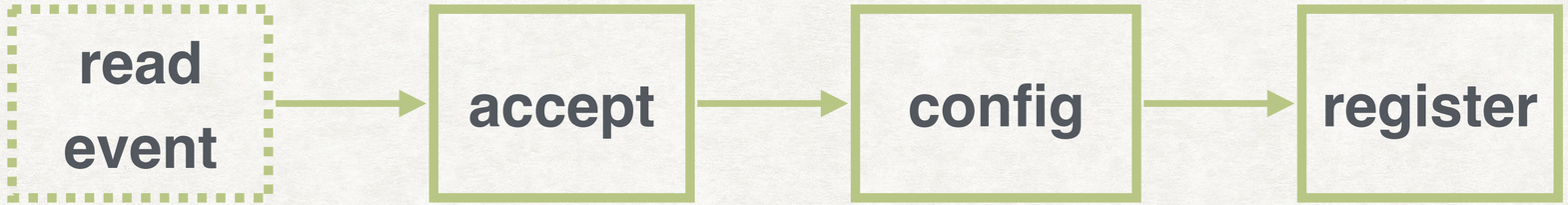
BANDWIDTH UTILIZATION WENT WAY UP, EVEN THOUGH REQUEST RATE WAS WAY LOWER.

”

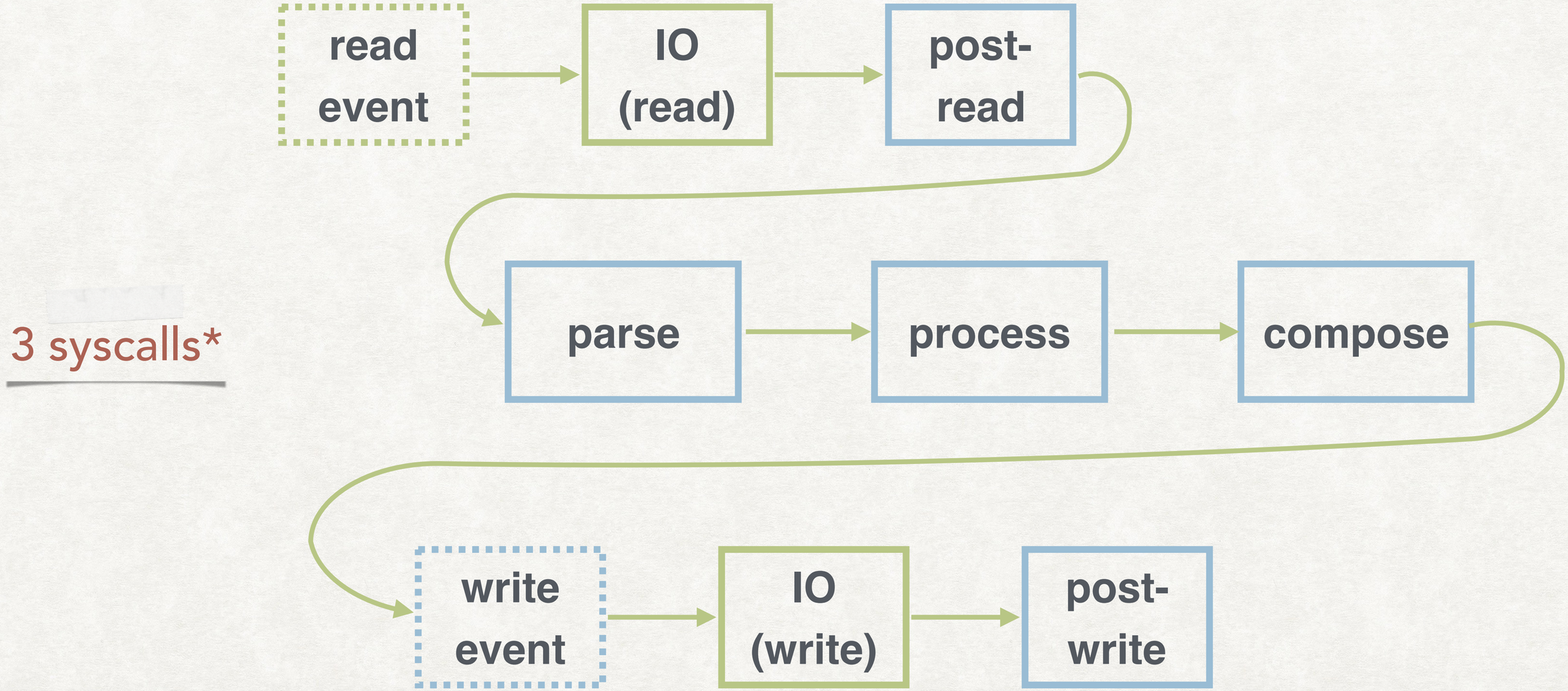
SYSCALLS

CONNECTING IS SYSCALL-HEAVY

4+ syscalls



REQUEST IS SYSCALL-LIGHT



3 syscalls*

*: event loop returns multiple read events at once, I/O syscalls can be further amortized by batching/pipelining

TWEMCACHE IS MOSTLY SYSCALLS

- 1-2 µs overhead per call
- dominate CPU time in simple cache
- What if we have 100k conns / sec?

count	pct	function
1572	52.4%	__sendmsg_nocancel
668	22.3%	__read_nocancel
82	2.7%	__lll_unlock_wake
78	2.6%	__epoll_wait_nocancel
66	2.2%	__pthread_mutex_lock
58	1.9%	assoc_find
48	1.6%	_IO_vfprintf
45	1.5%	__lll_lock_wait
36	1.2%	conn_add_iov
27	0.9%	memchr

source

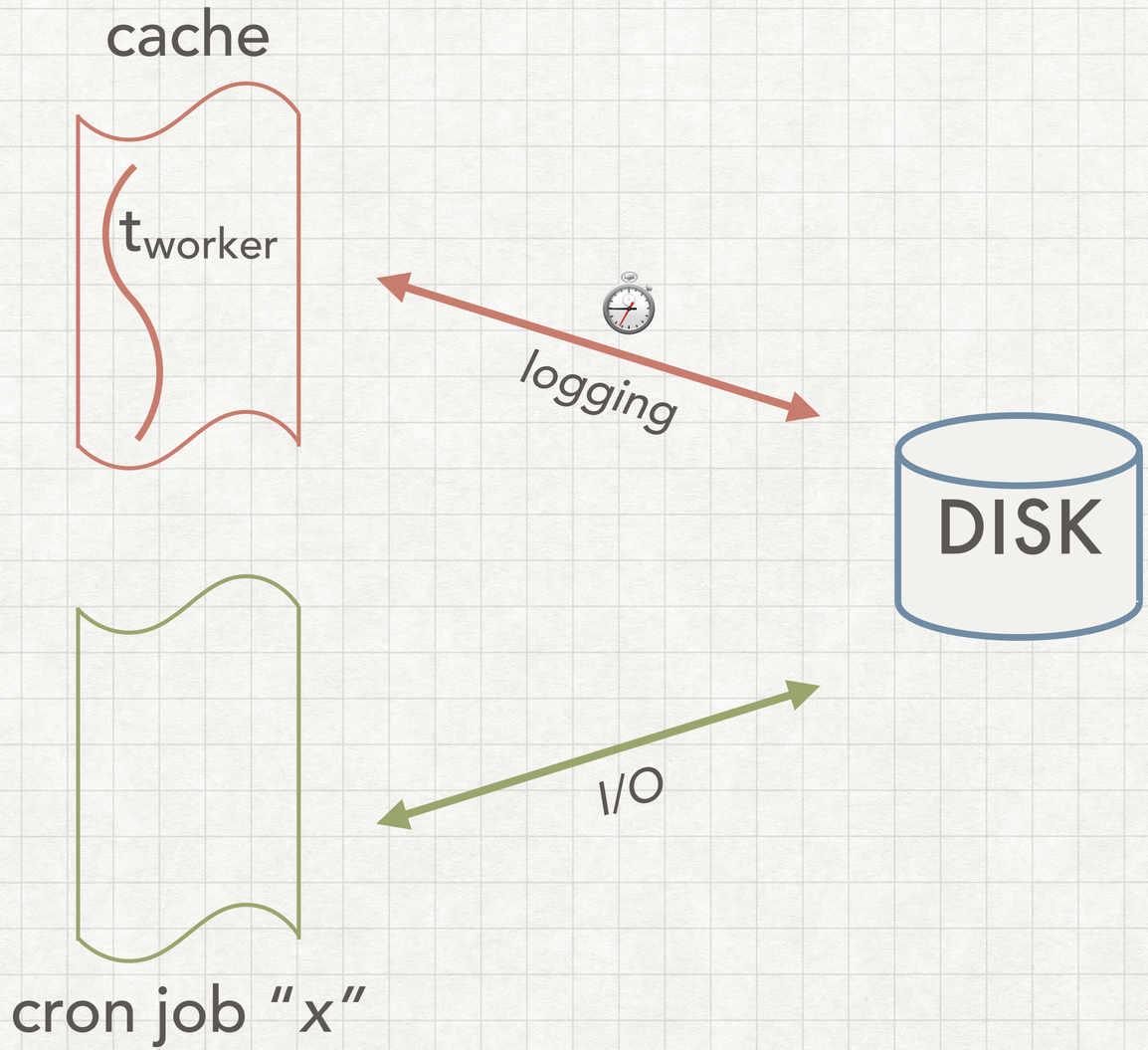
culprit:

CONNECTION STORM

“

…TWEMCACHE RANDOM HICCUPS, ALWAYS AT THE TOP OF THE HOUR.

”


cache
tworker
logging
DISK
I/O
cron job “x”

culprit:

BLOCKING I/O

“

WE ARE SEEING SEVERAL “BLIPS” AFTER EACH CACHE REBOOT…

”

A TIMELINE

MEMCACHE RESTART

...

MANY REQUESTS TIMED OUT

CONNECTION STORM

SOME MORE REQUESTS TIMED OUT

(REPEAT A FEW TIMES)

culprit:

LOCKING

LOCKING FACTS

- ~25ns per operation
- more expensive on NUMA
- much more costly when contended

version	avg	min	max	stddev	p95	p99	p999
top of trunk	0.343	0.065	15.473	0.146	0.570	0.787	1.200
lock-free hashtable	0.262	0.064	12.736	0.094	0.382	0.603	0.860
speedup	30.92%				49.21%	30.51%	39.53%

source

“

HOSTS WITH LONG RUNNING TWEMCACHE/REDIS TRIGGER OOM DURING LOAD SPIKES.

”

“

REDIS INSTANCES THAT STARTED EVICTING SUDDENLY GOT SLOWER.

”

culprit:

MEMORY LAYOUT / OPS

SUMMARY

CONNECTION STORM
BLOCKING I/O

LOCKING
MEMORY

HOW TO MITIGATE?

HIDE EXPENSIVE OPS

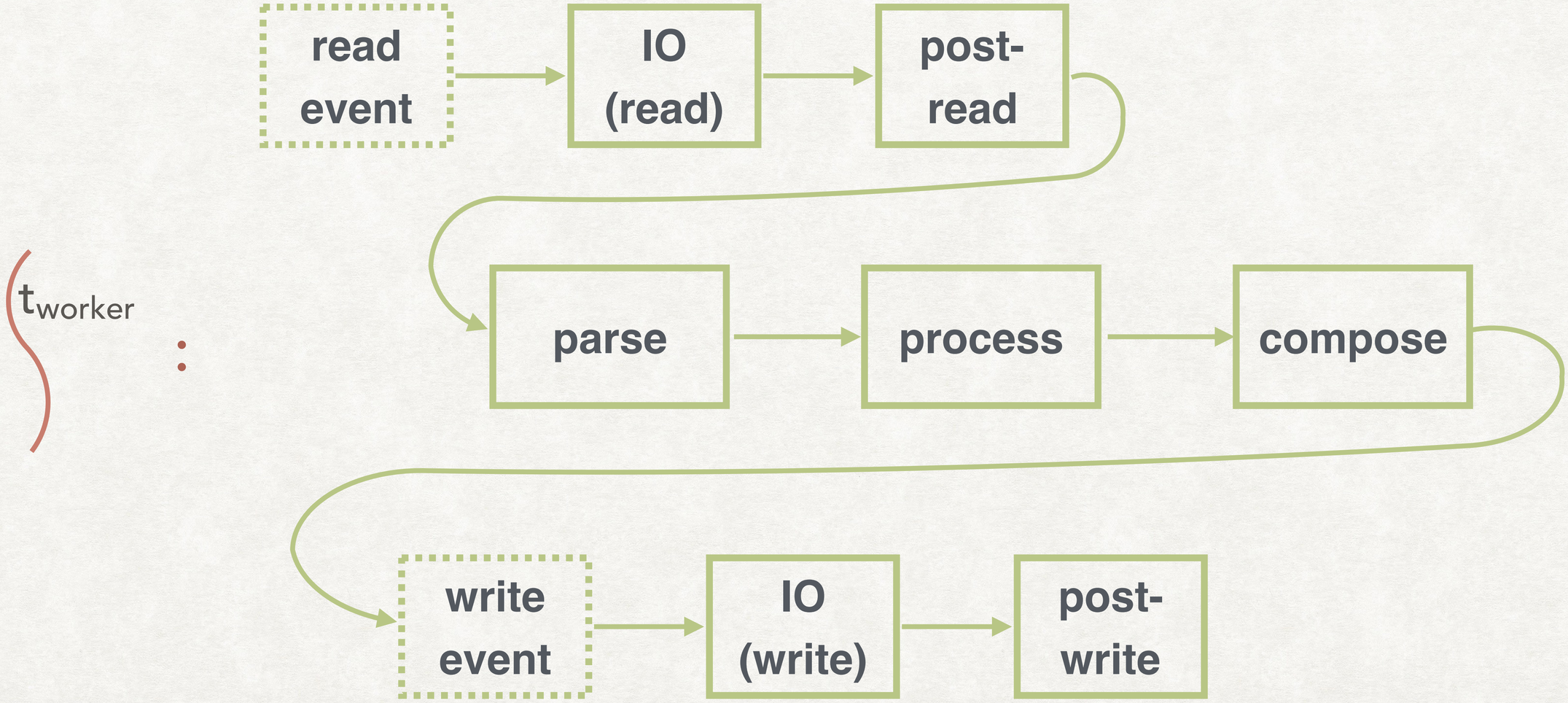
PUT OPERATIONS OF DIFFERENT NATURE / PURPOSE ON *SEPARATE THREADS*

DATA PLANE, CONTROL PLANE

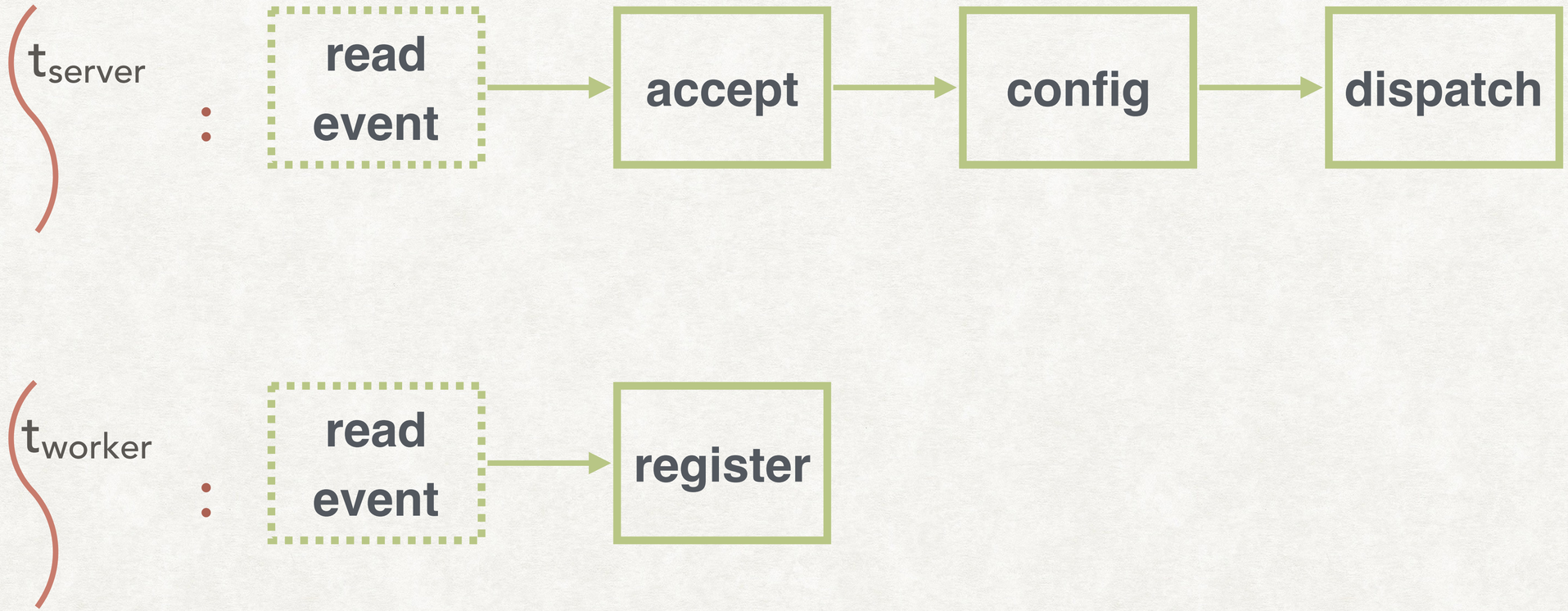
SLOW: CONTROL PLANE

STATS AGGREGATION
STATS EXPORTING
LOG DUMP
LOG ROTATION
...

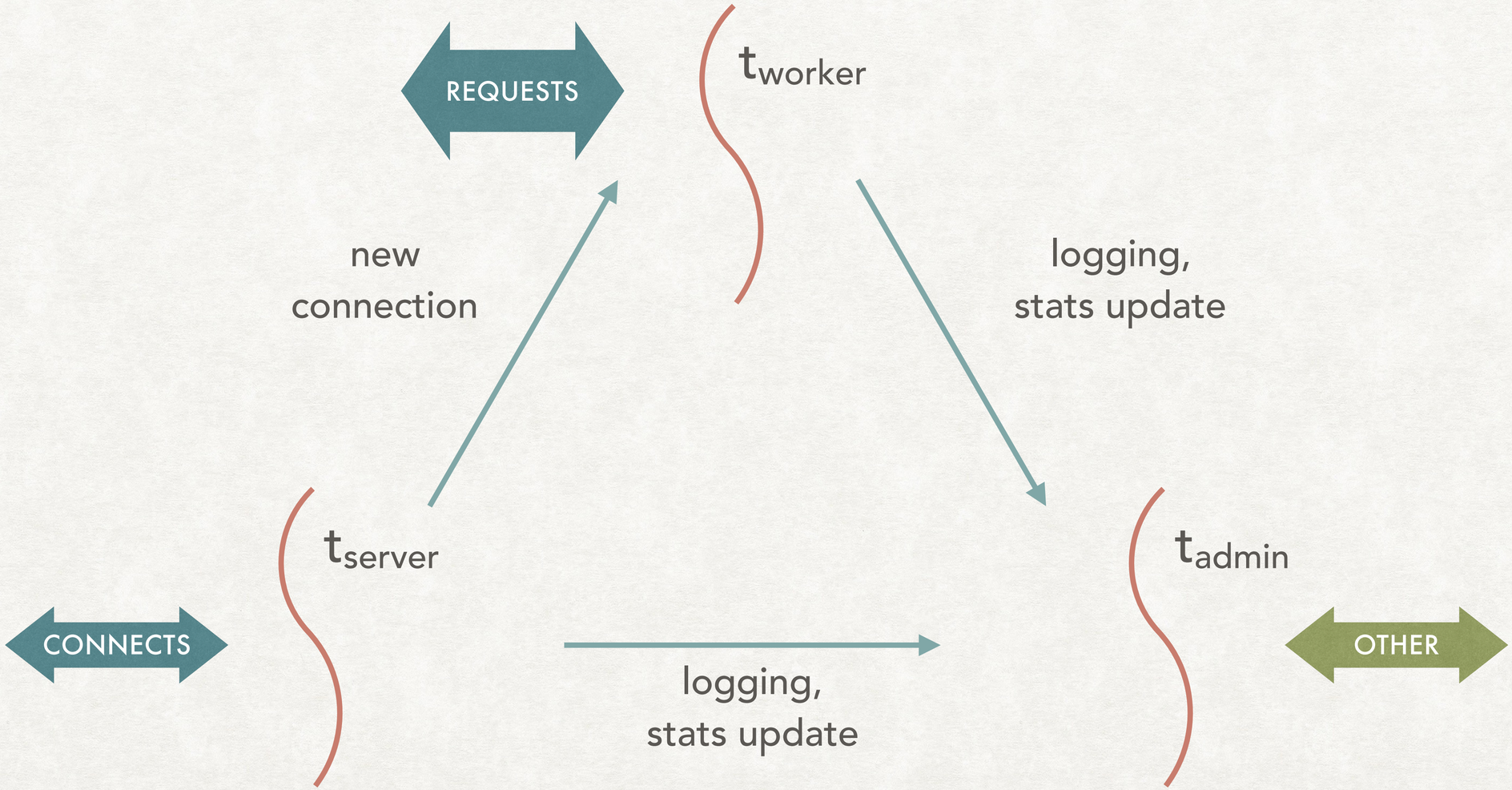
FAST: DATA PLANE / REQUEST



FAST: DATA PLANE / CONNECT



LATENCY-ORIENTED THREADING

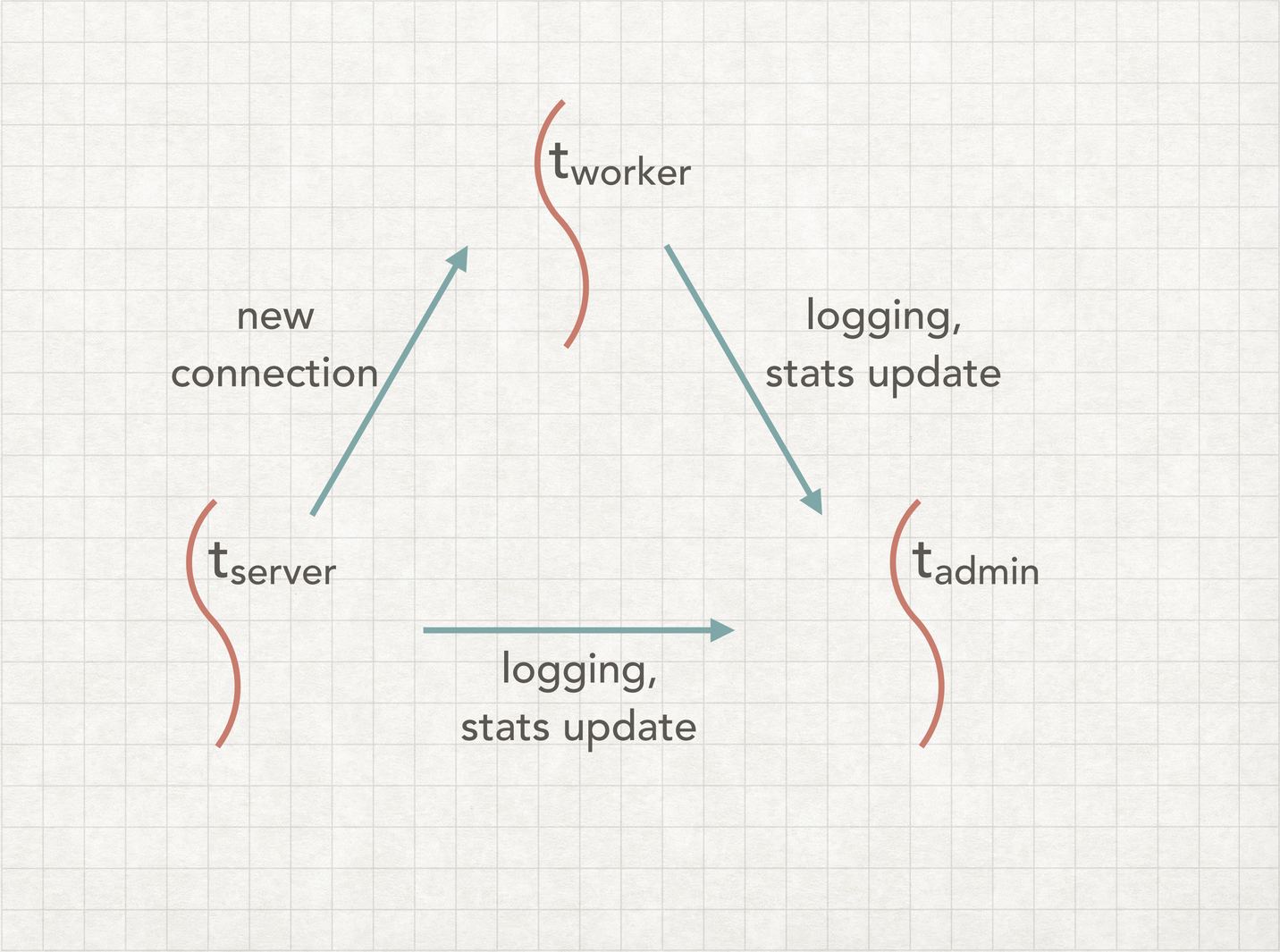


WHAT TO AVOID?

LOCKING

WHAT WE KNOW

- inter-thread communication in cache
  - stats
  - logging
  - connection hand-off
- locking propagates blocking/delay between threads



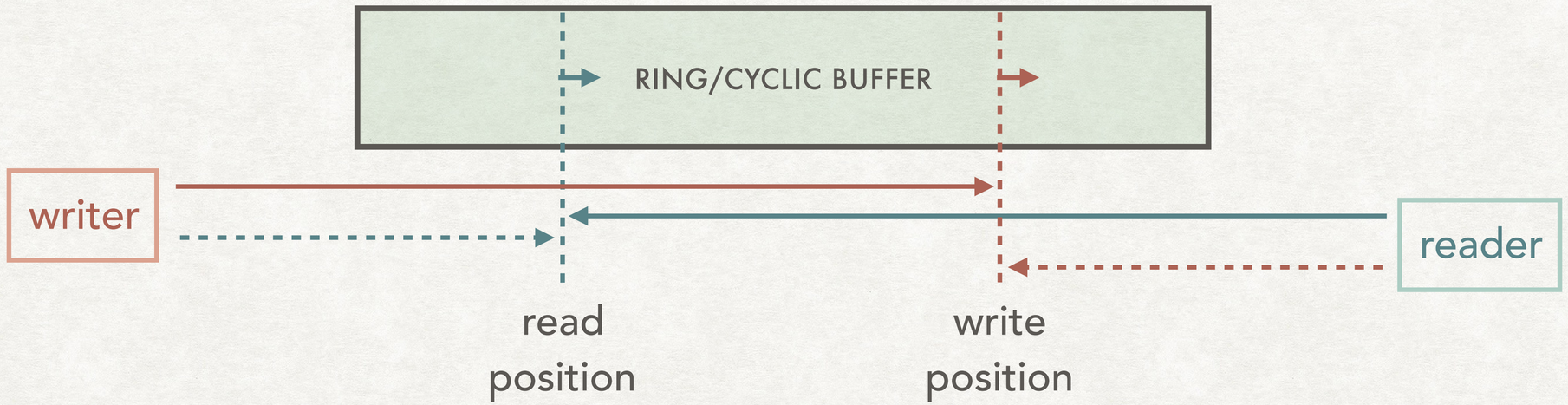
LOCKLESS OPERATIONS

MAKE STATS UPDATE LOCKLESS

w/ atomic instructions

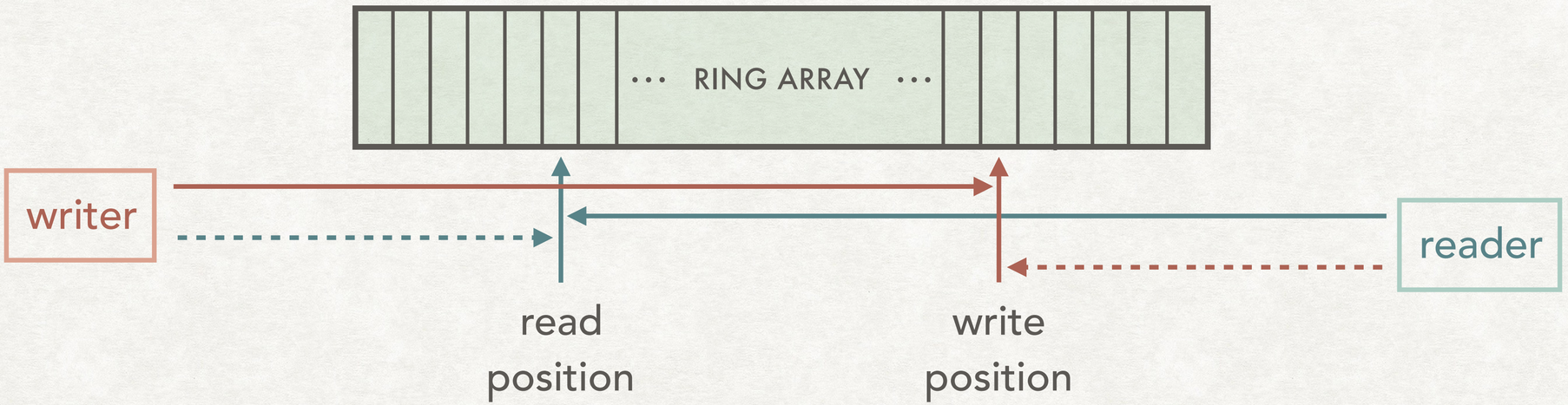
LOCKLESS OPERATIONS

MAKE LOGGING LOCKLESS



LOCKLESS OPERATIONS

MAKE CONNECTION HAND-OFF LOCKLESS

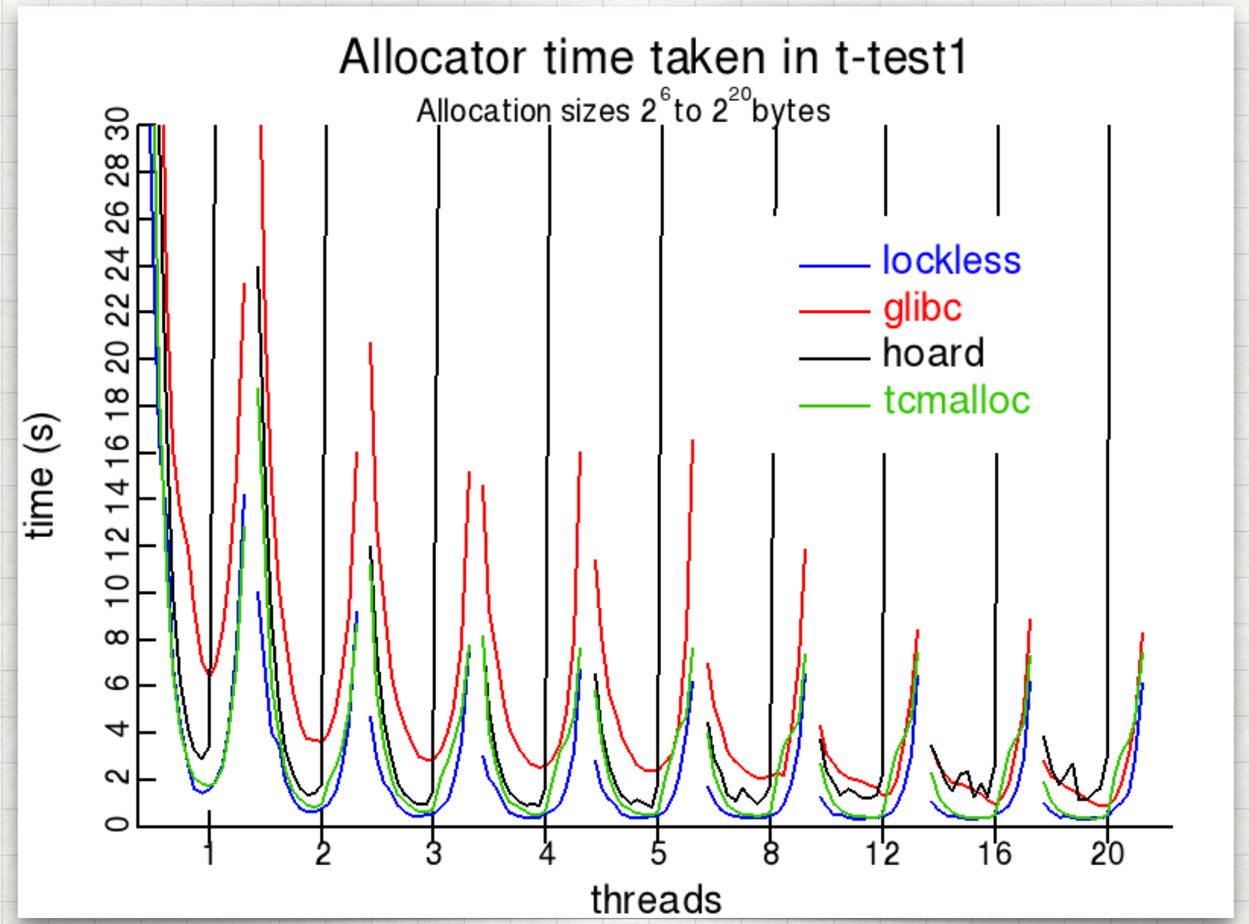


MEMORY

WHAT WE KNOW

- alloc-free cause fragmentation
- internal vs external fragmentation
- OOM/swapping is deadly

- memory alloc/copy relatively expensive



source

PREDICTABLE FOOTPRINT

AVOID EXTERNAL FRAGMENTATION
CAP ALL MEMORY RESOURCES

PREDICTABLE RUNTIME

REUSE BUFFER
PREALLOCATE

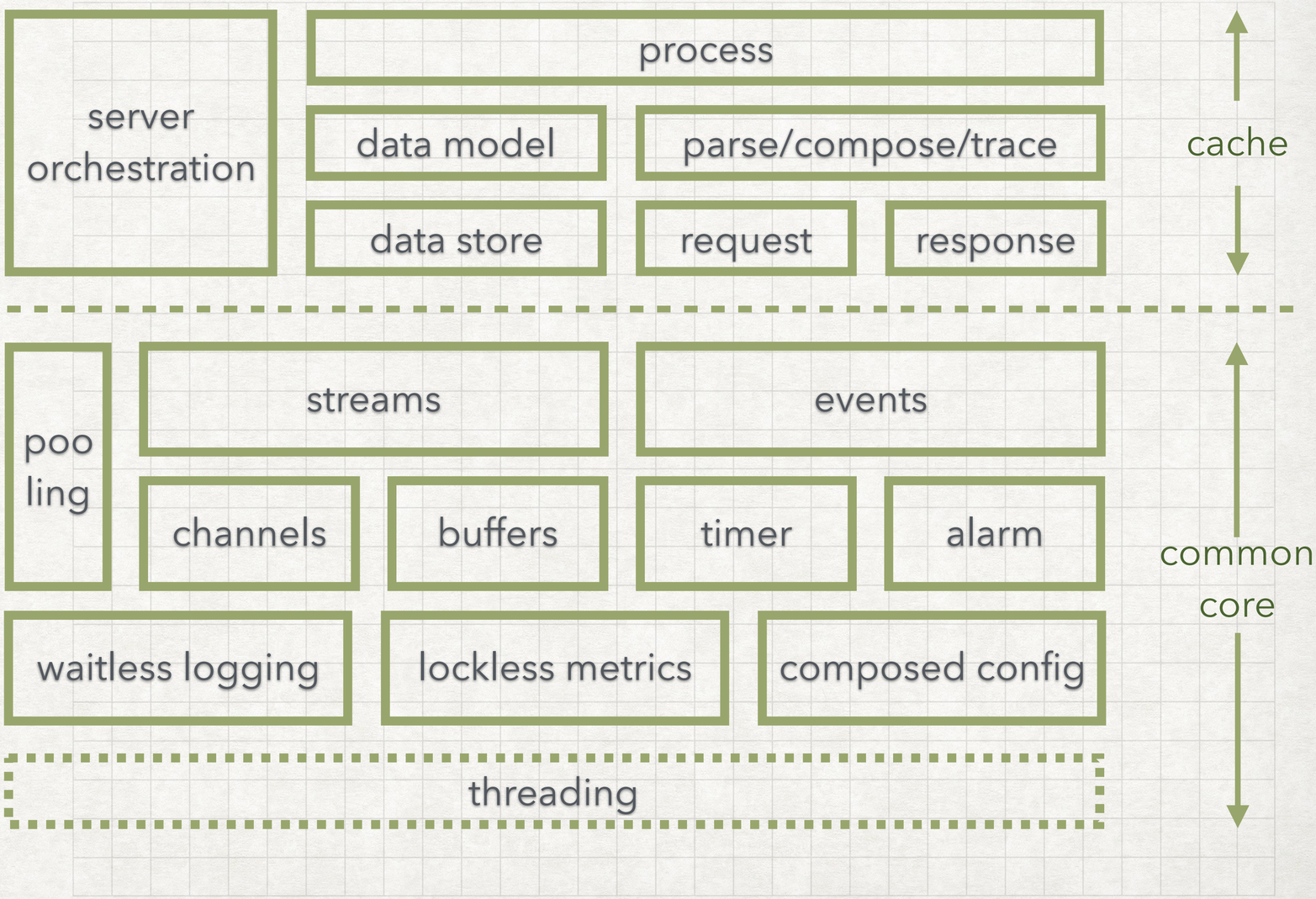
IMPLEMENTATION

PELIKAN CACHE

WHAT IS PELIKAN CACHE?

- (Datacenter-) Caching framework
- A summary of Twitter's cache ops
- Perf goal: deterministically fast
- Clean, modular design
- Open-source

pelikan.io



PERFORMANCE DESIGN DECISIONS

A COMPARISON

	latency-oriented threading	Memory/ fragmentation	Memory/ buffer caching	Memory/ pre-allocation, cap	locking
Memcached	partial	internal	partial	partial	yes
Redis	no->partial	external	no	partial	no->yes
Pelikan	yes	internal	yes	yes	no

TO BE FAIR...

MEMCACHED

- multiple threads can boost throughput
- binary protocol + SASL

REDIS

- rich set of data structures
- RDB
- master-slave replication
- redis-cluster
- modules
- tools

SCALABLE CACHE IS...

ALWAYS FAST

“

CAREFUL ABOUT MOVING TO MULTIPLE WORKER THREADS

”

QUESTIONS?