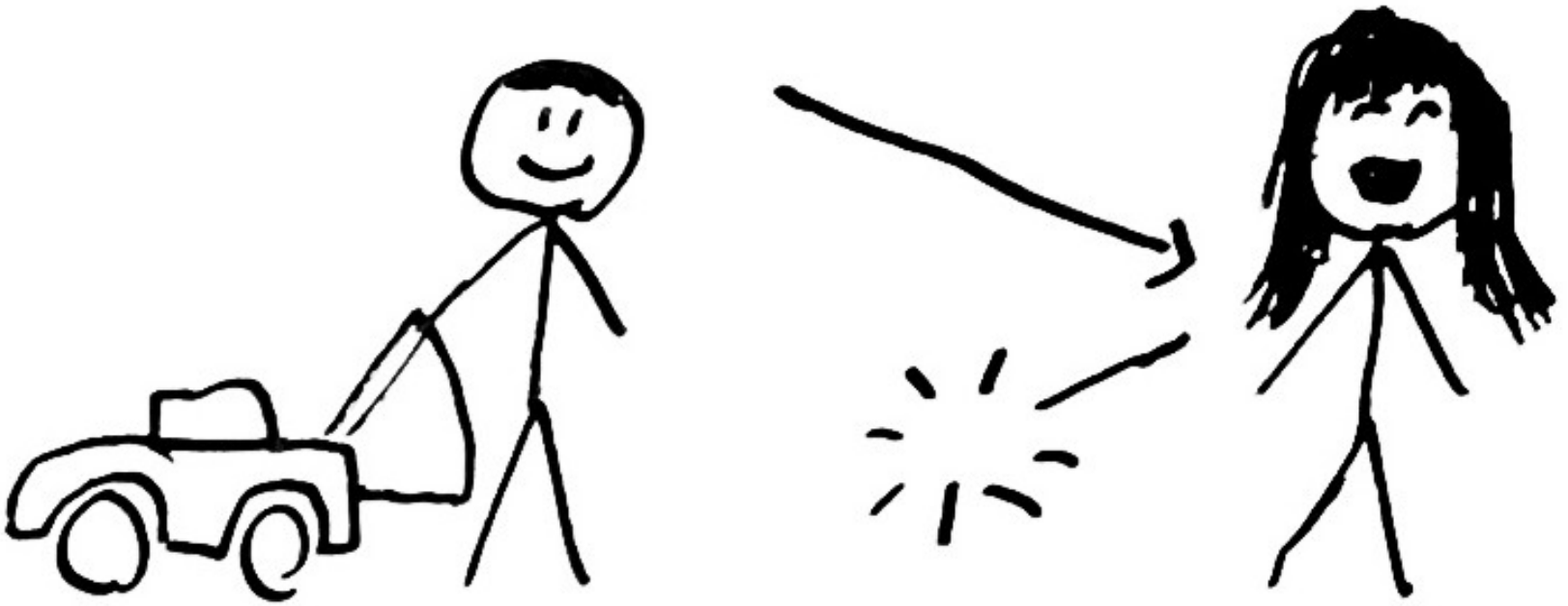
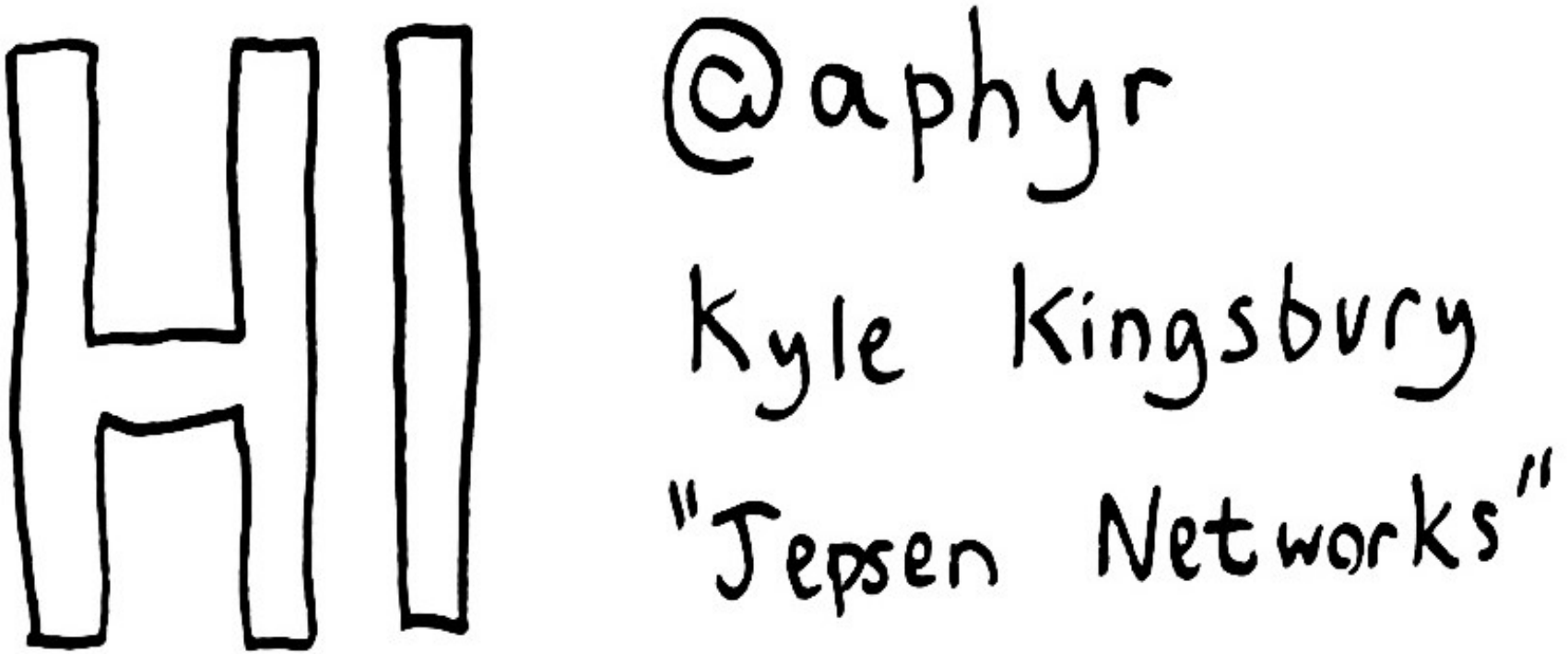
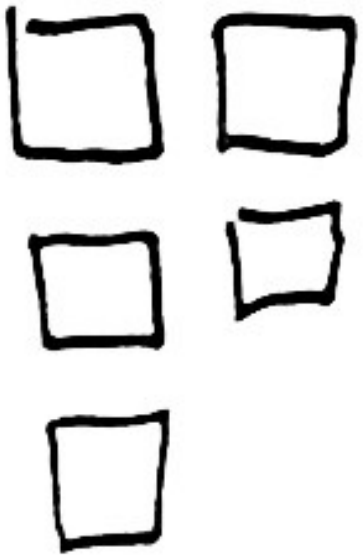
Call Me Maybe!




HI
@aphyr
Kyle Kingsbury
"Jepsen Networks"

Backend engineer at



Factual

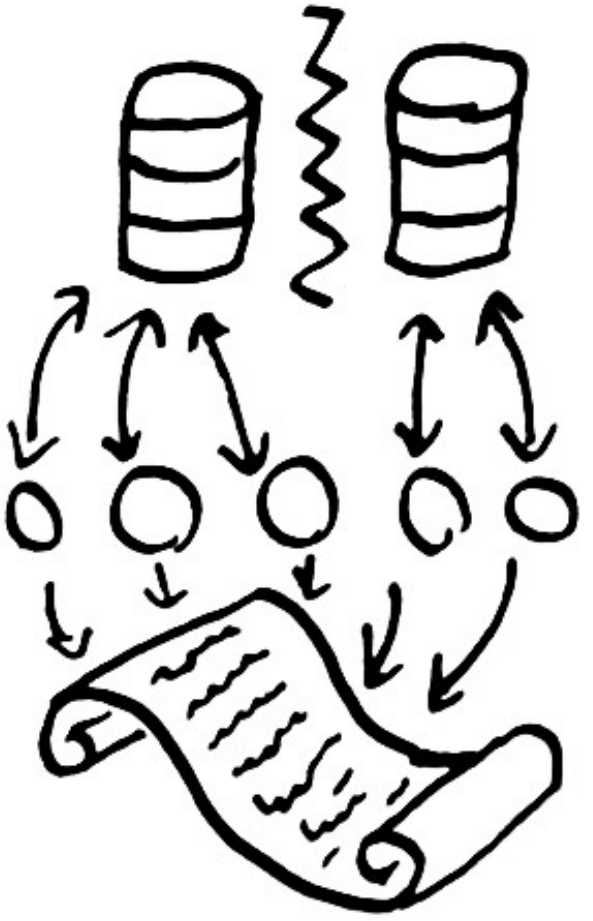
Jepsen II

Linearizable

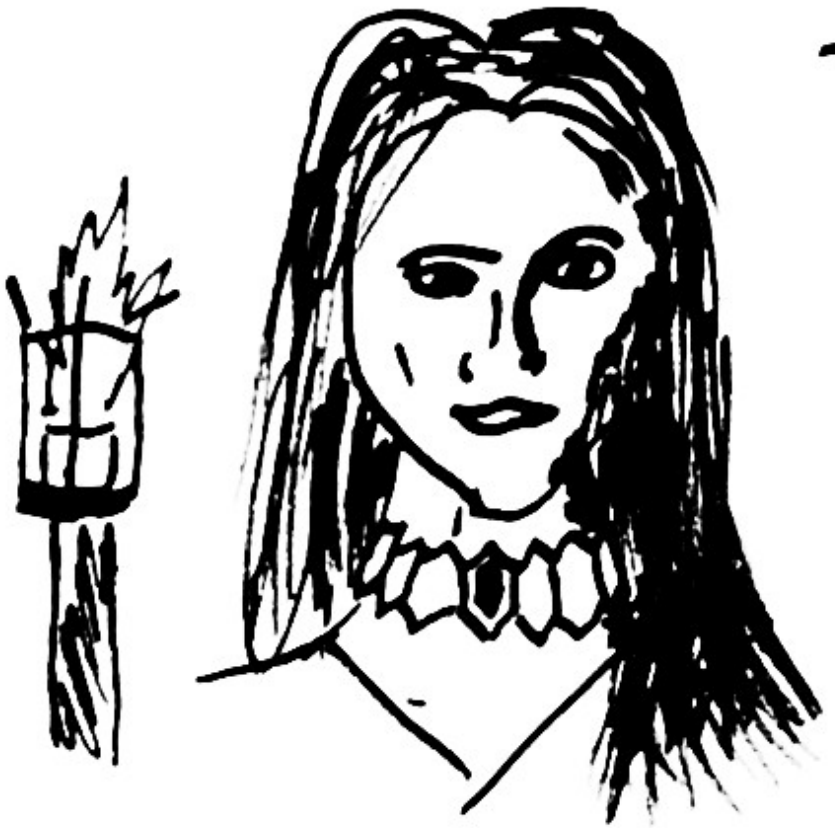
Boogaloo

Jepsen

Measures
Systems
Under
Stress

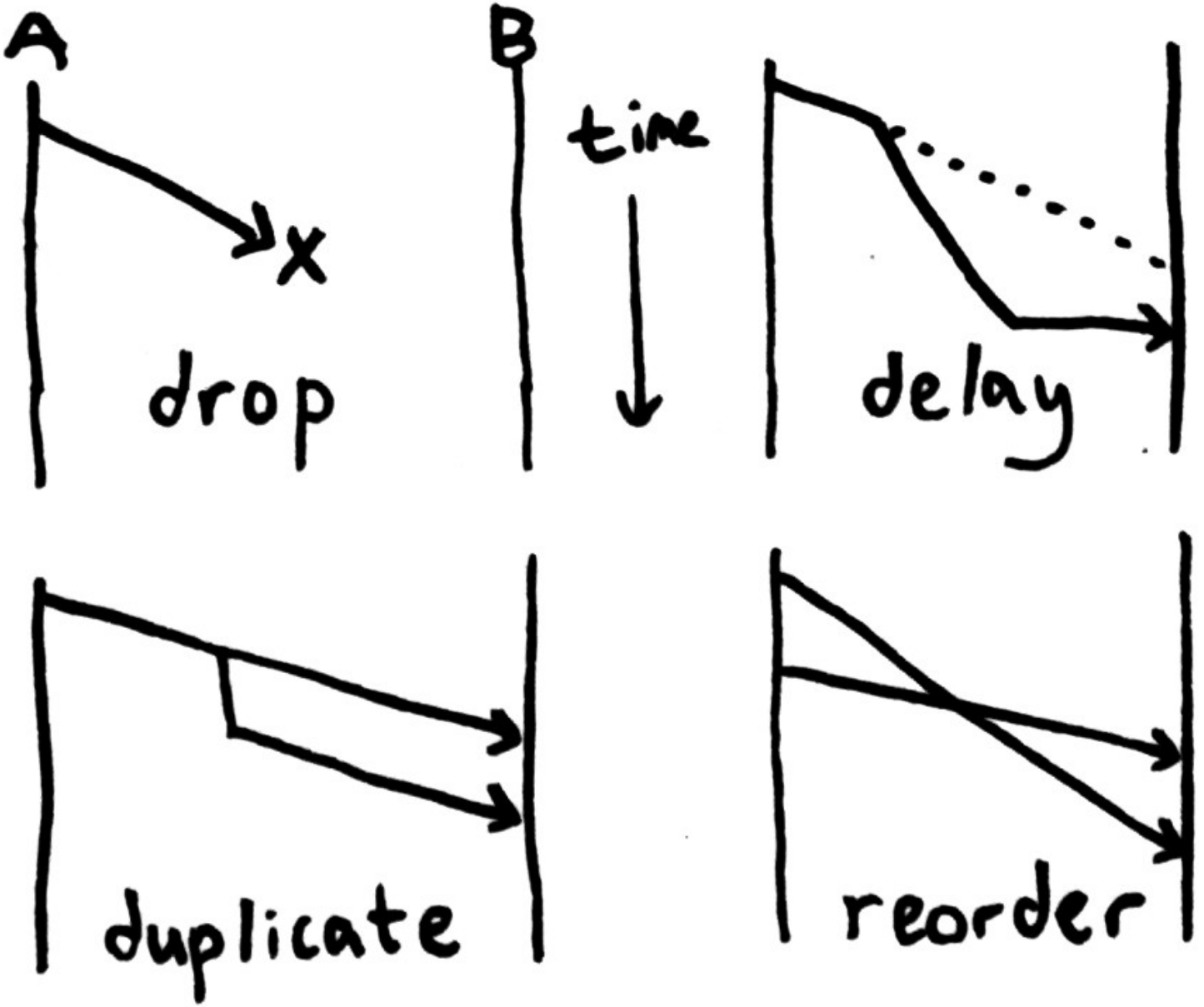


Why Care?



The net is dark, and full of TERAHS

Formally:



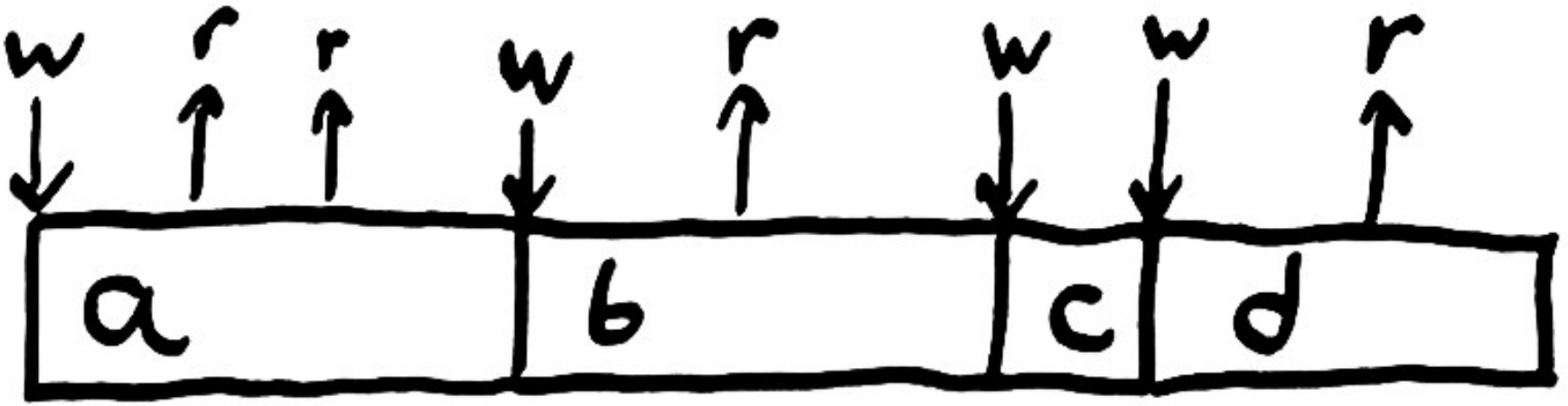
Causes

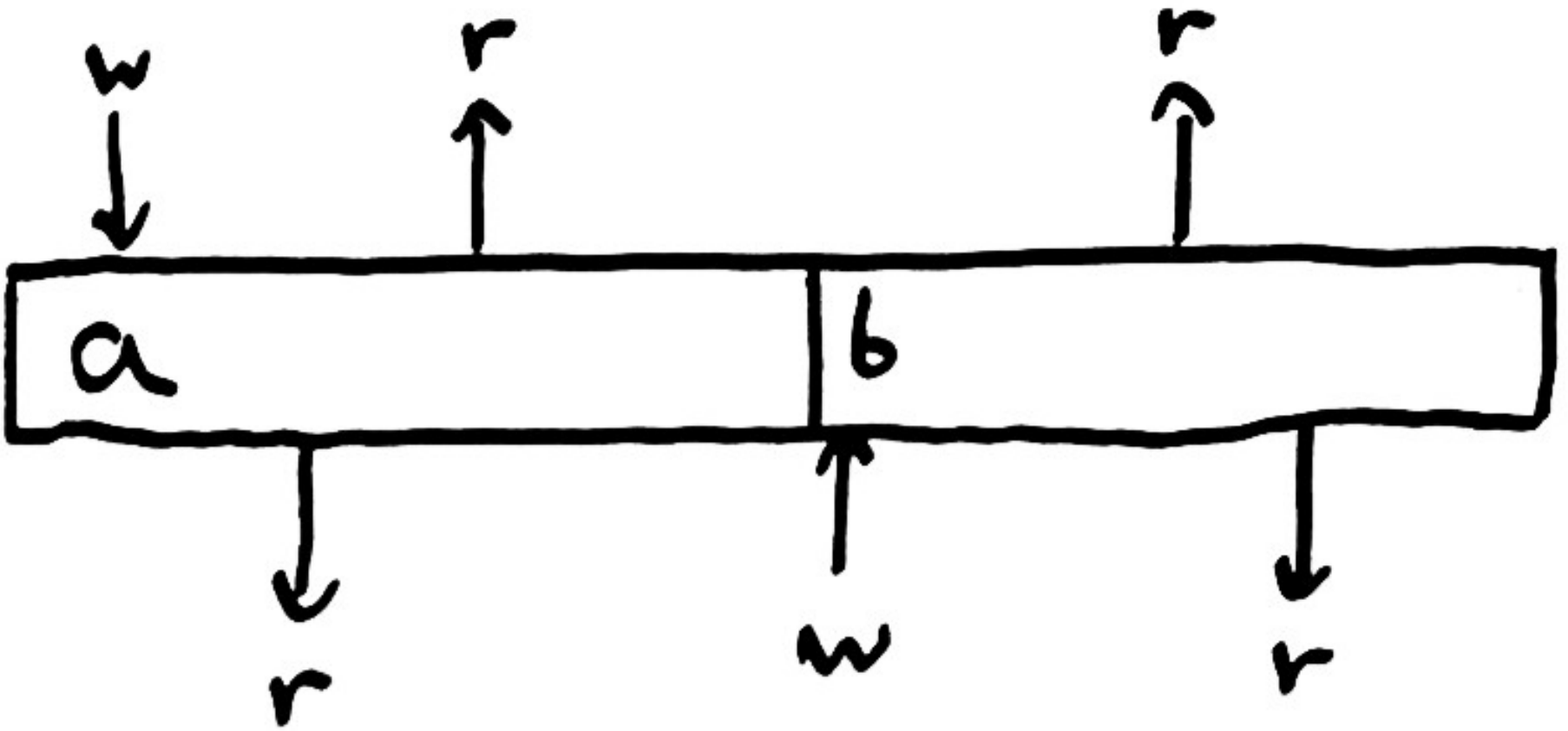
- GC pauses
- Network maint
- Segfaults & crashes
- Faulty NICs
- Bridge loops

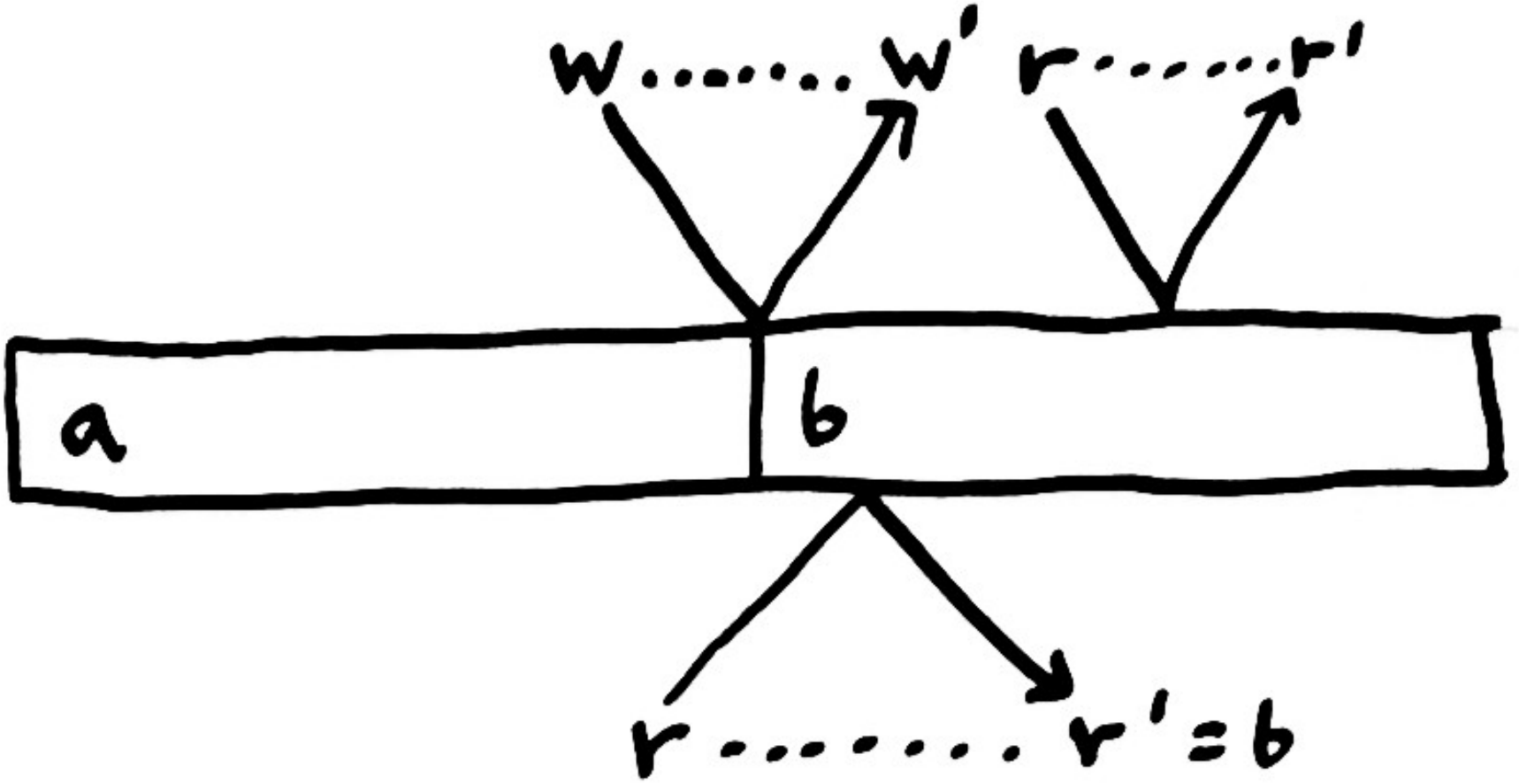
... continued

- Spanning tree
- Hosted networks
- The cloud
- WAN links & Backhoes

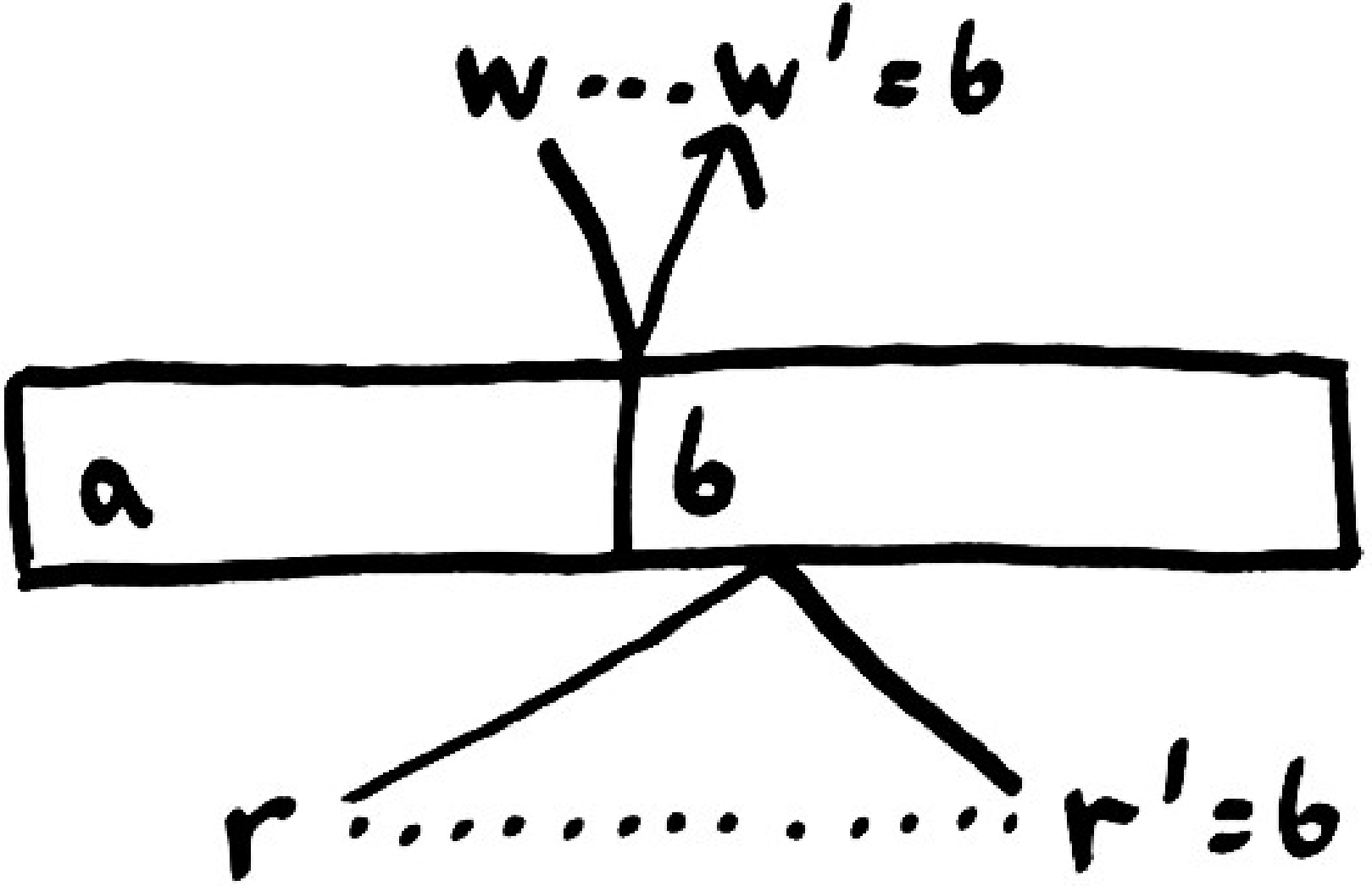
Intuitive Correctness


w
r
r
w
r
w
w
r
a
b
c
d

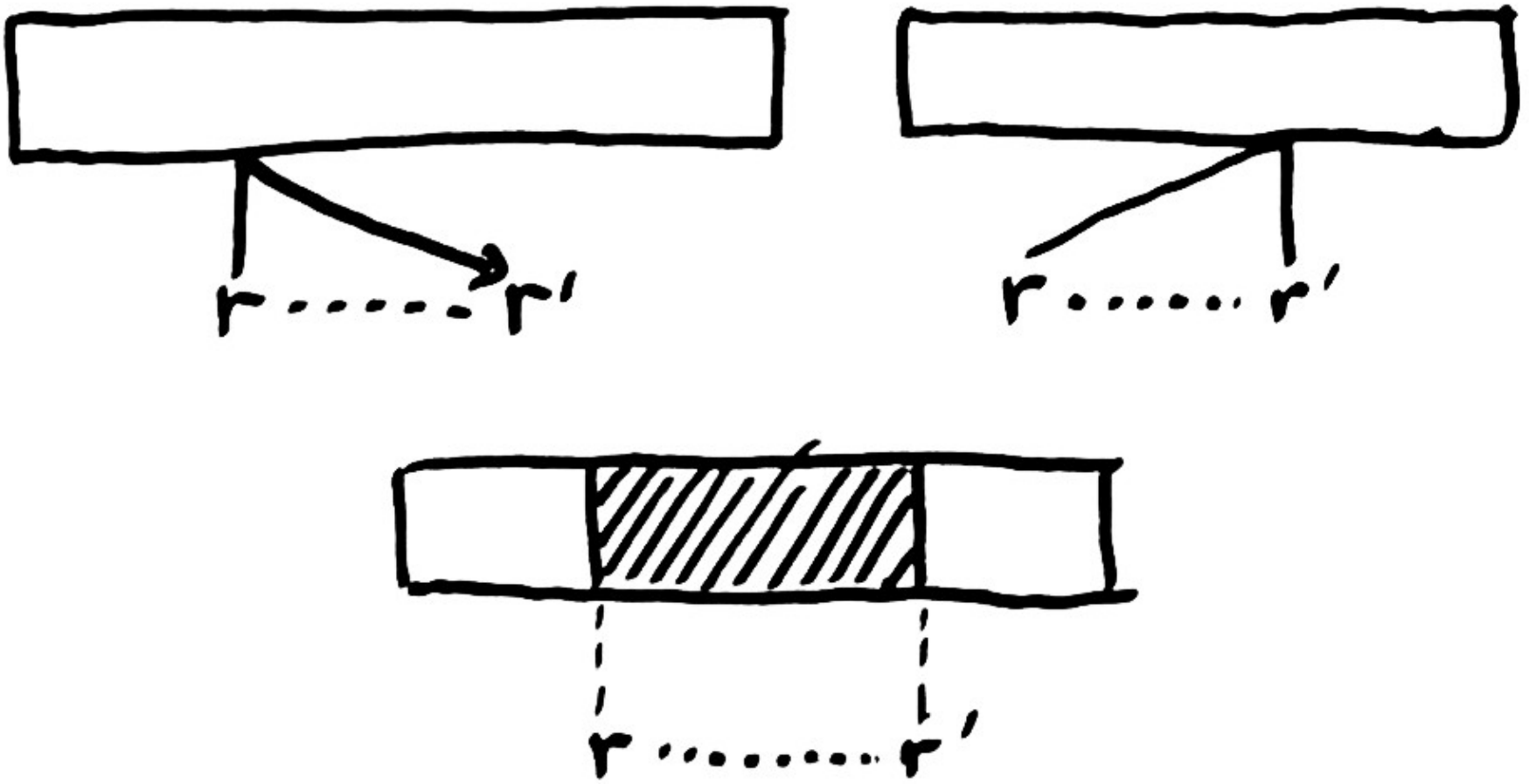

w
r
r
a
b
r
w
r


w
w'
r
r'
a
b
r
r'=b

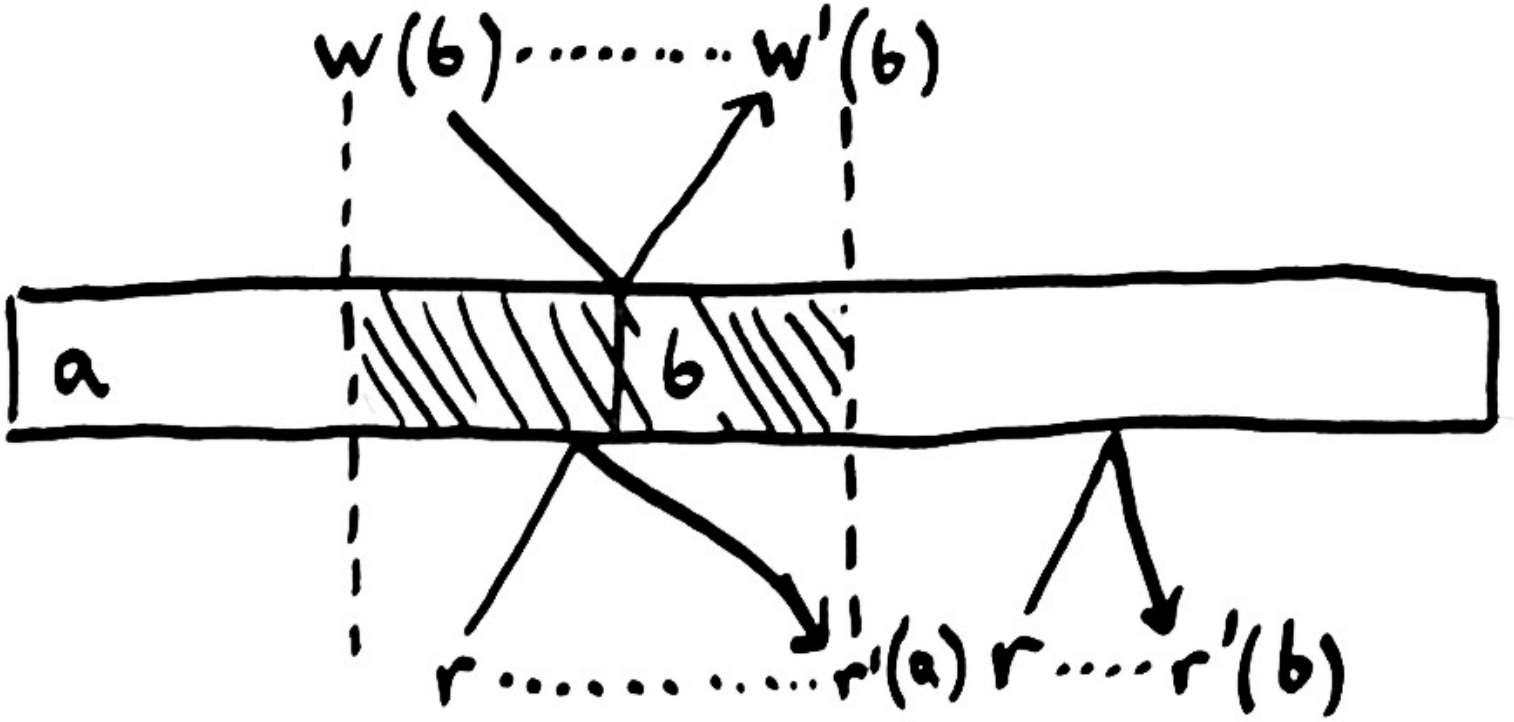
Synchronization
delay implies
ambiguous orders


w ... w' = b
a
b
r ..... r' = b

But there are finite bounds!



Once operation is complete,
it <u>will</u> be visible to all.



No stale reads

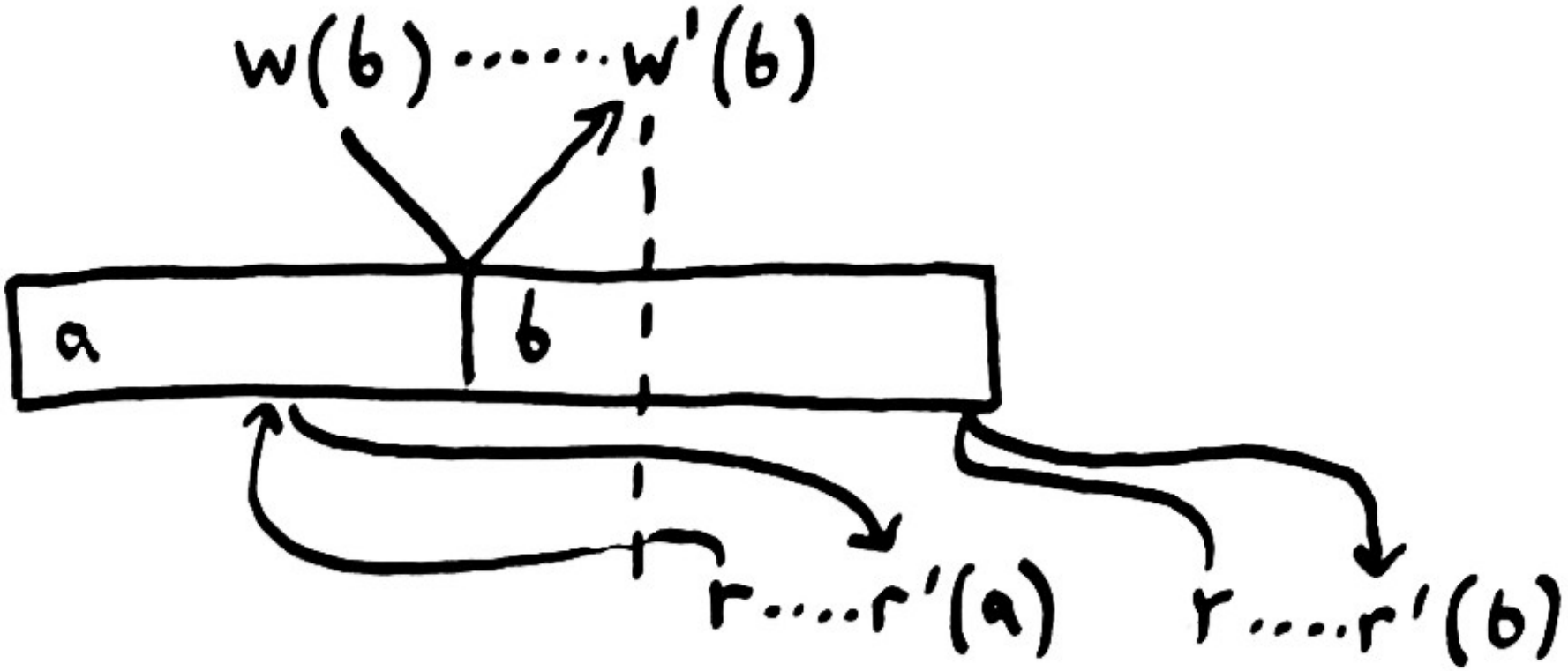
Safe state mutation

Everyone sees the same order of states.

Same properties as atomic ops in a multithreaded lang.

Sequential Consistency:

Everyone agrees on order, but not on time.

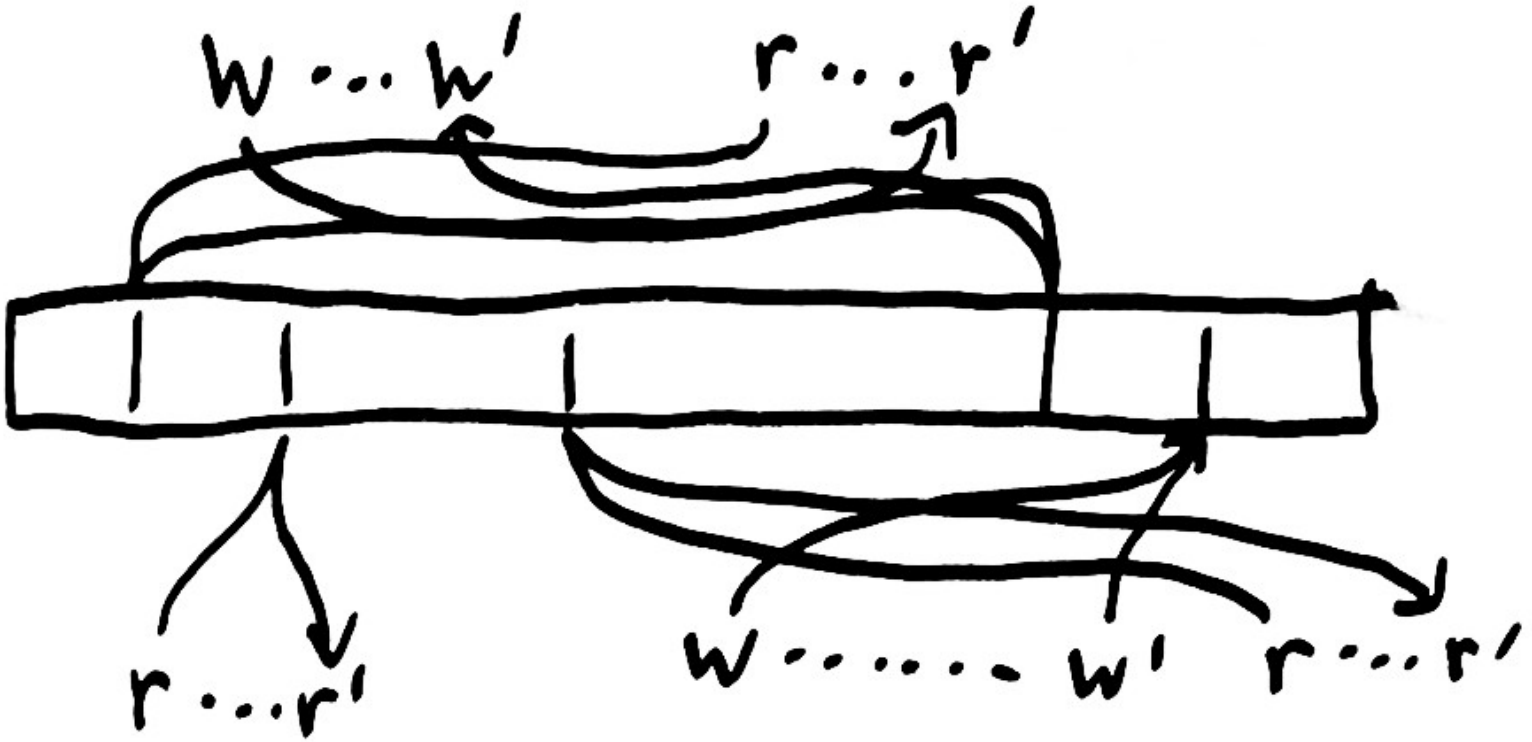


"lines can't cross"

Serializability

"Everything happens in some order, but nobody agrees on what that order was!"

Serializability



"ACID consistency/isolation"

"Acronyms are easy
to make shit up about."

-MRB

CAP:

C = Linearizability

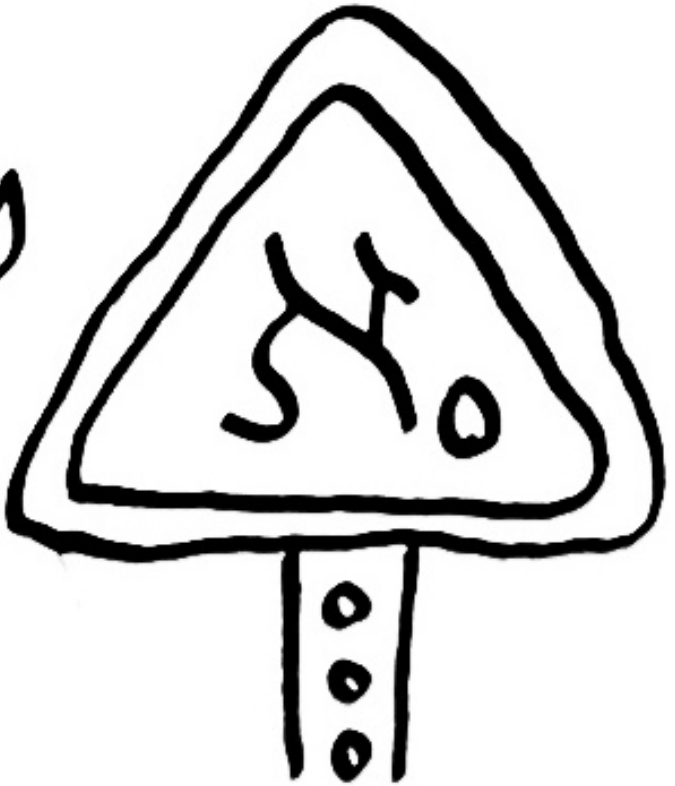
A = All up nodes can satisfy all requests

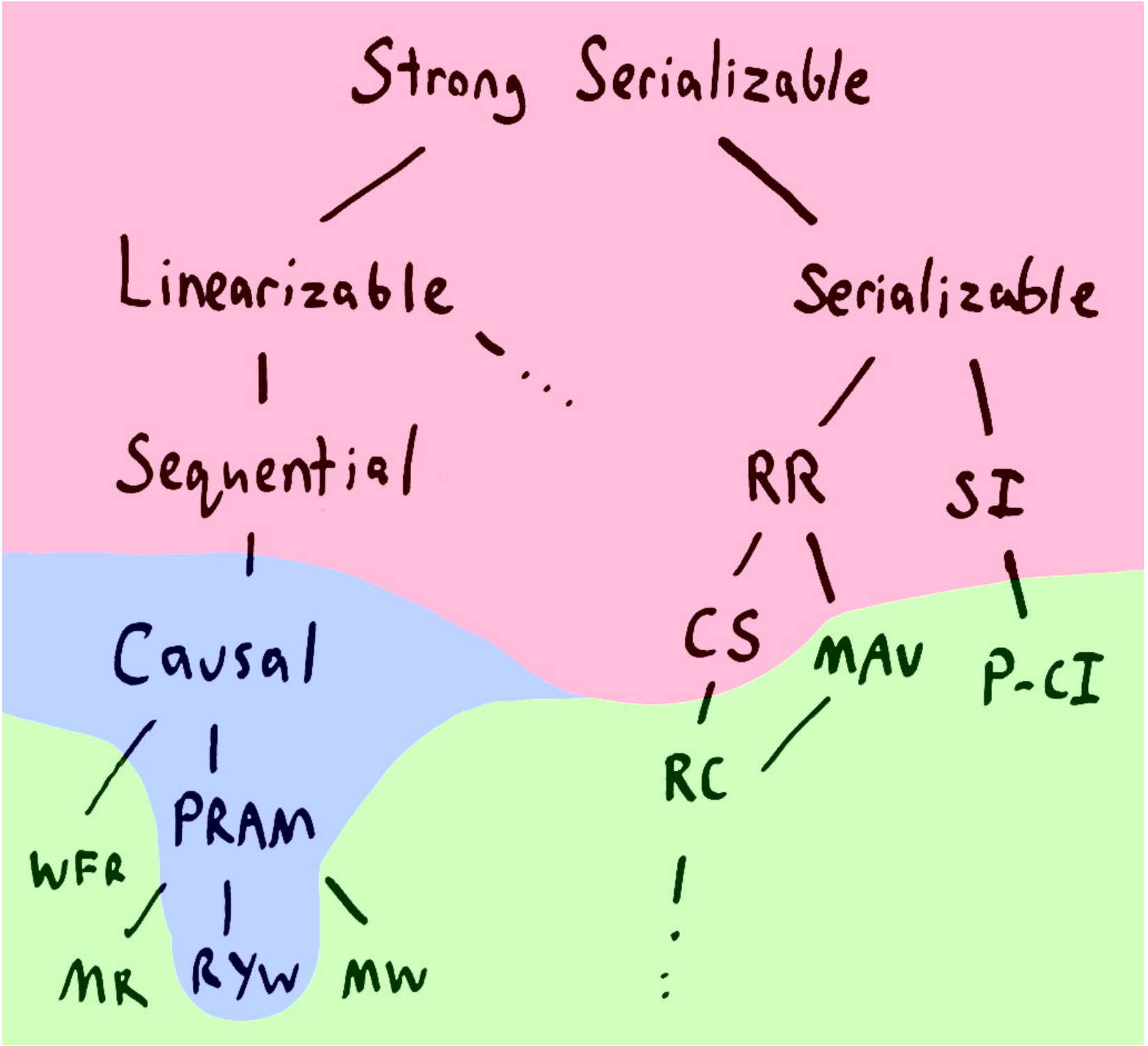
P = Mandatory

Pick AP or CP.

Anyone who claims CA has no idea what they're doing.

Infinitely many models!




Strong Serializable
Linearizable
Serializable
Sequential
RR
SI
Causal
CS
MAV
P-CI
RC
PRAM
WFR
MR
RYW
MW

Weaker consistency models are more available in failure.

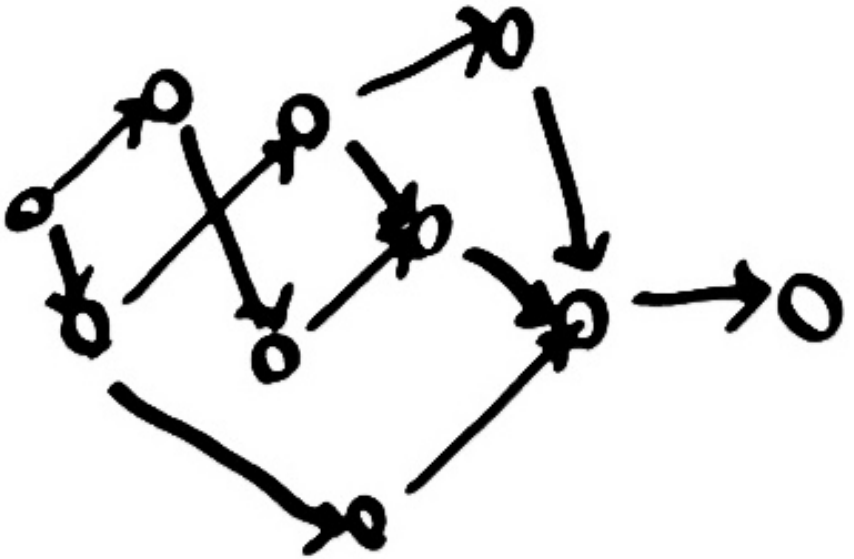
Weaker consistency models are less intuitive.

Weaker consistency models are faster.

- CPU memory model

"Weak" ≠ Unsafe

CRDTs



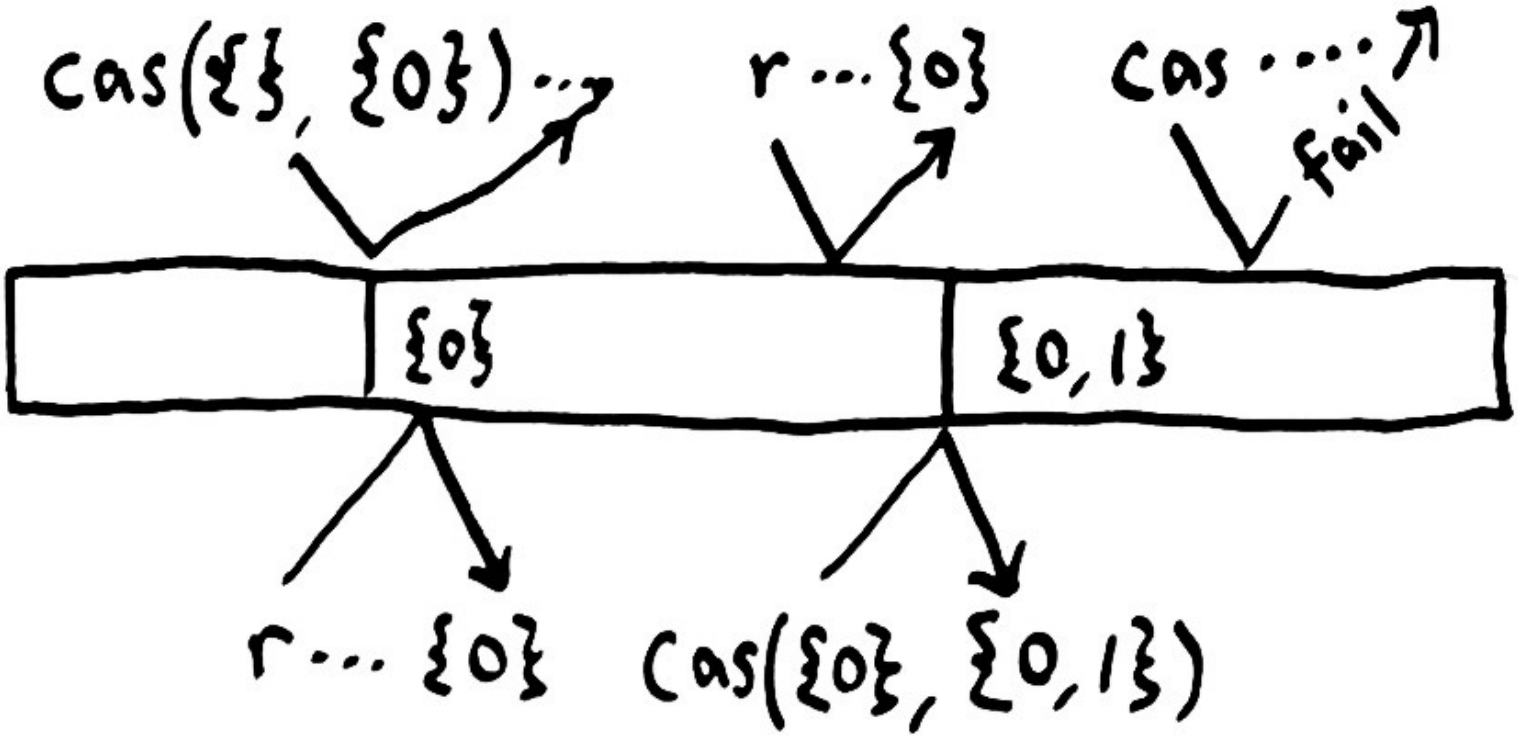
Zookeeper

NuoDB

Kafka

Cassandra

Jepsen failed to find problems in etcd, and other strongly consistent systems. Why?



cas, cas, cas, cas, ... read final

+0 +1 +2 +3 ↑ {0, 1, 2, 3, ...}

?

Jepsen: DB automation, testing, failures, scheduling ops.

↓

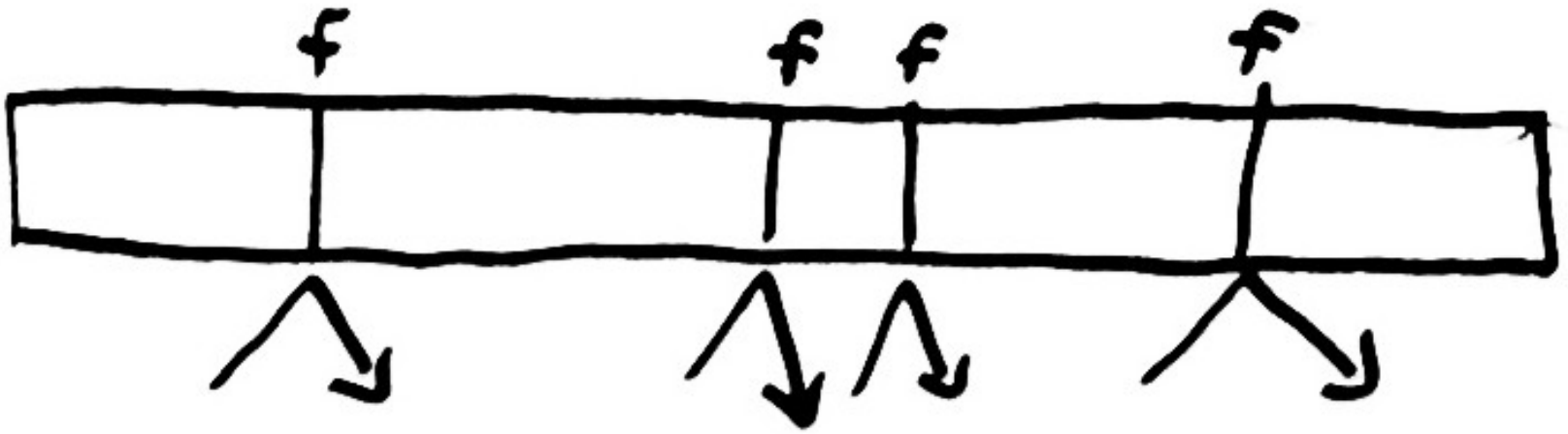
History of Ops

↓

Knossos: Verifies Linearizability

Model

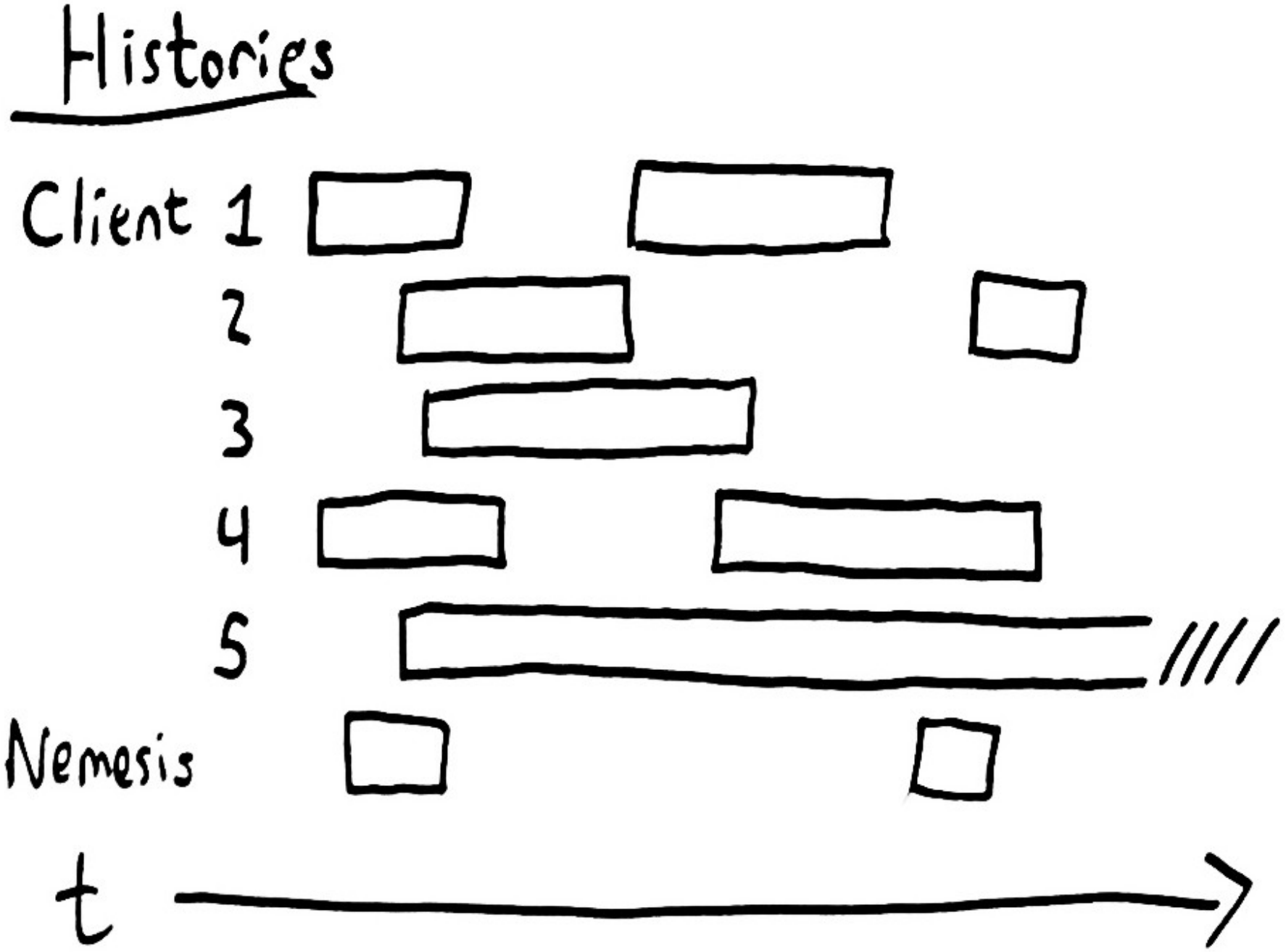
f(state, op) → state'

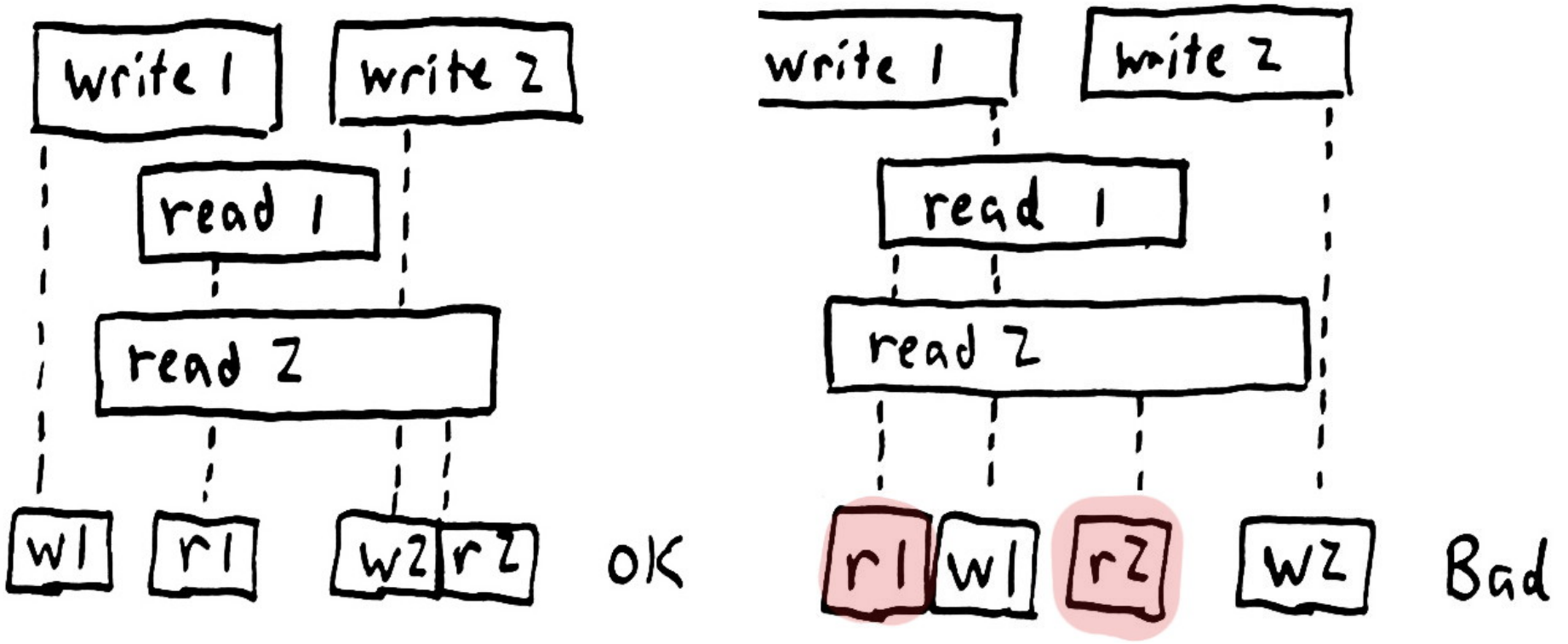


```clojure
(defrecord Mutex [locked?]
  Model
  (step [r op]
    (condp = (:f op)
      :acquire (if locked?
                 (inconsistent "already held")
                 (Mutex. true))
      :release (if locked?
                 (Mutex. false)
                 (inconsistent "not held")))))
```

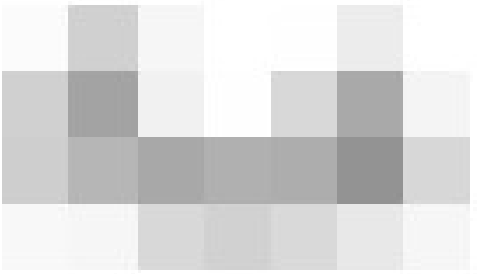
Model verifies that a single threaded history is OK.

But we don't Know the singlethreaded history...


Histories
Client 1
2
3
4
5
Nemesis
t


write 1
write 2
read 1
read 2
w1
r1
w2
r2
OK
write 1
write 2
read 1
read 2
r1
w1
r2
w2
Bad

O(n!)

(oh_ )

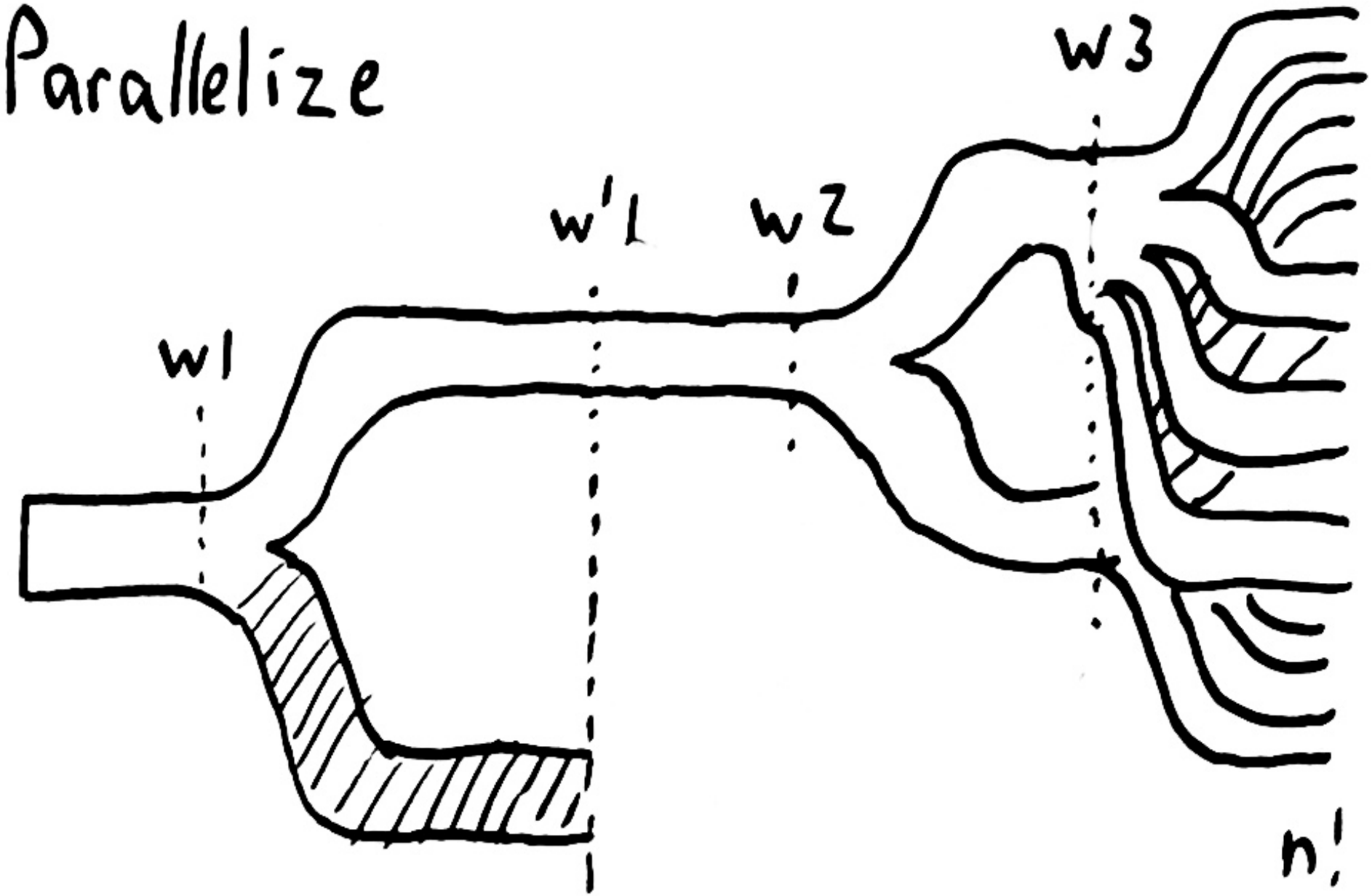
Techniques:

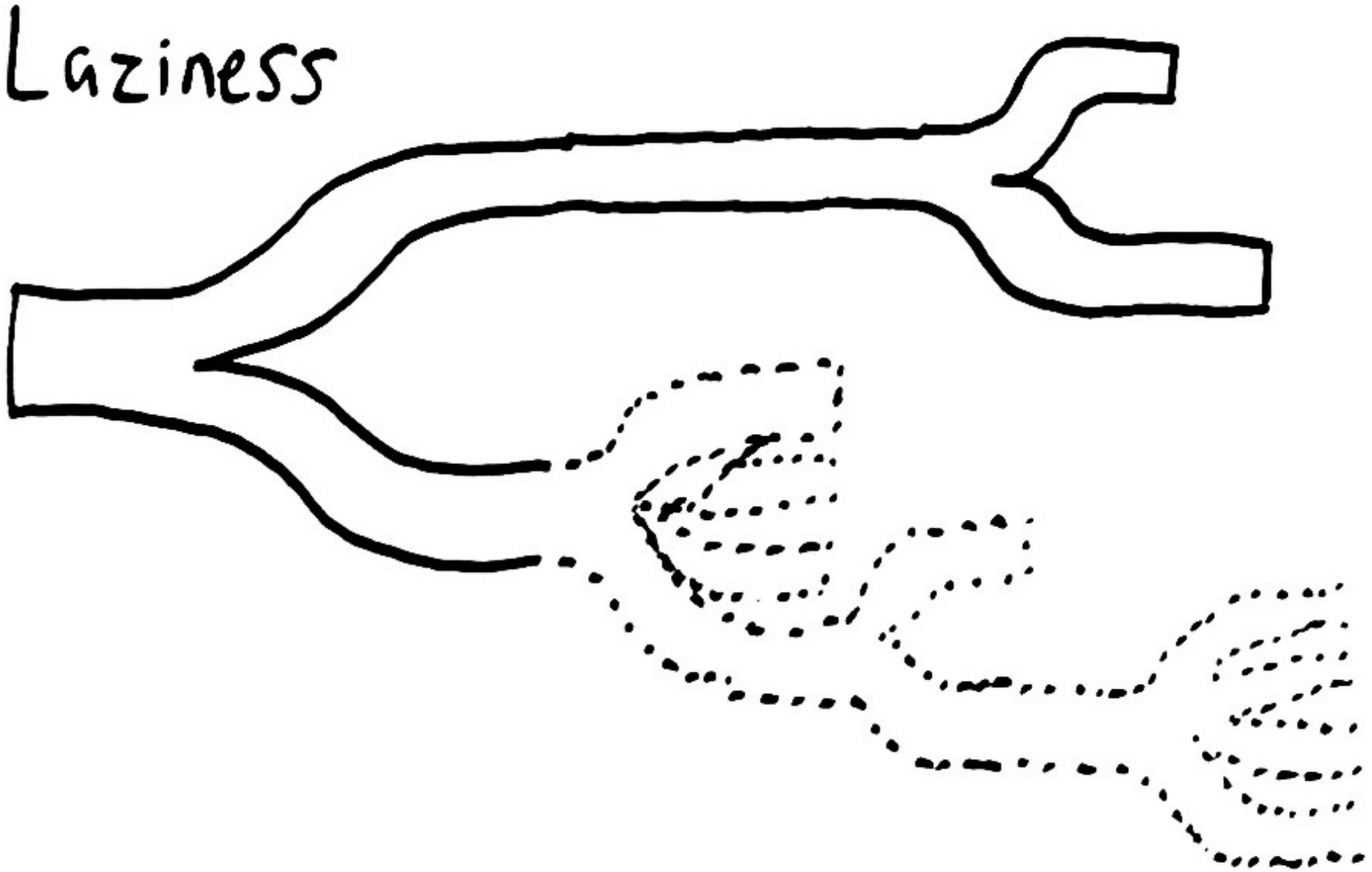
- Exploit degeneracy
- Parallelize
- Laziness + memoization
- Immutable, shared-structure representation of each world

Degeneracy

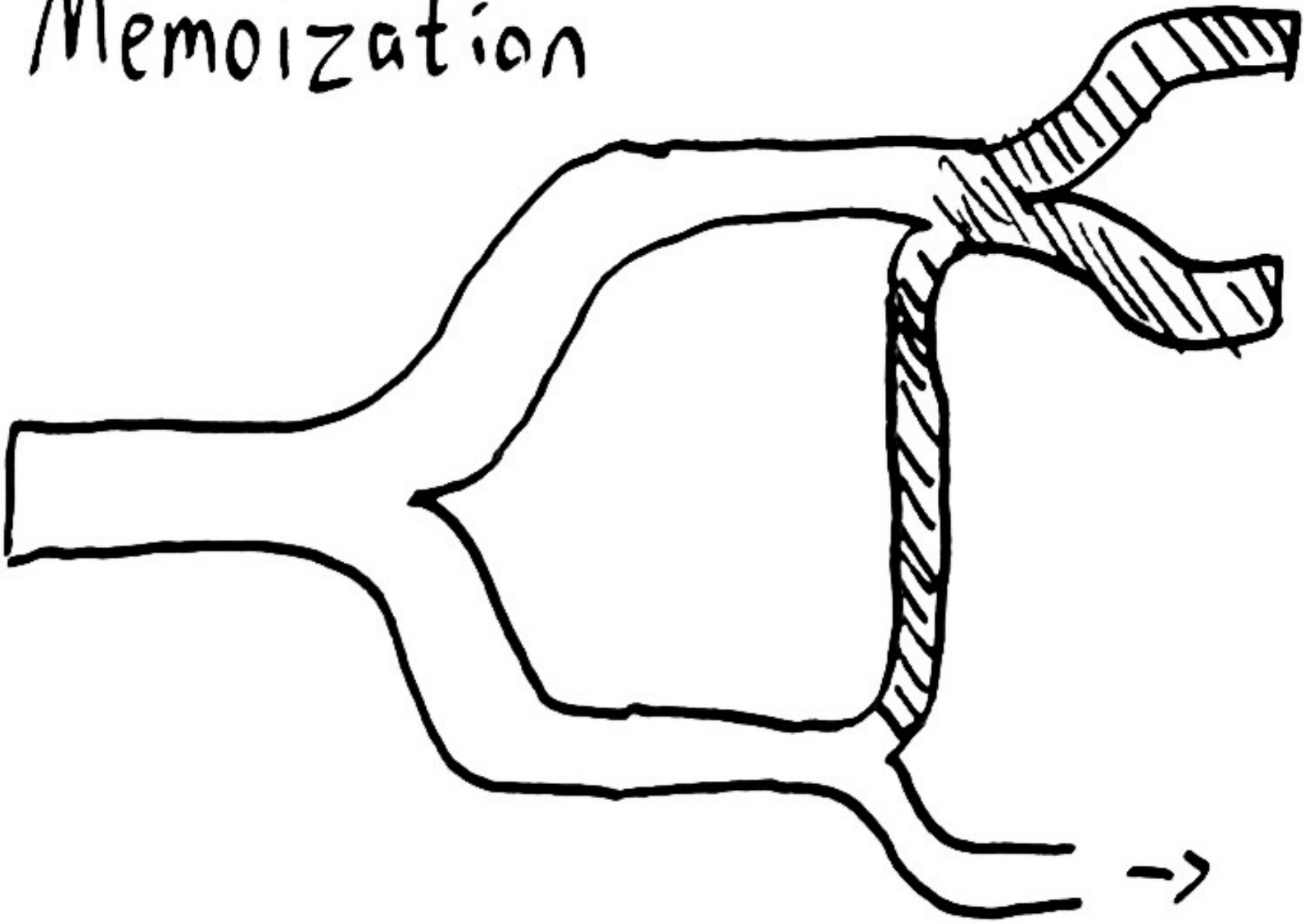
history	state	future
w'1, r'1	1	r'1, w2, r1, w'2
w'1	1	r'1, w2, r1, w'2
w'1, w'2	2	r'1, w2, r1, w'2
w'2	2	w2, r'1, w2, r1, w'2

The past is a fiction.

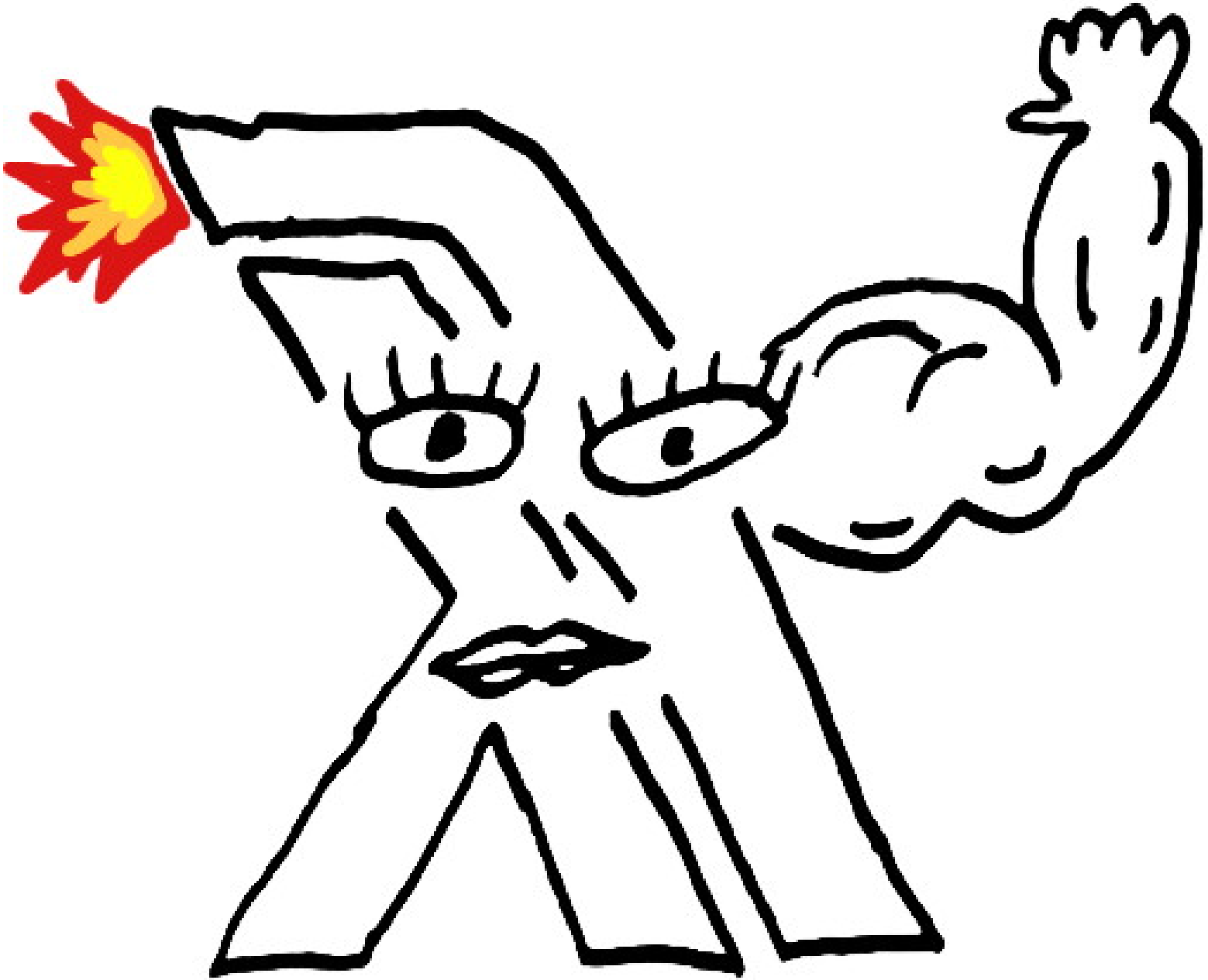

Parallelize
w3
w'1
w2
w1
n!

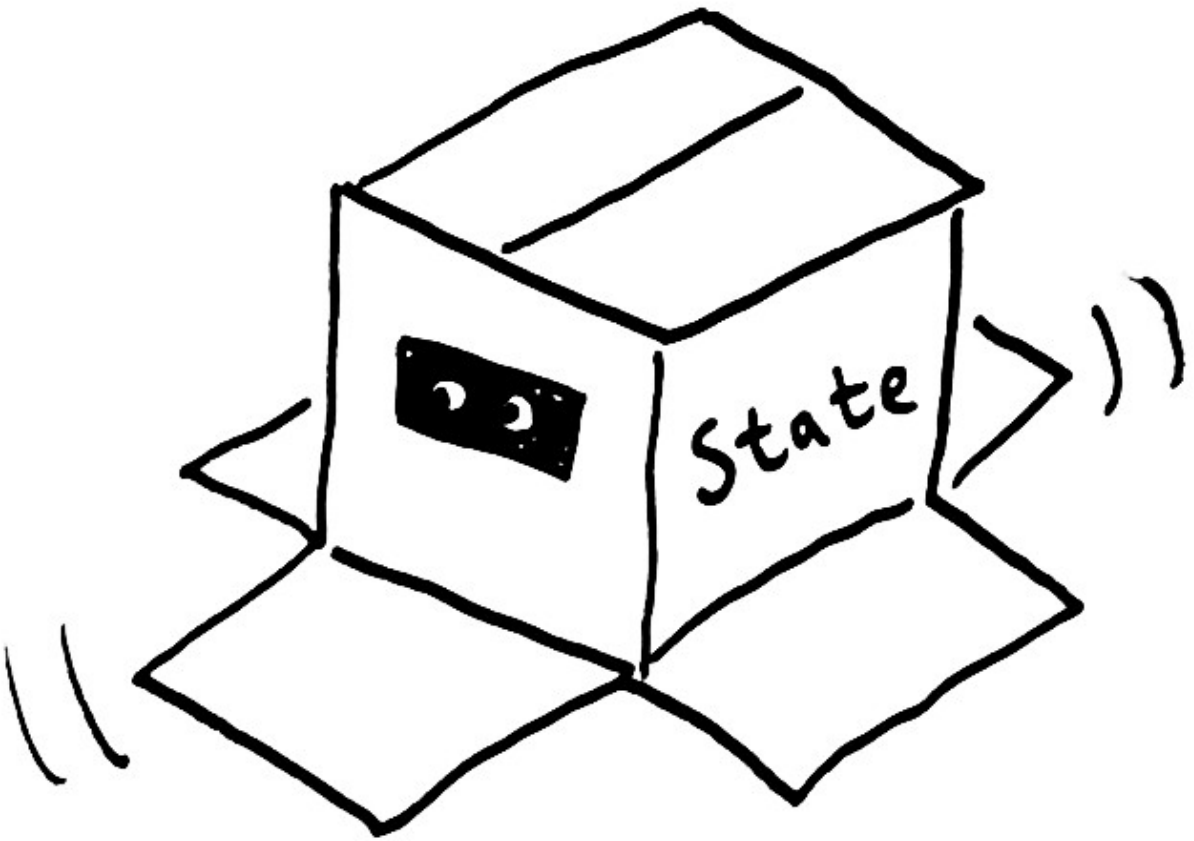

Laziness

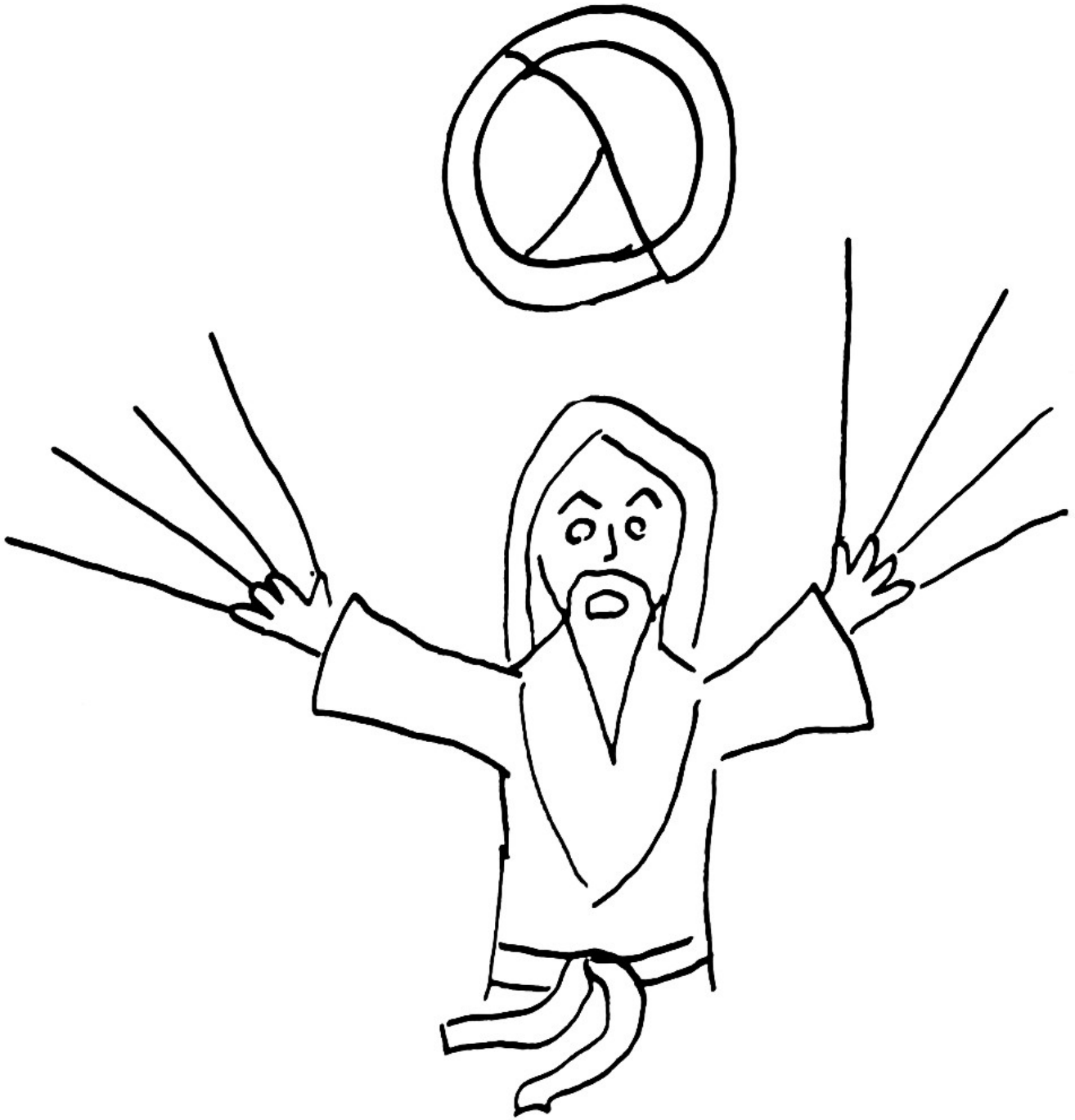
Memoization



- All parallel-safe
- Execution as data
- Adapted from Margo Seltzer's talk on autoparallelizing x86




State



Shared structure

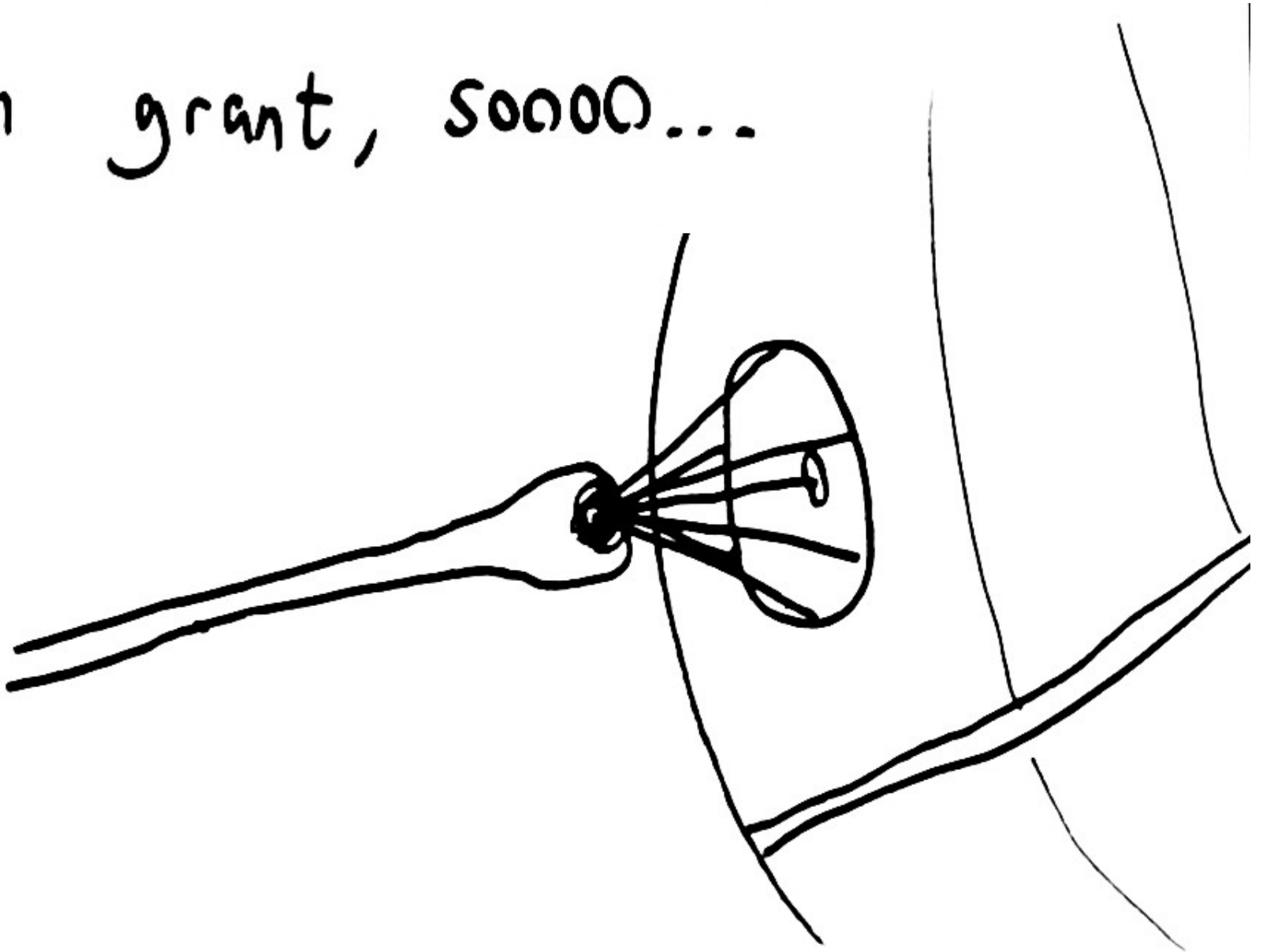
Clojure's maps/vectors only store deltas; reduces memory overhead from world forks by orders of magnitude.

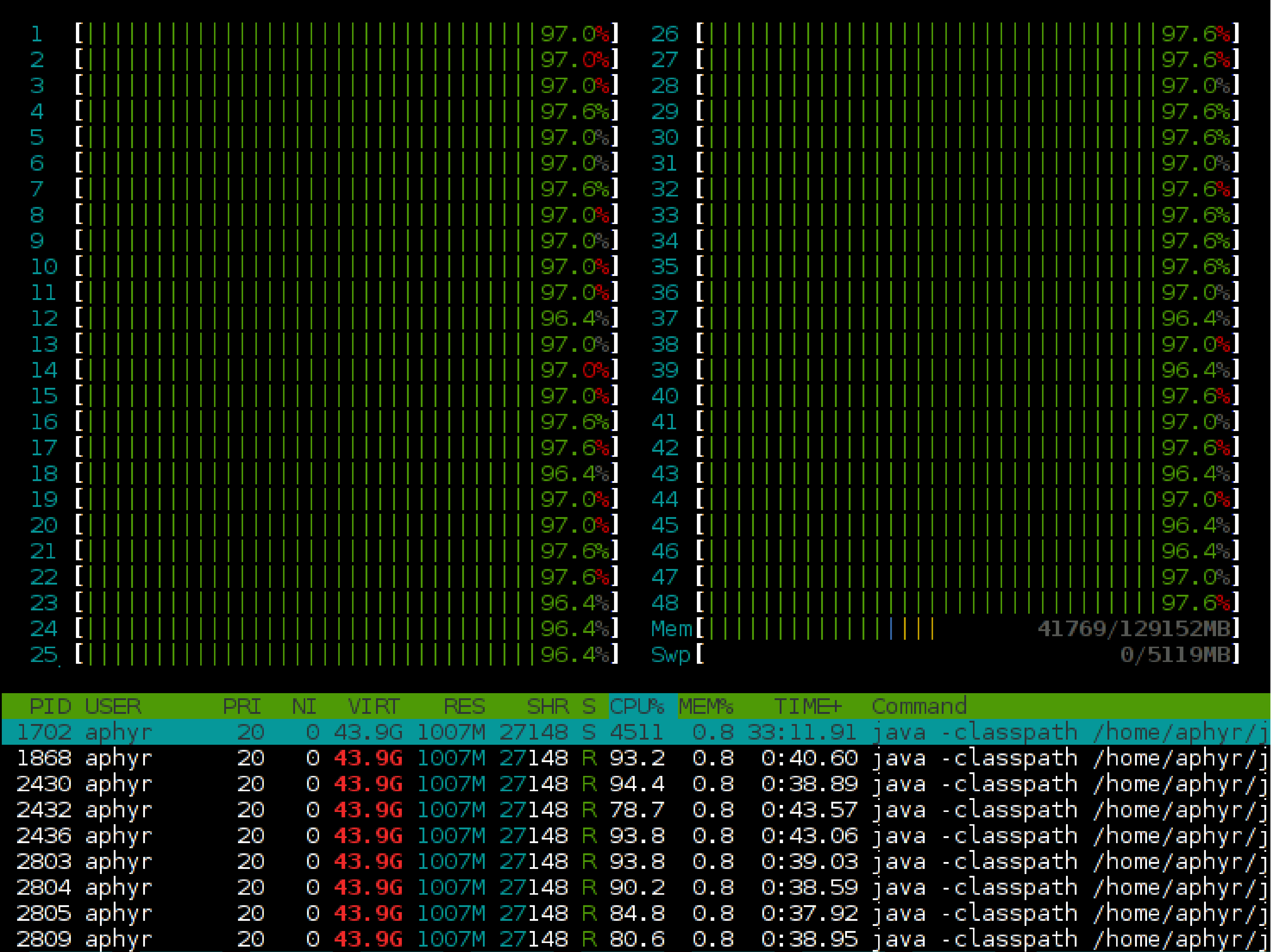

PUT A (pmap) ON IT!

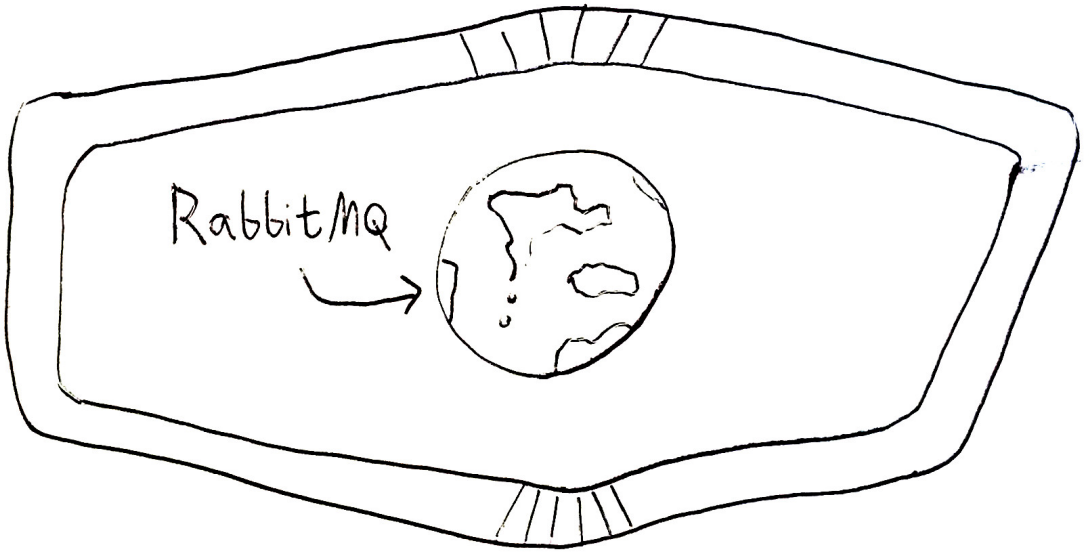
Controlled use of JVM mutable types where perf > determinism

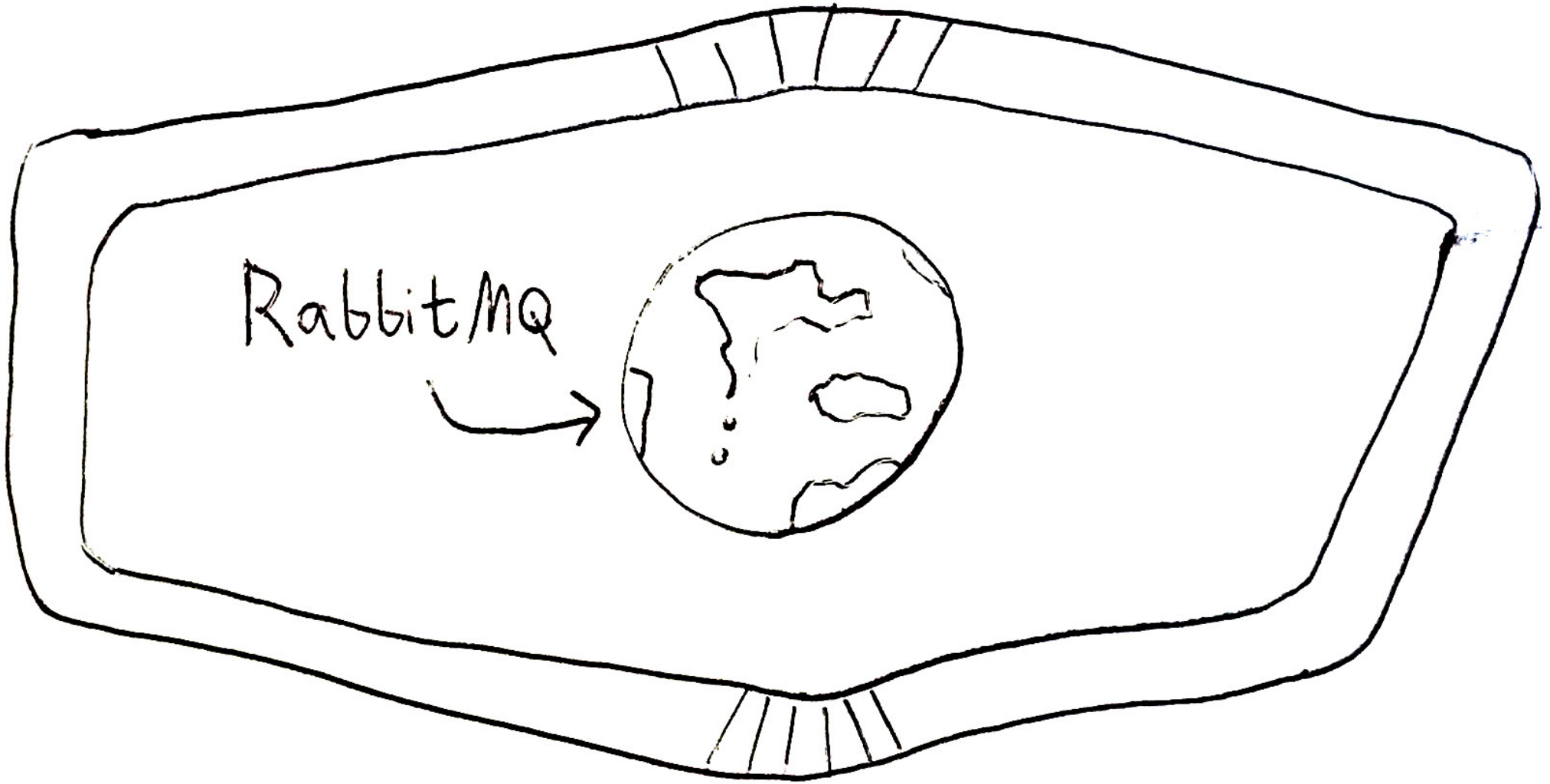
- Cliff Click's high_scale_lib
- Striped wrapper for j.u.c. PrioQueue
- j.u.c. atomics for cas state
- Clojure atoms & futures too

Comcast gave me a research grant, soooo...




1 97.0%
2 97.0%
3 97.0%
4 97.6%
5 97.0%
6 97.0%
7 97.6%
8 97.0%
9 97.0%
10 97.0%
11 97.0%
12 96.4%
13 97.0%
14 97.0%
15 97.0%
16 97.6%
17 97.6%
18 96.4%
19 97.0%
20 97.0%
21 97.6%
22 97.6%
23 96.4%
24 96.4%
25 96.4%
26 97.6%
27 97.6%
28 97.0%
29 97.6%
30 97.6%
31 97.0%
32 97.6%
33 97.6%
34 97.6%
35 97.6%
36 97.0%
37 96.4%
38 97.0%
39 96.4%
40 97.6%
41 97.0%
42 97.6%
43 96.4%
44 97.0%
45 96.4%
46 96.4%
47 97.0%
48 97.6%
Mem 41769/129152MB
Swp 0/5119MB
PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command
1702 aphyr 20 0 43.9G 1007M 27148 S 4511 0.8 33:11.91 java -classpath /home/aphyr/j
1868 aphyr 20 0 43.9G 1007M 27148 R 93.2 0.8 0:40.60 java -classpath /home/aphyr/j
2430 aphyr 20 0 43.9G 1007M 27148 R 94.4 0.8 0:38.89 java -classpath /home/aphyr/j
2432 aphyr 20 0 43.9G 1007M 27148 R 78.7 0.8 0:43.57 java -classpath /home/aphyr/j
2436 aphyr 20 0 43.9G 1007M 27148 R 93.8 0.8 0:43.06 java -classpath /home/aphyr/j
2803 aphyr 20 0 43.9G 1007M 27148 R 93.8 0.8 0:39.03 java -classpath /home/aphyr/j
2804 aphyr 20 0 43.9G 1007M 27148 R 90.2 0.8 0:38.59 java -classpath /home/aphyr/j
2805 aphyr 20 0 43.9G 1007M 27148 R 84.8 0.8 0:37.92 java -classpath /home/aphyr/j
2809 aphyr 20 0 43.9G 1007M 27148 R 80.6 0.8 0:38.95 java -classpath /home/aphyr/j


RabbitMQ


RabbitMQ

- Message Queue
- Knows about worker tasks
- Lets make a lock out of it!

- Rabbit docs are very clear: partitions ruin all guarantees

```
=ERROR REPORT==== 10-Apr-2014::13:16:08 ===
** Node rabbit@n3 not responding **
** Removing (timedout) connection **

=INFO REPORT==== 10-Apr-2014::13:16:29 ===
rabbit on node rabbit@n5 down

=ERROR REPORT==== 10-Apr-2014::13:16:45 ===
Mnesia(rabbit@n1): ** ERROR ** mnesia_event got
{inconsistent_database,
running_partitioned_network, rabbit@n3}
```

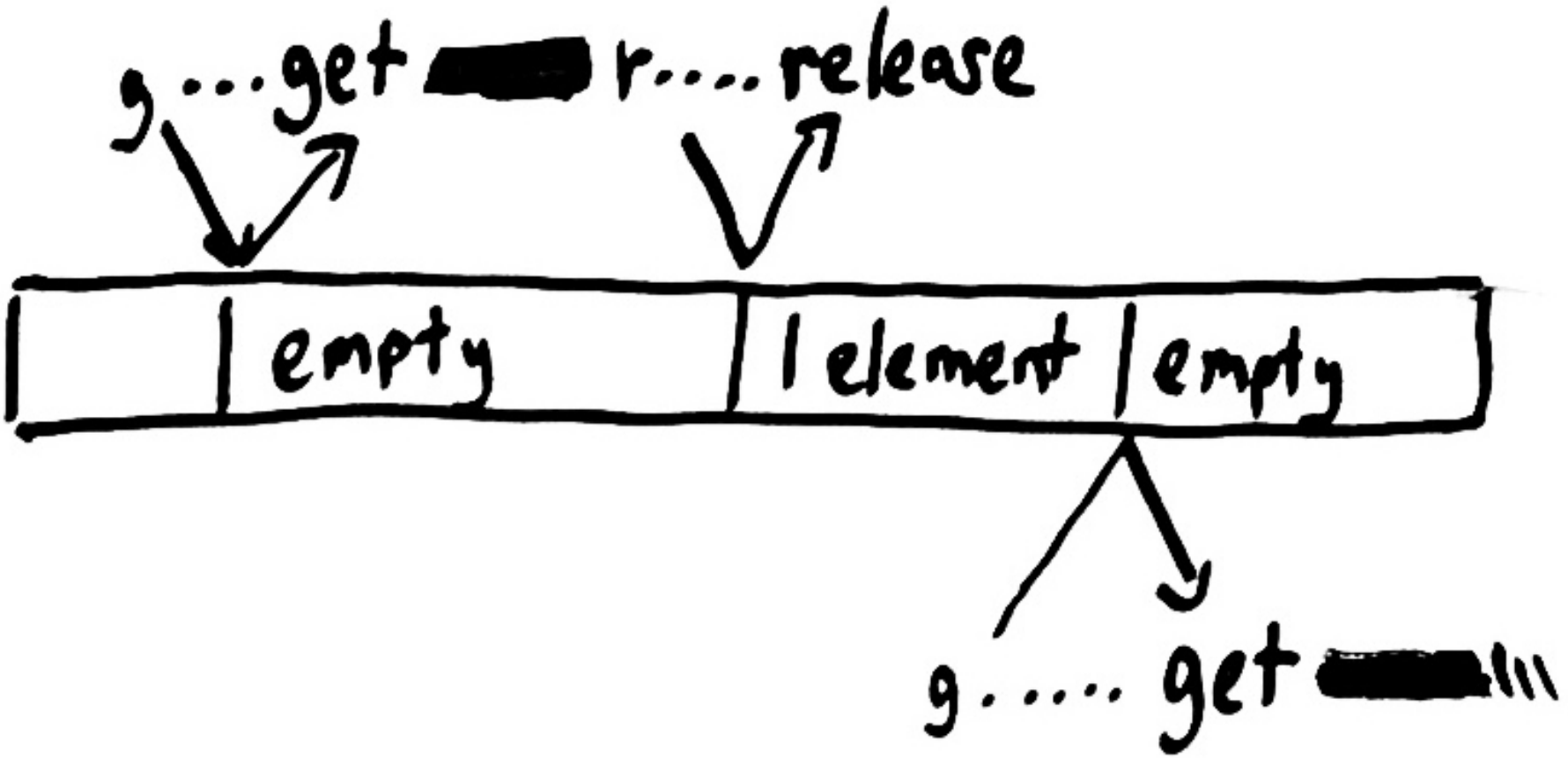
Distributed Semaphores with RabbitMQ

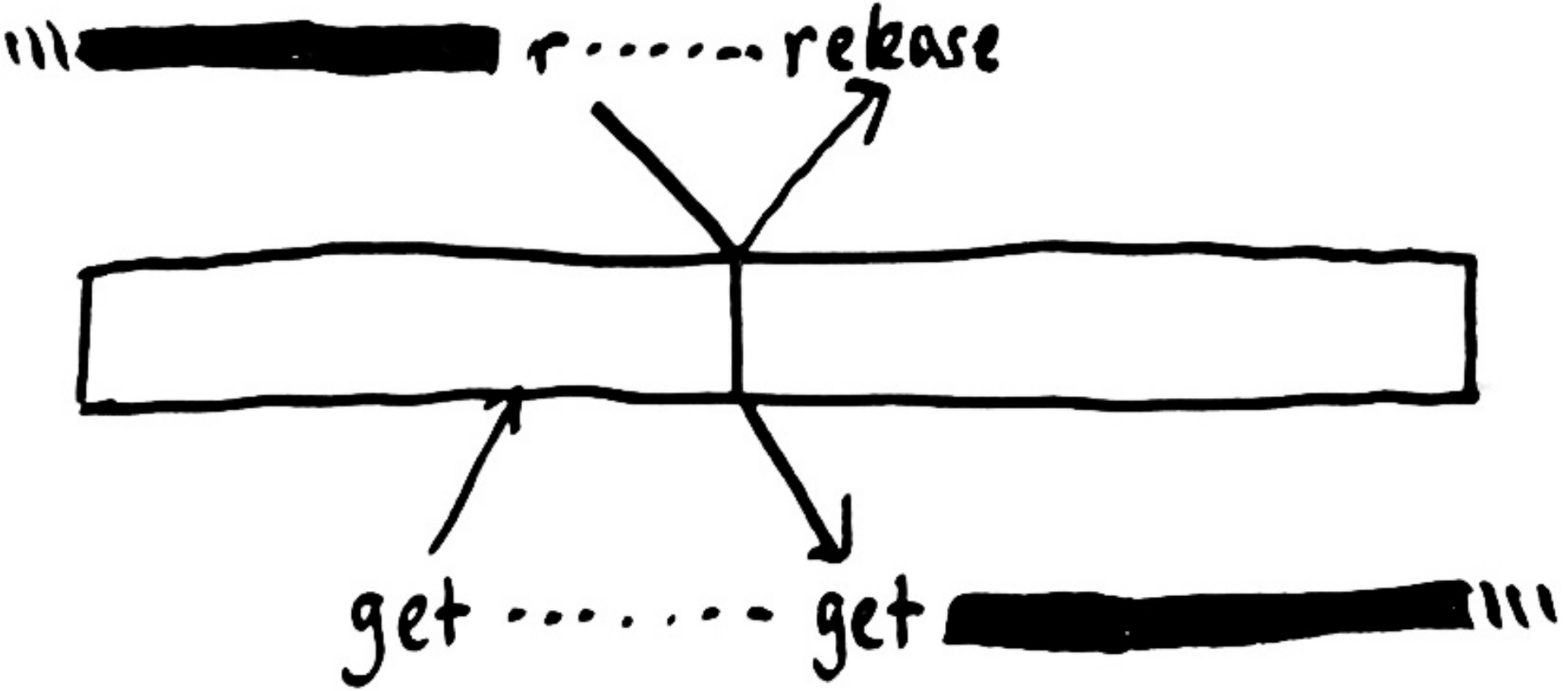
In this blog post we are going to address the problem of controlling the access to a particular resource in a distributed system. The technique for solving this problem is well know in computer science, it's called Semaphore and it was invented by Dijkstra in 1965 in his paper called "Cooperating Sequential Processes". We are going to see how to implement it using AMQP's building blocks, like consumers, producers and queues.

The Need for Semaphores

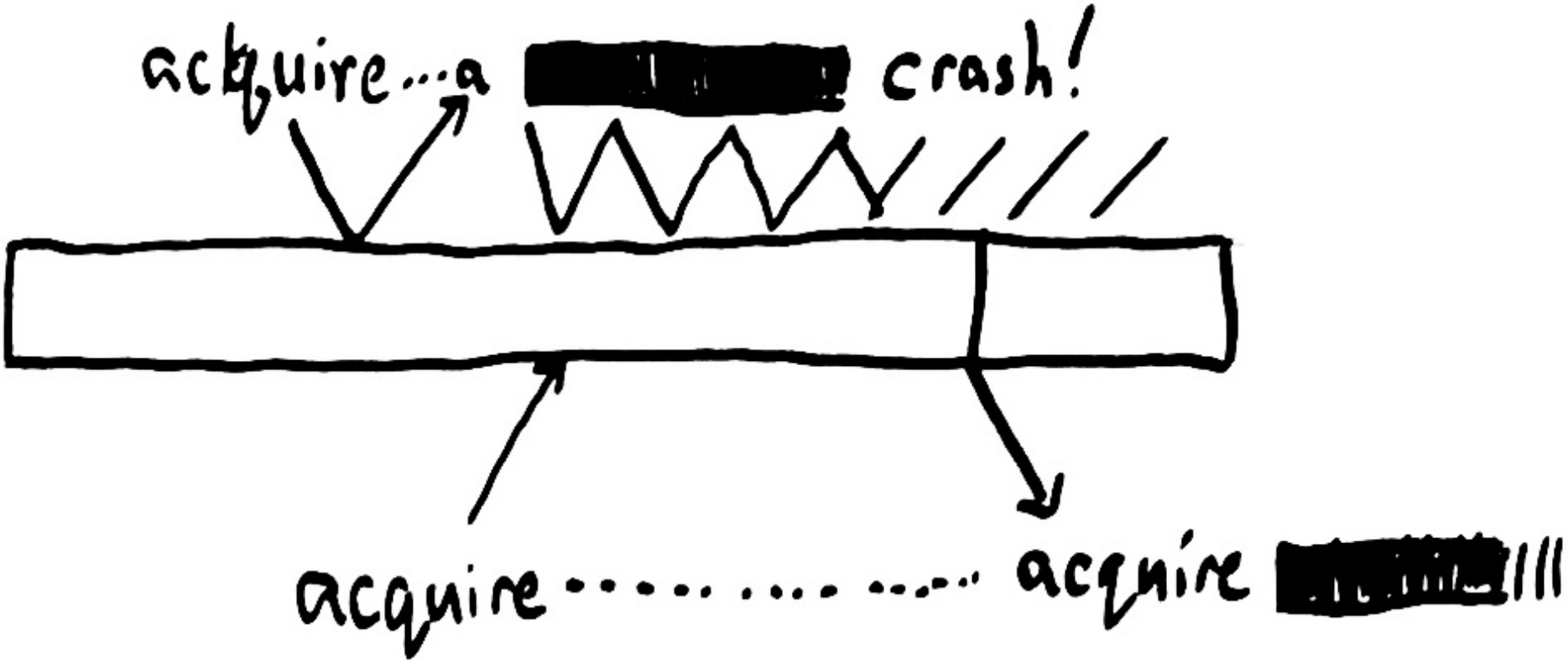
Before going into the actual solution, let's see when we might actually need something like this:

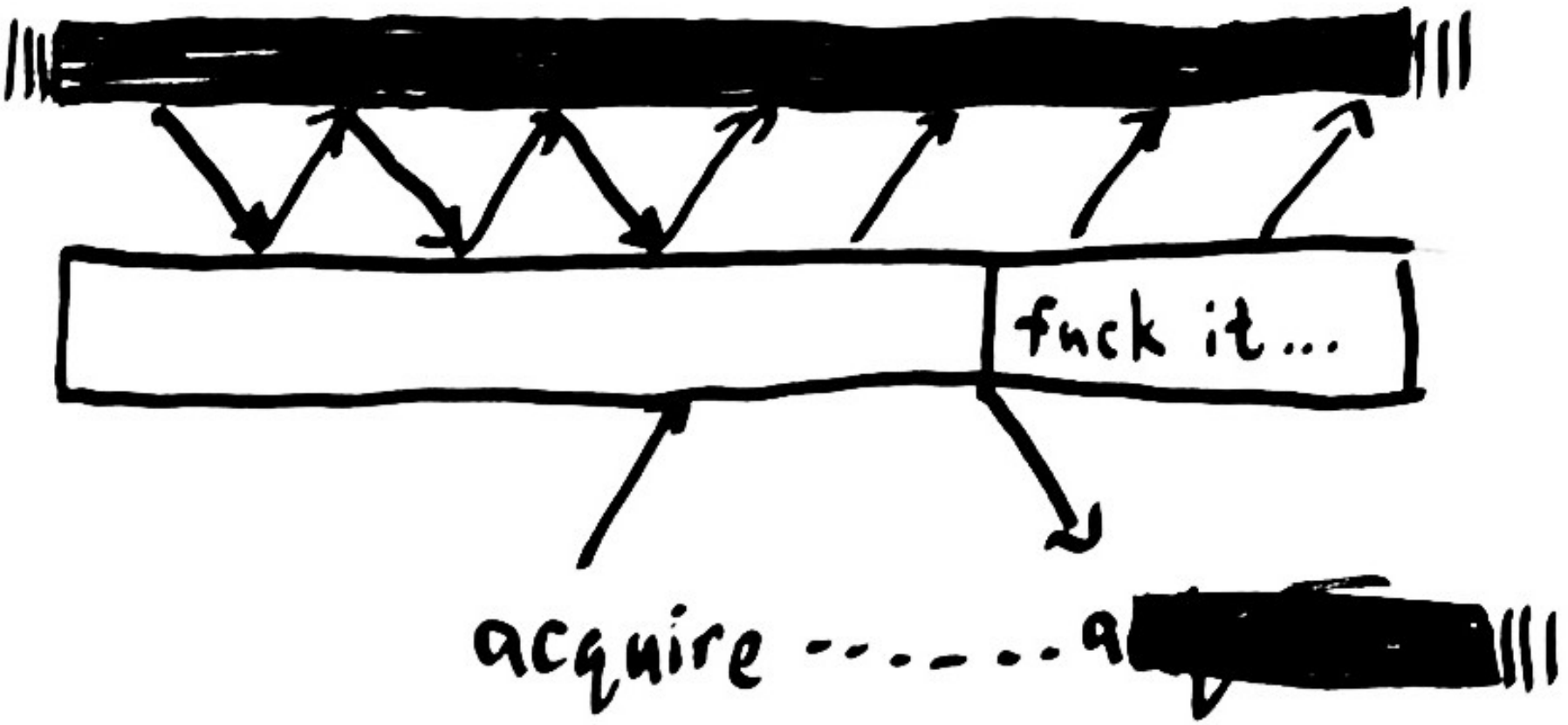
Let's say our application has many processes taking jobs from a queue and then inserting records to a database, we might need to limit how many of them do it concurrently.


g ... get
r .... release
empty
1 element
empty
g ..... get


release
get
get

But Rabbit is not a linearizable system!




fuck it...
acquire ------ a

```
Not linearizable. Linearizable prefix was:
...
:nemesis  :info :stop    nil
:nemesis  :info :stop    "fully connected"
:nemesis  :info :start   nil
:nemesis  :info :start   "partitioned into
                         [(:n1 :n4)
                          (:n3 :n5 :n2)]"
3 :invoke :acquire       nil
2 :invoke :acquire       1
2 :ok     :acquire       1
1 :invoke :acquire       2

Followed by inconsistent operation:
1 :ok :acquire           2
```

```
Last consistent worlds were: ----------------

World from fixed history:
1 :invoke :acquire  1
1 :invoke :release  nil
2 :invoke :acquire  1

and current state  #jepsen.model.Mutex{:locked true}

with pending operations:
3 :invoke :acquire  nil
1 :invoke :acquire  2
0 :invoke :acquire  nil
4 :invoke :acquire  nil
----------------------------------------

Inconsistent state transitions:
([{:locked true} "already held"])
```

Queues often sacrifice linearizability to provide fault tolerance.

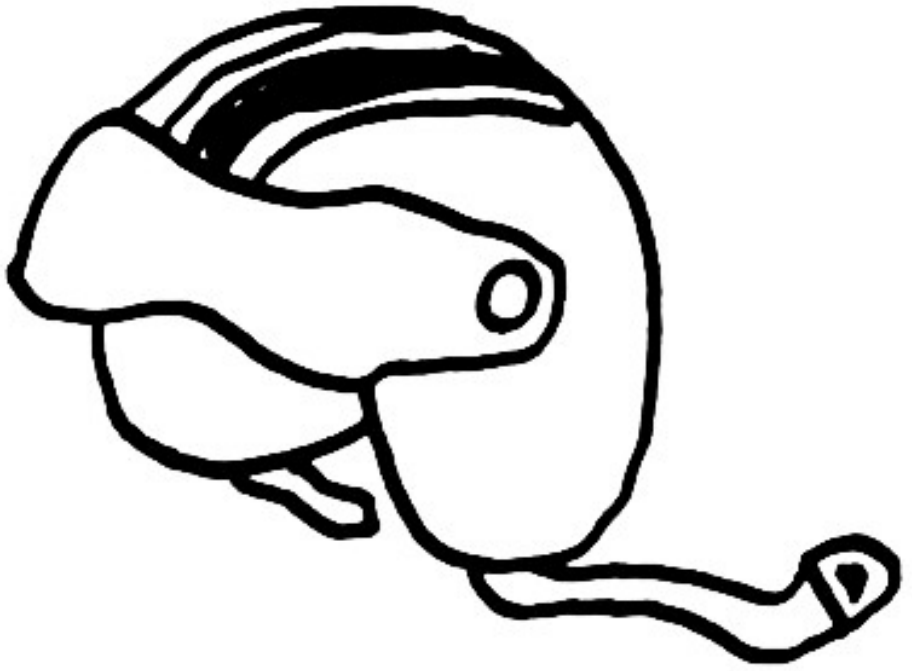
Perfectly fine queue.

Not a lock service.


etcd

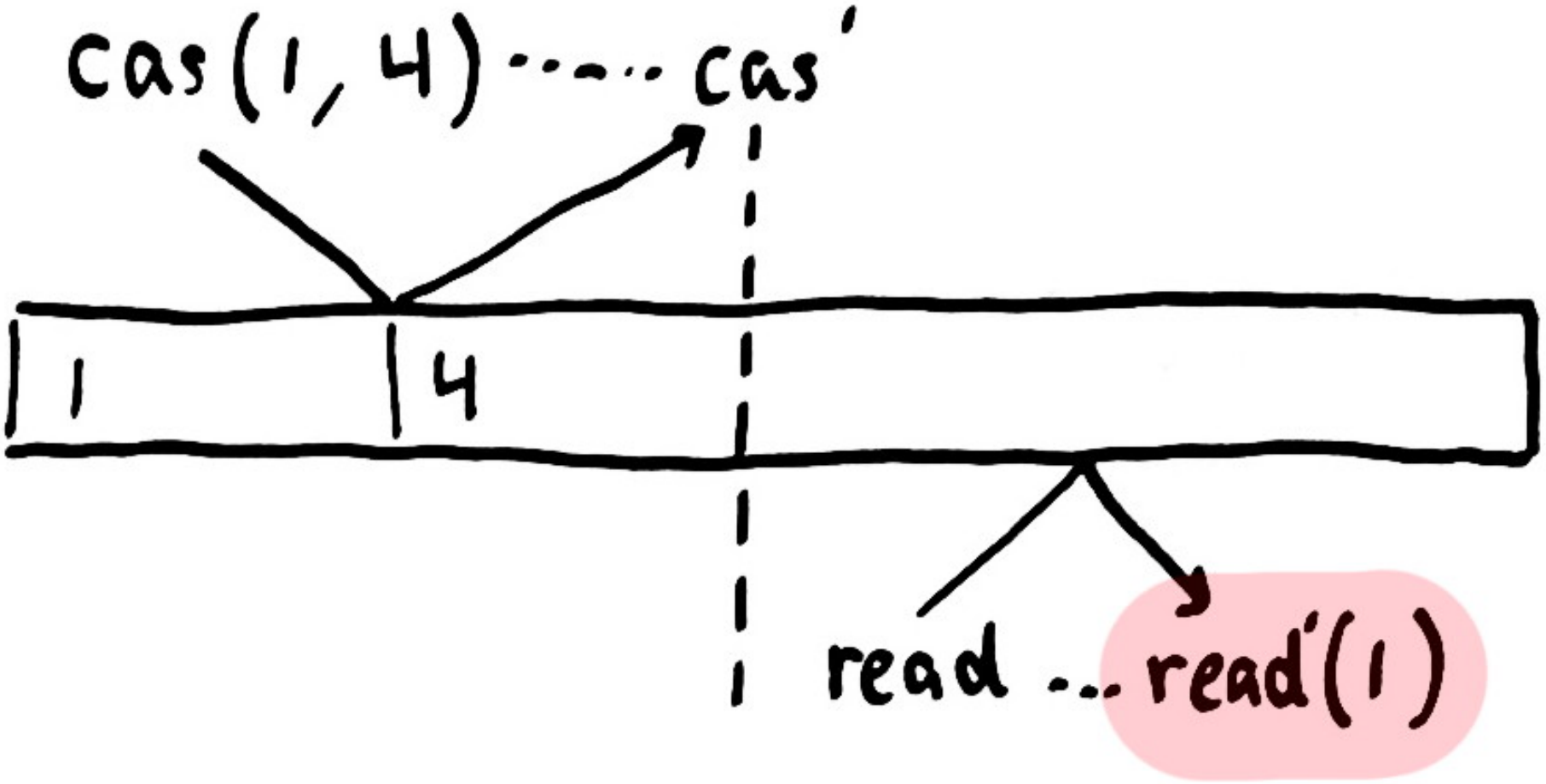
- Based on Raft
- HTTP API
- Configuration & Coordination
- Reads, writes, value & index-based CaS

Race conditions in Cluster join!



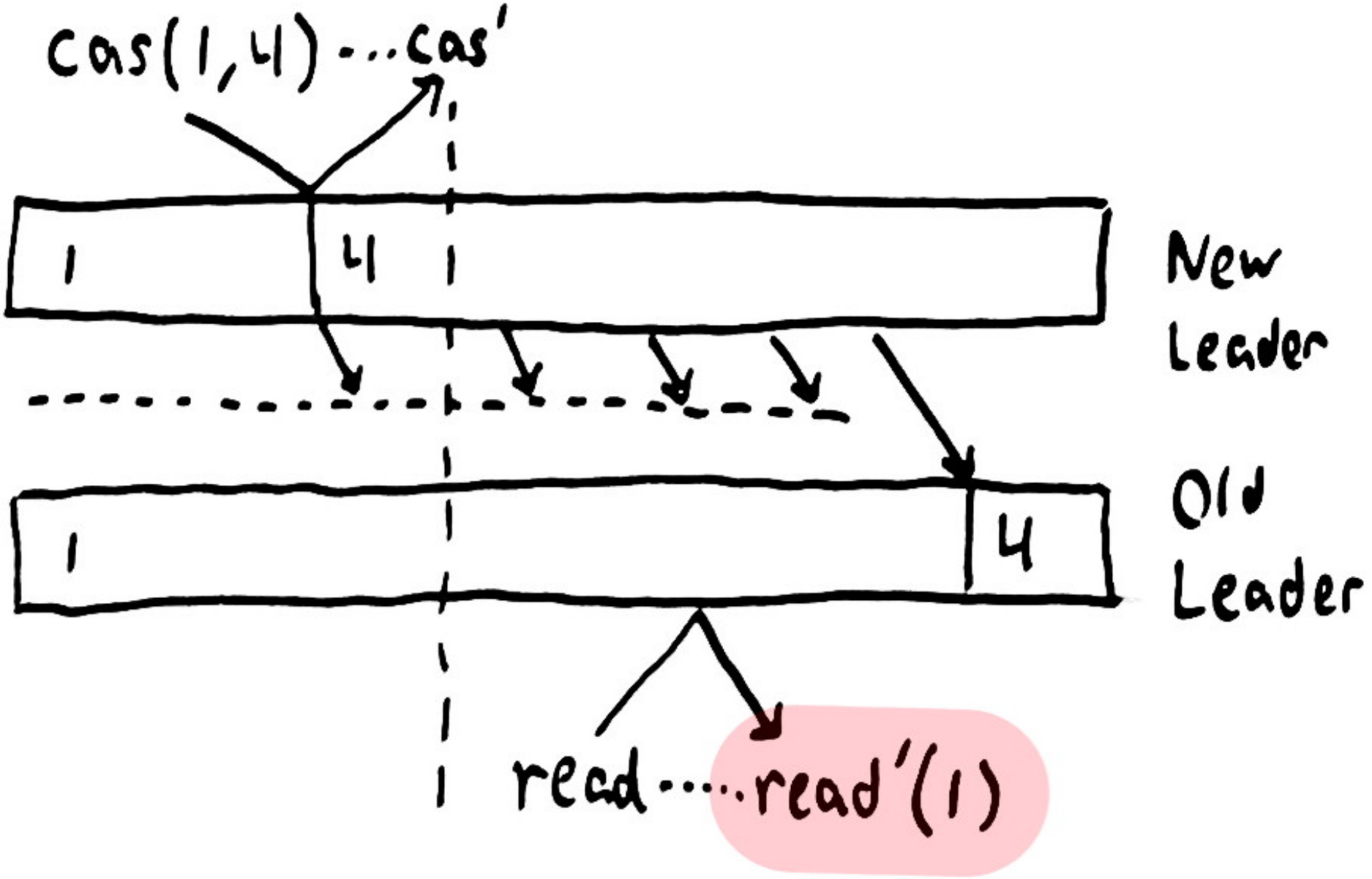
```
FAIL in (register-test) (etcd_test.clj:45)
expected: (:valid? (:results test))
  actual: false
Not linearizable. Linearizable prefix was:
…
:nemesis :info :start  "Cut off {:n5 #{:n4 :n1}, :n2
#{:n4 :n1}, :n3 #{:n4 :n1}, :n1 #{:n3 :n2 :n5}, :n4
#{:n3 :n2 :n5}}"
2  :invoke    :cas    [1 4]
4  :invoke    :read   1
4  :ok        :read   1
2  :ok        :cas    [1 4]

Followed by inconsistent operation:
0  :invoke    :read   1
```

```
World with fixed history:
...
4  :invoke :write  1
0  :invoke :read 1
3  :invoke :read 1
1  :invoke :read 1
4  :invoke :read 1
2  :invoke :cas  [1 4]

led to state:
{:value 4}

with pending operations:

(and 12928 more worlds, elided here
------------------------------------------------

Inconsistent state transitions:
([{:value 4} "can't read 1 from register 4"])
```


cas(1, 4) ..... cas'
1
4
read ... read'(1)

Raft guarantees committed log entries are linearizable.

But etcd "consistent" reads don't go through the Raft log! They read current primary state & return!


cas(1,4)...cas'
1
4
New Leader
1
4
Old Leader
read.....read'(1)

Stale reads violate linearizability, sequential, & causal consistency. Also RYW. Also MW. Also MR.

As part of our Consul testing, we ran it against Jepsen to determine if any consistency issues could be uncovered. In our testing, Consul gracefully recovered from partitions without introducing any consistency issues.

As part of our Consul testing, we ran it against Jepsen to determine if any consistency issues could be uncovered. In our testing, Consul gracefully recovered from partitions without introducing any consistency issues.

```
LeaderLeaseTimeout: time.Second,
LeaderLeaseTimeout: 300 * time.Millisecond,
```

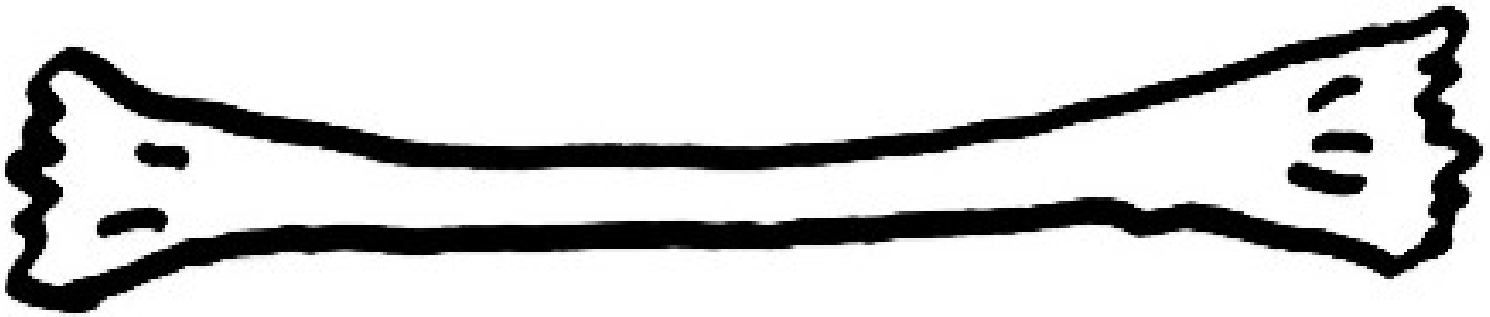
Suggestion: timeouts in a non-realtime system can't guarantee correctness, only improve odds.

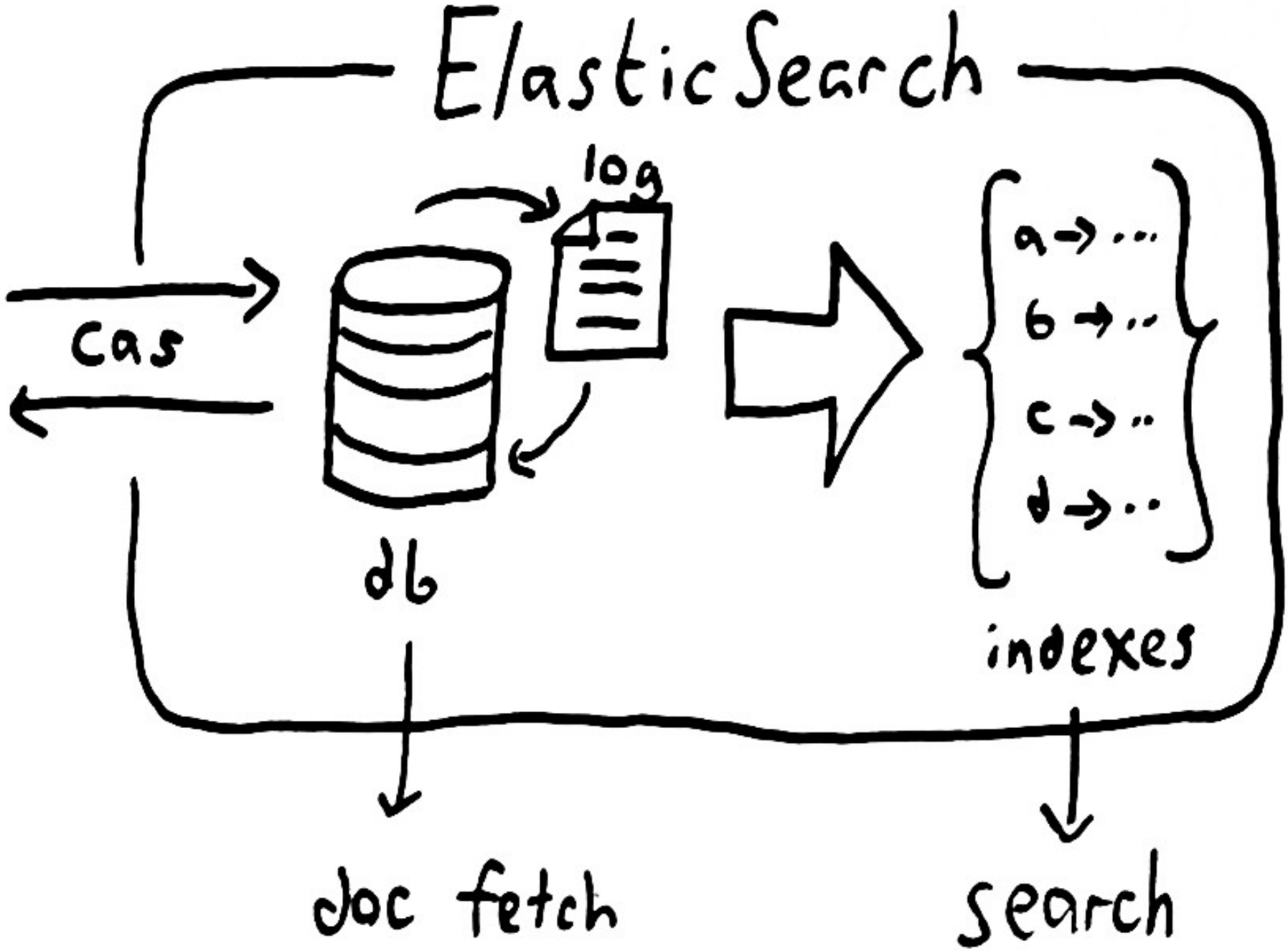
→ Elasticsearch GC partitions...

Both etcd & Consul are considering making reads actually consistent

- Latency cost!
- Many apps can tolerate stale reads.

ElasticSearch!




ElasticSearch
log
cas
db
a → ...
b → ..
c → ..
d → ..
indexes
doc fetch
search

Elasticsearch offers atomic compare-and-set.

=> linearizability

write consistency

To prevent writes from taking place on the "wrong" side of a network partition, by default, index operations only succeed if a quorum (>replicas/2+1) of active shards are available. This default can be overridden on a node-by-node basis using the `action.write_consistency` setting. To alter this behavior per-operation, the `consistency` request parameter can be used.

When it comes to CAP, in a very high level, elasticsearch gives up on partition tolerance. This is for several reasons:

1. I personally believe that *within the same data center", network partitions very rarely happen, and when they do, its a small set (many times single) machine that gets "partitioned out of the network". When a single machine gets disconnected from the network, then thats not going to affect elasticsearch. When it come to cross data centers, a solution that gives up on consistency can be built on top of elasticsearch, either by elasticsearch (in the future), or now, by using messaging in some form to replicate changes between two data centers.

ES gives up on partition tolerance, it means, if enough nodes fail, cluster state turns red and ES does not proceed to operate on that index.

ES is not giving up on availability. Every request will be responded, either true (with result) or false (error). In a system being not available, you would have to expect the property of having some requests that can no longer be answered at all (they hang forever or the responder is gone).

Consistency is not given up by ES. First, on doc level, you have "write your own read" consistency implemented as versioning - a doc read followed by a doc write is guaranteed to be consistent if both read and write versions match (MVCC). Write-your-read consistency is only eventually consistent because other clients can not be sure when to read the old and when to read the new value, and this is different from for example ACID (causal consistency). But it's just another model of consistency.

And second, on index level, after a node crash, ES tries an index recovery that should end up always in a consistent index state. This is possible because of the WAL (translog) at each shard.

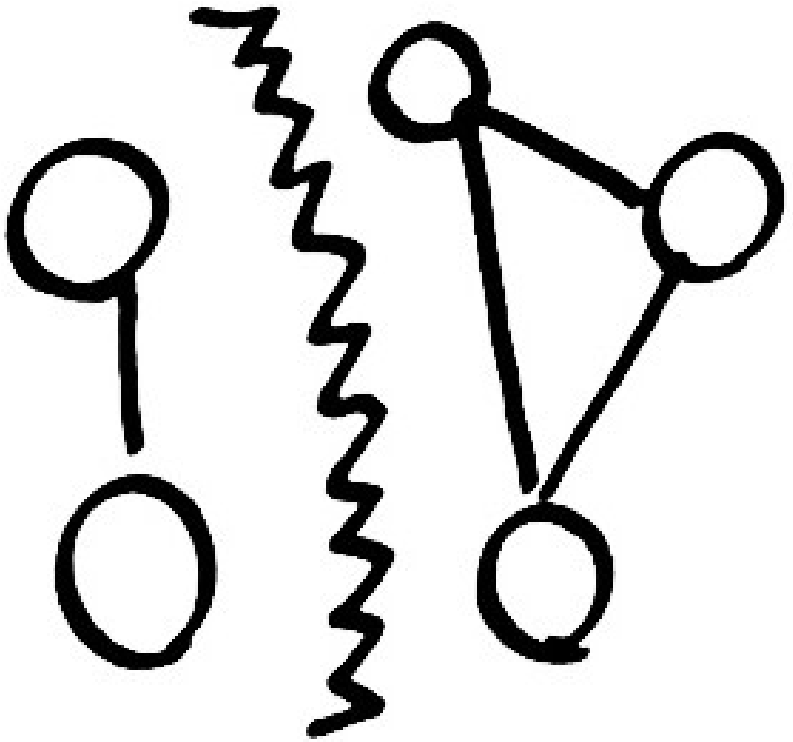
Short windows of write loss during cluster transitions.

```
2   :ok       :add   46
1   :invoke   :add   47
4   :invoke   :add   48
4   :ok       :add   48
0   :invoke   :add   49
:nemesis  :info  :start "Cut off {:n5 #{:n2 :n1}, :n4 #{:n2 :n1},
:n1 #{:n4 :n5}, :n2 #{:n4 :n5}}"
3   :invoke   :add   50
2   :invoke   :add   51
4   :invoke   :add   52
1   :info     :add   :timed-out
0   :info     :add   :timed-out
3   :info     :add   :timed-out
2   :info     :add   :timed-out
4   :info     :add   :timed-out
5   :invoke   :add   53
6   :invoke   :add   54
8   :invoke   :add   55
7   :invoke   :add   56
9   :invoke   :add   57
5   :info     :add   :timed-out
```

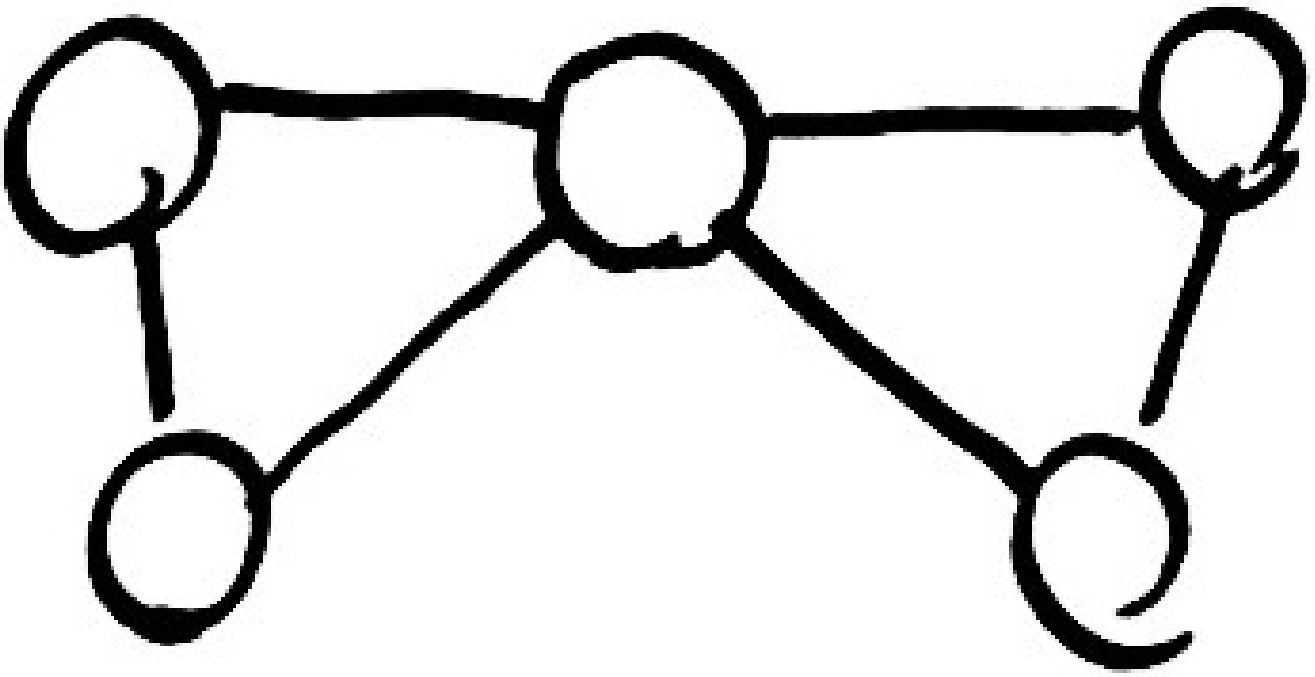
```
186 :invoke   :add  273
186 :fail     :add  273
173 :invoke   :add  274
173 :ok       :add  274
185 :info     :add  :timed-out
:nemesis      :info :stop nil
:nemesis      :info :stop "fully connected"
:nemesis      :info :stop nil
:nemesis      :info :stop "fully connected"
174 :invoke   :read nil
174 :ok       :read #{0 1 2 3 4 5 6 9 10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 26 27 28 29 30 31 32 33 35 36 37 38 39 40 41 42 43
44 45 46 48 50 51 52 55 56 57 60 61 62 65 67 68 71 72 73 76 77 78
81 82 83 86 87 88 91 92 93 96 97 98 101 102 103 106 107 108 111
112 113 116 117 118 121 122 123 126 127 128 131 132 133 136 137
138 141 142 143 146 147 148 151 152 153 154 155 156 157 158 159
160 161 162 163 164 166 168 169 170 171 173 174 175 270 271 272
274}
```

```
{:valid? false,
 :ok          #{0..6 9..24 26..33 35..46 48 50..52 55..57 60..62 65
                67..68 71..73 76..78 81..83 86..88 91..93 96..98
                101..103 106..108 111..113 116..118 121..123
                126..128 131..133 136..138 141..143 146..148
                151..164 166 168..171 173..175 270..272 274},
 :recovered   #{50..52 55..57 60..62 65 67..68 71..73 76..78 81..83
                86..88 91..93 96..98 101..103 106..108 111..113
                116..118 121..123 126..128 131..133 136..138
                141..143 146..147},
 :lost        #{176..178 250..251},
 :unexpected #{},
 :ok-frac          26/55,
 :recovered-frac  59/275,
 :lost-frac        1/55,
 :unexpected-frac 0}
```

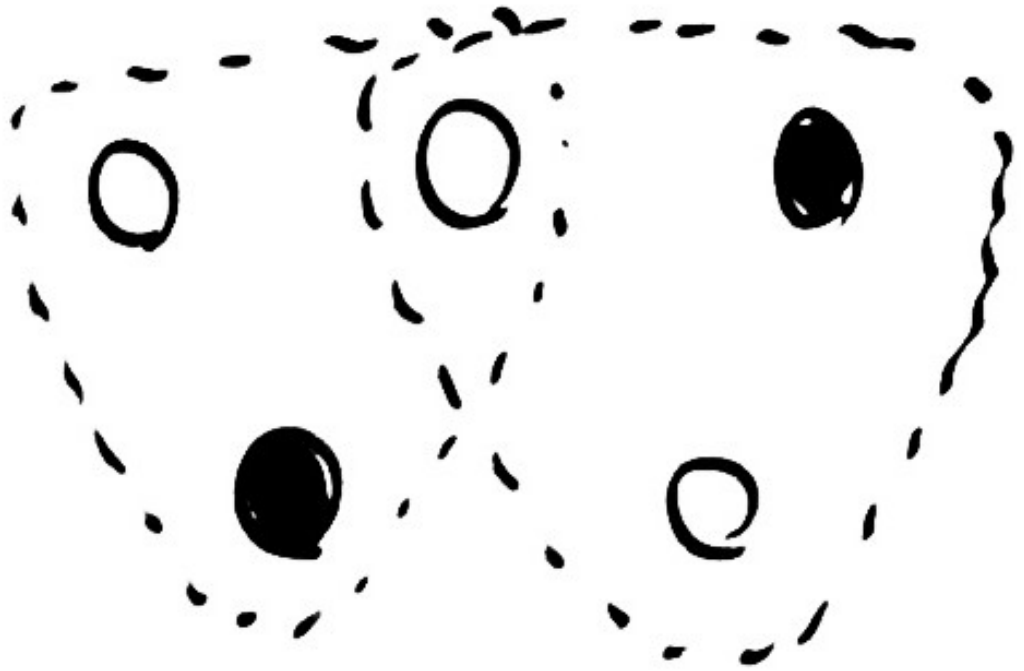
Assumes partitions are symmetric & complete



What about...



Every node thinks it has majority.



2 primaries

But wait –

There's More!

Insert a document in the next five minutes and we'll lose

HALF YOUR DATA!

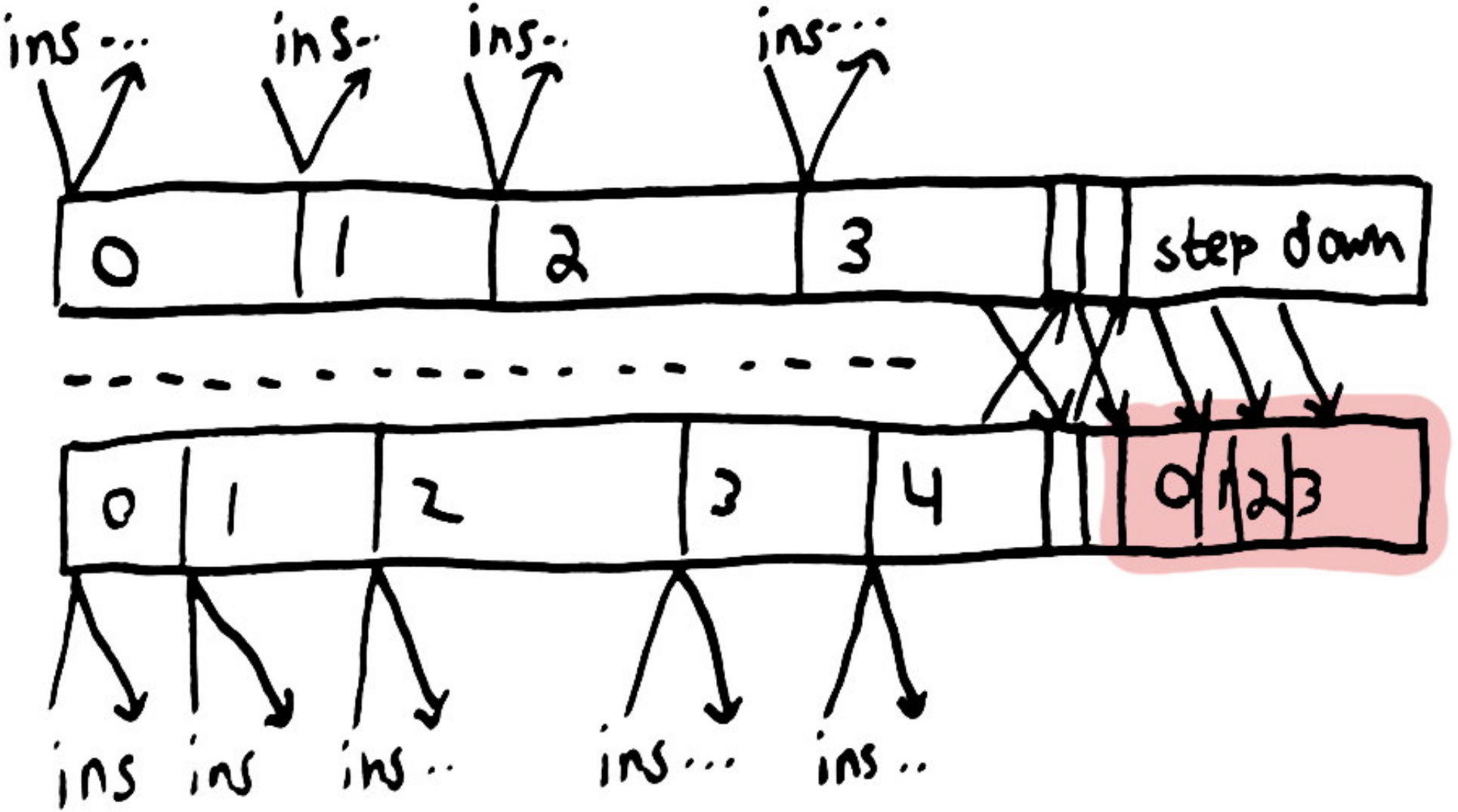
```
"RemoteTransportException[[Cadaver]
[inet[/192.168.122.12:9300]][index]]; nested:
RemoteTransportException[[Death Adder]
[inet[/192.168.122.11:9300]][index]]; nested:
DocumentAlreadyExistsException[[jepsen-index][1]
[number][EpVU56YERBOfRqyVc-_hAg]: document already
exists];
```

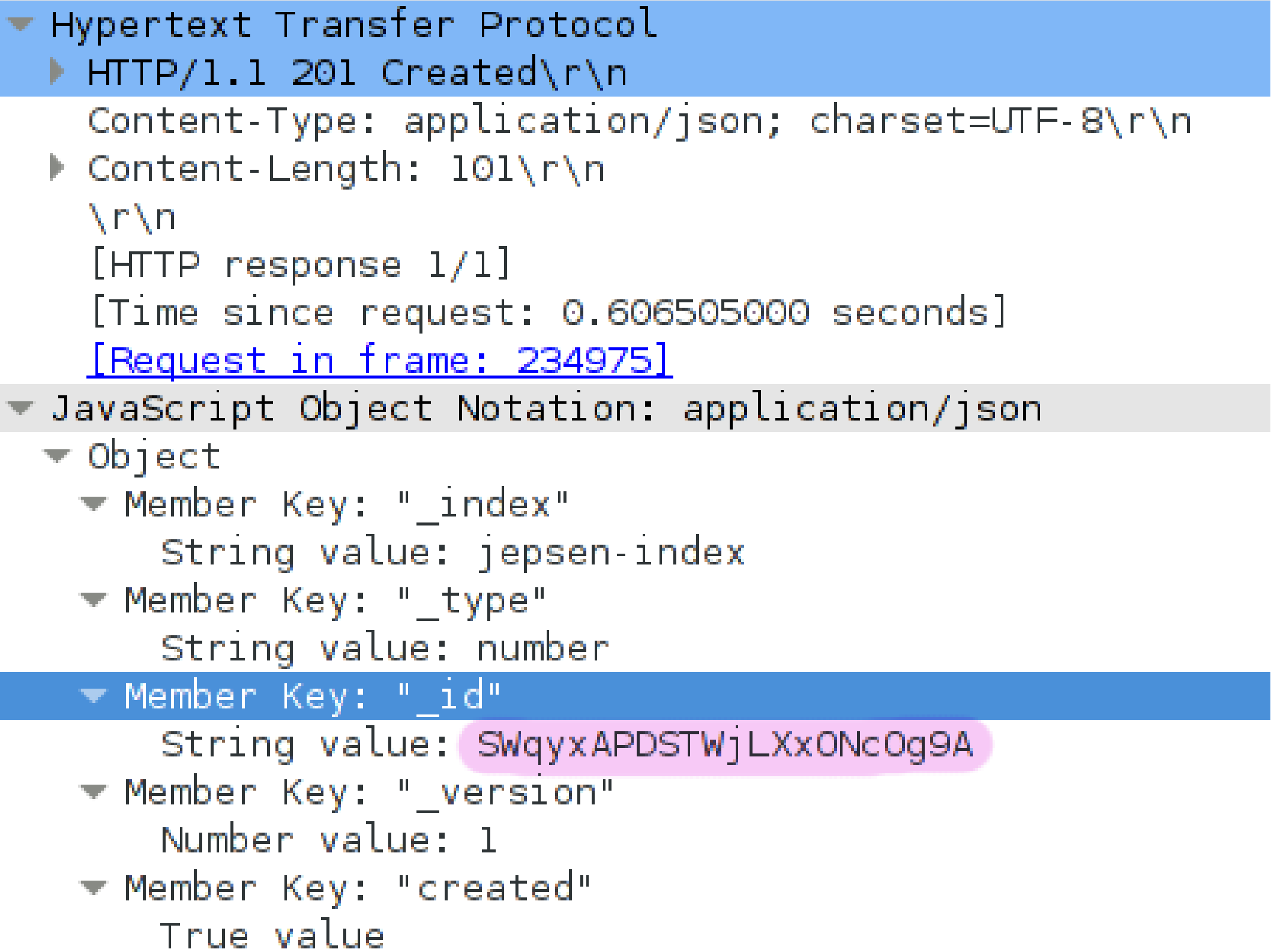
The response is similar to what we saw before, except that the `_id` field has been generated for us:

```
{
   "_index":    "website",
   "_type":     "blog",
   "_id":       "wM0OSFhDQXGZAWDf0-drSA",
   "_version":  1,
   "created":   true
}
```

Auto-generated IDs are 22 character long, URL-safe, Base64-encoded string *universally unique identifiers*, or UUIDs.

```
FAIL in (register-test) (elasticsearch_test.clj:49)
expected: (:valid? (:results test))
  actual: false
 {:valid?    false,
  :ok        …,
  :recovered #{420 438 457 475},
  :lost      #{418..419 421..427 429..437 439..444 446..456 458..463 465..474
476..481 483..493 495..500 502..503 505..506 509..510 514 516 520..521 524 527 529
531 534 536 539 542 544 547..548 552 554 557..558 562 564 567..568 572 574 577 579
582..583 587..588 592..593 597 599 602 604 607 609 612 614 617 619 622 624 626 629
631 634 636 638 642 644 647 649 652..653 657 659 662 664 667 669 671 674 676 679 681
684 686 689 691 694 696 698 701 704 706 709 711 715..716 720..721 725 727 730 732
735 737 741..742 746..747 749 752 754 757..758 762..763 767 769 772 775 777 779
782..783

…

1706..1707 1710 1712 1716..1717 1721..1722 1726..1727 1731..1732 1736..1737
1741..1742 1746..1747 1751..1752 1756 1761 1766 1771..1772 1776..1777 1781..1782
1786..1787 1791..1792 1796..1797 1801 1806..1807 1811..1812 1816..1817 1821
1826..1827 1831..1832 1836..1837 1841..1842 1846..1847 1851..1852 1856..1857
1861..1862 1866..1867 1871..1872 1877..1878 1882..1883 1887..1888 1892..1893
1897..1898 1902..1903 1907 1912..1913 1917..1918 1922..1923 1926 1931..1932
1936..1937 1941 1946..1947 1951 1953 1957},
  :unexpected       #{},
  :ok-frac          1103/1961
  :recovered-frac   4/1961,
  :lost-frac        645/1961,
  :unexpected-frac  0})
```


ins
ins
ins
ins
0
1
2
3
step down
0
1
2
3
4
0
1
2
3
ins
ins
ins
ins
ins


Hypertext Transfer Protocol
HTTP/1.1 201 Created\r\n
Content-Type: application/json; charset=UTF-8\r\n
Content-Length: 101\r\n
\r\n
[HTTP response 1/1]
[Time since request: 0.606505000 seconds]
[Request in frame: 234975]
JavaScript Object Notation: application/json
Object
Member Key: "_index"
String value: jepsen-index
Member Key: "_type"
String value: number
Member Key: "_id"
String value: SWqyxAPDSTWjLXxONcOg9A
Member Key: "_version"
Number value: 1
Member Key: "created"
True value

```
if (result.hasConflicts()) {
    // TODO: What should we do???
```

Multiple production user reports - caused by network failure or GC.

tallpsmith

11 months ago

suffered from this the other day when an accidental provisioning error had a 4GB ES Heap instance running on a 4GB O/S memory, which was always going to end up in trouble. The node swapped, process hung, and the intersection issue described here happened.

trevorreeves

6 months ago

We have been frequently experiencing this 'mix brain' issue in several of our clusters - up to 3 or 4 times a week. We have always had dedicated master eligible nodes (i.e. master=true, data=false), correctly configured minimum_master_nodes and have recently moved to 0.90.3, and seen no improvement in the situation.

Our situation appears to be IOWait related, in that a master node (also a data-node) hits an issue that causes extensive IOWait (a _scroll based search can trigger this, we already cap the # streams and Mb/second recovery rate through settings), the JVM becomes unresponsive. The other nodes that are doing the Master Fault Detection are configured with 3 x 30 second ping timeouts, all of which fail, and then they give up on the master.

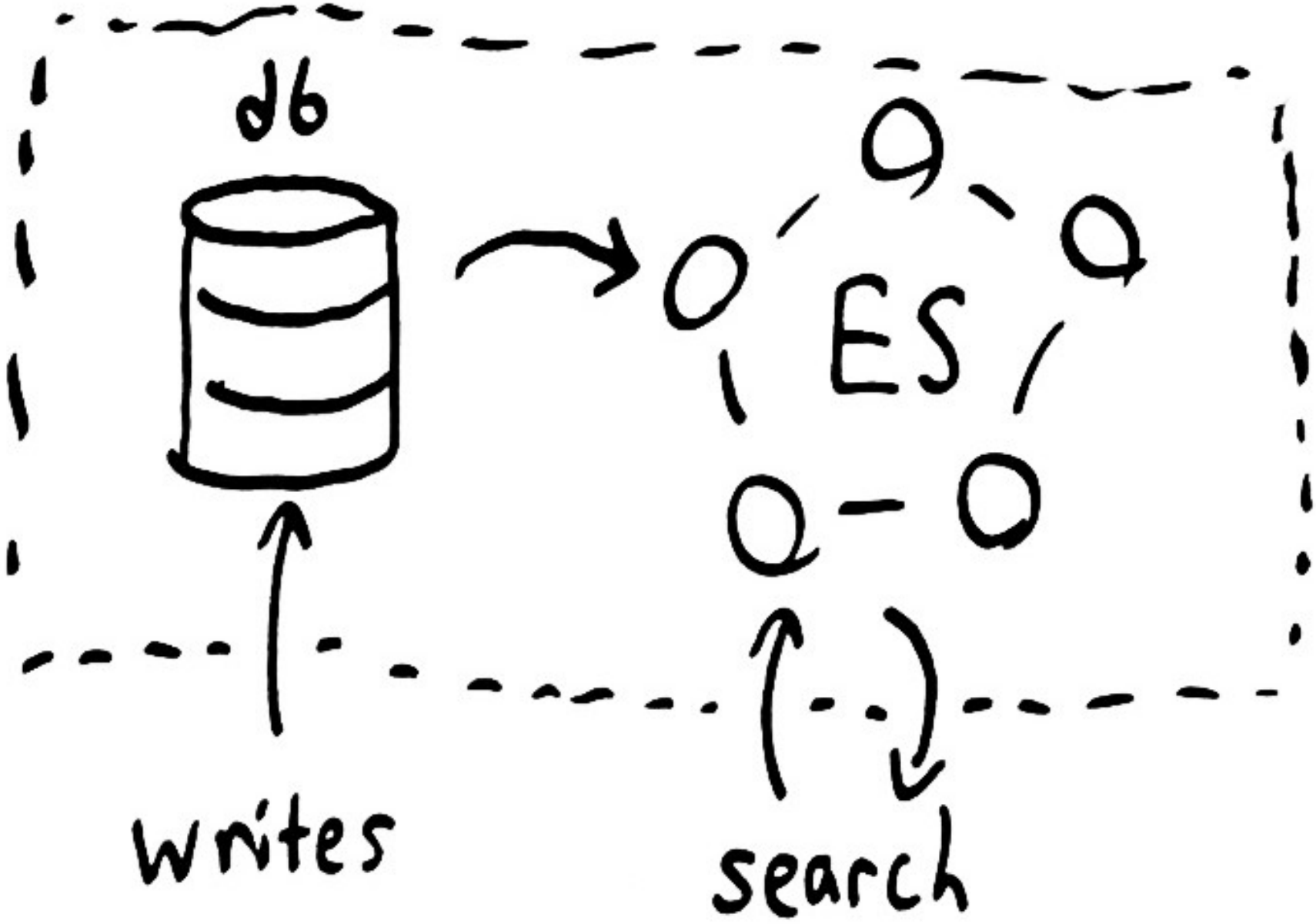
For my clusters, split-brains always occur when a node becomes isolated and then elects themselves as master. More visibility (logging) of the election process would be helpful. Re-discovery would be helpful as well since I rarely see the cluster self heal despite being in erroneous situations (nodes belongs to two clusters_. I am on version 0.90.2, so I am not

I am experiencing at random time spike in heap usage followed by long GC.
after the nodes finished the long GC cluster is getting into split brain situation with weird states where one of the cluster nodes is member of both sides of the split.
minimum_master_nodes option does not help in this case since the node exists in two of the different cluster states.

Zendisco is terrible.

Use ZK instead!

Always treat ES as a derivative datastore. Keep Primary data somewhere safe, & periodically resync.


db
ES
writes
search

In summary...

Jepsen tells you about

1 Particular

SYSTEM

Is YOUR system safe? I dunno!

Consider...

- Runtime pauses
- Network slowness
- One-way partitions

"How do we rely on clocks?"

Clients are
a part of the
distributed system!

Reason carefully about state transitions with critical impact

How much safety do you really need?

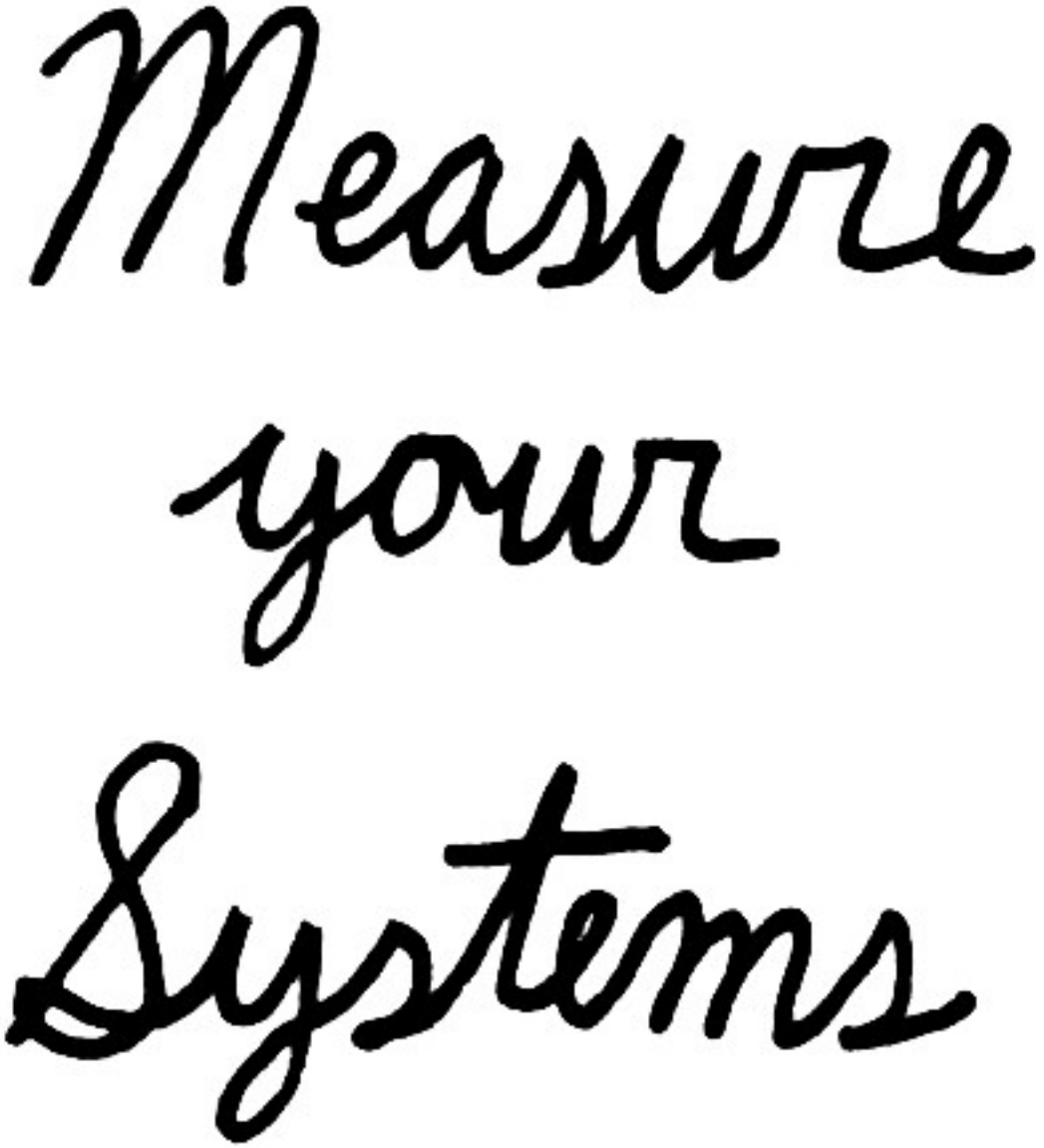
Data loss or inconsistency can be OK!

Balance

Complexity

— with —

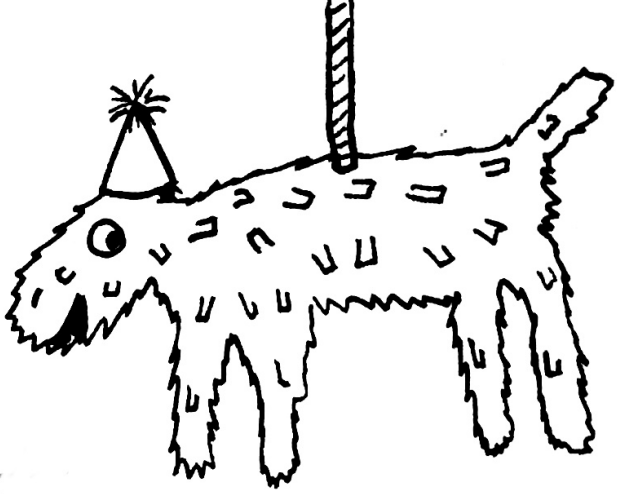
Correctness


Measure
your
Systems

Use iptables to simulate failures in prod, and record results!

- Most systems have gone untested under failure
- You'd be surprised what well-designed systems will do during a network partition
- Test <u>your</u> systems.

Thanks!



Peter Bailis

Bob Poekert

Kelly Sommers

Armon Dadgar

Joeseph Blomstedt

Comcast + Factual + et al

Aaron France

Alvaro Videla

Jacques Fuentes

Xiang Li

Tyler Schvett