Aperture

An algorithm for non-cooperative, client-side load balancing.

Ruben Oanta
@rubenoanta

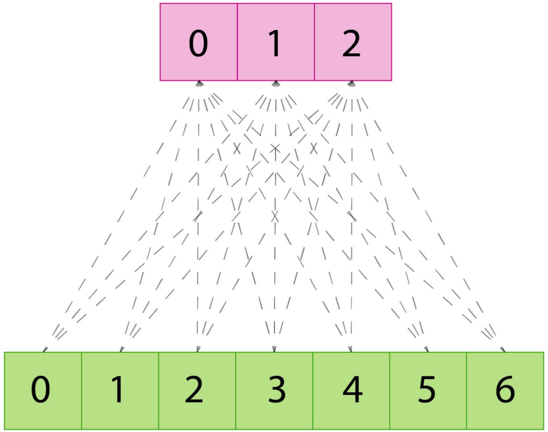
Bryce Anderson
@brycelanderson

TWITTERS CLIENT-SIDE LOAD BALANCER EVOLUTION

TWITTERS CLIENT-SIDE LOAD BALANCER EVOLUTION

1. A simple and fair load balancer

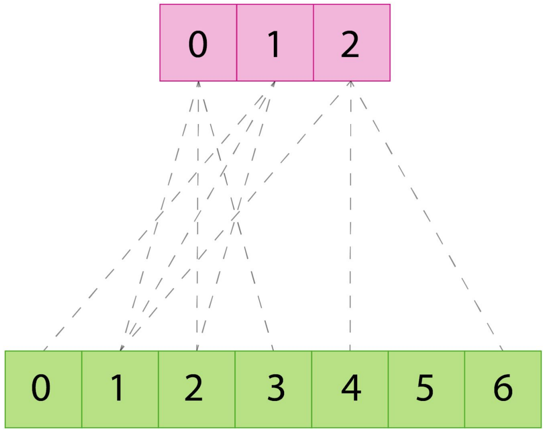
P2C



TWITTERS CLIENT-SIDE LOAD BALANCER EVOLUTION

1. A simple and fair load balancer
2. A scalable but *unfair* load balancer

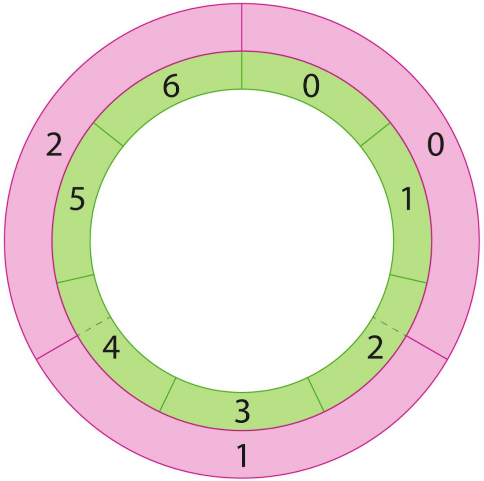
Random Aperture



TWITTERS CLIENT-SIDE LOAD BALANCER EVOLUTION

1. A simple and fair load balancer
2. A scalable but unfair load balancer
3. A scalable *and* fair load balancer

Deterministic Aperture



SERVICE-TO-SERVICE LOAD BALANCING

capacity utilization
safely make use of aggregate capacity of replicas

failure management
route around replicas when they inevitably fail

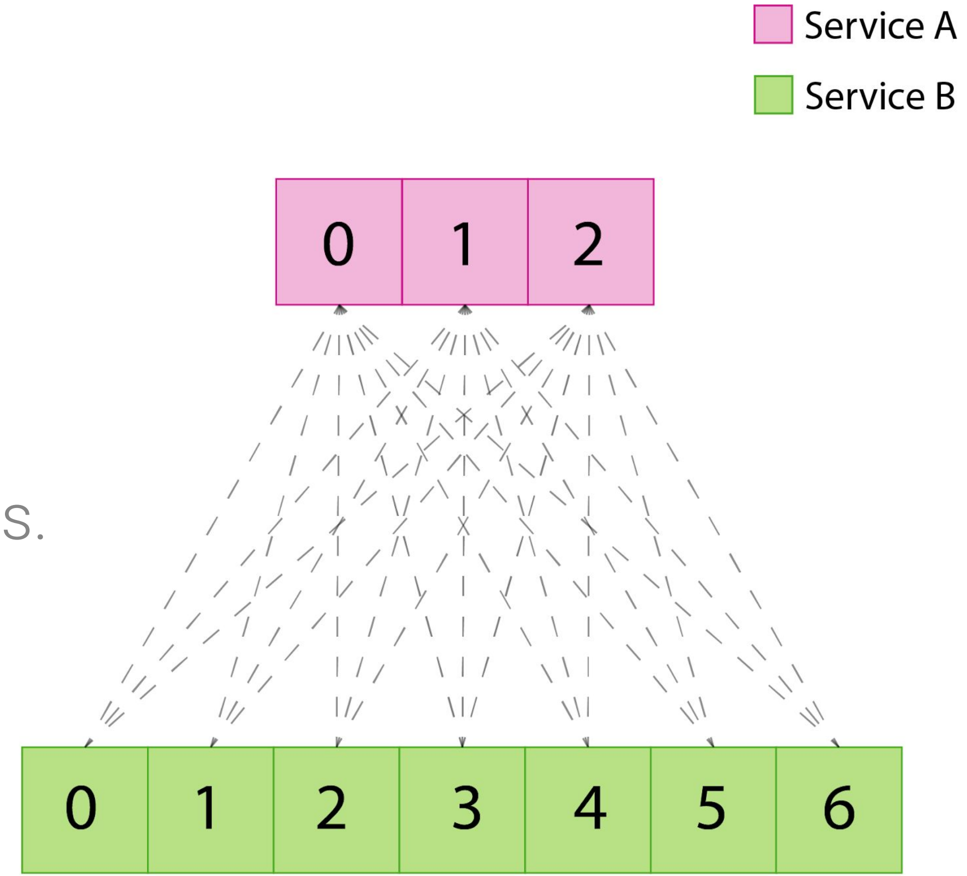
SERVICE-TO-SERVICE LOAD BALANCING

non-cooperative
multiple load balancers which make decisions independently

client-side
embedded within each replica of a service

load balancing
over sessions (OSI L5) and requests (OSI L7)

EXAMPLE SERVICE TOPOLOGY

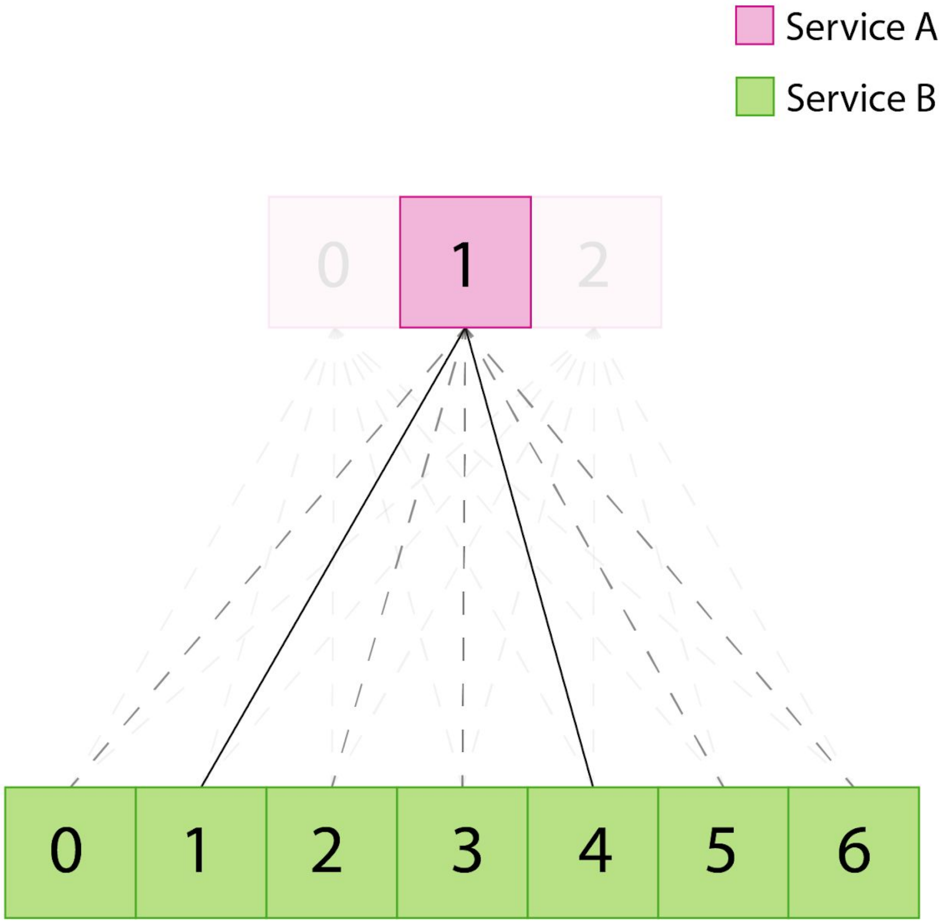


All clients connect to all servers.

PER REQUEST: P2C

1. Select two instances uniformly and randomly.

2. Of the two, select the 'best' instance.



M. Mitzenmacher, "The power of two choices in randomized load balancing", *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094-1104, October 2001.

```scala
// select two indices within `vec`, uniformly
// and randomly, without replacement.
val a = rng.nextInt(vec.size)
var b = rng.nextInt(vec.size - 1)
if (b >= a) { b = b + 1 }

val nodeA = vec(a)
val nodeB = vec(b)

// If both nodes are in the same health status, we pick
// the least loaded one. Otherwise we pick the one
// that's healthier.
val aStatus = nodeA.status
val bStatus = nodeB.status
if (aStatus == bStatus) {
  if (nodeA.load <= nodeB.load) nodeA else nodeB
} else {
  if (Status.best(aStatus, bStatus) == aStatus) nodeA else nodeB
}
```

```
// select two indices within `vec`, uniformly
// and randomly, without replacement.
val a = rng.nextInt(vec.size)
var b = rng.nextInt(vec.size - 1)
if (b >= a) { b = b + 1 }

val nodeA = vec(a)
val nodeB = vec(b)

// If both nodes are in the same health status, we pick
// the least loaded one. Otherwise we pick the one
// that's healthier.
val aStatus = nodeA.status
val bStatus = nodeB.status
if (aStatus == bStatus) {
  if (nodeA.load <= nodeB.load) nodeA else nodeB
} else {
  if (Status.best(aStatus, bStatus) == aStatus) nodeA else nodeB
}
```

```scala
// select two indices within `vec`, uniformly
// and randomly, without replacement.
val a = rng.nextInt(vec.size)
var b = rng.nextInt(vec.size - 1)
if (b >= a) { b = b + 1 }

val nodeA = vec(a)
val nodeB = vec(b)

// If both nodes are in the same health status, we pick
// the least loaded one. Otherwise we pick the one
// that's healthier.
val aStatus = nodeA.status
val bStatus = nodeB.status
if (aStatus == bStatus) {
  if (nodeA.load <= nodeB.load) nodeA else nodeB
} else {
  if (Status.best(aStatus, bStatus) == aStatus) nodeA else nodeB
}
```

PER REQUEST: P2C

fair request distribution
request load is even with homogenous replicas

efficient
fully concurrent, constant time for selection + comparison

decoupled selection + comparison
allows for sophisticated definitions of load

PER SESSION: IT'S A MESH!

wasted resources
everyone talks to everyone

no isolation
independently discover the same problems

low concurrency
poor load metric performance without concurrent requests

How can we reduce the number of sessions?

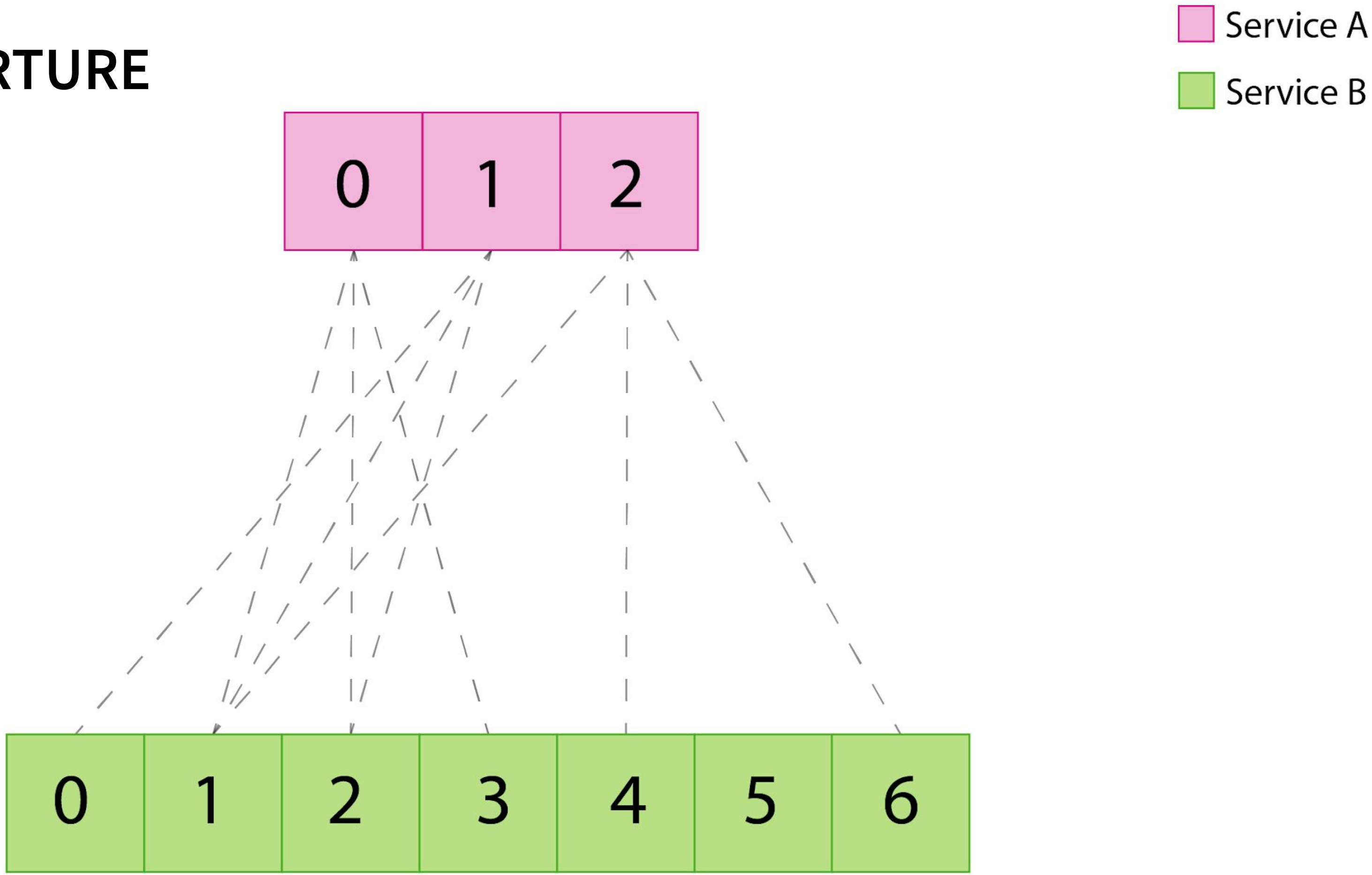
RANDOM APERTURE

random
replicas selected within a random window

dynamic sizing
can grow or shrink based on feedback controller

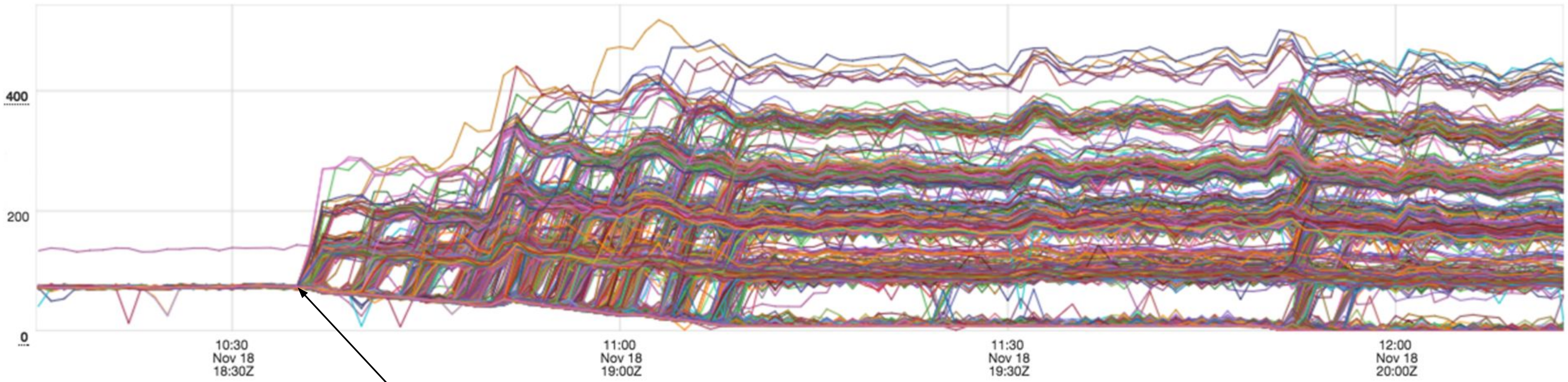
highly concurrent
aperture is smallest subset to satisfy concurrency

RANDOM APERTURE



RANDOM APERTURE: UNFAIR

RPS PER SERVER / TIME

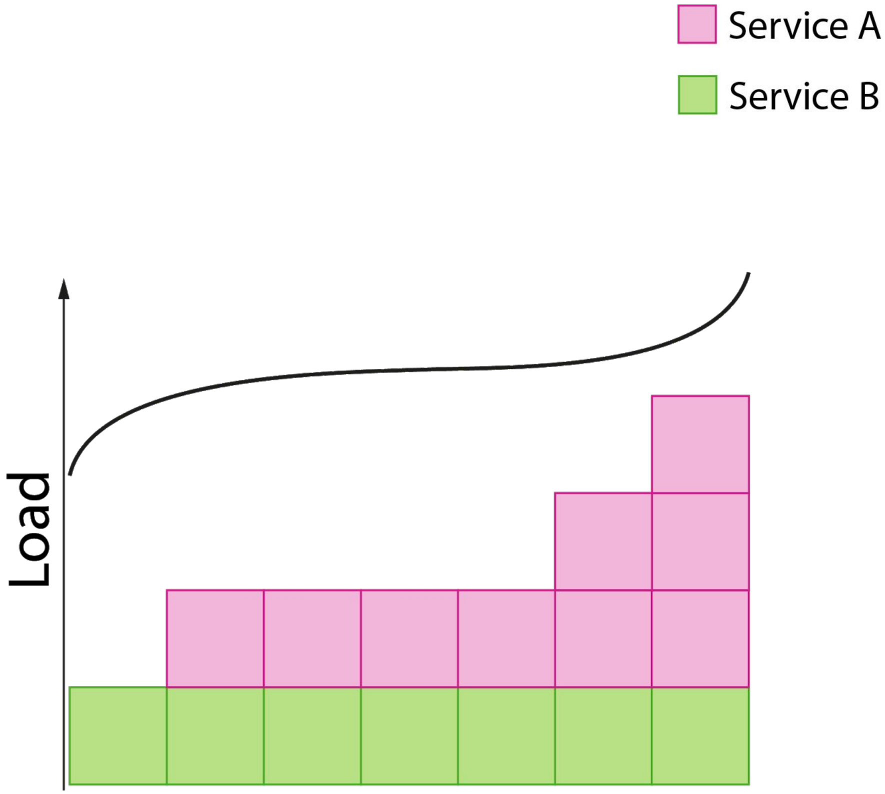


clients deployed
random aperture

RANDOM IS STATISTICAL

Results in a load distribution that closely resembles a binomial distribution.

Minimizing the “banding” requires tuning which can only be eliminated when the aperture is the size of all the backend replicas.



CONFIGURED RANDOM APERTURE

WolframAlpha computational intelligence.

binomial distribution(500000, 0.001)

Browse Examples

Surprise Me

Input:

binomial distribution	number of trials	$n = 500\,000$
	probability of success	$p = 0.001$

Open code

Statistical properties:

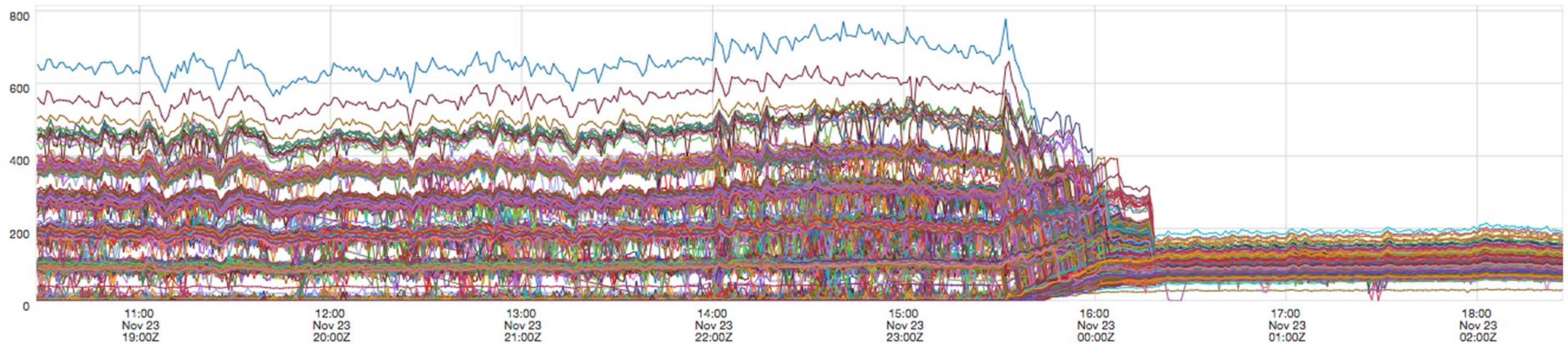
More

mean	500
standard deviation	22.3495
variance	499.5
skewness	0.0446542
kurtosis	3.00199

Probability density function (PDF):

CONFIGURED RANDOM APERTURE

RPS PER SERVER / TIME



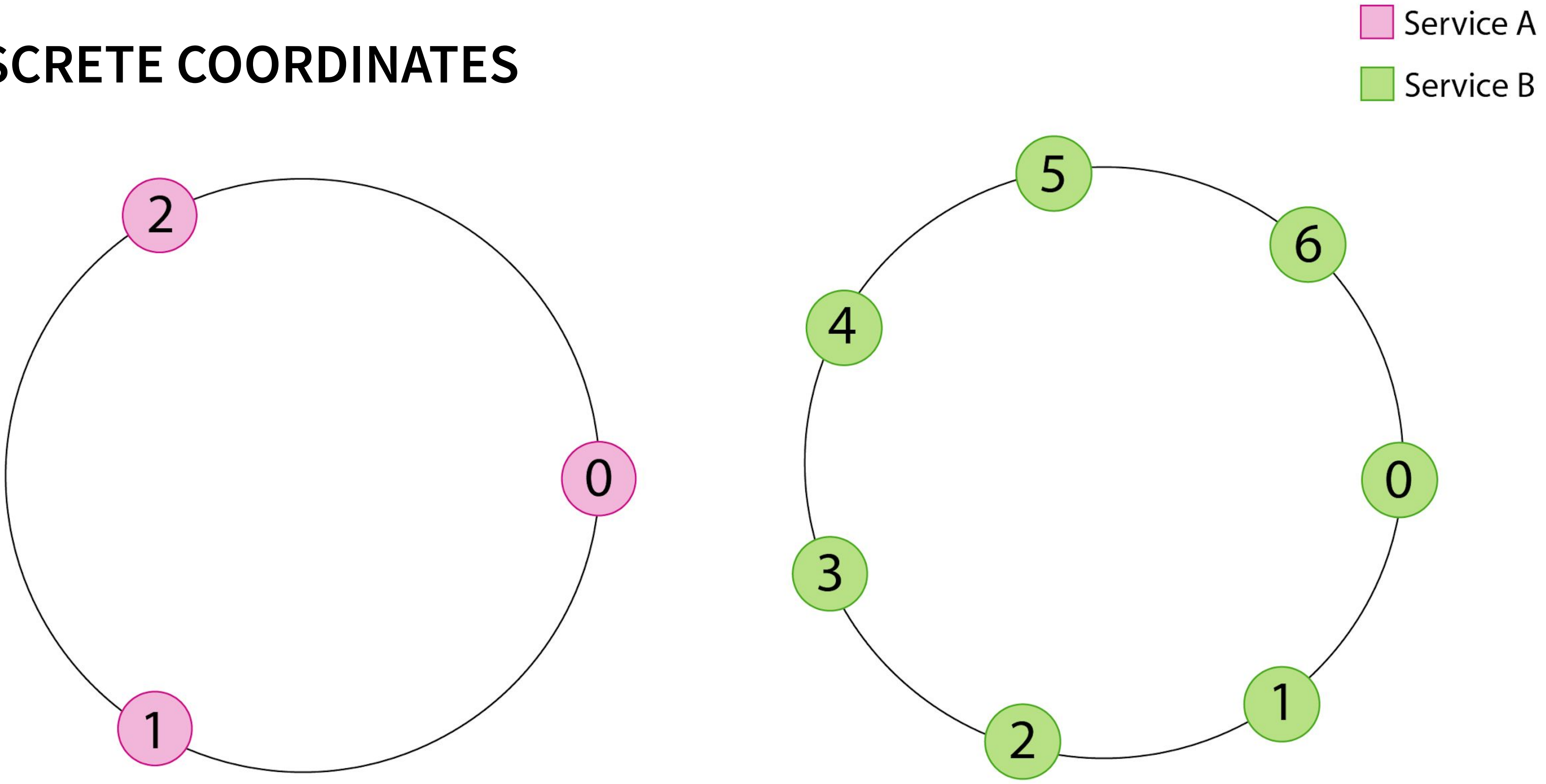
Distributing the configuration burden for core pieces of infrastructure will likely converge to poorly configured infrastructure.

How can we improve aperture?

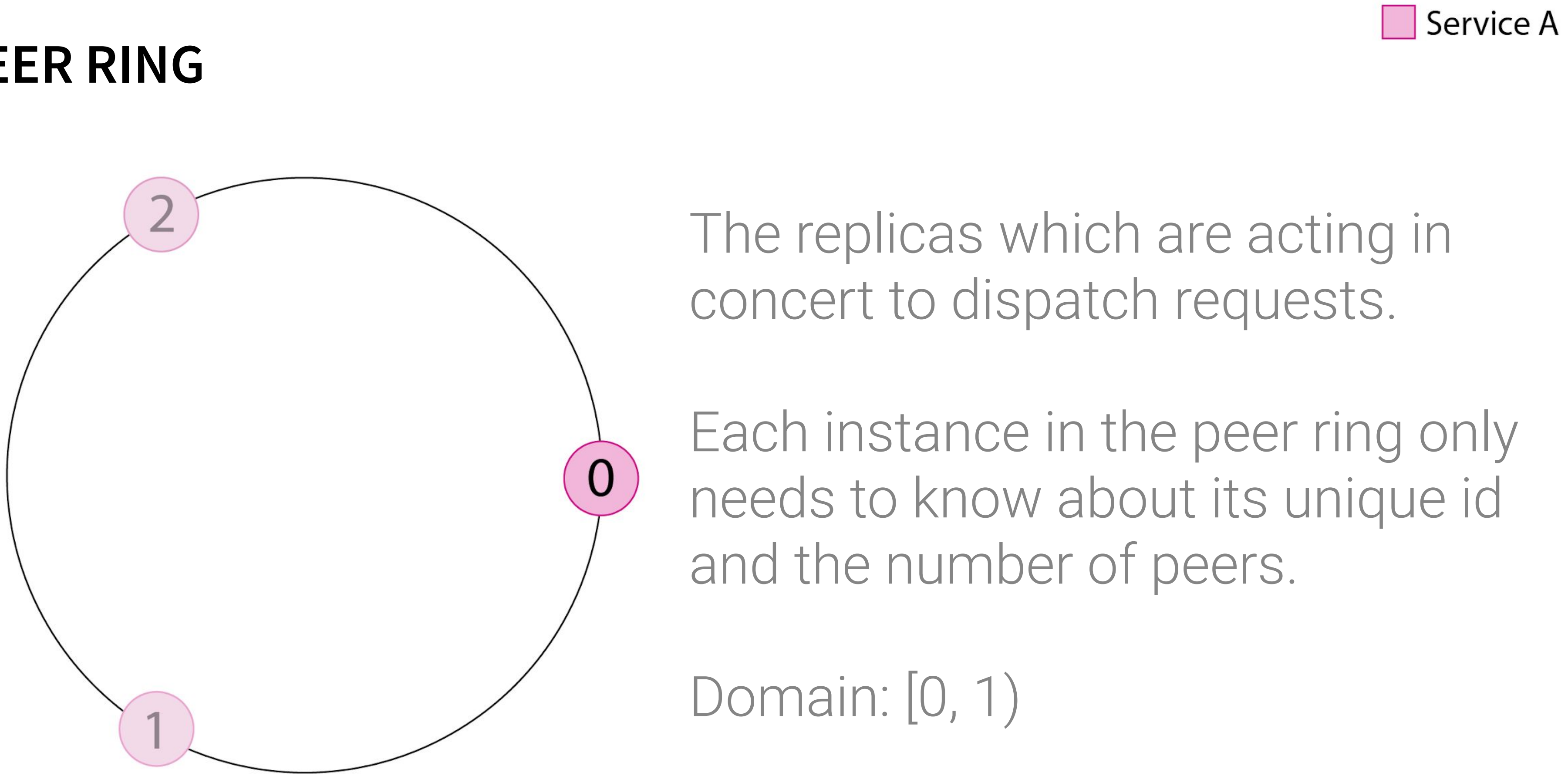
fairer
less config

distributed
subset

DISCRETE COORDINATES



PEER RING



The replicas which are acting in concert to dispatch requests.

Each instance in the peer ring only needs to know about its unique id and the number of peers.

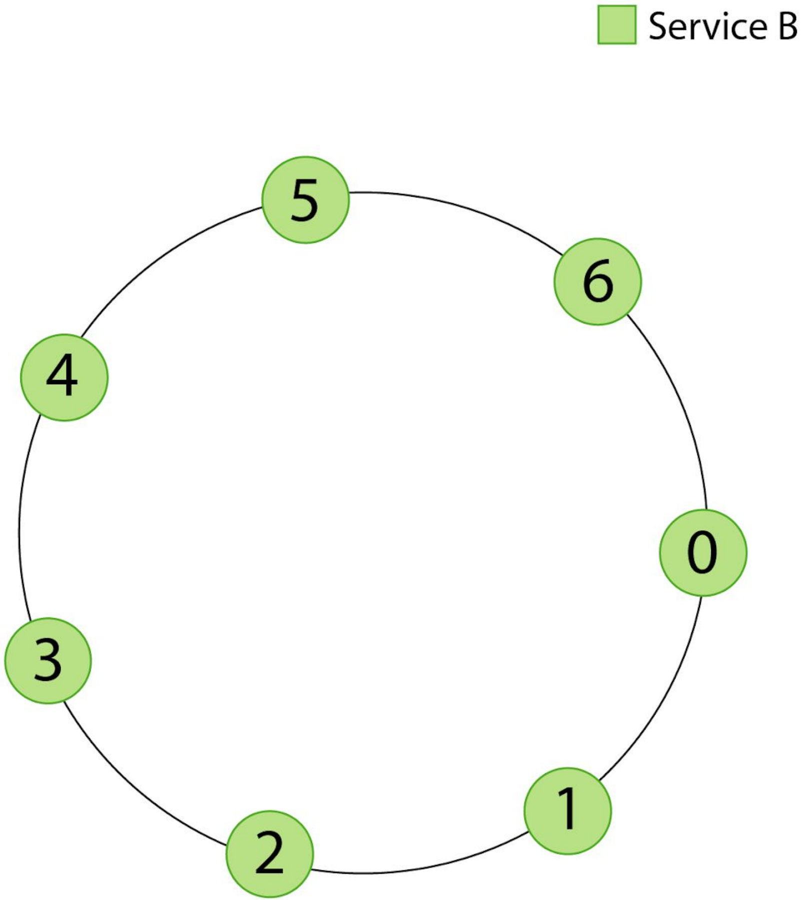
Domain: [0, 1)

DESTINATION RING

The ring which will be receiving requests.

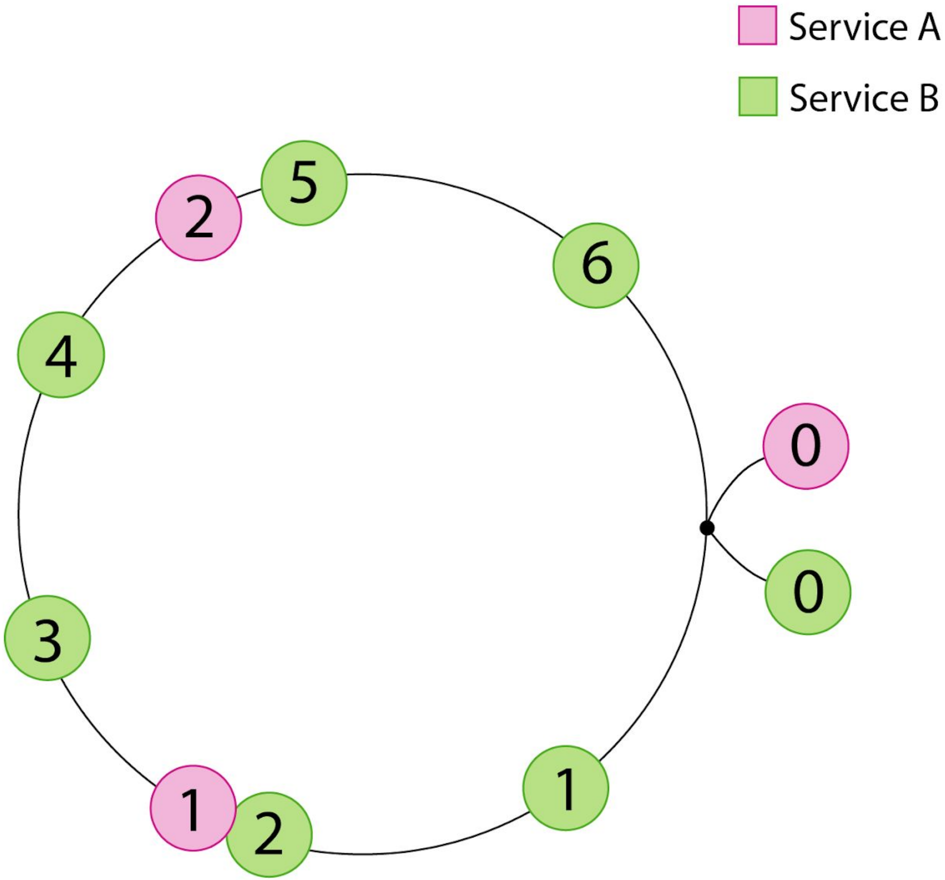
Each peer computes this ring via metadata received from service discovery.

Domain: [0, 1)



COMPOSITE RINGS

0: [0, 1, 2, 3, 4, 5, 6]
1: [3, 4, 5, 6, 0, 1, 2]
2: [5, 6, 0, 1, 2, 3, 4]

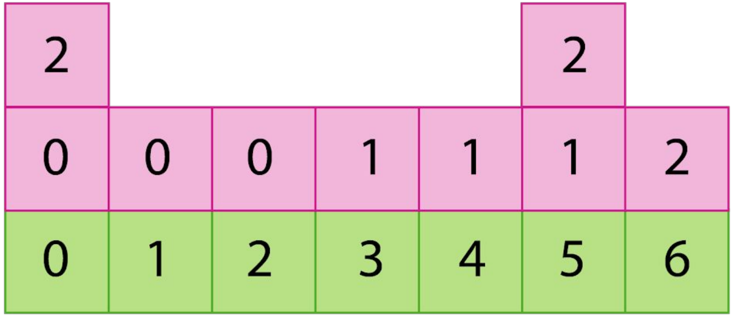


SESSION HISTOGRAM

Service A
Service B

Service A

0: [0, 1, 2, 3, 4, 5, 6]
1: [3, 4, 5, 6, 0, 1, 2]
2: [5, 6, 0, 1, 2, 3, 4]



MULTIPLE SERVICE RINGS

Service A

0: [0, 1, 2, 3, 4, 5, 6]

1: [3, 4, 5, 6, 0, 1, 2]

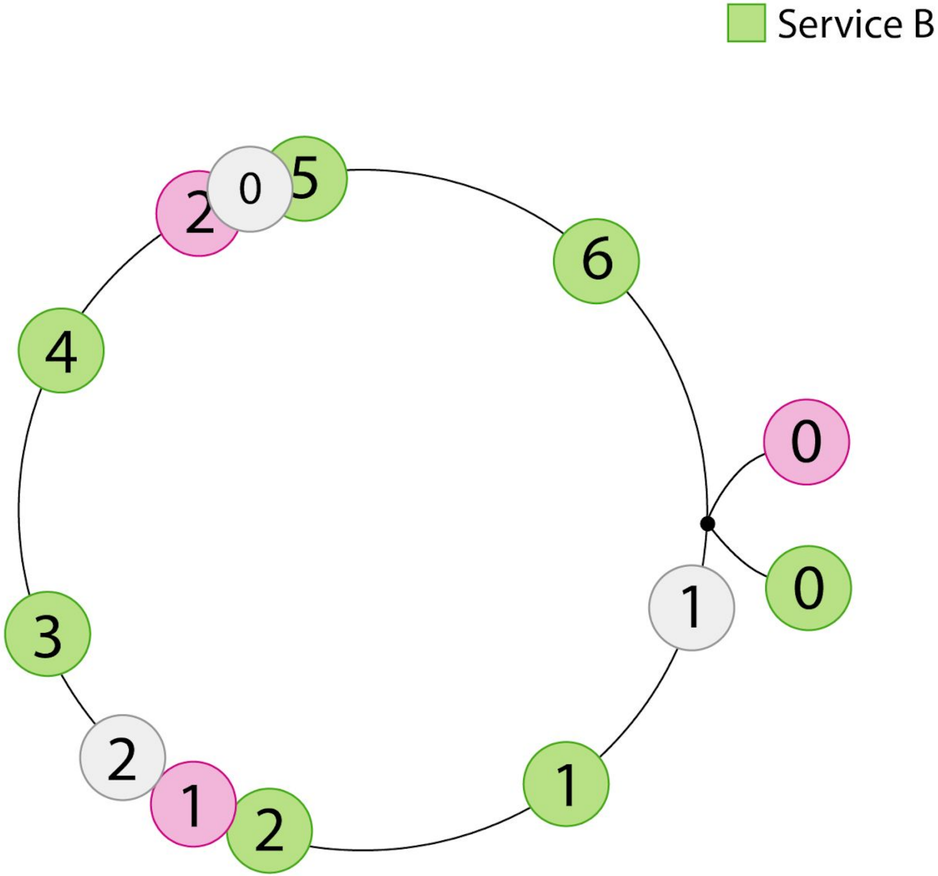
2: [5, 6, 0, 1, 2, 3, 4]

Service C

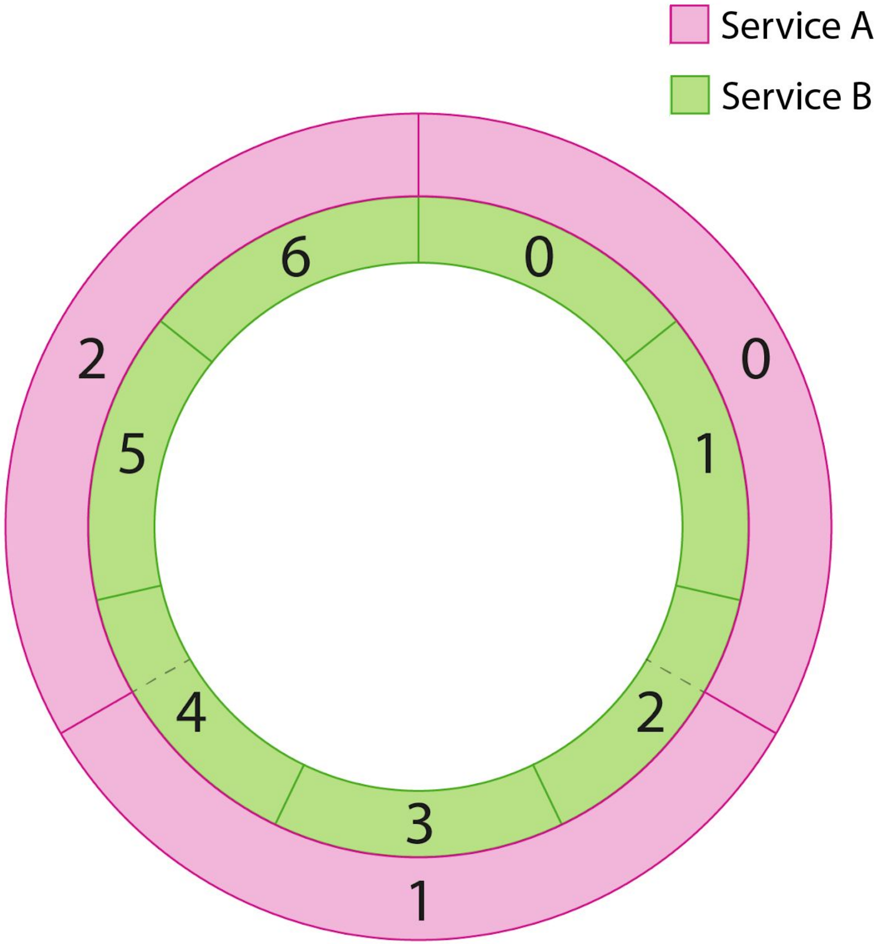
0: [5, 6, 0, 1, 2, 3, 4]

1: [1, 2, 3, 4, 5, 6, 0]

2: [3, 4, 5, 6, 0, 1, 2]



CONTINUOUS COORDINATES



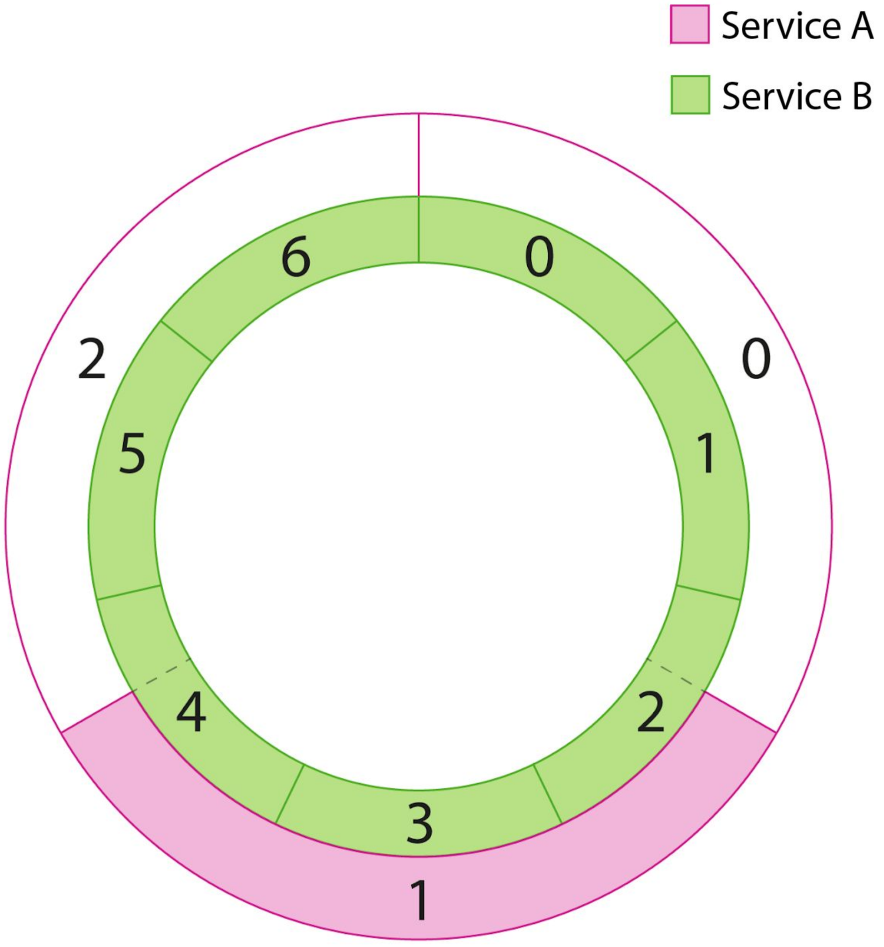
Services fully occupy the same domain.

Load balancers can map from their respective range to discrete destinations.

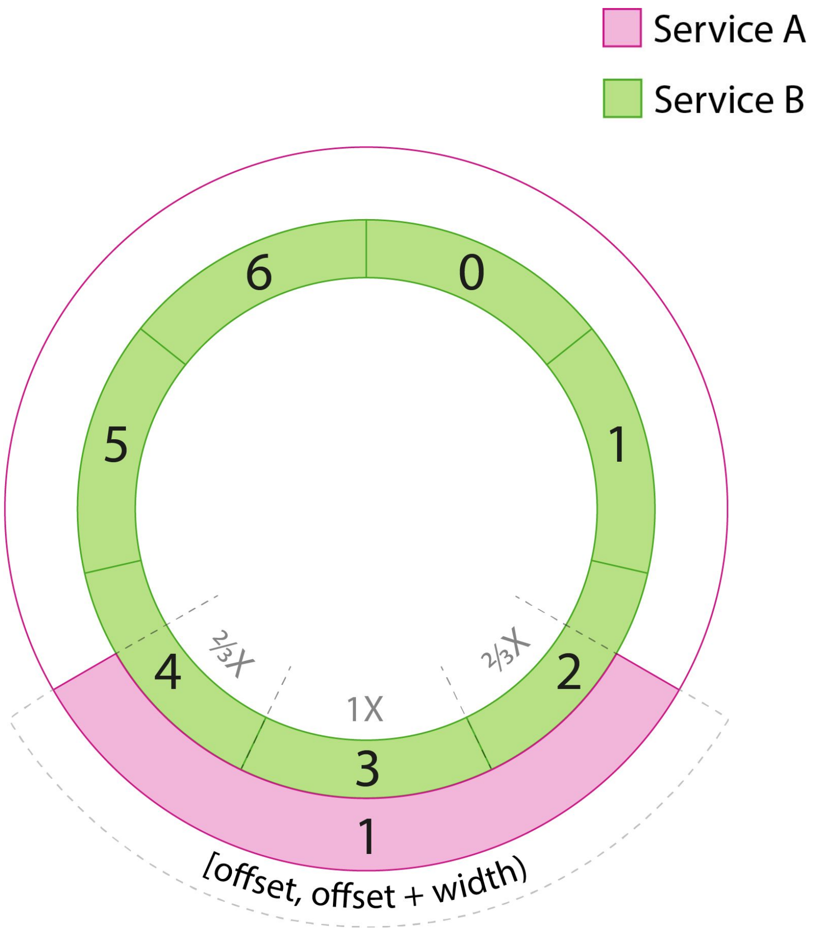
P2C + FRACTIONAL LOAD

Each load balancer picks two coordinates randomly within its range and maps them to discrete destinations.

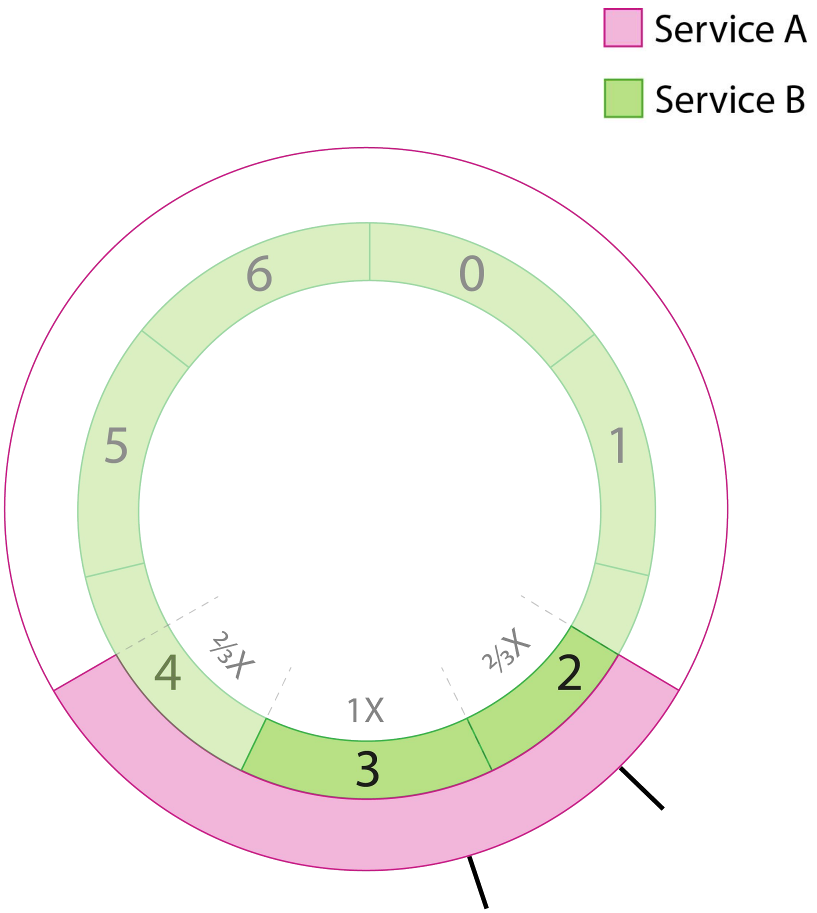
This inherently respects the fractional boundary conditions.



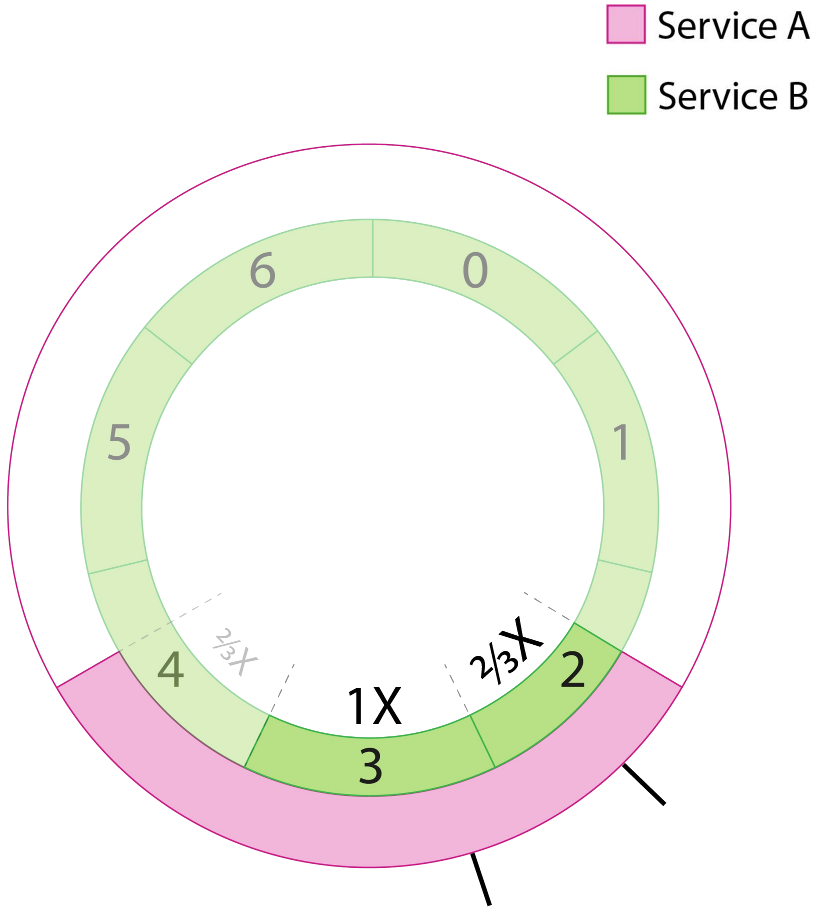
```scala
// compute the offset and width of this balancer.
val offset = coord.offset
val width = apertureWidth

// select two coordinates, randomly and uniformly,
// within our range [offset, offset + width) and map
// them to the destination ring.
val (a, b) = destRing.pick2(offset, width)

val nodeA = vector(a)
val nodeB = vector(b)

val aStatus = nodeA.status
val bStatus = nodeB.status
if (aStatus == bStatus) {
  // what proportion of a and b, respectively,
  // fall within [offset, offset + width)?
  val aw = destRing.weight(a, offset, width)
  val bw = destRing.weight(b, offset, width)
  // weight the load w.r.t to the ring proportions
  // to avoid biasing towards the node picked less often.
  if (nodeA.load / aw <= nodeB.load / bw) nodeA else nodeB
} else {
  if (Status.best(aStatus, bStatus) == aStatus) nodeA else nodeB
}
```



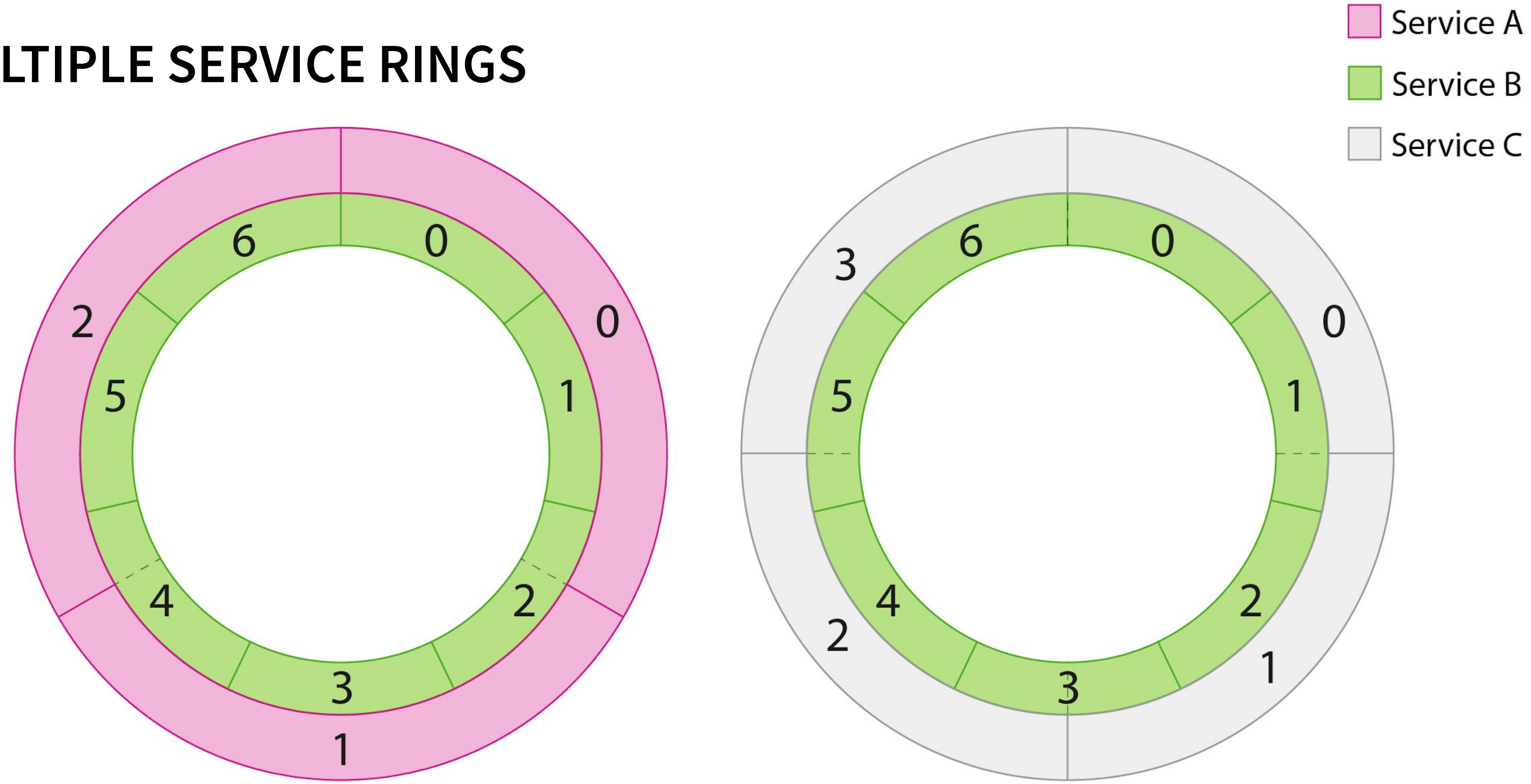
```scala
// compute the offset and width of this balancer.
val offset = coord.offset
val width = apertureWidth

// select two coordinates, randomly and uniformly,
// within our range [offset, offset + width) and map
// them to the destination ring.
val (a, b) = destRing.pick2(offset, width)

val nodeA = vector(a)
val nodeB = vector(b)

val aStatus = nodeA.status
val bStatus = nodeB.status
if (aStatus == bStatus) {
  // what proportion of a and b, respectively,
  // fall within [offset, offset + width)?
  val aw = destRing.weight(a, offset, width)
  val bw = destRing.weight(b, offset, width)
  // weight the load w.r.t to the ring proportions
  // to avoid biasing towards the node picked less often.
  if (nodeA.load / aw <= nodeB.load / bw) nodeA else nodeB
} else {
  if (Status.best(aStatus, bStatus) == aStatus) nodeA else nodeB
}
```



```
// compute the offset and width of this balancer.
val offset = coord.offset
val width = apertureWidth

// select two coordinates, randomly and uniformly,
// within our range [offset, offset + width) and map
// them to the destination ring.
val (a, b) = destRing.pick2(offset, width)

val nodeA = vector(a)
val nodeB = vector(b)

val aStatus = nodeA.status
val bStatus = nodeB.status
if (aStatus == bStatus) {
  // what proportion of a and b, respectively,
  // fall within [offset, offset + width)?
  val aw = destRing.weight(a, offset, width)
  val bw = destRing.weight(b, offset, width)
  // weight the load w.r.t to the ring proportions
  // to avoid biasing towards the node picked less often.
  if (nodeA.load / aw <= nodeB.load / bw) nodeA else nodeB
} else {
  if (Status.best(aStatus, bStatus) == aStatus) nodeA else nodeB
}
```



MULTIPLE SERVICE RINGS



CONTINUOUS COORDINATE MODEL

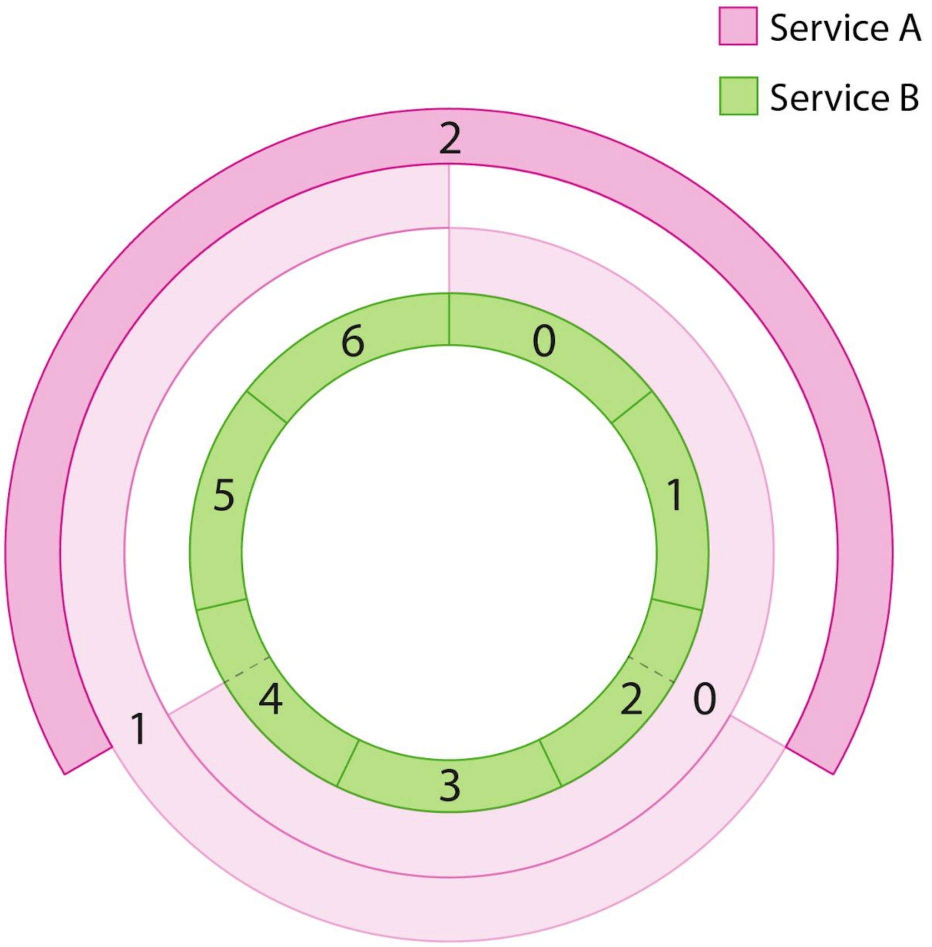
fair request distribution
with distinct services talking to the same destination ring

distributed
light coordination around metadata to construct rings

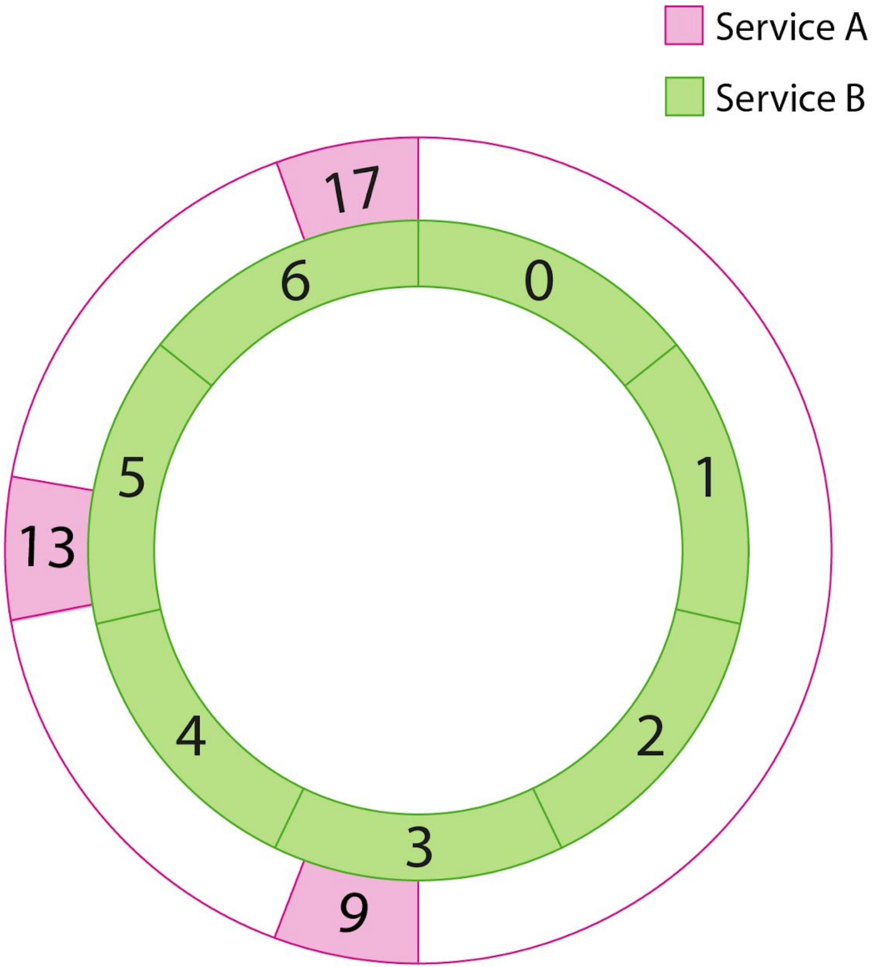
fewer sessions
aperture size naturally falls out of representation

DYNAMIC APERTURE SIZE

The aperture can grow/shrink so long as the peer ring completes whole rotations around destination ring.



RESILIENCY



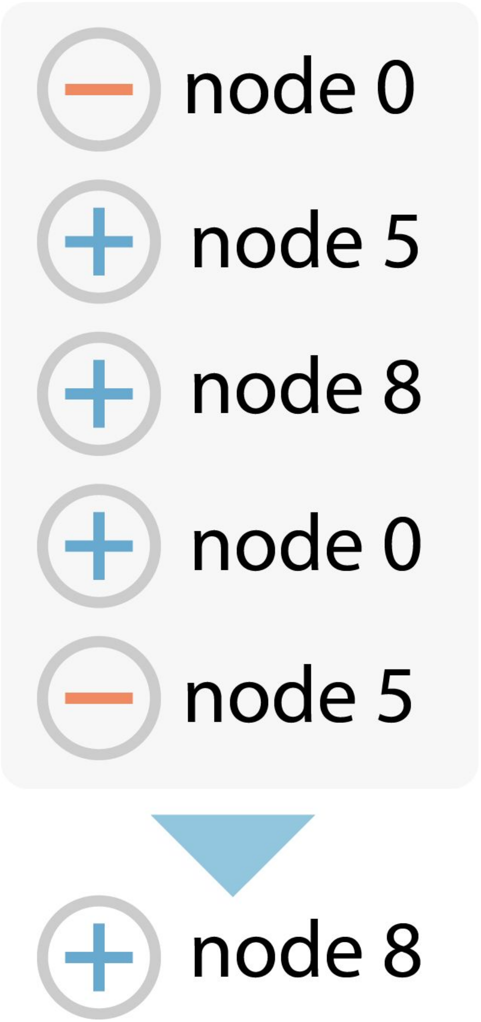
peer size heuristics

Nodes are placed closer to their final position by inferring the size of the ring when receiving updates.

RESILIENCY

coalesce updates

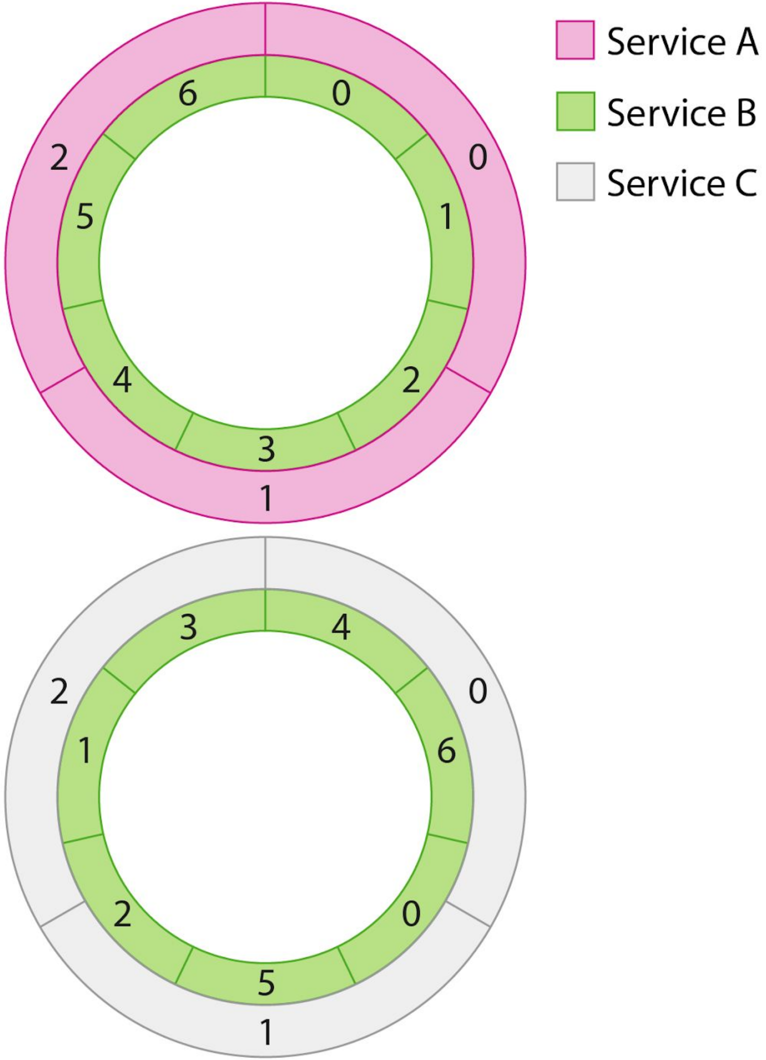
Changes are buffered and combined in order to avoid transient ring states.



RESILIENCY

entropy

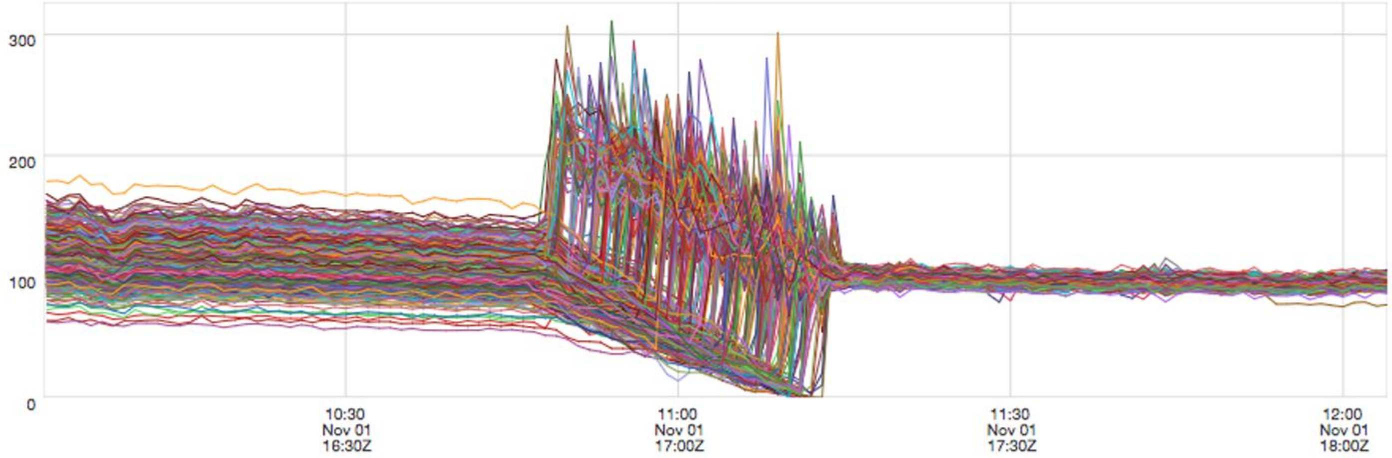
The destination ring is pseudo-randomized to avoid any synchronization across distinct peer rings.



Production Results

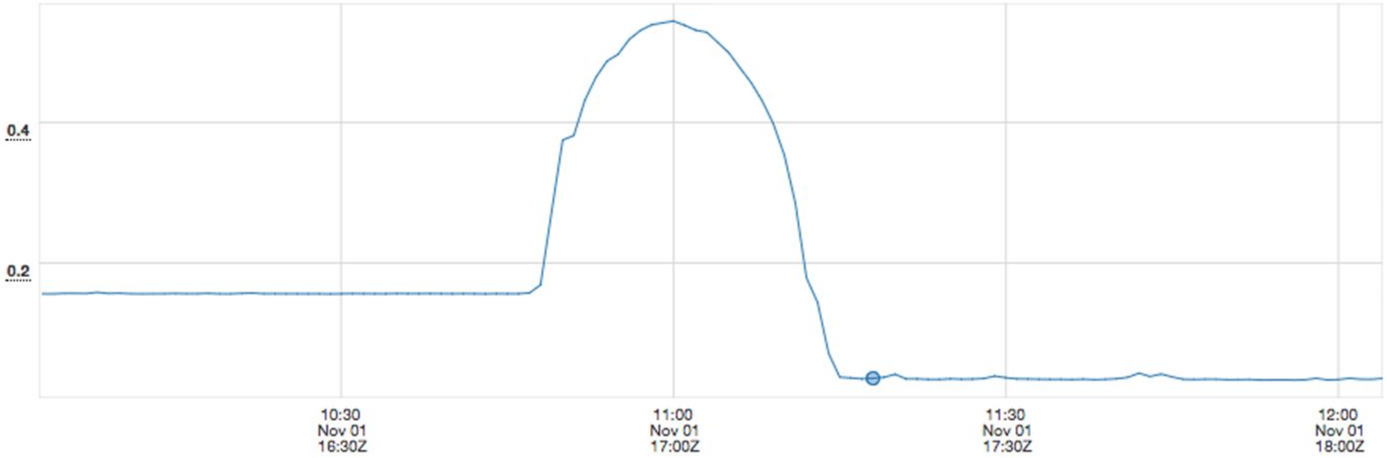
MIGRATION FROM RANDOM APERTURE TO D-APERTURE

RPS PER SERVER / TIME



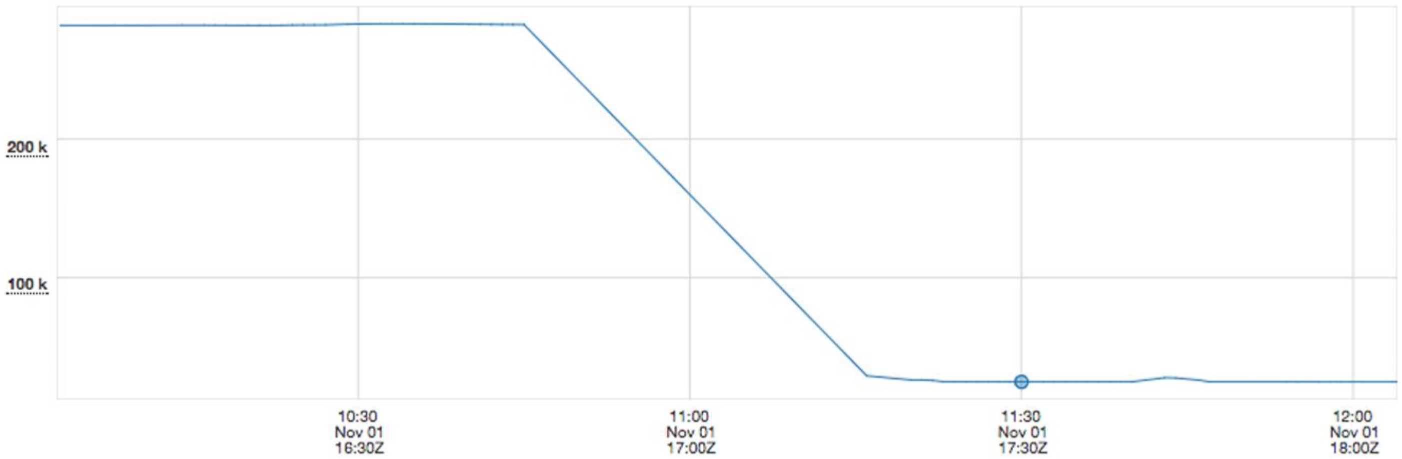
MIGRATION FROM RANDOM APERTURE TO D-APERTURE

78% reduction in relative standard deviation request rate



MIGRATION FROM RANDOM APERTURE TO D-APERTURE

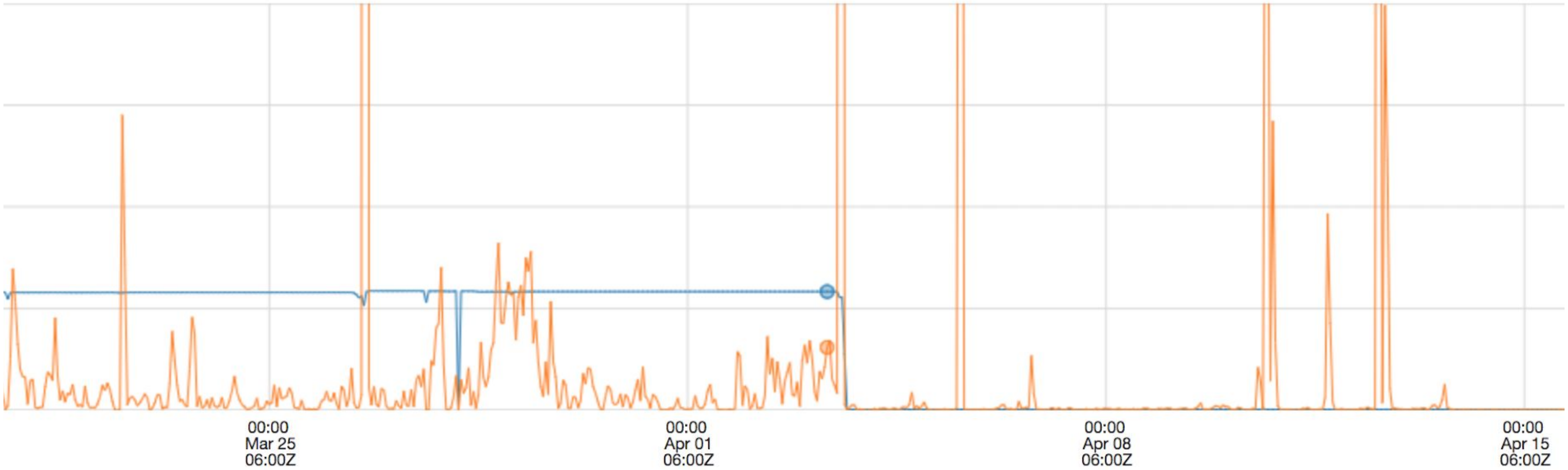
Drop from ~280K to ~25K aggregate connections (91%)



SECOND-ORDER RESULTS

20-25% less CPU used
Total garbage collection (GC) time cut in half
75% fewer failures
~20% reduction in latency at 99.9th percentile

REDUCTION IN REQUEST RETRIES



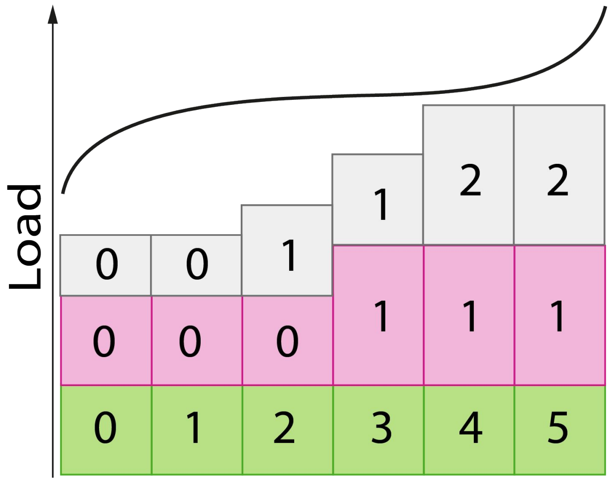
LIMITATIONS

unequal workloads

If different clients have unequal demands of the client we again get to unbalanced load on the backend.

bursty traffic

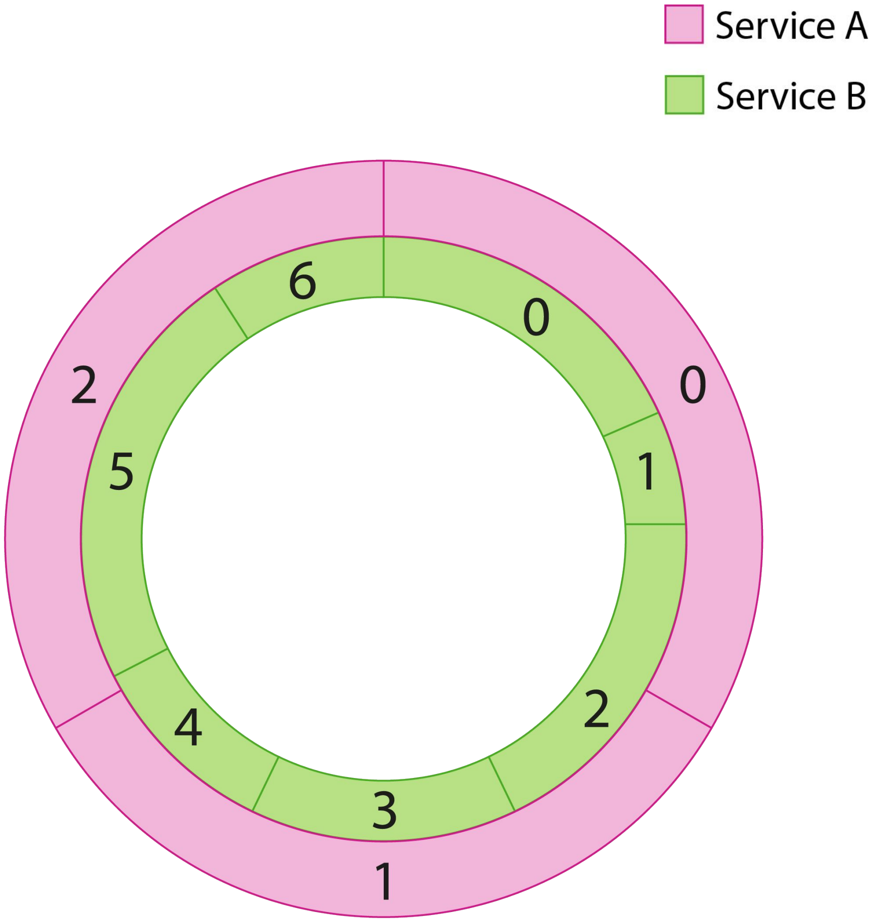
Bursts of traffic break the assumption that incoming load is ‘smooth’.



FUTURE WORK

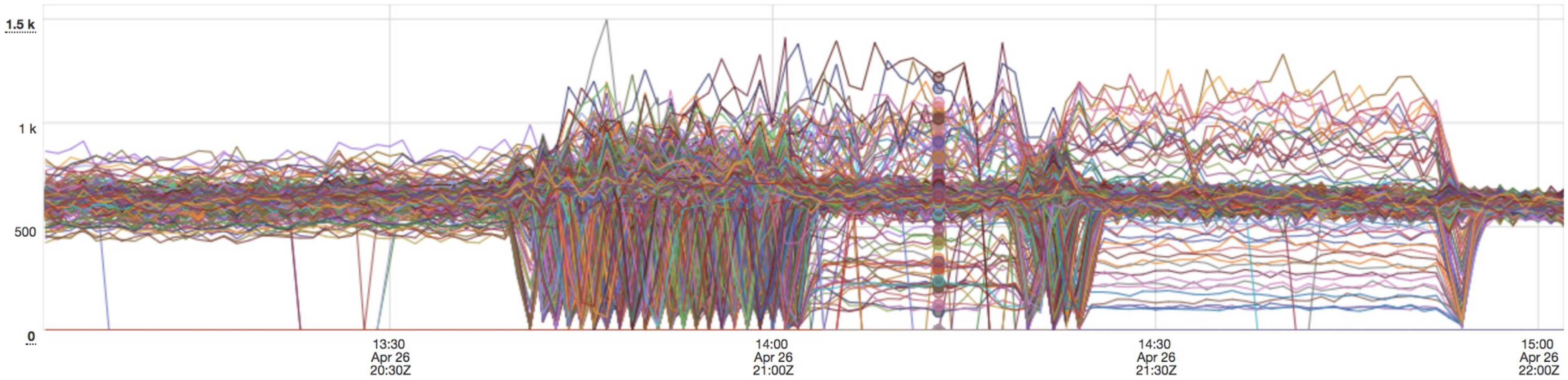
flexible node capacity

Some nodes will be better than others, heterogeneous hardware etc, and we can size serving units accordingly.



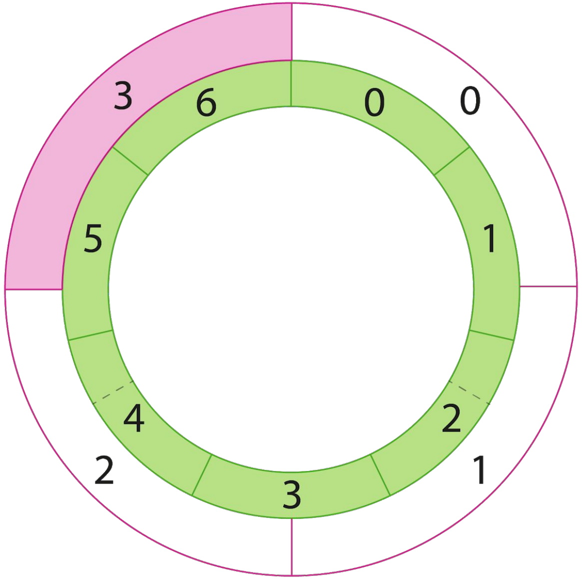
THIS IS FINE

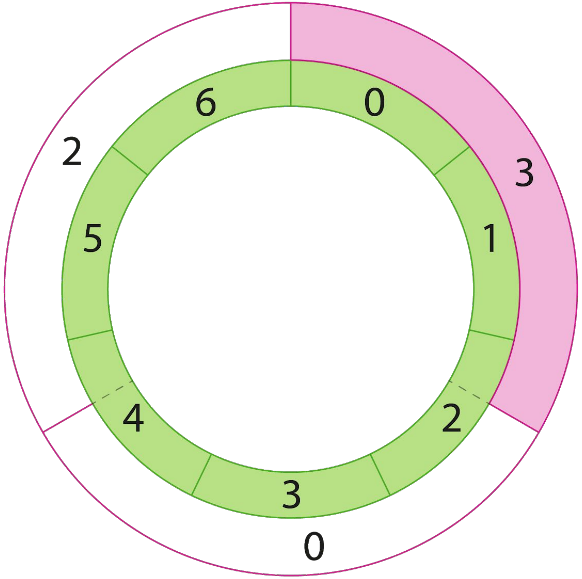
THIS IS FINE - LOOK MA! I'M INSTANCE 100 OF 90!



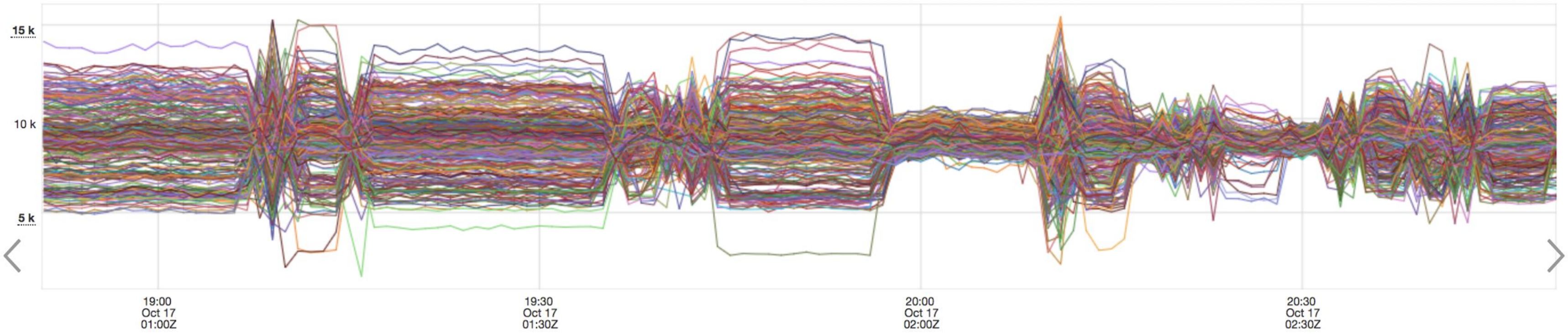
THIS IS FINE - LOOK MA! I’M INSTANCE 100 OF 90!

As we re-deploy instance 1, instance 3 overflows around the ring.

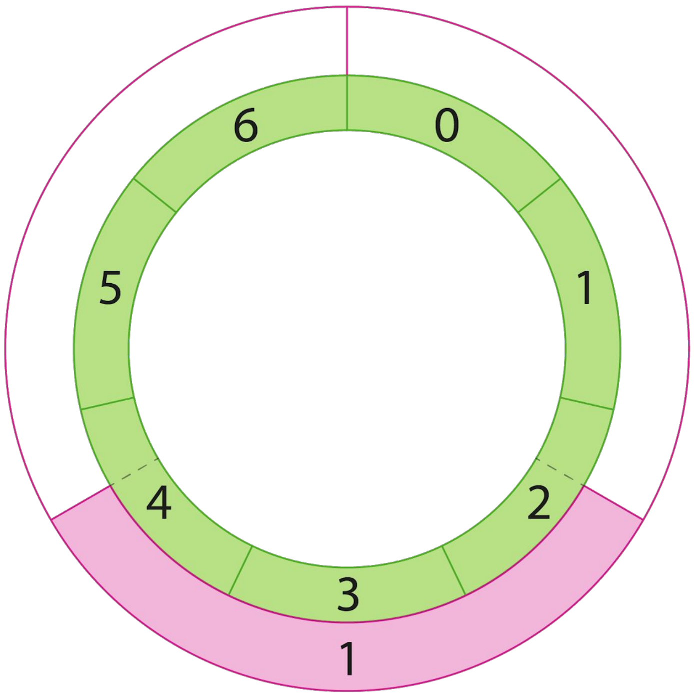




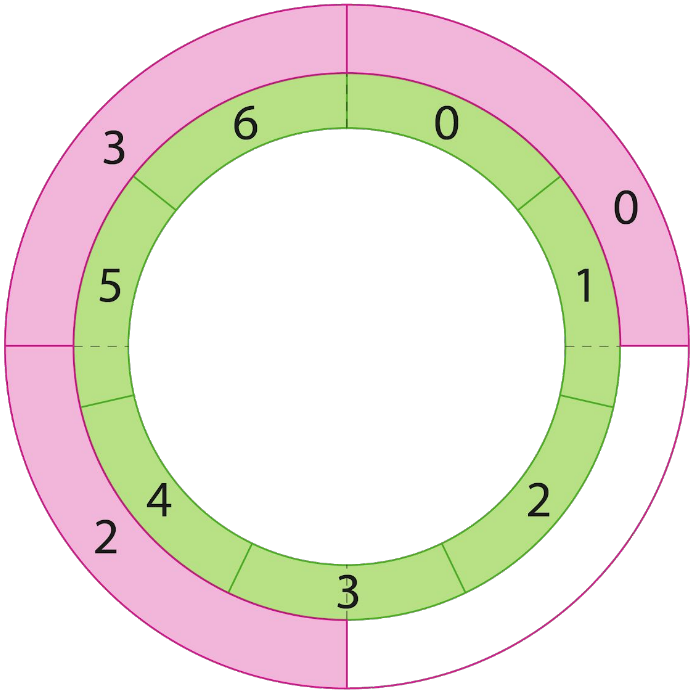
THIS IS FINE – UPDATES, SMUPDATES…



THIS IS FINE – UPDATES, SMUPDATES...



Peer Instance 1



Everyone Else

attribution

Billy Becker, Marius Eriksen, Daniel Furse, Steve Gury, Eugene Ma, Nick Matheson, Moses Nakamura, Kevin Oliver, Brian Rutkin, Daniel Schobel

code

github.com/twitter/finagle

We’re hiring – including in Singapore!