Building a Skyscraper with Legos:

The Anatomy of a Distributed System

Tyler McMullen

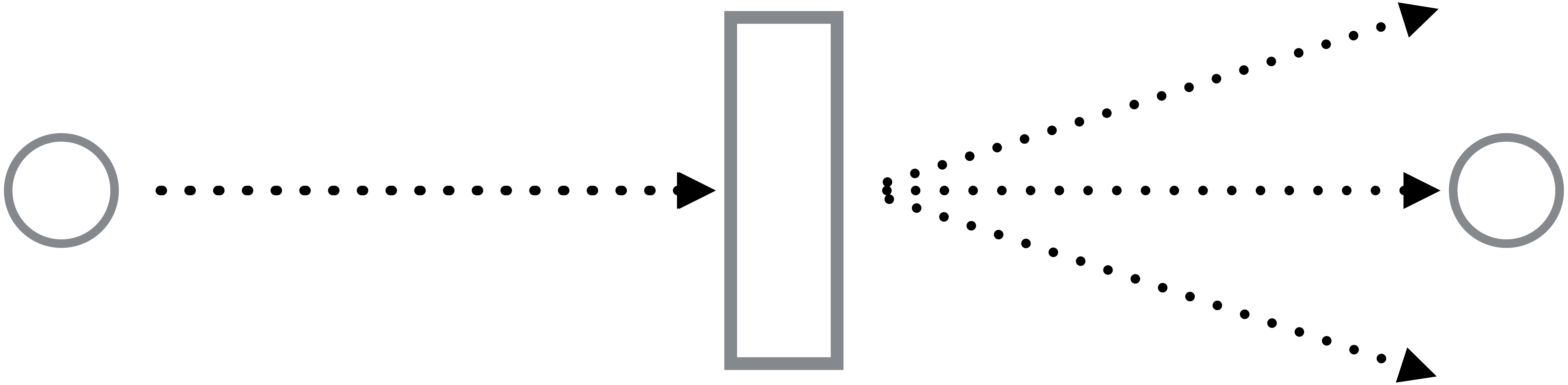
@tbmcmullen

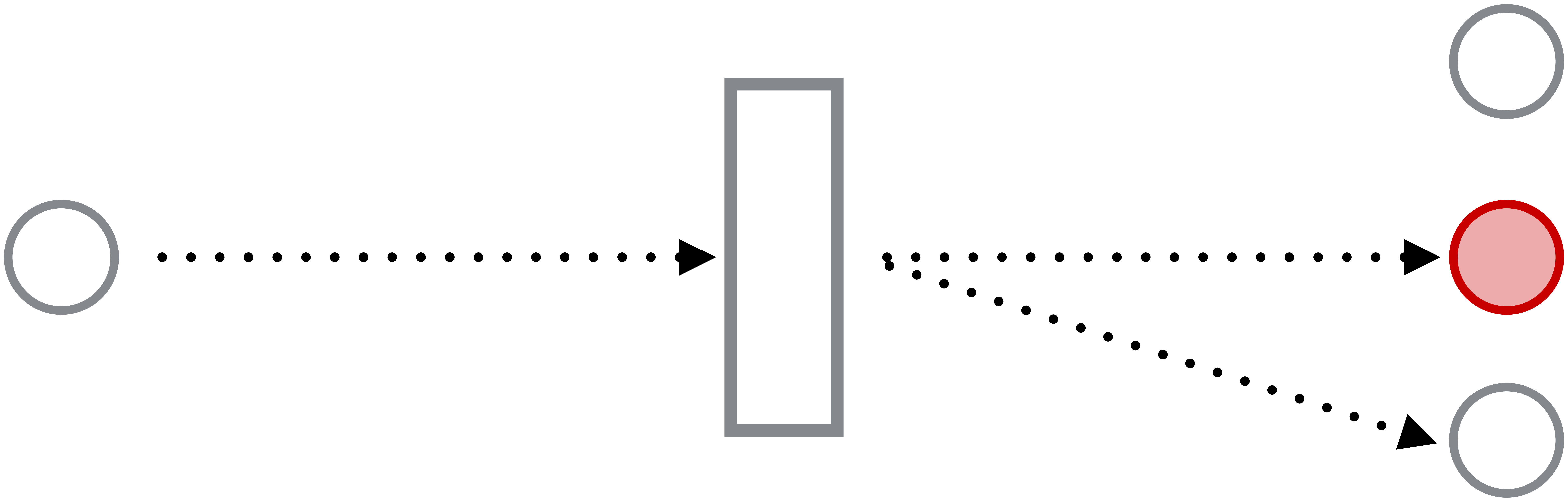
fastly®

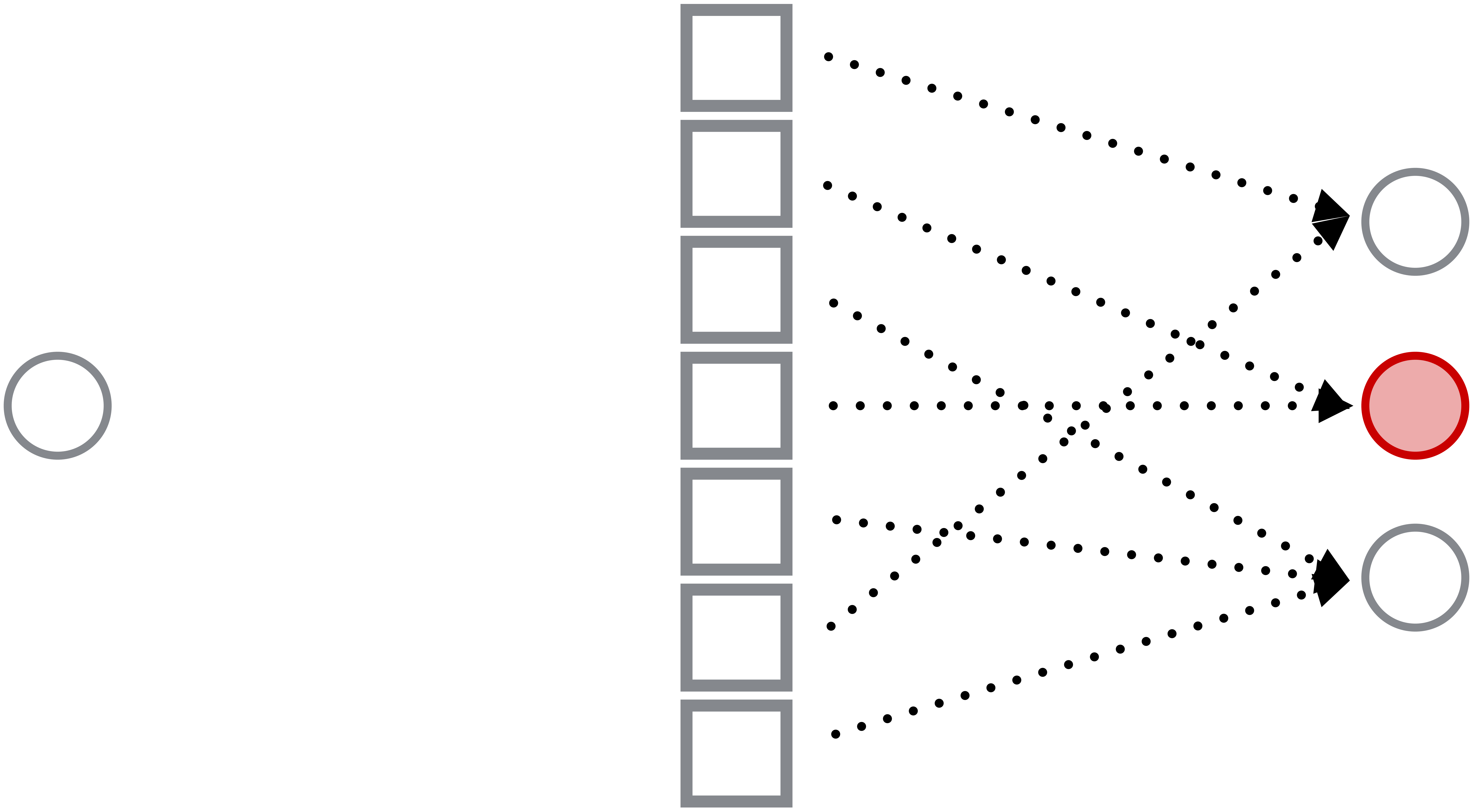
ok, what’s up?

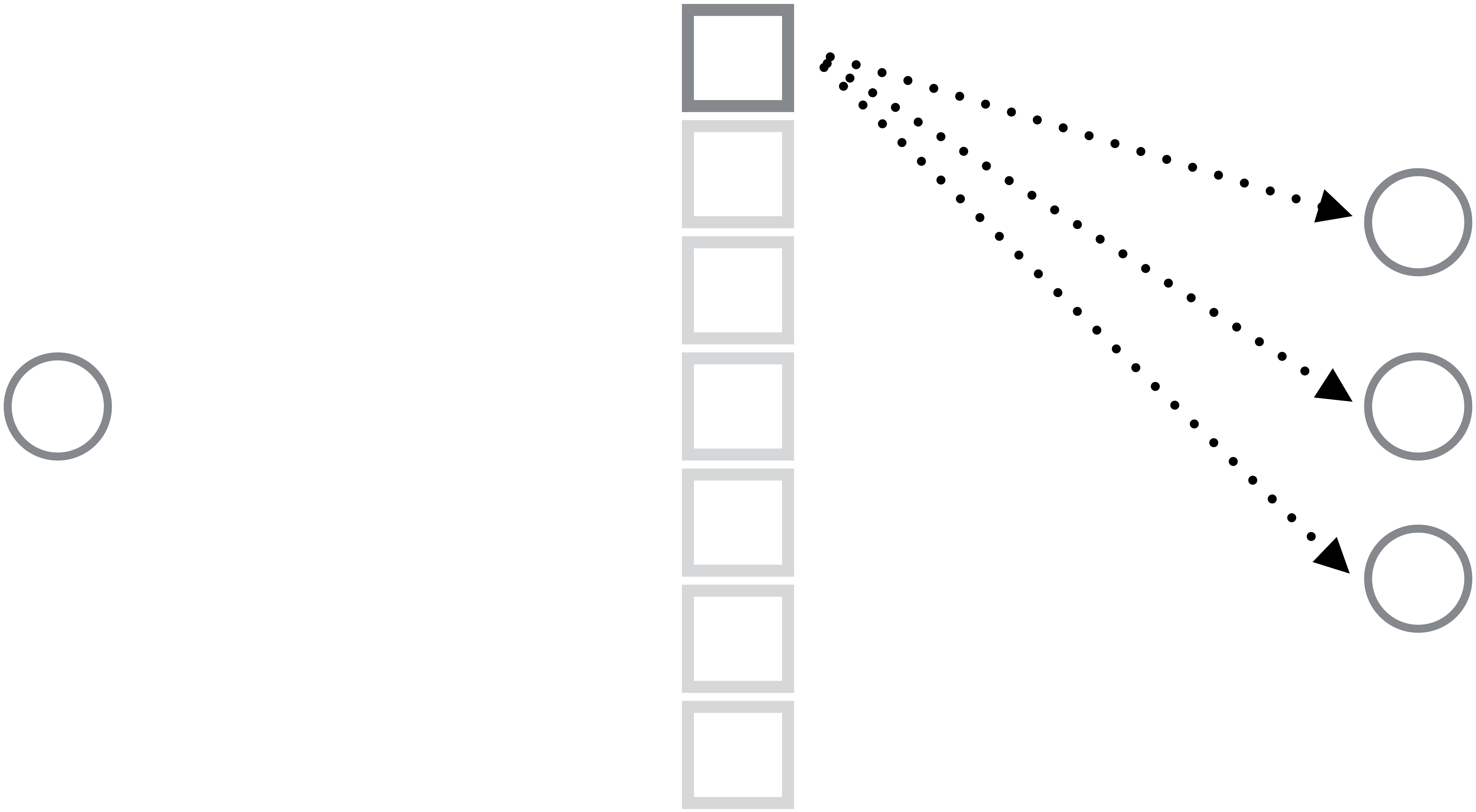
Let’s build a distributed system!

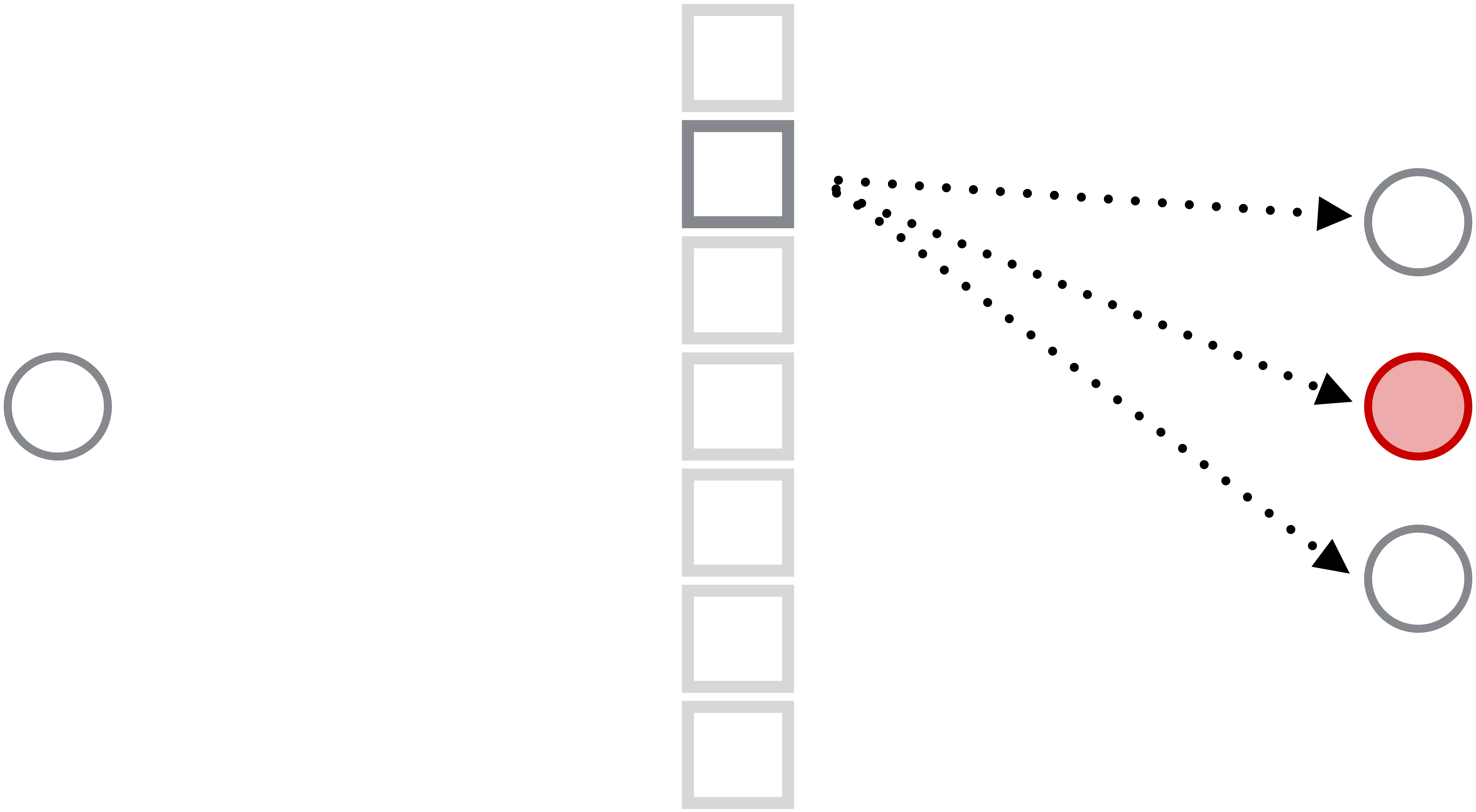
The Project

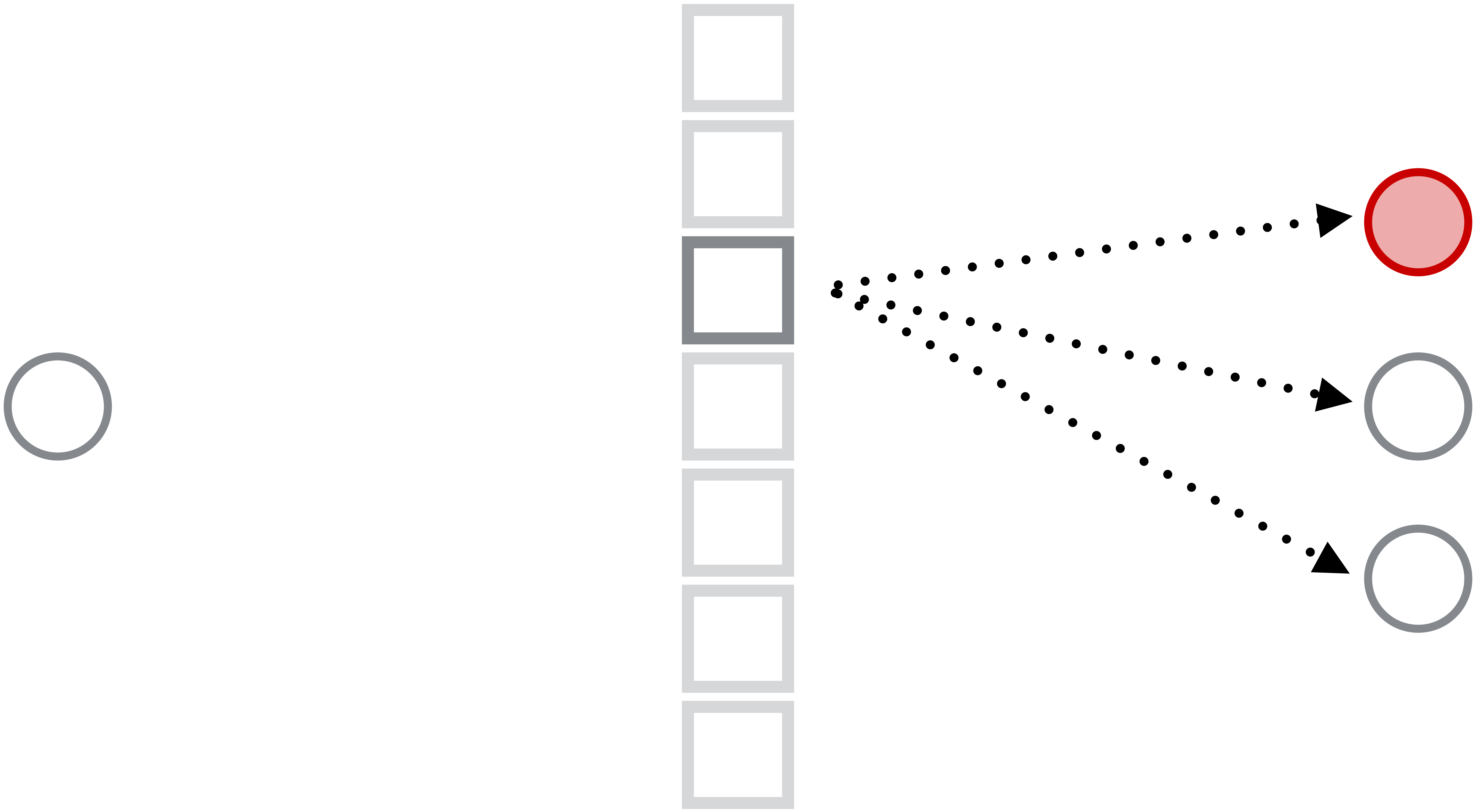


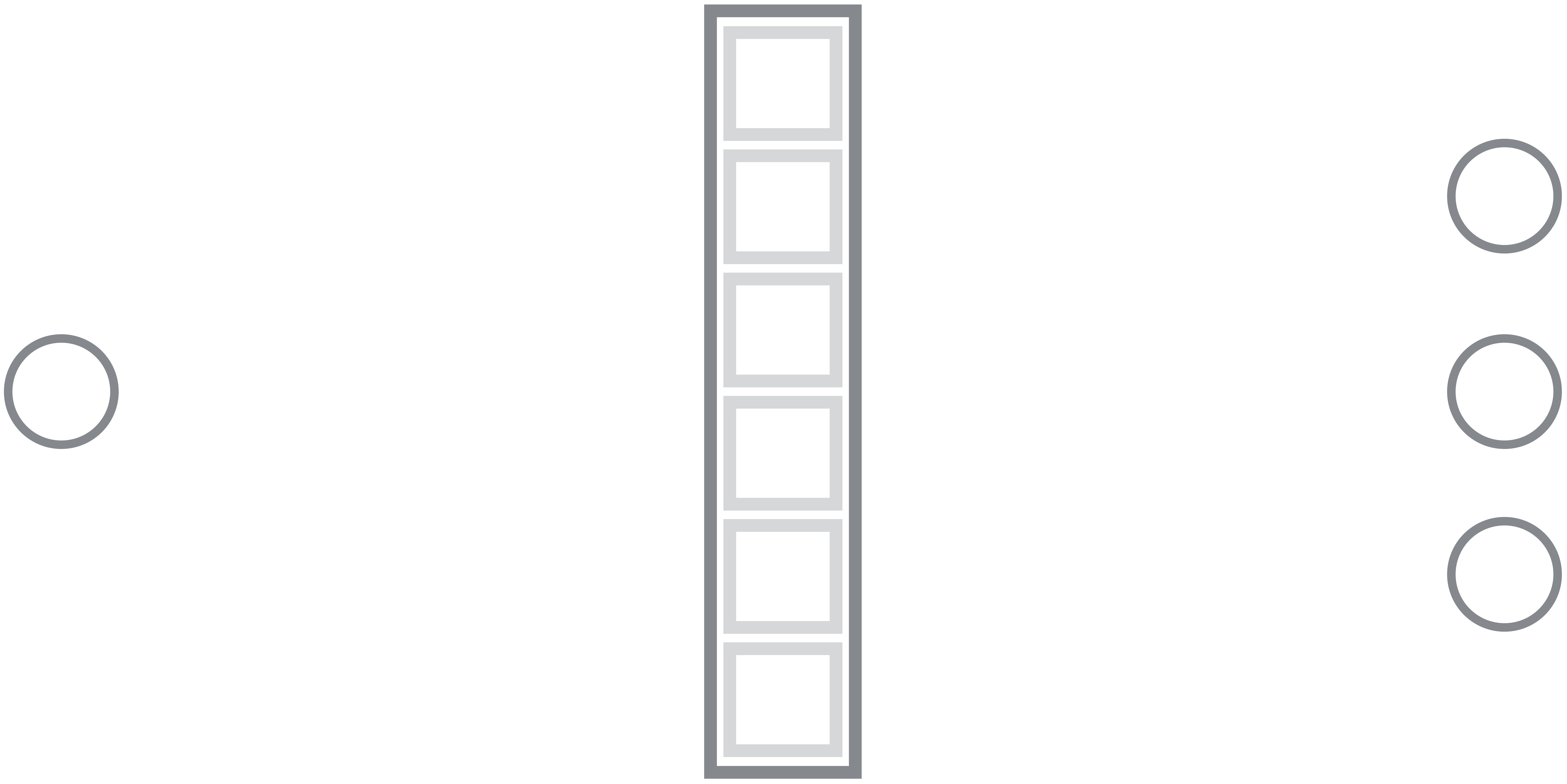


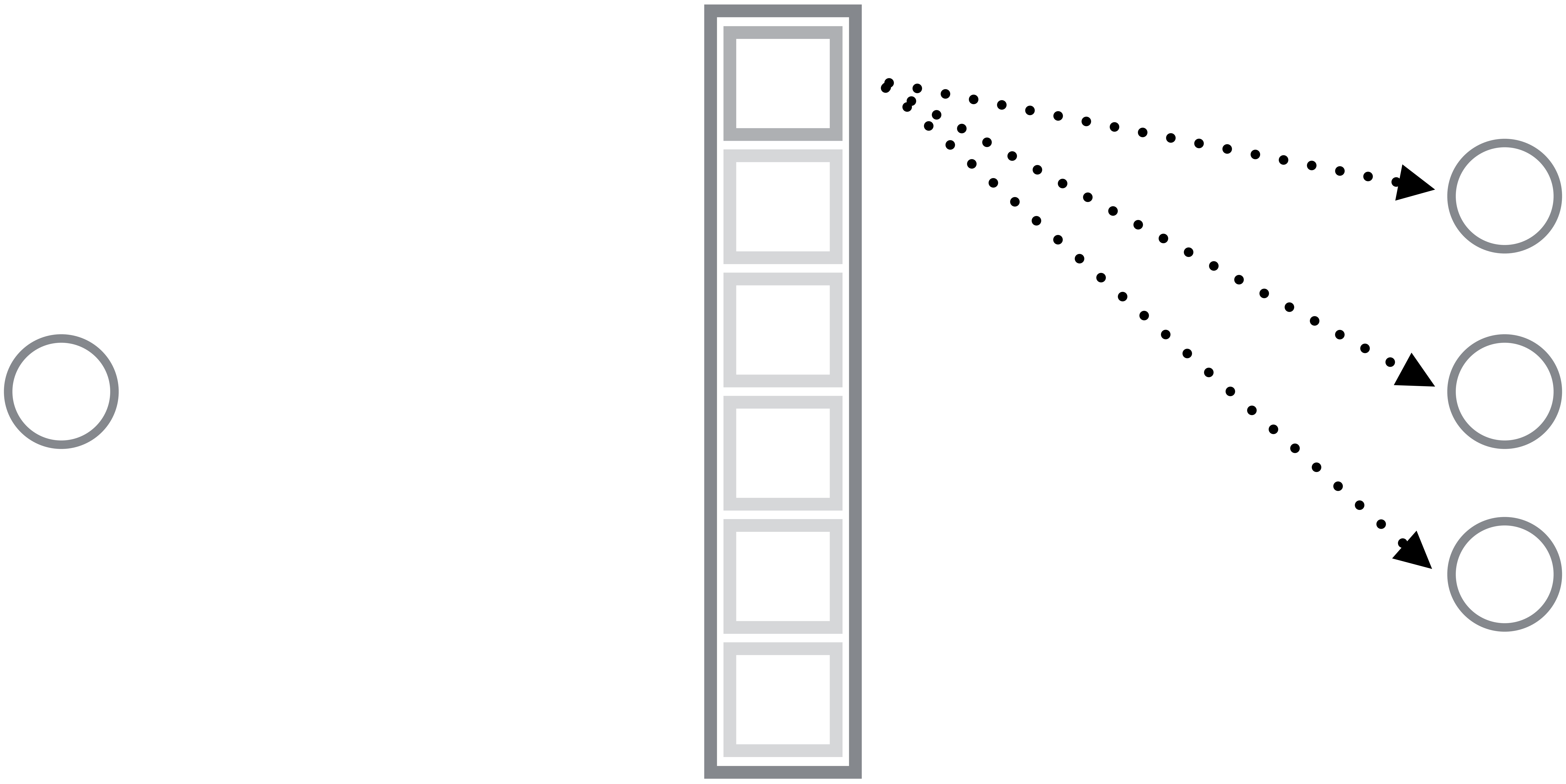


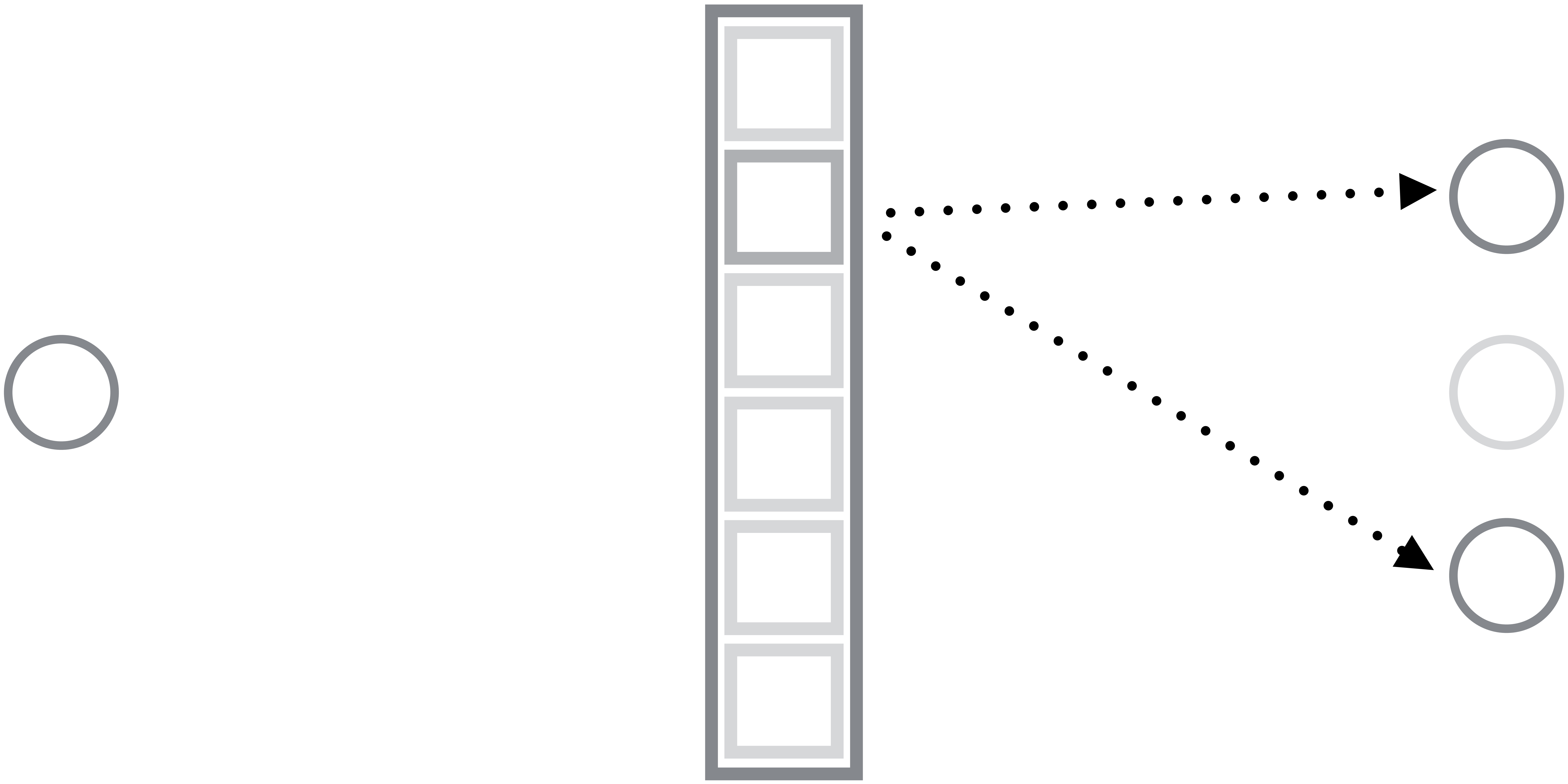


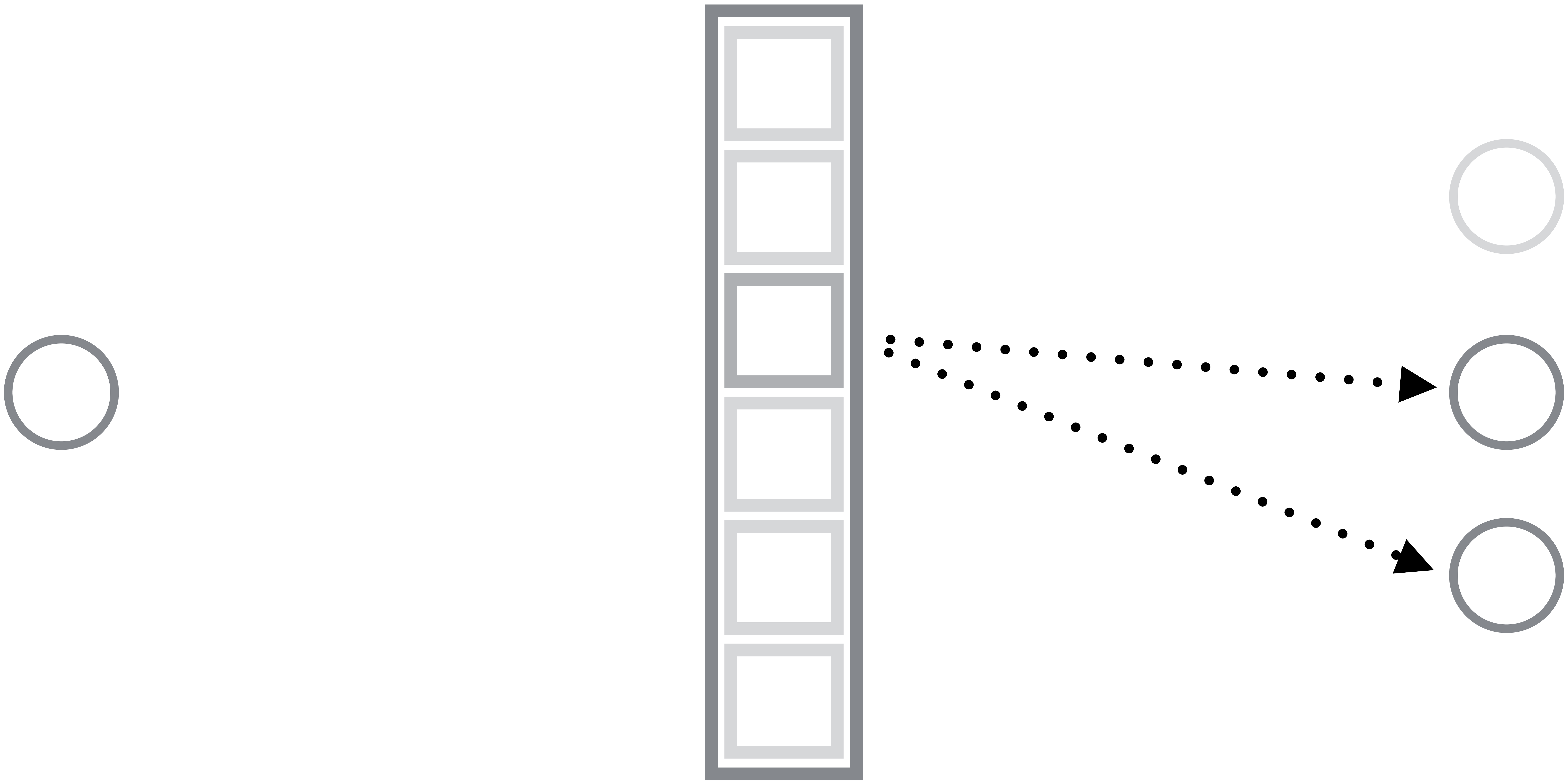


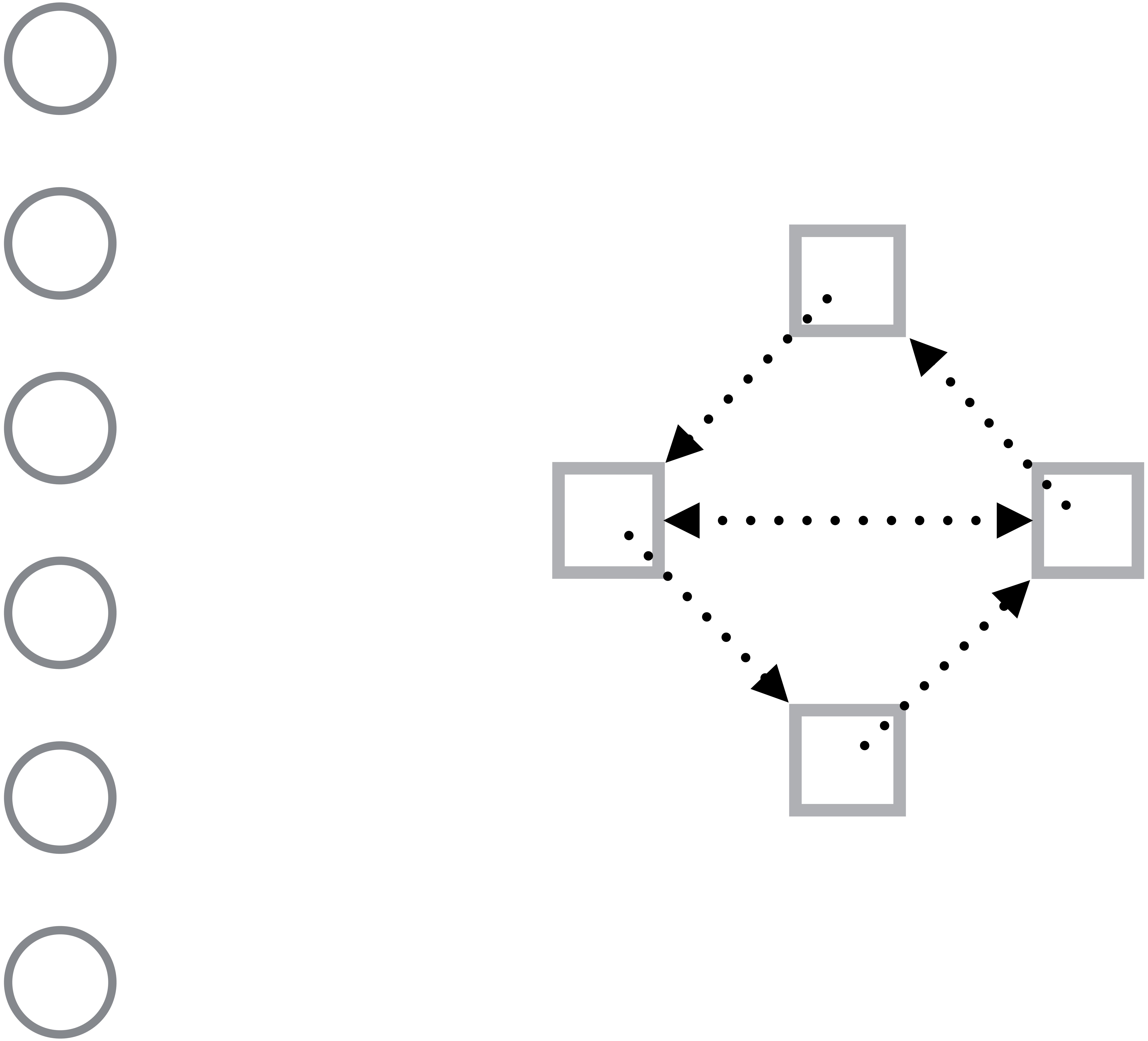












50+ Datacenters
Thousands of bare metal servers
up to 64 Servers per Datacenter

Ten of Thousands of Origin Servers

Recap

Still with me?

One Possible Solution

Consul

etcd

(Too) Strong Consistency

Eventual Consistency

Forward Progress

Coordination Avoidance in Database Systems

Peter Bailis, Alan Fekete[†], Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, Ion Stoica
UC Berkeley and [†]University of Sydney

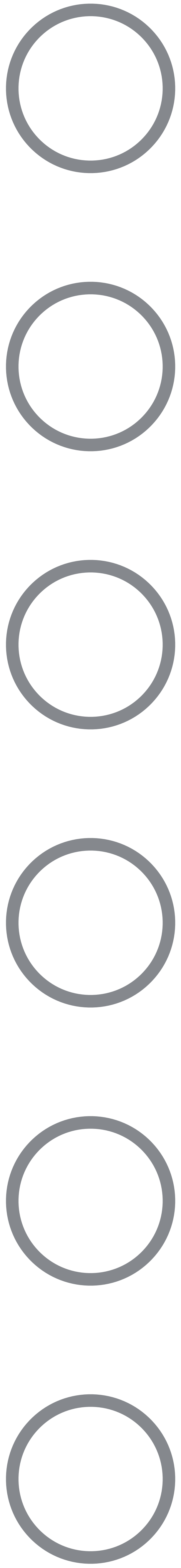
ABSTRACT

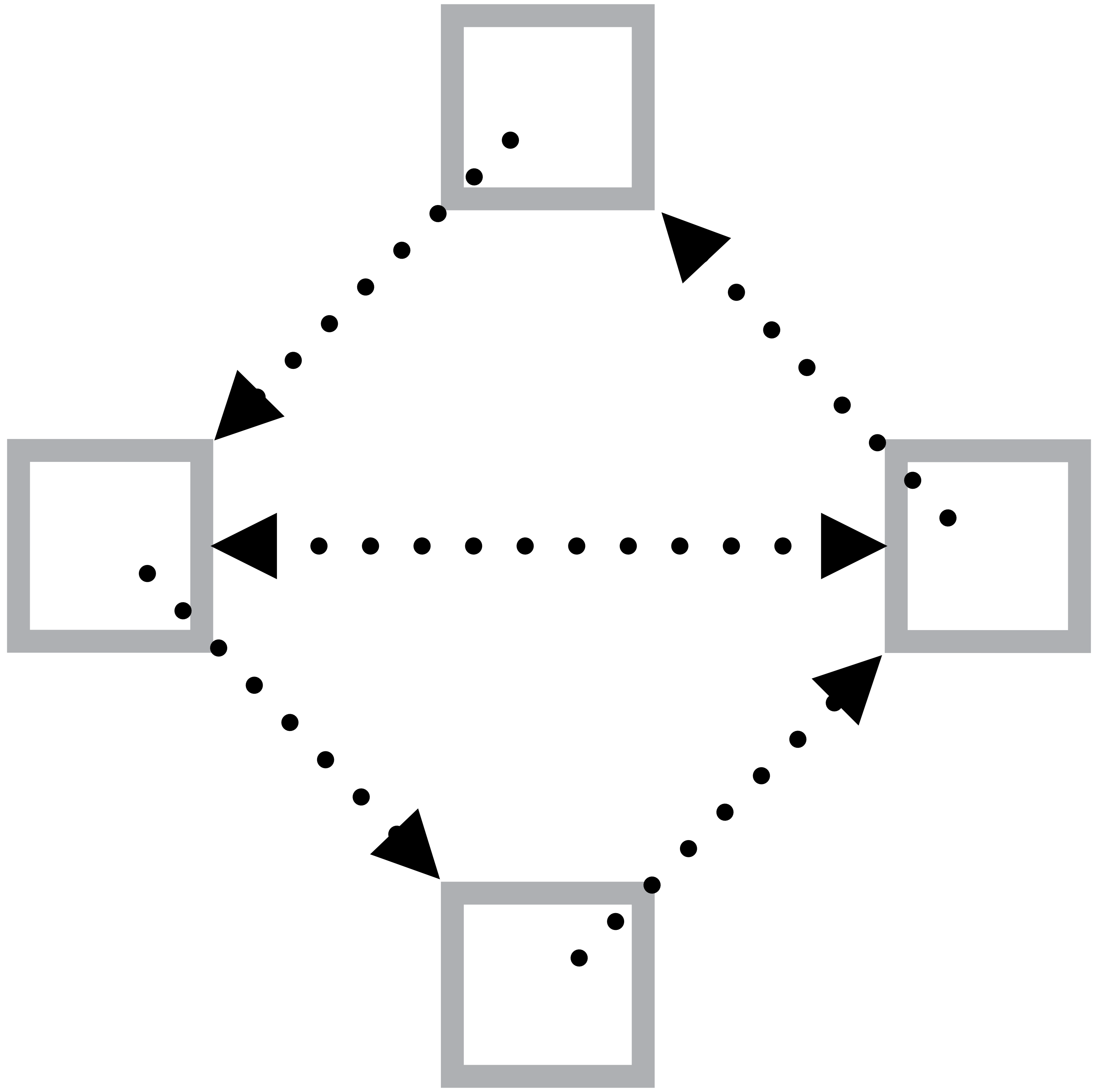
Minimizing coordination, or blocking communication between concurrently executing operations, is key to maximizing scalability, availability, and high performance in database systems. However, uninhibited coordination-free execution can compromise application correctness, or consistency. When is coordination necessary for correctness? The classic use of serializable transactions is sufficient to maintain correctness but is not necessary for all applications, sacrificing potential scalability. In this paper, we develop a formal framework, invariant confluence, that determines whether an application requires coordination for correct execution. By operating on application-level invariants over database states (e.g., integrity constraints), invariant confluence analysis provides a necessary and sufficient condition for safe, coordination-free execution. When programmers specify their application invariants, this analysis allows databases to coordinate only when anomalies that might violate invariants are possible. We analyze the invariant confluence of common invariants and operations from real-world database systems (i.e., integrity constraints) and applications and show that many are invariant confluent and therefore achievable without coordination. We apply these results to a proof-of-concept coordination-avoiding database prototype and demonstrate sizable performance gains compared to serializable execution, notably a 25-fold improvement over prior TPC-C New-Order performance on a 200 server cluster.

level correctness, or consistency.[1] In canonical banking application examples, concurrent, coordination-free withdrawal operations can result in undesirable and "inconsistent" outcomes like negative account balances—application-level anomalies that the database should prevent. To ensure correct behavior, a database system must coordinate the execution of these operations that, if otherwise executed concurrently, could result in inconsistent application state.

This tension between coordination and correctness is evidenced by the range of database concurrency control policies. In traditional database systems, serializable isolation provides concurrent operations (transactions) with the illusion of executing in some serial order [15]. As long as individual transactions maintain correct application state, serializability guarantees correctness [30]. However, each pair of concurrent operations (at least one of which is a write) can potentially compromise serializability and therefore will require coordination to execute [9, 21]. By isolating users at the level of reads and writes, serializability can be overly conservative and may in turn coordinate more than is strictly necessary for consistency [29, 39, 53, 58]. For example, hundreds of users can safely and simultaneously retweet Barack Obama on Twitter without observing a serial ordering of updates to the retweet counter. In contrast, a range of widely-deployed weaker models require less coordination to execute but surface read and write behavior that may in turn compromise consistency [2, 9, 22, 48]. With these alternative models, it is up to users to decide when weakened guarantees are

Ownership





Rendezvous Hashing

A Name-Based Mapping Scheme for Rendezvous

David G. Thaler and Chinya V. Ravishankar
Electrical Engineering and Computer Science Department
The University of Michigan, Ann Arbor, Michigan 48109-2122
thalerd@eecs.umich.edu ravi@eecs.umich.edu

November 13, 1996

Abstract

Clusters of identical intermediate servers are often created to improve availability and robustness in many domains. The use of proxy servers for the WWW and of Rendezvous Points [1] in multicast routing are two such situations. However, this approach is inefficient if identical requests are received and processed by multiple servers. We present an analysis of this problem, and develop a method called the Highest Random Weight (HRW) Mapping that eliminates these difficulties. Given an object name, HRW maps it to a server within a given cluster using the object name, rather than any *a priori* knowledge of server states. Since HRW always maps a given object name to the same server within a given cluster, it may be used locally at client sites to achieve consensus on object-server mappings.

origin server we're
deciding on the owner of

hash function we decided upon

$$h(S_n,O) = W_n$$

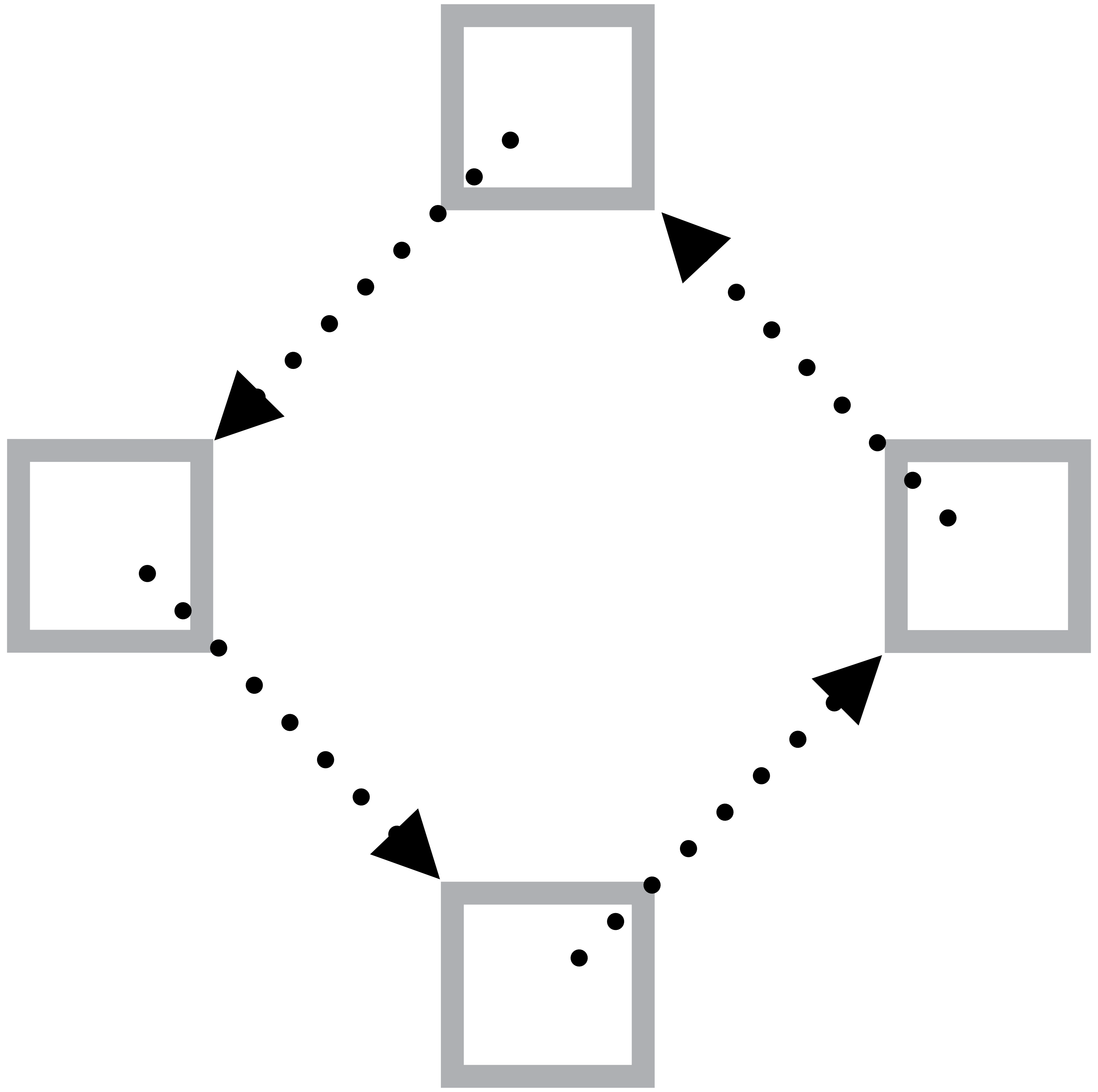
set of live servers

the weight or priority

$$\text{priorities}(O_1) = [S_2,\ S_3,\ S_1,\ S_4,\ \ldots]$$

$$\text{priorities}(O_2) = [S_4,\ S_2,\ S_3,\ S_1,\ \ldots]$$

Failure Detection



SWIM: *S*calable *W*eakly-consistent *I*nfection-style Process Group *M*embership Protocol

Abhinandan Das, Indranil Gupta, Ashish Motivala*
Dept. of Computer Science, Cornell University
Ithaca NY 14853 USA
{asdas,gupta,ashish}@cs.cornell.edu

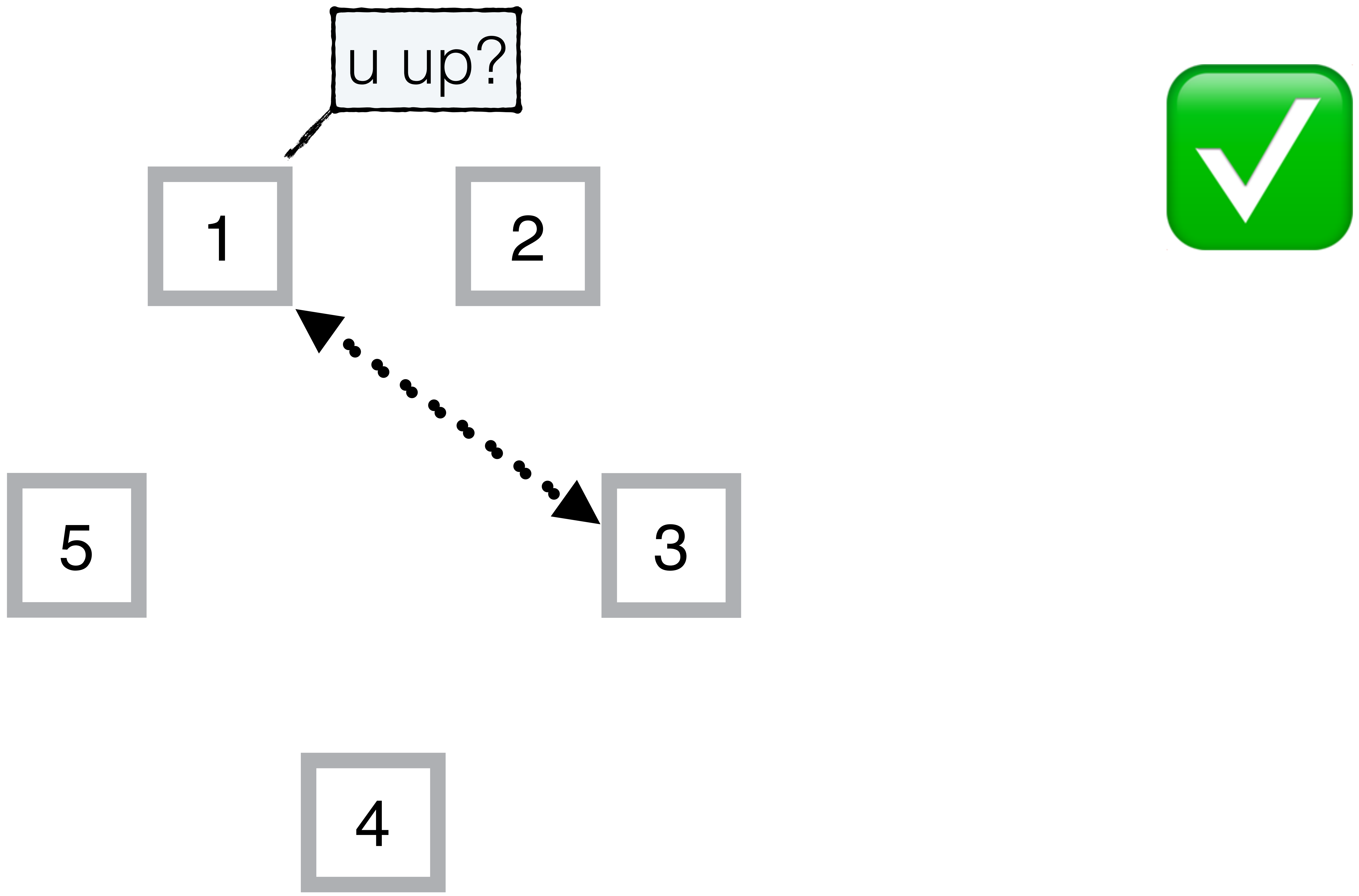
Abstract

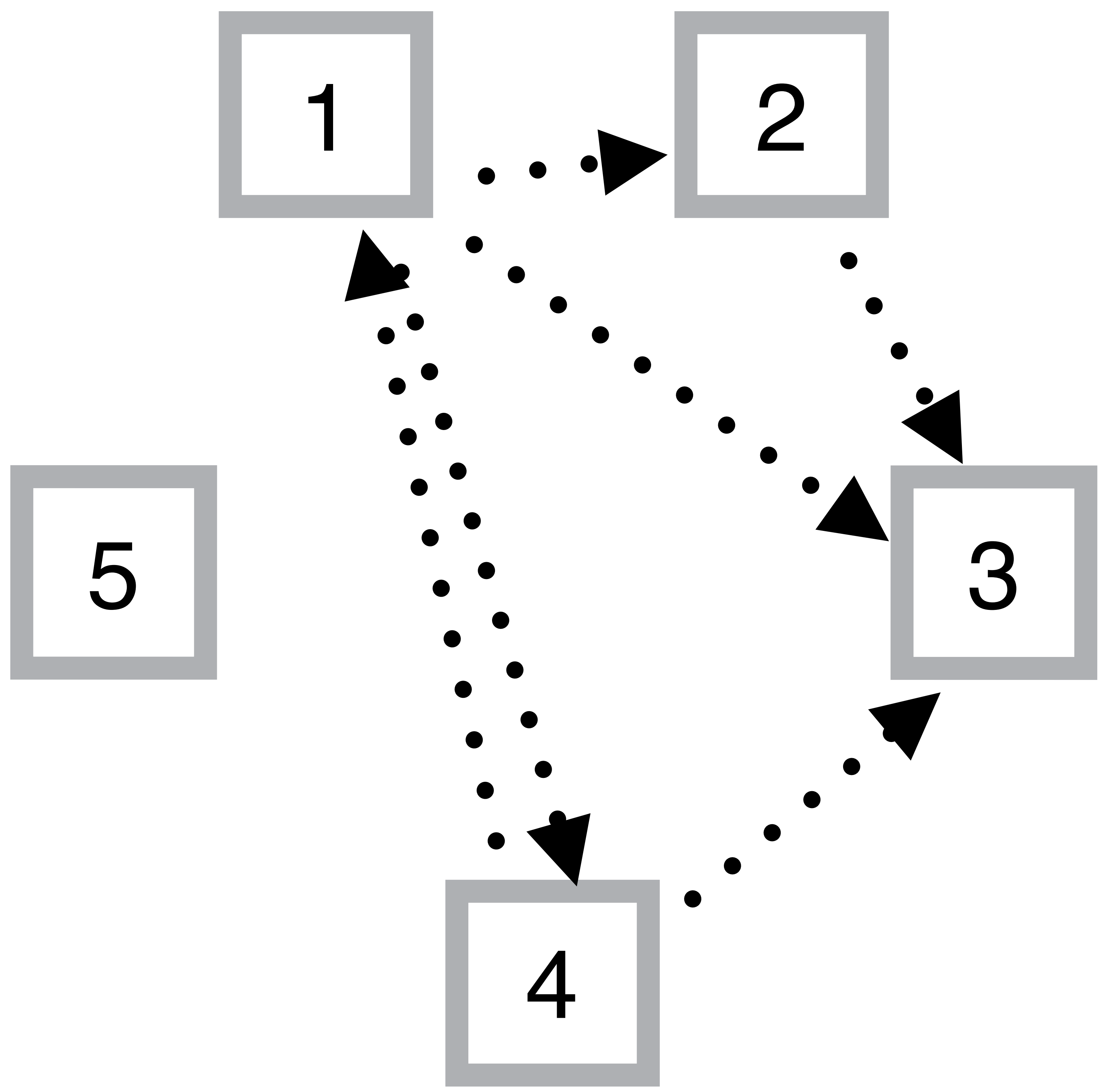
Several distributed peer-to-peer applications require weakly-consistent knowledge of process group membership information at all participating processes. SWIM is a generic software module that offers this service for large-scale process groups. The SWIM effort is motivated by the unscalability of traditional heart-beating protocols, which either impose network loads that grow quadratically with group size, or compromise response times or false positive frequency w.r.t. detecting process crashes. This paper reports on the design, implementation and performance of the

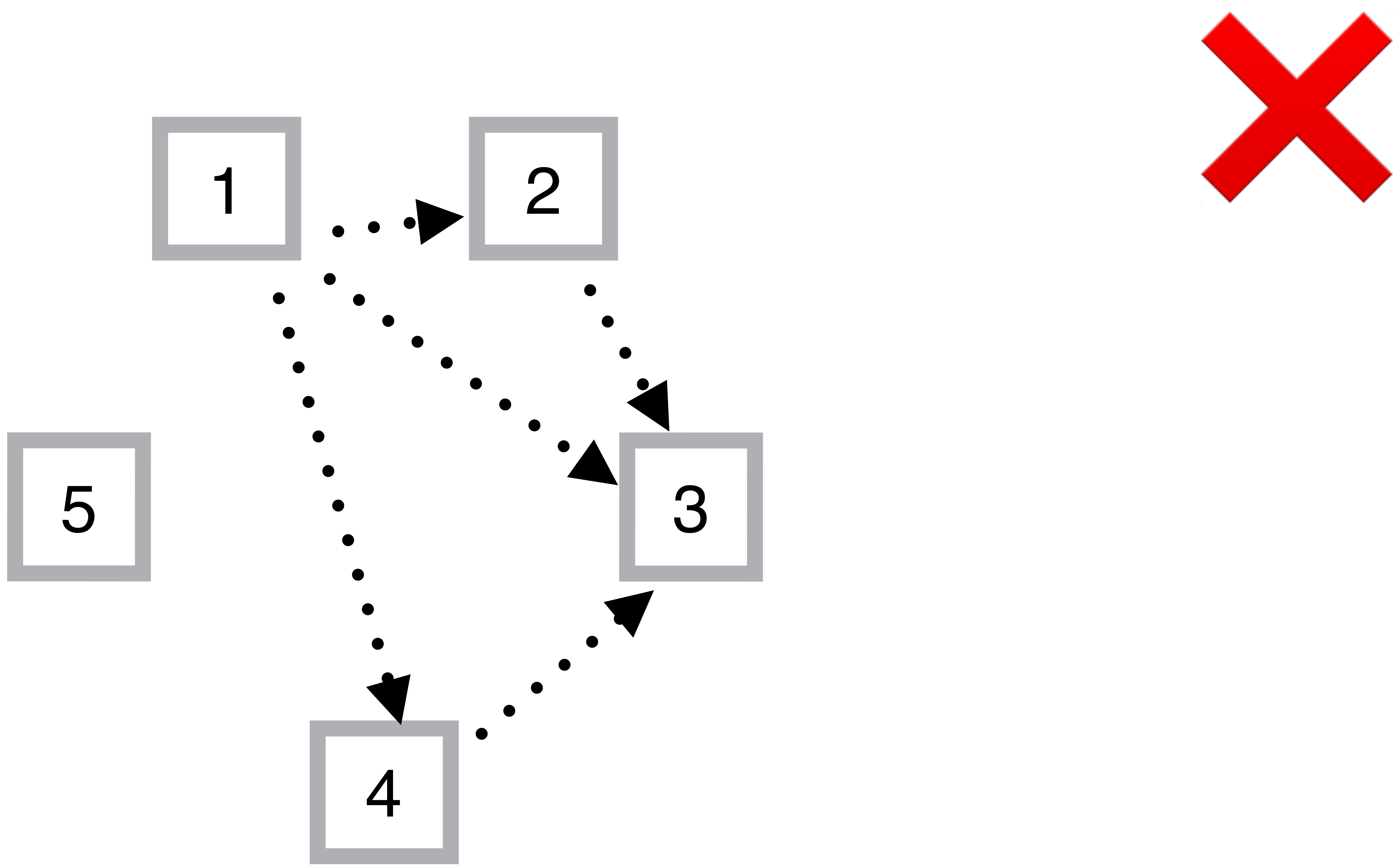
1. Introduction

As you swim lazily through the milieu,
The secrets of the world will infect you.

Several large-scale peer-to-peer distributed process groups running over the Internet rely on a distributed membership maintenance sub-system. Examples of existing middleware systems that utilize a membership protocol include reliable multicast [3, 11], and epidemic-style information dissemination [4, 8, 13]. These protocols in turn find use in applications such as distributed databases that need to reconcile recent disconnected updates [14], publish-subscribe systems,


u up?
1
2
5
3
4


1
2
3
4
5


1
2
5
3
4

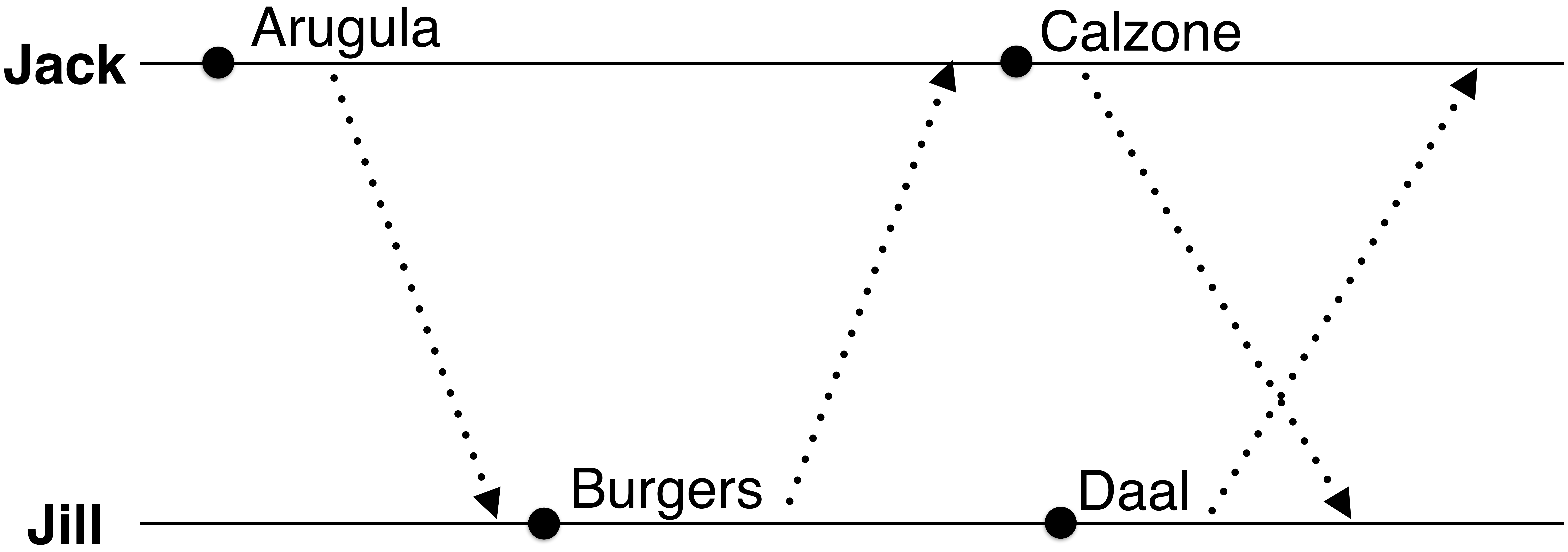
Memberlist

Gossip

Push and Pull

Convergence

Causality



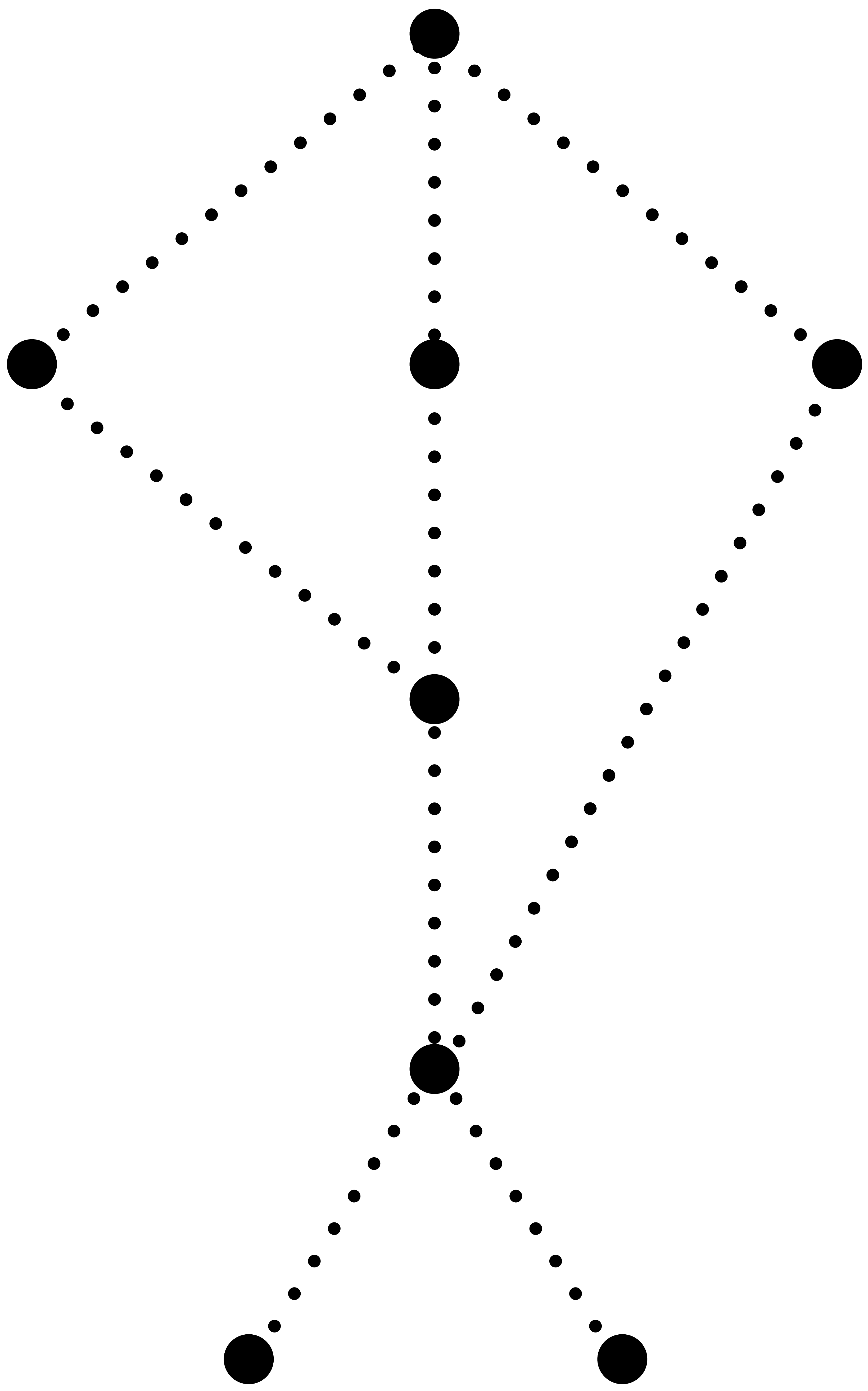
Burgers happened-before **Calzone**

Arugula happened-before **Burgers**

Calzone is concurrent with **Daal**

Burgers happened-before **Daal**

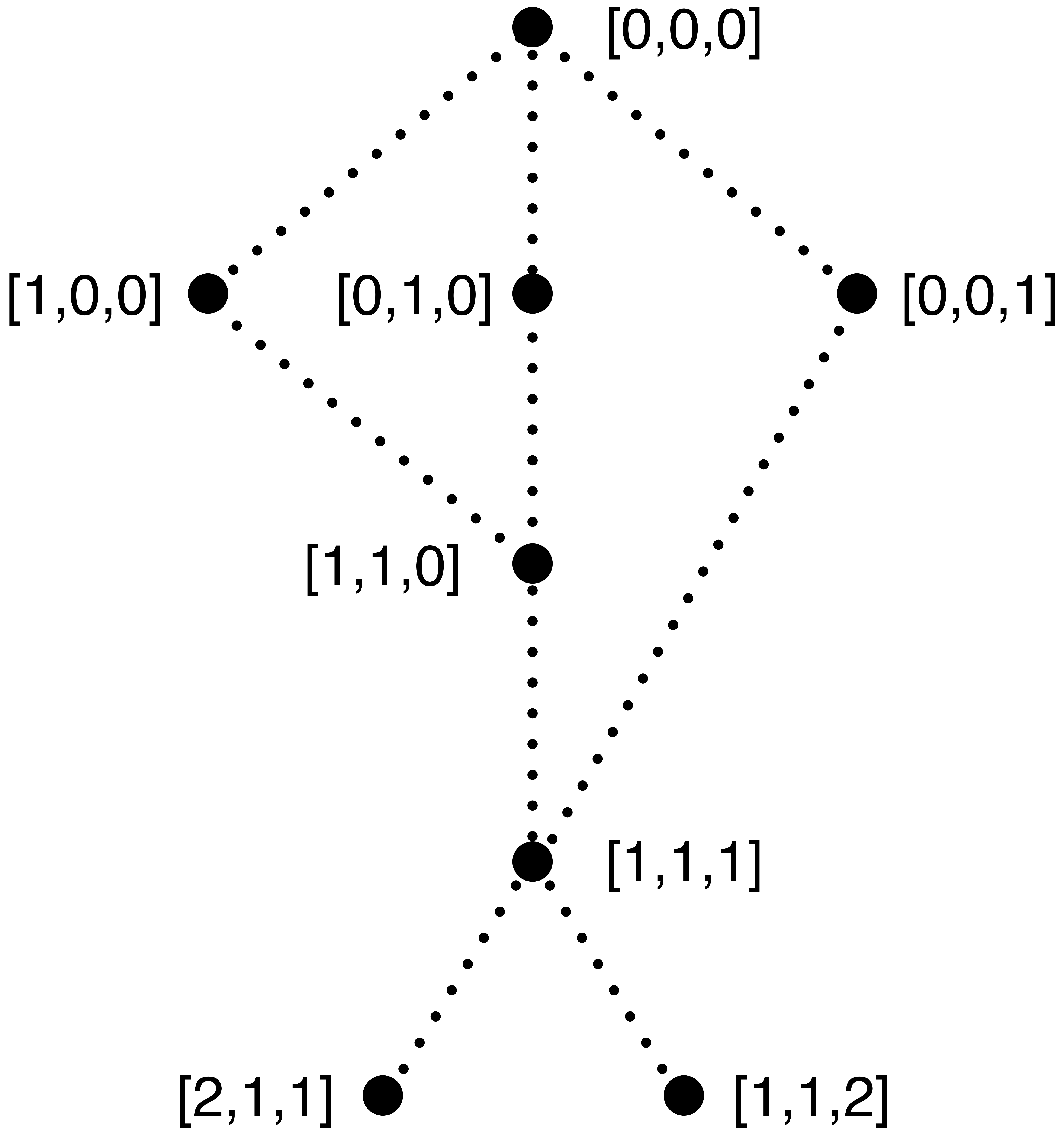
Lattices



Causality

Version Vectors


[1,0,0]
S1
[1,0,0]
[1,1,0]
[1,1,1]
[1,2,1]
S2
[0,0,1]
S3


[0,0,0]
[1,0,0]
[0,1,0]
[0,0,1]
[1,1,0]
[1,1,1]
[2,1,1]
[1,1,2]

Coordination-free Distributed Map


Version
Merge
Delta

Δ-CRDT Map

```go
type SharedMap struct {
    storage map[Key]SharedMapRecord
    v       clock.VersionVector
}

type SharedMapRecord struct {
    value Value
    dot   clock.VVDot
}
```

```
Send_Version:
  Send our Version Vector.
```

```
Received_Version(V):
  For each record(R) in our map:
    If (V happened-before    R.Dot) OR
       (V is-concurrent-with R.Dot):
      Add R to Delta.

  Send Delta.
```

```
Received_Delta(D):
  V = Our Version Vector

  For each record(R) in D:
    If R.Dot happened-before V:
      Skip it.

    R’ = Local Record

    If R’.Dot happened-before D.Version:
      Merge it.


    R and R’ are concurrent: ✨
```

✨ = Rendesvous

Delta-state CRDT Map

Δ-CRDTs: Makin

Albert van der Linde
a.linde@campus.fct.unl.pt

jc.le
NOV
Universi

Algorithm 1: Δ-CRDT replication

upon *onVersionVector*(VV, REPLICA) **do**
  Δ ⟵ getDelta(vv)
  if Δ.size() > 0
    REPLICA.send(Δ)
  optionally do (*push model*)
    if vv **after** self.versionVector
      REPLICA.send(self.versionVector)

upon *delta*(Δ) **do**
  self.state.applyDelta(Δ)
  self.versionVector.update(Δ)

periodically do (*pull model*)
  r ⟵ randomReplica()
  r.send(self.versionVector)

on local operation do (*push model*)
  r ⟵ randomReplica()
  r.send(self.versionVector)

ABSTRACT

Replication is a key technique for providing both fau
erance and availability in distributed systems. Ho
managing replicated state, and ensuring that these
cas remain consistent, is a non trivial task, in par
in scenarios where replicas can reside on the clien
as clients might have unreliable communication ch
and hence, exhibit highly dynamic communication pa
One way to simplify this task is to resort to CRDTs,
are data types that enable replication and operatio
replicas with no coordination, ensuring eventual stat
vergence when these replicas are synchronized. Ho
when the communication patters, and therefore synchro-
nization patterns, are highly dynamic, existing designs of

not only introduces additional overhead (to keep track of
causality) but also fits poorly in scenarios where there are

Ok, why?

Edge Compute

Coordination-free Distributed Systems

Single System Image

A Certain Tendency Of The Database Community*

Christopher S. Meiklejohn
Université catholique de Louvain
ch

We posit tha
system image" se
systems operate i
timization of thi
that facilitates c
in a system. We
to address the problems of computation over "eventually consistent" information in a large-scale distributed system.

We posit that striving for distributed systems that provide "single system image" semantics is fundamentally flawed and at odds with how systems operate in the physical world.

We need new metaphors.

We need new intuition.

Thank You.

@tbmcmullen