Scaling Out Key-Value Storage: Dynamo

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

CS 240: Computing Systems and Concurrency
Lecture 8

Marco Canini

Availability: vital for web applications

- Web applications are expected to be "always on"
  - Down time → pisses off customers, costs $
- System design considerations relevant to availability
  - **Scalability**: always on under growing demand
  - **Reliability**: always on despite failures
  - *Performance*: 10 sec latency considered available?

    "an availability event can be modeled as a long-lasting performance variation"
    (Amazon Aurora SIGMOD '17)

Scalability: up or out?

- Scale-up (vertical scaling)
  - Upgrade hardware
  - E.g., MacBook Air → MacBook Pro
  - Down time during upgrade; stops working quickly

- **Scale-out** (horizontal scaling)
  - Add machines, divide the work
  - E.g., a supermarket adds more checkout lines
  - No disruption; works great with careful design

Reliability: available under failures

- More machines, more likely to fail
  - p = probability a machine fails in given period
  - n = number of machines
  - Probability of any failure in given period = $1-(1-p)^n$

- For **50K machines**, each with **99.99966% available**
  - **16%** of the time, **data center experiences failures**

- For **100K machines,** failures happen **30%** of the time!

Two questions (challenges)

- How is data partitioned across machines so the system scales?
- How are failures handled so the system is always on?

Today: Amazon Dynamo

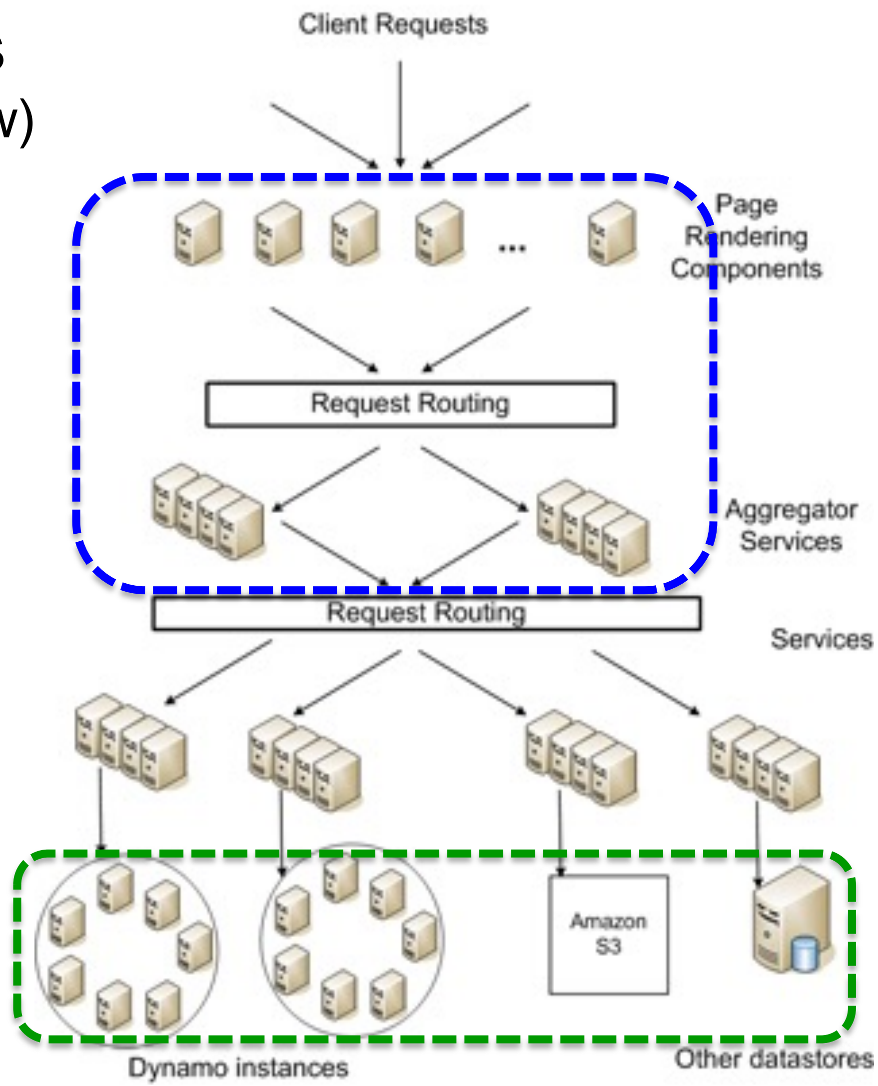
1. **Background and system model**

2. Data partitioning

3. Failure handling

Amazon in 2007

- 10^4s of servers in multiple DCs
  - 10^6s of servers, 120+ DCs (as of now)
- 10^7s of customers at peaks
  - 89M+ reqs/s (Prime Day '21)
- Tiered architecture (similar today)
  - Service-oriented architecture
  - Stateless web servers & aggregators
  - Stateful storage servers



Dynamo requirements

- **Highly available writes** despite failures
  - Despite disks failing, network routes flapping, “data centers destroyed by tornadoes”
  - Always respond quickly, even during failures → replication
- **Low request-response latency:** focus on **99.9%** SLA
  - E.g., “provide a response within 300ms for 99.9% of its requests for peak client load of 500 reqs/s”
- **Incrementally scalable** as servers grow to workload
  - Adding “nodes” should be seamless
- Comprehensible **conflict resolution**
  - High availability in above sense implies conflicts

Basics in Dynamo

- Basic interface is a **key-value store** (vs. relational DB)
  - **get(k)** and **put(k, v)**
  - Keys and values opaque to Dynamo

- Nodes are symmetric
  - P2P and DHT context

Today: Amazon Dynamo

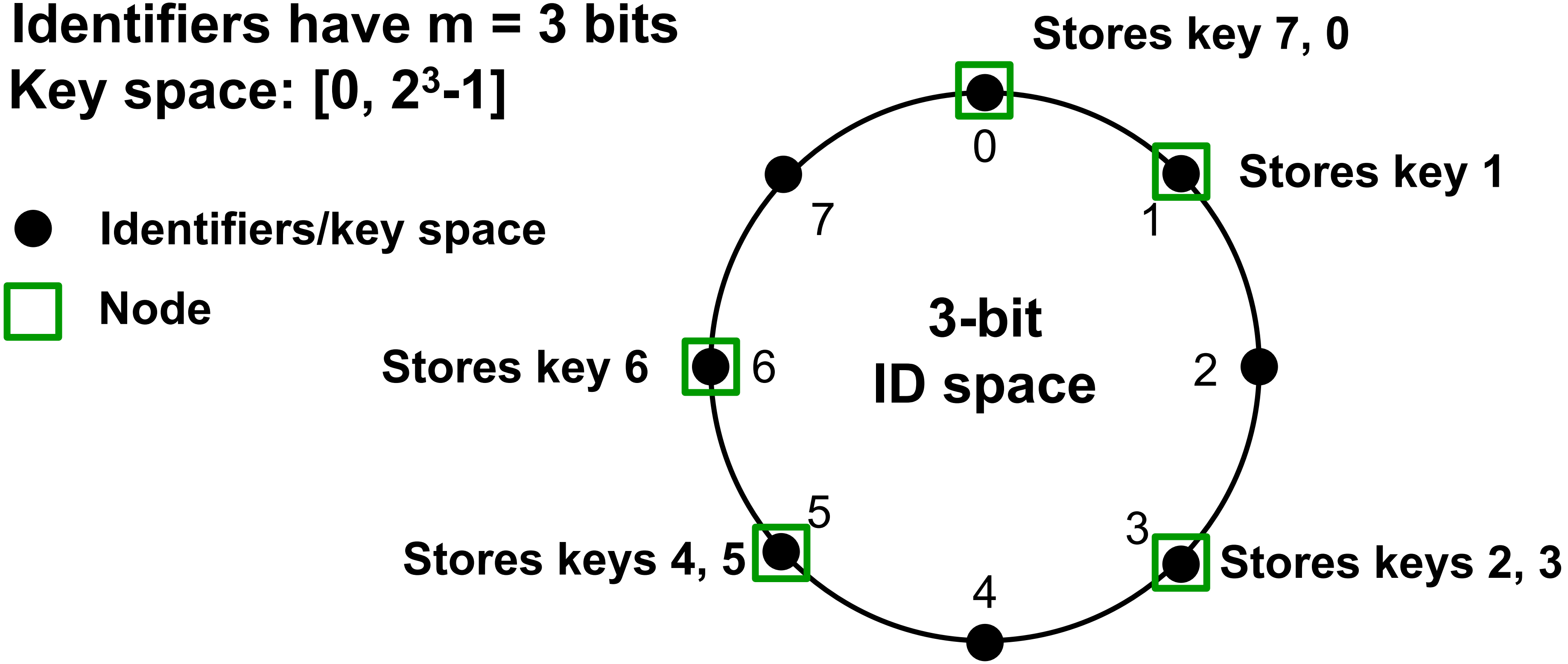
1. Background and system model

2. **Data partitioning**

3. Failure handling

Consistent hashing recap

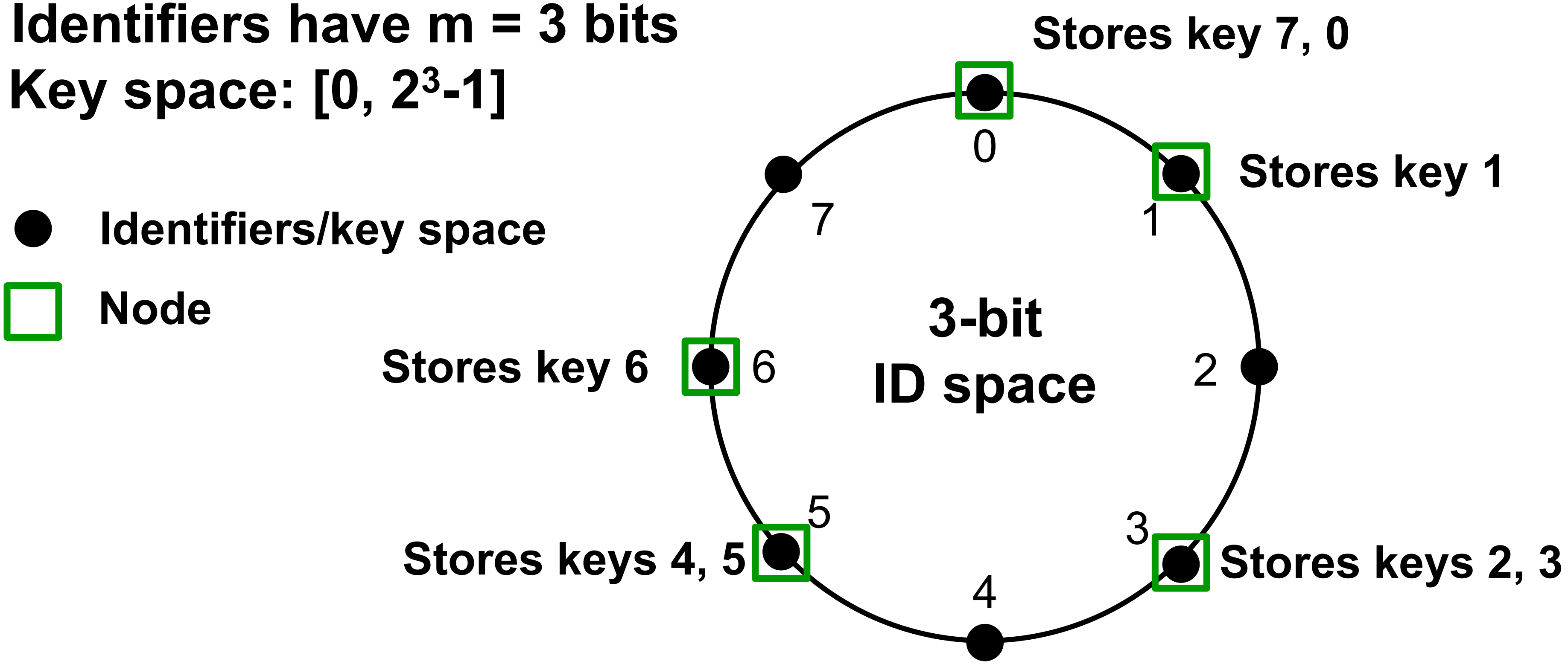
Identifiers have m = 3 bits
Key space: [0, 2^3-1]



Key is stored at its **successor:** node with next-higher ID

Incremental scalability (why consistent hashing)

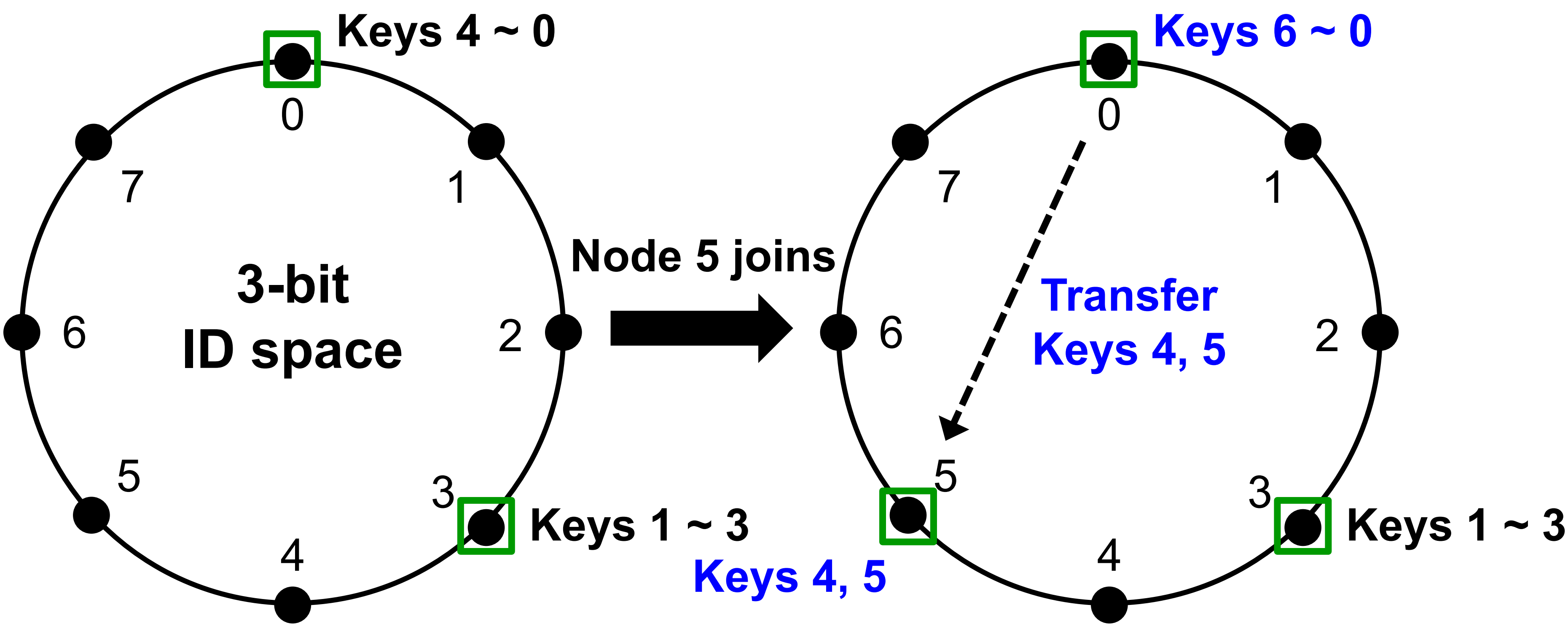
Identifiers have m = 3 bits
Key space: [0, 2^3-1]



Key is stored at its **successor:** node with next-higher ID

Incremental scalability (why consistent hashing)

- Minimum data is moved around when nodes join and leave
- Unlike modular hashing (see next slide)



Modulo hashing

- Consider problem of data partition:
  – Given **object id X**, choose one of ***k*** servers to use

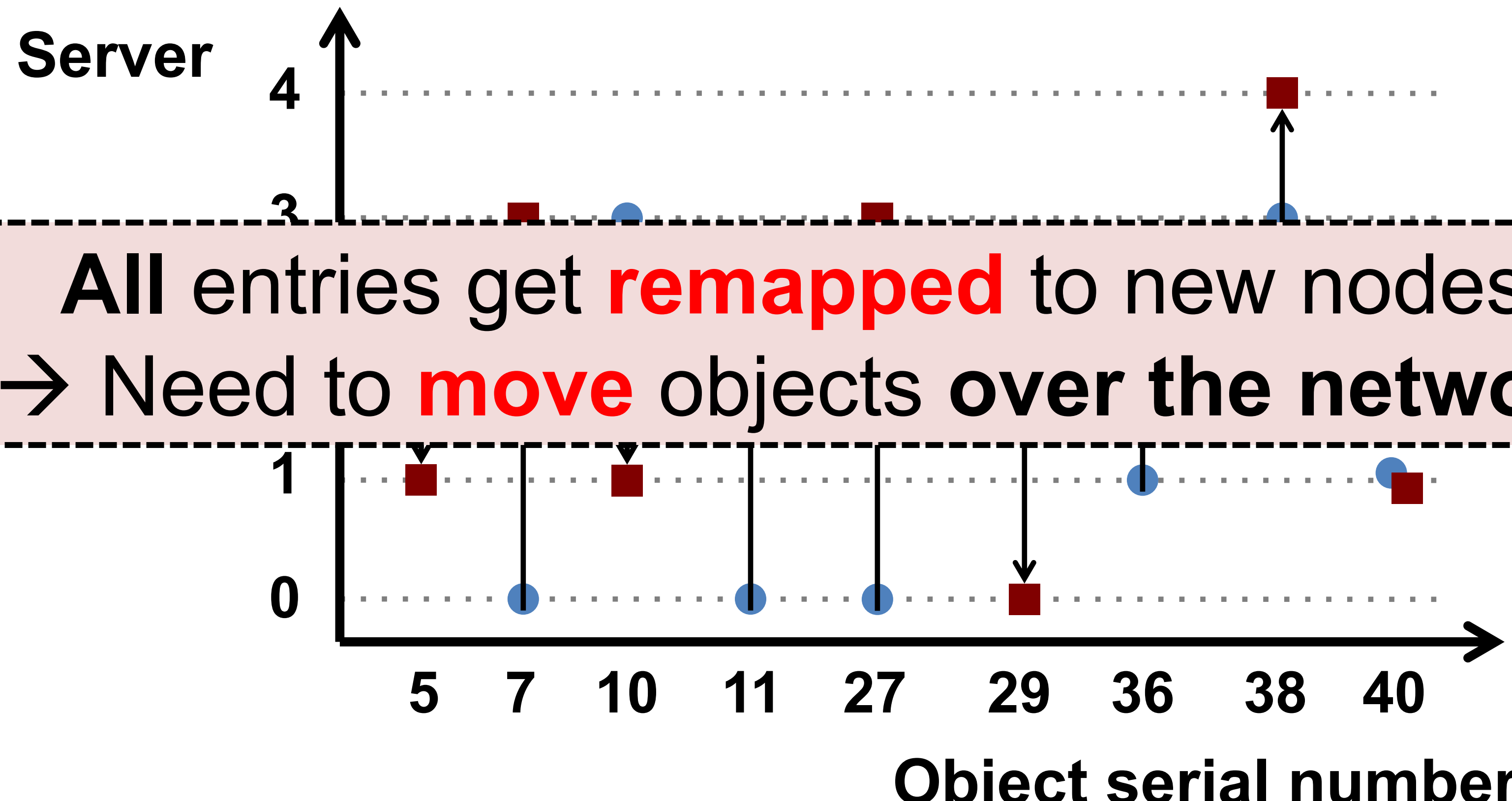
- Suppose instead we use **modulo hashing:**
  – Place ***X*** on server ***i* = hash(*X*) mod *k***

- What happens if a server fails or joins (k ← k±1)?
  – or different clients have **different estimate** of k?

Problem for modulo hashing: Changing number of servers

$h(x) = x + 1 \pmod 4$

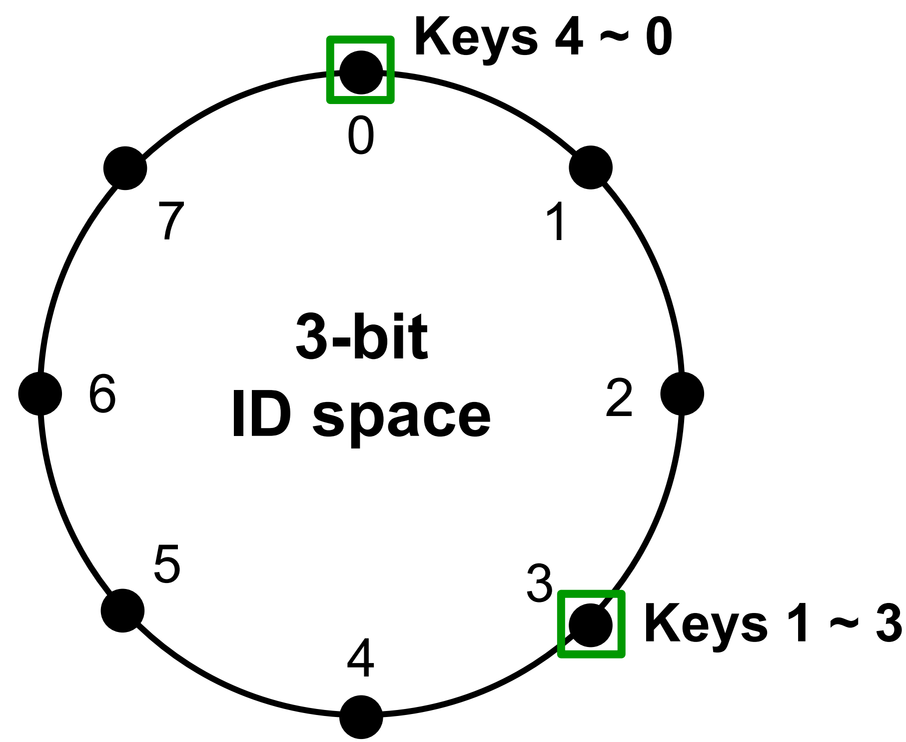
Add one machine: $h(x) = x + 1 \pmod 5$



All entries get **remapped** to new nodes!
→ Need to **move** objects **over the network**

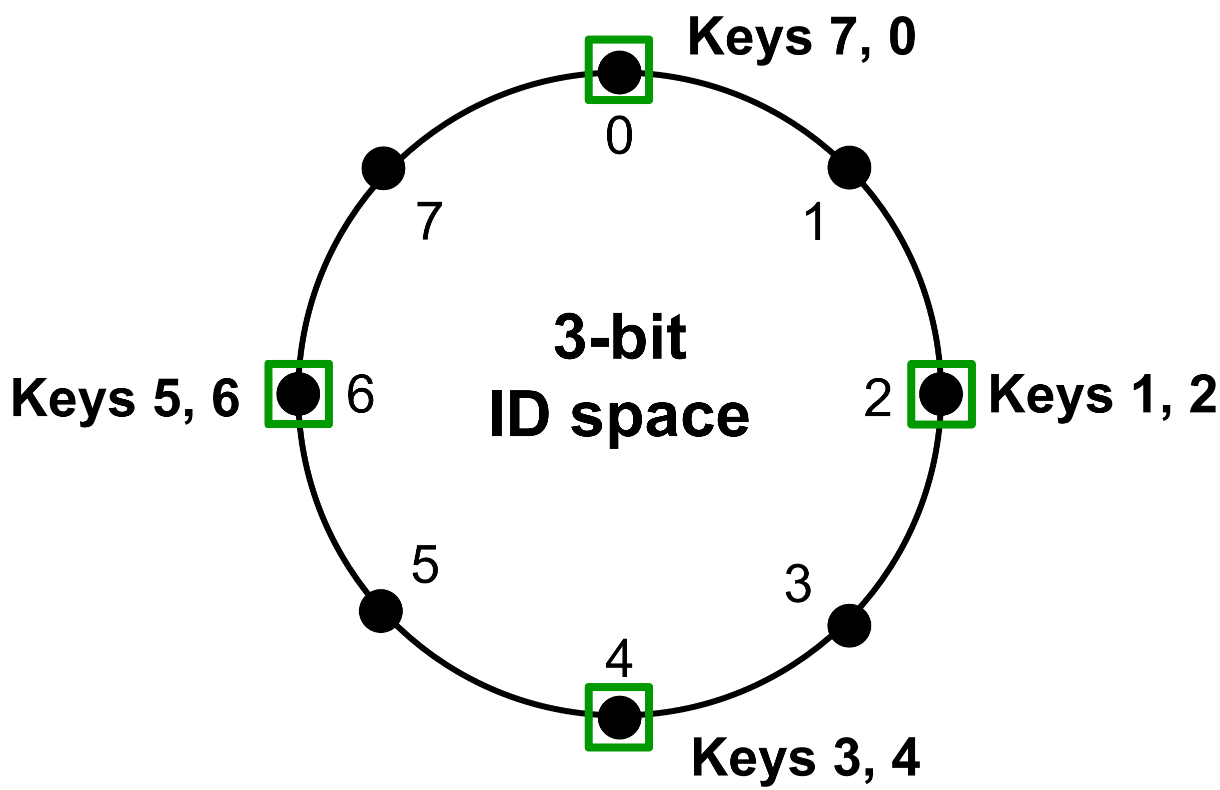
Challenge: unbalanced load

- Nodes are assigned different # of keys



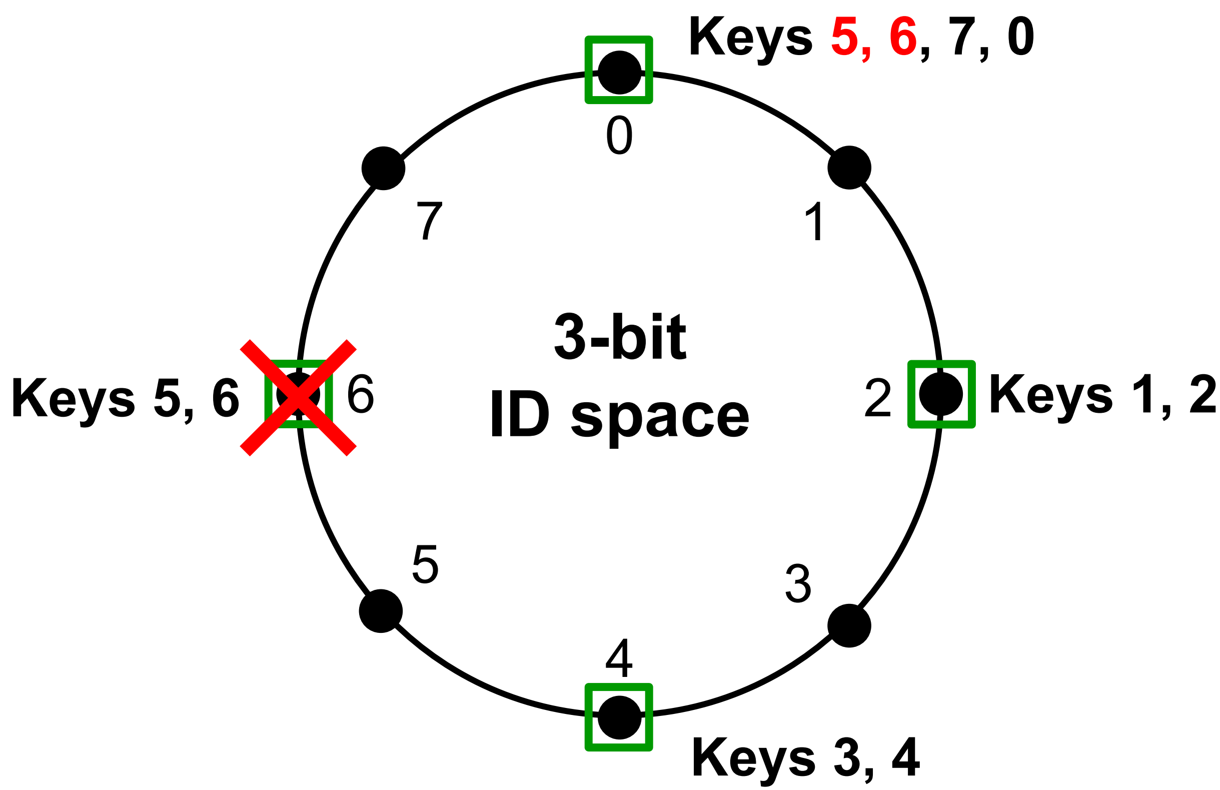
Challenge: unbalanced load

- Nodes are assigned different # of keys
- Unbalanced with nodes join/leave



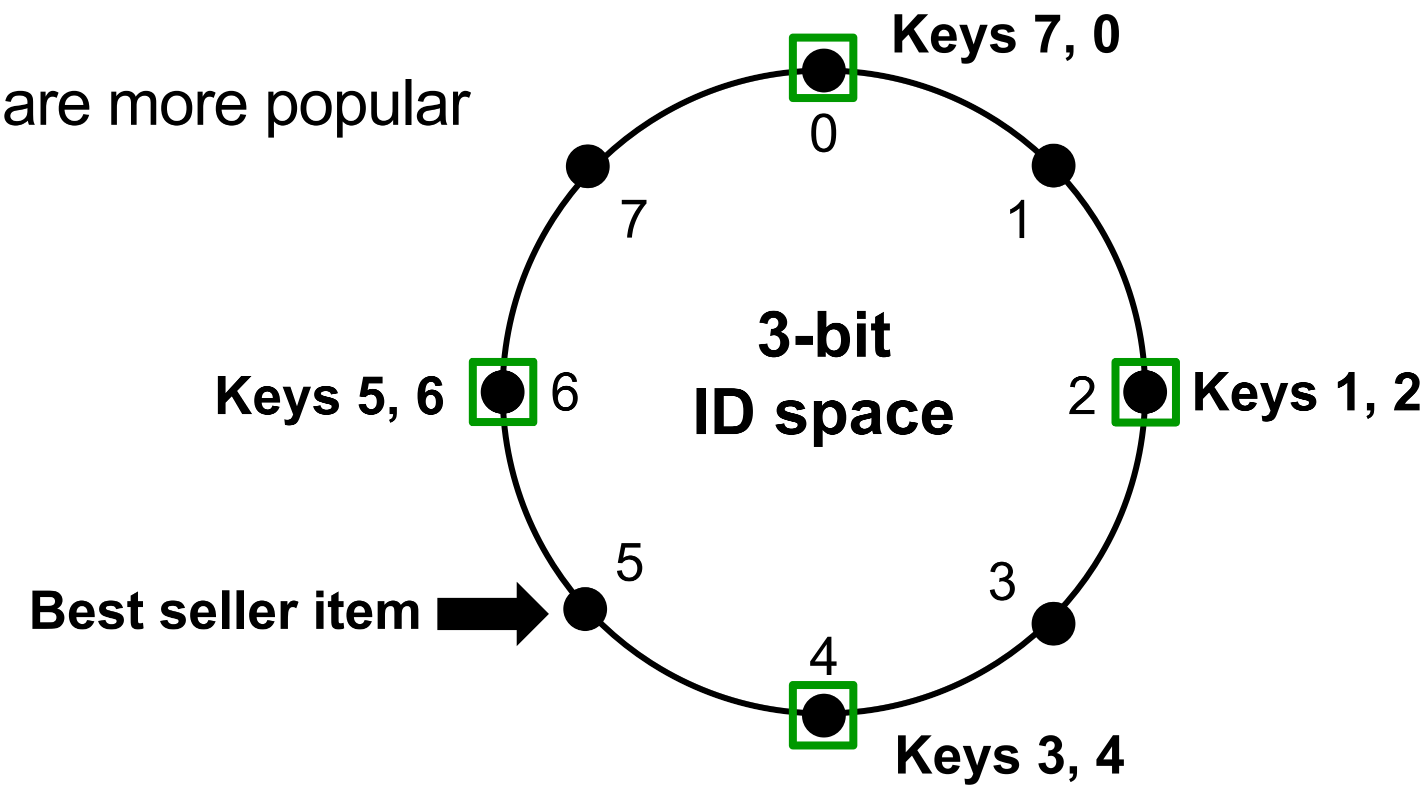
Challenge: unbalanced load

- Nodes are assigned different # of keys
- Unbalanced with nodes join/leave



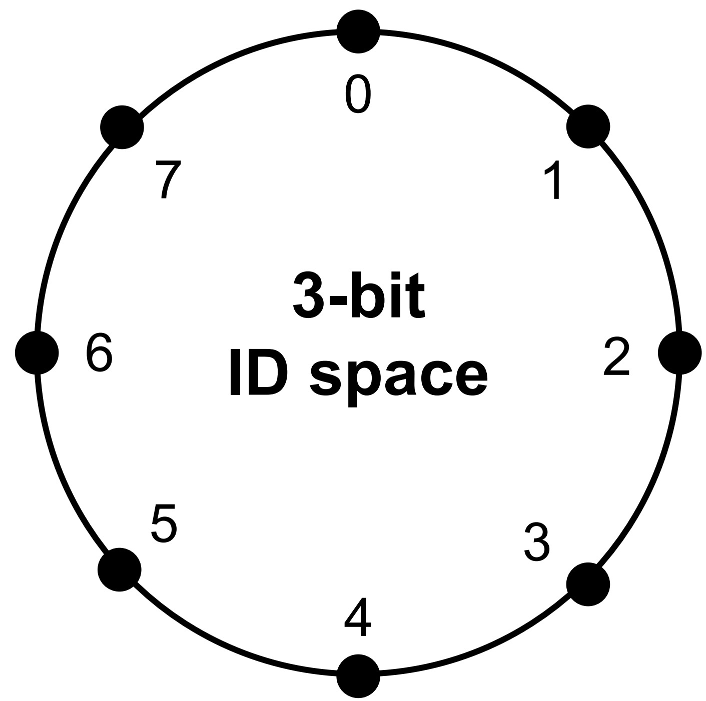
Challenge: unbalanced load

- Nodes are assigned different # of keys
- Unbalanced with nodes join/leave
- Some keys are more popular



Solution: virtual nodes (vnodes)

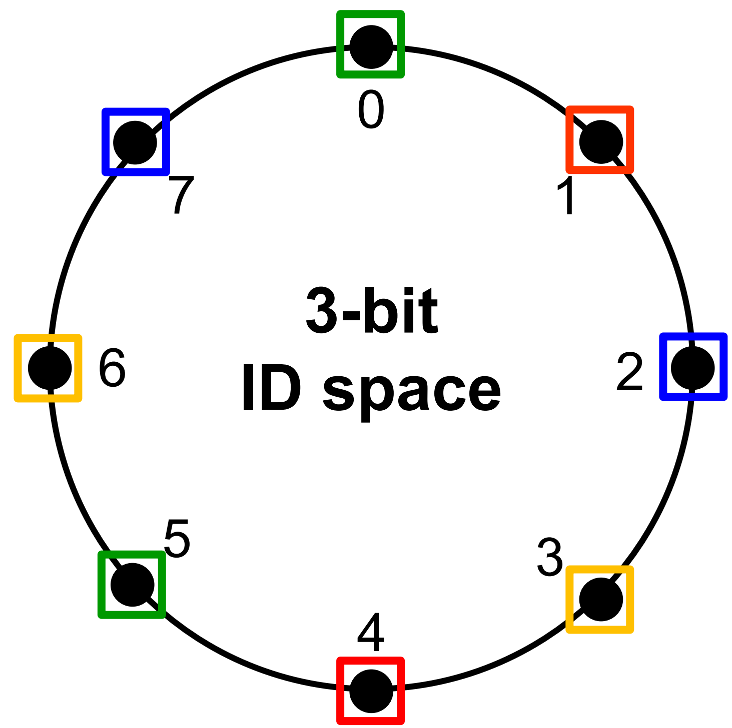
- An extra level of mapping
  - From node id in the ring to physical node
  - Node ids are now virtual nodes (tokens)
  - Multiple node ids → same physical node



Solution: virtual nodes (vnodes)

- An extra level of mapping
  - From node id in the ring to physical node
  - Node ids are now virtual nodes (tokens)
  - Multiple node ids → same physical node

4 physical nodes (servers)
2 vnodes / server

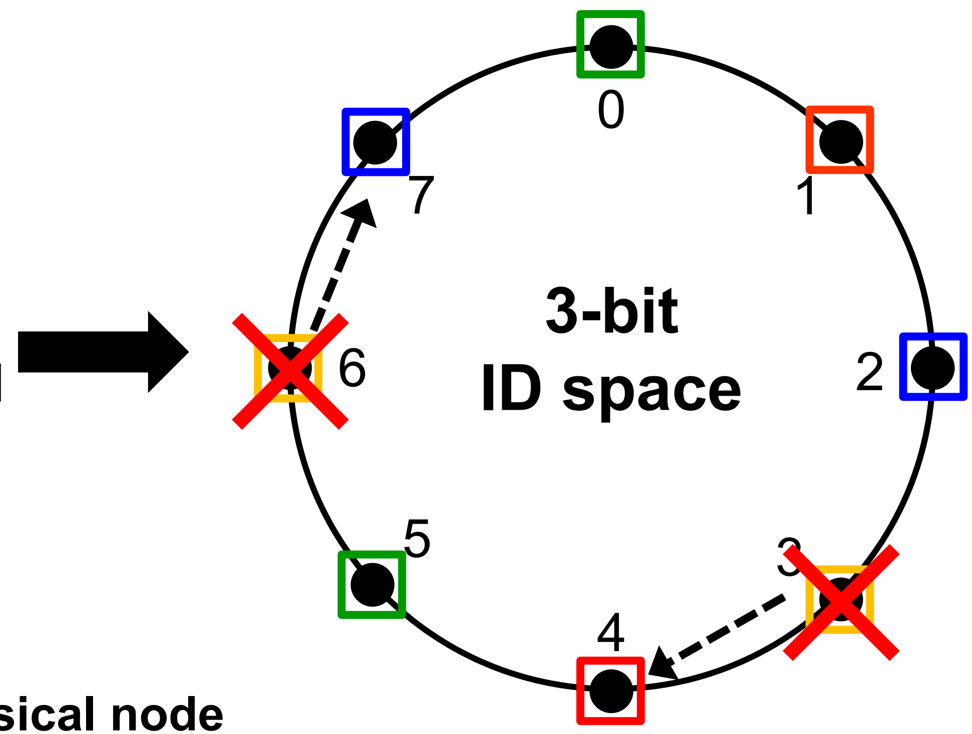


Virtual node:
same color → same physical node

Solution: virtual nodes (vnodes)

- An extra level of mapping
  - From node id in the ring to physical node
  - Node ids are now virtual nodes (tokens)
  - Multiple node ids → same physical node

Orange server leaves
Keys moved to blue and red

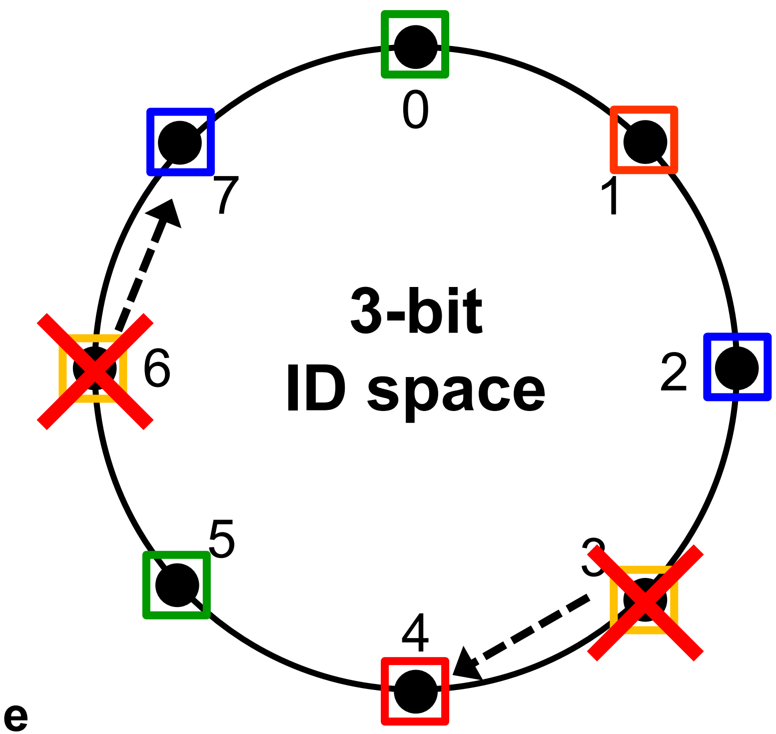


Virtual node:
same color → same physical node

Solution: virtual nodes (vnodes)

- An extra level of mapping
  - From node id in the ring to physical node
  - Node ids are now virtual nodes (tokens)
  - Multiple node ids → same physical node

- More virtual nodes, more balanced
- Faster data transfer for join/leave
- Controllable # of vnodes / server
  - Server capacity: e.g., CPU, memory, network



Virtual node: same color → same physical node

Gossip and “lookup”

- **Gossip:** Once per second, each node contacts a **randomly chosen other node**
  - They **exchange their lists of known nodes** (including virtual node IDs)
- Assumes all nodes will come back eventually, doesn’t repartition
- Each node **learns** which others handle all **key ranges**

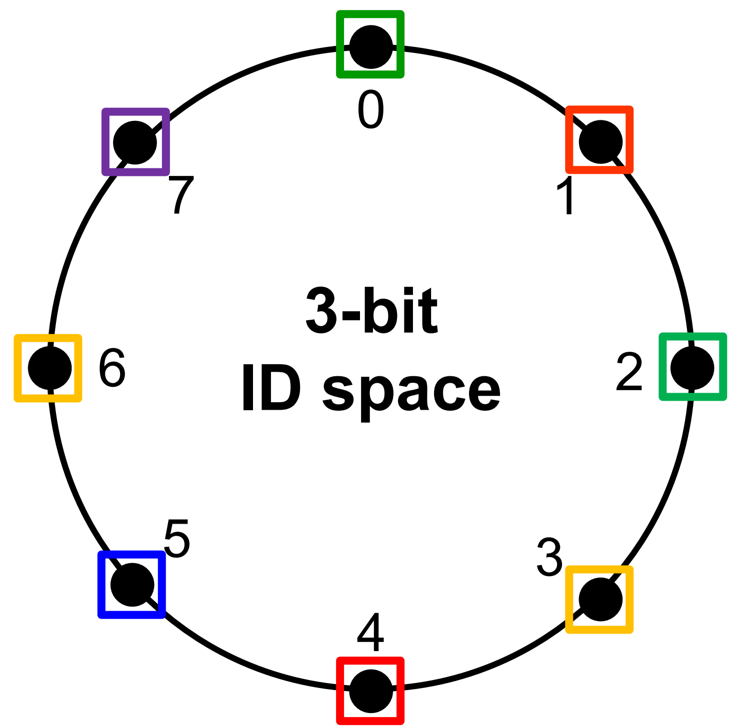
  - **Result: All** nodes can send **directly to any key’s coordinator (“zero-hop DHT”)**
    - **Reduces variability** in response times

Today: Amazon Dynamo

1. Background and system model
2. Data partitioning
3. **Failure handling**

Preference list (data replication)

- Key replicated on M vnodes
  - Remember "r-successor" in DHT?
- All M vnodes on distinct servers across different datacenters

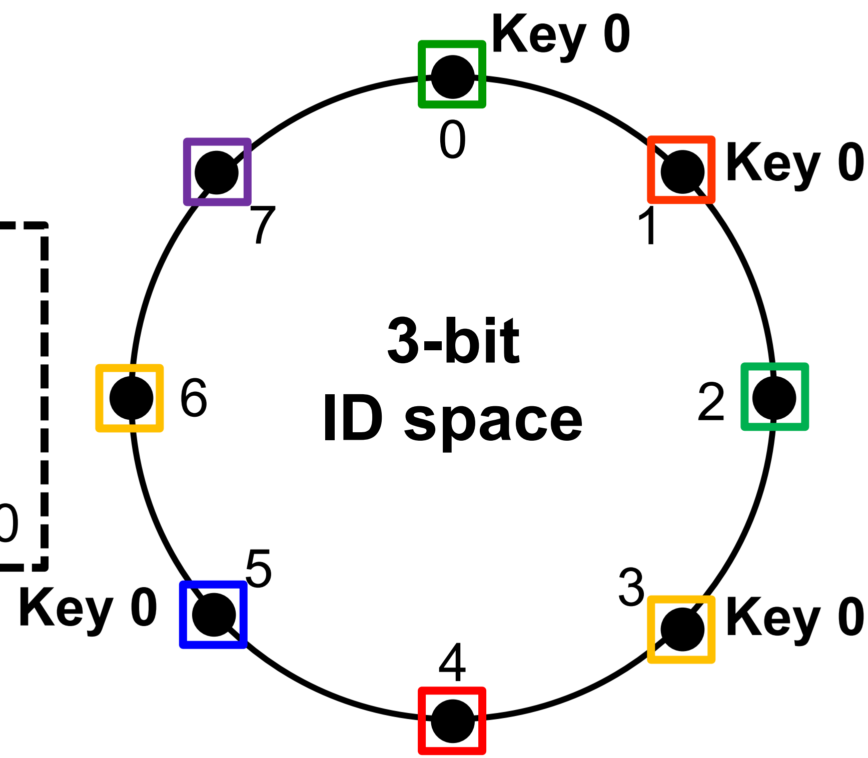


Virtual node:
5 colors → 5 physical nodes

Preference list (data replication)

- Key replicated on M vnodes
  - Remember “r-successor” in DHT?
- All M vnodes on distinct servers across different datacenters

M = 4
Key 0’s Preference list could be vnodes: {0, 1, 3, 5} mapping to servers: {green, red, orange, blue}
Green is the **coordinator** server of key 0



Virtual node:
5 colors → 5 physical nodes

Read and write requests

- Received by the coordinator (this is not Chord)
  - Either the client (web server) knows the mapping or re-routed
- Sent to first N “healthy” servers in preference list (coordinator incl.)
  - Durable writes: my updates recorded on multiple servers
  - Fast reads: possible to avoid straggler
- A write creates a new immutable version of the key (no overwrite)
  - Multi-versioned data store
- Quorum-based protocol
  - A write succeeds if W out of N servers reply (write quorum)
  - A read succeeds if R out of N servers reply (read quorum)
  - **W + R > N**

Quorum implications (W, R, and N)

- N determines the durability of data (Dynamo N = 3)

- W and R adjust the **availability-consistency tradeoff**
  - W = 1 (R = 3): fast write, weak durability, slow read
  - R = 1 (W = 3): slow write, good durability, fast read
  - Dynamo: W = R = 2

- Why W + R > N ?
  - Read and write quorums overlap **when there are no failures**!
  - Reads see all updates without failures
    - What if there are failures?

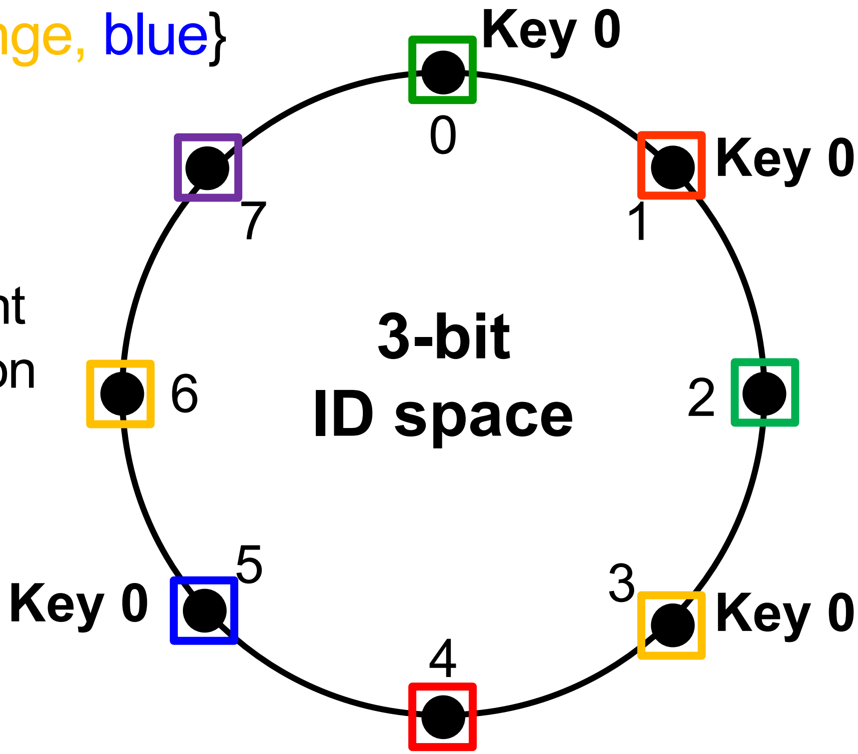
Failure handing: sloppy quorum + hinted handoff

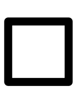
- Sloppy: not always the same servers used in N
  - First N servers in the preference list without failures
  - Later servers in the list take over if some in the first N fail

- Consequences
  - Good performance: no need to wait for failed servers in N to recover
  - Eventual (weak) consistency: conflicts are possible, versions diverge
  - Another decision on **availability-consistency tradeoff**!

Failure handing: sloppy quorum + hinted handoff

- Key 0's preference list {green, red, orange, blue}
- N = 3: {green, red, orange} without failures
- If red fails, requests go to {green, orange, blue}
- **Hinted handoff**
  - Blue temporarily serves requests
  - Hinted that red is the intended recipient
  - Send replica back to red when red is on





Wide-area replication

- Last ¶, § 4.6: **Preference lists always** contain nodes from **more than one data center**
  - **Consequence:** Data likely to **survive failure** of **entire data center**

- Blocking on **writes to a remote data center** would incur unacceptably high latency
  - **Compromise:** **W < N**, eventual consistency
  - Better **durability** & **latency** but worse **consistency**

Conflicts

- Suppose **N = 3, W = R = 2,** nodes are **A, B, C, D, E**
  - CL1 put(k, …) completes on **A** and **B**
  - CL2 put(k, …) completes on **C** and **D**

- **Conflicting results** from **A, B** and **C, D**
  - Each has seen a **different put(k, …)**

- How does Dynamo handle conflicting versions?

An example of conflicting writes (versions)

Preference list (M = 5, N = 3)

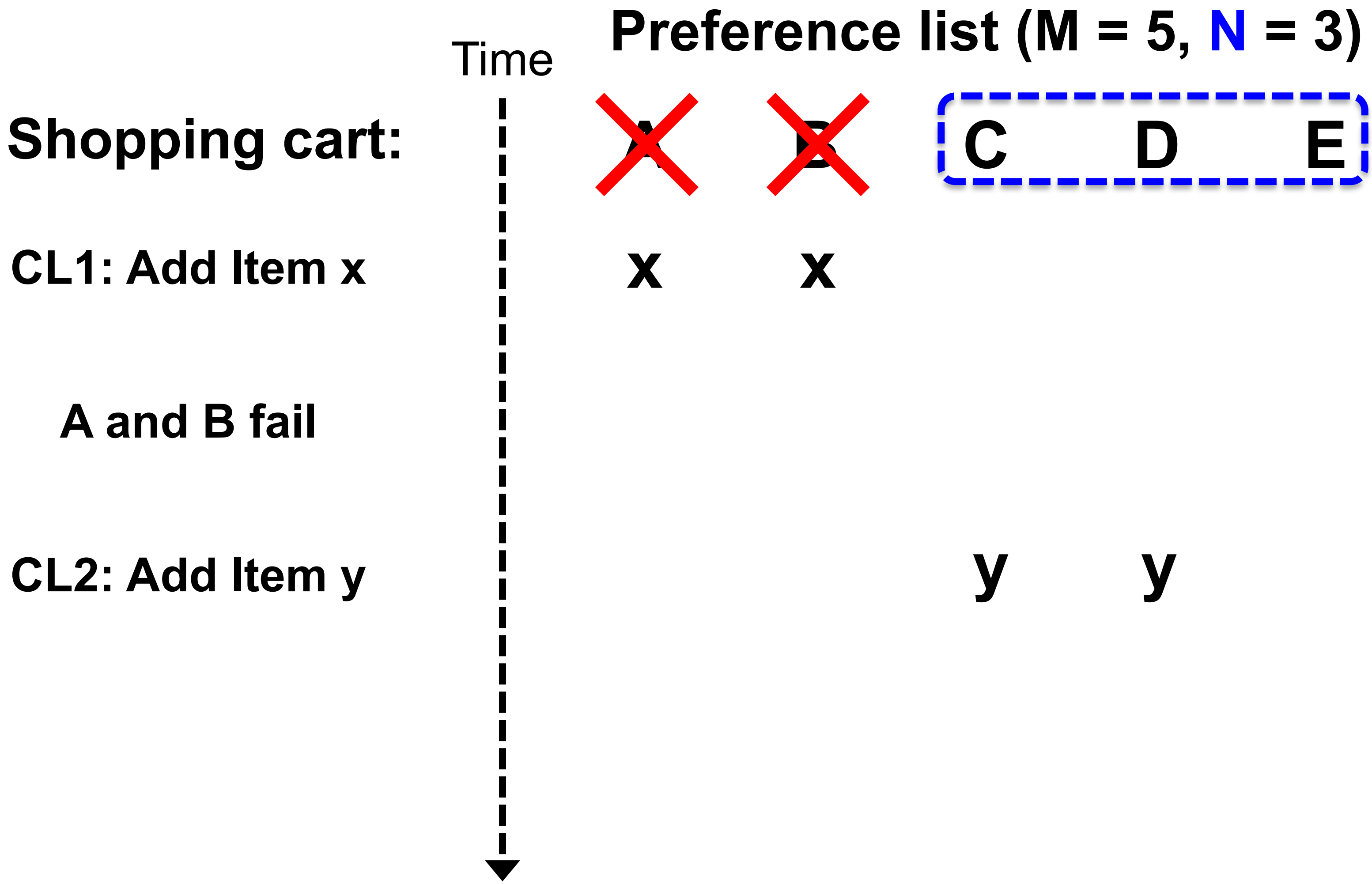
Time

Shopping cart: **A B C D E**

CL1: Add Item x **x x**

A and B fail

An example of conflicting writes (versions)



An example of conflicting writes (versions)

Preference list (M = 5, N = 3)

Time ↓

	A	B	C	D	E
Shopping cart:	A	B	C	D	E
CL1: Add Item x	x	x			
A and B fail					
CL2: Add Item y			y	y	
A and B recover					
CL1: Read cart		read	read		

Conflicting versions only possible under failures

Vector clocks: handling conflicting versions

Preference list (M = 5, N = 3)

Time

Shopping cart: A B C D E

CL1: Add Item x

x (A,1) x (A,1)

Read returns
x (A,1) and **y (C,1)**
(A,1) and (C,1) are
not causally related:
conflicts!

y (C,1) y (C,1)

CL1: Read cart

read read

Can we use Lamport clocks?

Version vectors (vector clocks)

- ***Version vector:*** List of **(coordinator node, counter)** pairs
  - *e.g.*, [(A, 1), (B, 3), …]

- Dynamo stores a version vector with **each stored** key-value **pair**

- **Idea:** track "ancestor-descendant" relationship between different versions of data stored under the same key **k**

Dynamo's system interface

- get(key) → value, **context**
  - Returns one value or multiple conflicting values
  - Context describes version(s) of value(s)

- put(key, **context**, value) → "OK"
  - **Context** indicates **which versions** this version supersedes or merges

Version vectors: Dynamo's mechanism

- **Rule:** If vector clock comparison of v1 < v2, then the first is an ancestor of the second – **Dynamo can forget v1**

- Each time a put() occurs, Dynamo increments the counter in the V.V. for the coordinator node

- Each time a get() occurs, Dynamo returns the V.V. for the value(s) returned (in the "context")

  – Then users **must supply that context** to put()s that modify the same key

Conflict resolution (reconciliation)

- If vector clocks show causally related (not really conflicting)
  - System overwrites with the later version
- For conflicting versions
  - *System handles it automatically, e.g., last-writer-wins (limited use case)*
  - **Application specific resolution (most common)**
    - Clients resolve the conflict via reads, e.g., merge shopping cart

Vector clocks: handling conflicting versions

Preference list (M = 5, N = 3)

Time

	A	B	C	D	E
Shopping cart:	A	B	C	D	E
CL1: Add Item x	x (A,1)	x (A,1)			
CL2: Add Item y			y (C,1)	y (C,1)	
CL1: Read cart x (A,1), y (C,1)					

Vector clocks: handling conflicting versions

Time

Preference list (M = 5, N = 3)

Shopping cart: A B C D E

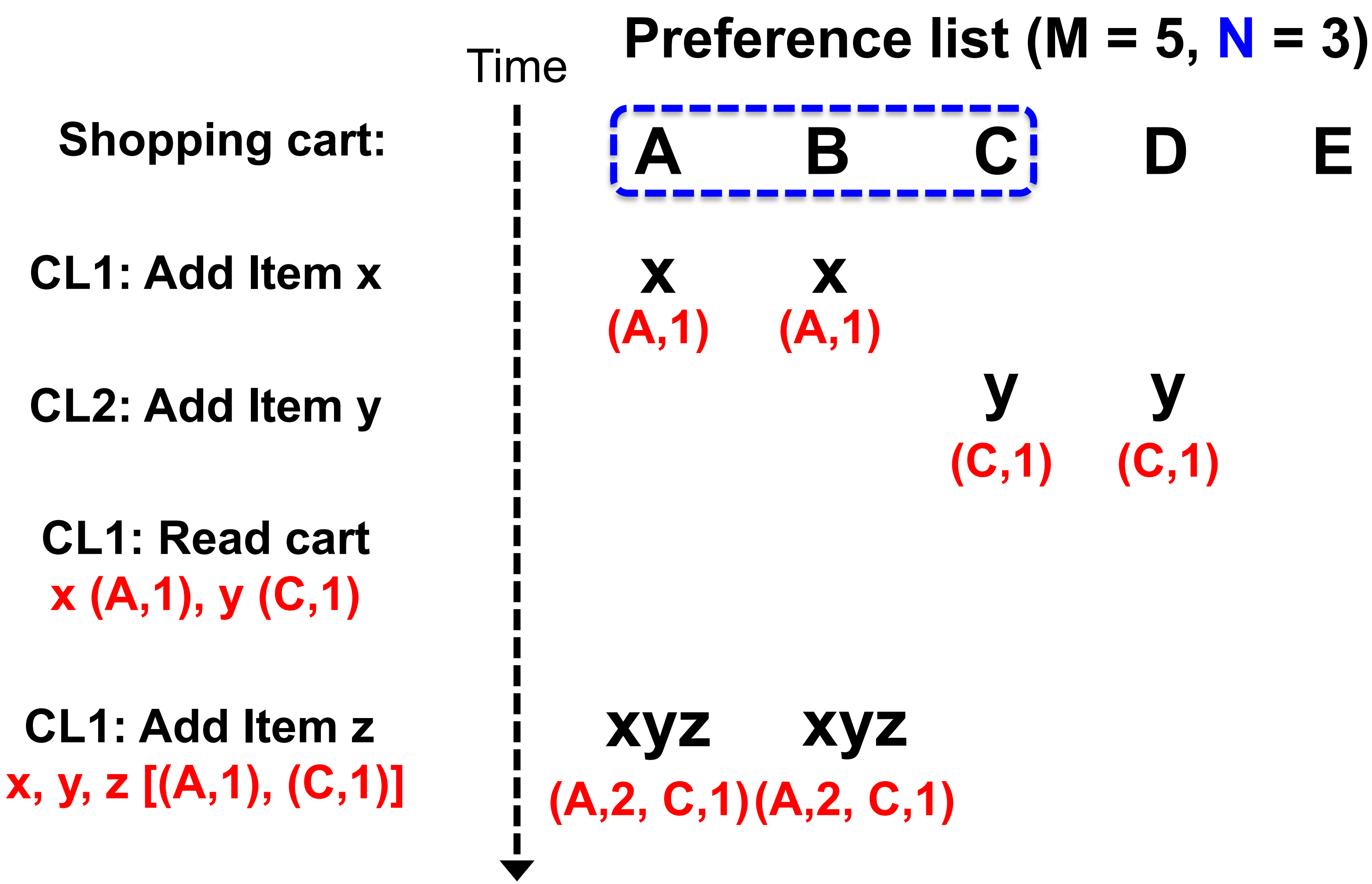
CL1: Add Item x — A: x (A,1); B: x (A,1)

CL2: Add Item y — C: y (C,1); D: y (C,1)

CL1: Read cart
x (A,1), y (C,1)

CL1: Add Item z
x, y, z [(A,1), (C,1)]

Vector clocks: handling conflicting versions



How useful is it to vary N, R, W?

N	R	W	Behavior
3	**2**	**2**	**Parameters from paper:** **Good durability, good R/W latency**
3	3	1	
3	1	3	
3	3	3	
3	1	1	

How useful is it to vary N, R, W?

N	R	W	Behavior
3	**2**	**2**	**Parameters from paper:** **Good durability, good R/W latency**
3	3	1	Slow reads, **weak durability, fast writes**
3	1	3	**Slow writes,** strong durability, fast reads
3	3	3	More likely that **reads see all prior writes**?
3	1	1	Read quorum **may not overlap** write quorum

Failure detection and ring membership

- Server A considers B has failed if B does not reply to A's message
  - Even if B replies to C
  - A then tries alternative nodes

- With servers join and permanently leave
  - Servers periodically send gossip messages to their neighbors to sync who are in the ring
  - Some servers are chosen as seeds, i.e., common neighbors to all nodes

Anti-entropy (replica synchronization)

- Hinted handoff node **crashes before it can replicate data** to node in **preference list**
  - Need another way to **ensure** that each key-value pair is **replicated N times**

- **Mechanism: replica synchronization**
  - Nodes nearby on ring periodically **gossip**
    - **Compare** the (k, v) pairs they hold
    - **Copy** any missing keys the other has

How to **compare and copy** replica state **quickly and efficiently?**

Efficient synchronization with Merkle trees

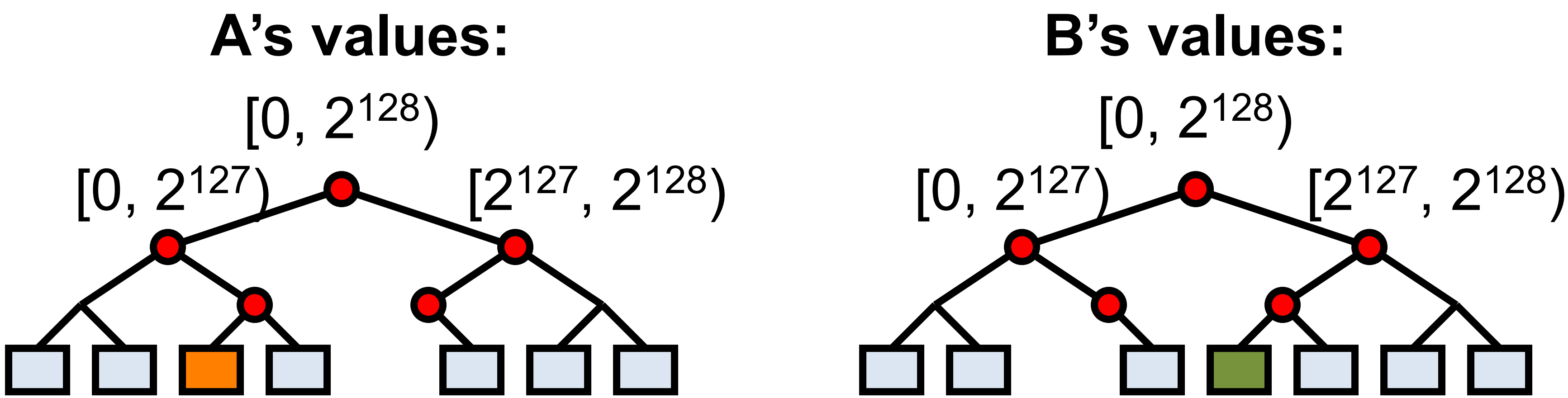
- **Merkle trees hierarchically summarize** the key-value pairs a node holds

- One Merkle tree for each **virtual node key range**
  - **Leaf node** = hash of **one key's value**
    (# of leaves = # keys on the virtual node)
  - **Internal node** = hash of **concatenation of children**

- Replicas exchange trees from top down, depth by depth
  - **If root nodes match**, then identical replicas, stop
  - **Else**, go to next level, compare nodes pair-wise

Merkle tree reconciliation

- **B** is missing orange key; **A** is missing green one
- Exchange and compare hash nodes from root downwards, **pruning when hashes match**



Finds differing keys **quickly** and with minimum information exchange

Dynamo: Take-aways ideas

- Availability is important
  - Systems need to be scalable and reliable
- Dynamo is eventually consistent
  - Many design decisions **trade consistency for availability**
- Core techniques
  - **Consistent hashing**: data partitioning
  - **Replication, preference list, sloppy quorum, hinted handoff**: availability under failures
  - **Vector clocks**: conflict resolution (partly automatic, rest app.)
  - **Anti-entropy**: synchronize replicas
  - **Gossip**: synchronize ring membership