CS X55: DISTRIBUTED SYSTEMS
[DYNAMO & GFS]

Go virtual!
Looking to scale
As things fail?
Go with nodes that are virtual
And, yes, they are just as real

Allowing nodes to take on load
That balance without arduous code
Commensurate with ability
With hotspots a *very* slim possibility

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- Do sloppy quorums need DHTs as its underlying structure?
- Virtual nodes …?

Topics covered in this lecture

- Amazon's Dynamo
  - Partitioning Algorithm
  - Replication & Versioning
  - Experiences
- The Google File System

My room is round when I lay down, when I wake up it's square
When I go outside it's on a spiral set of stairs
The people that surround me are waiting out there
In a round room they can't find me anywhere

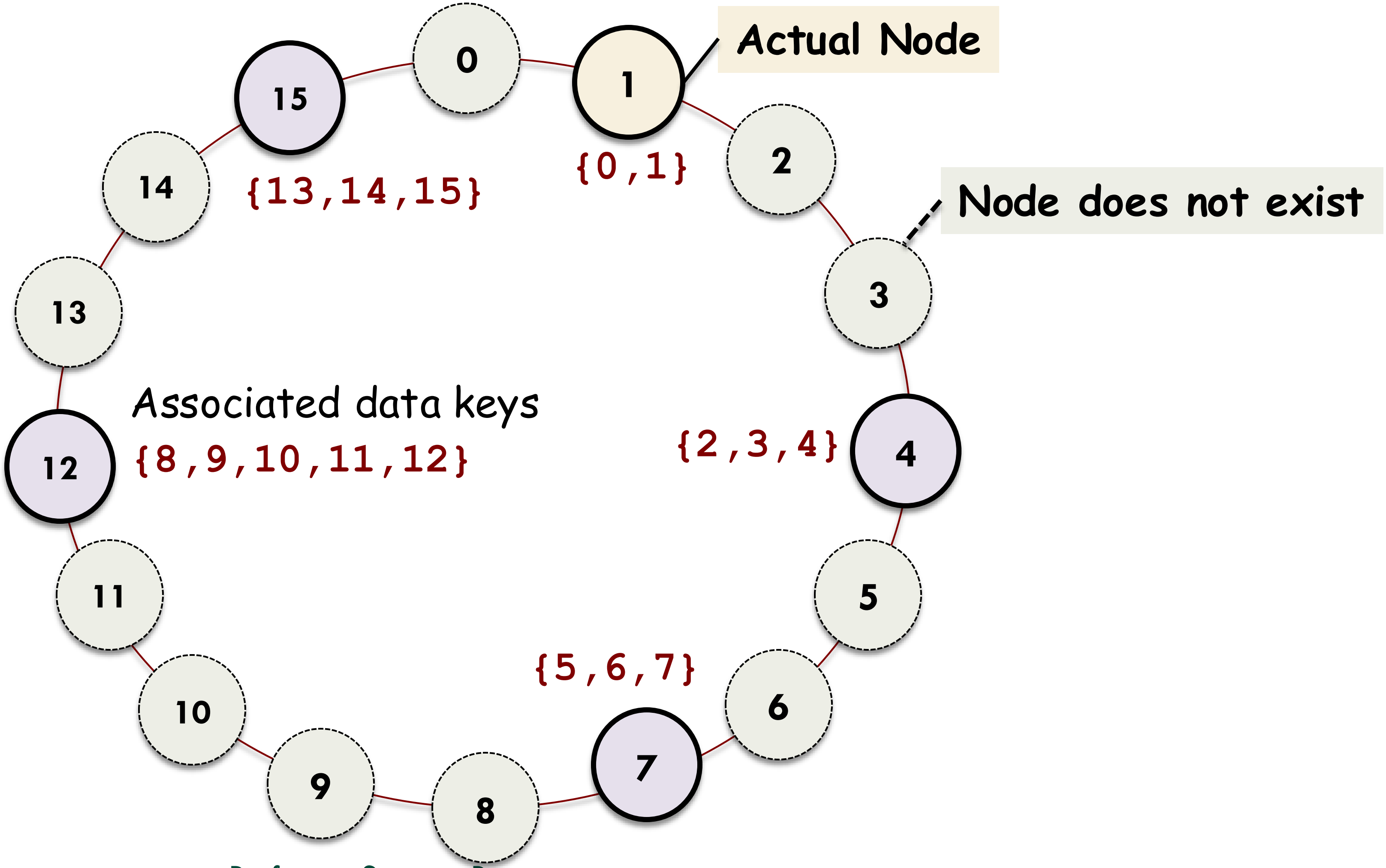
The Round Room; Mike Gordon; Phish

CHORD

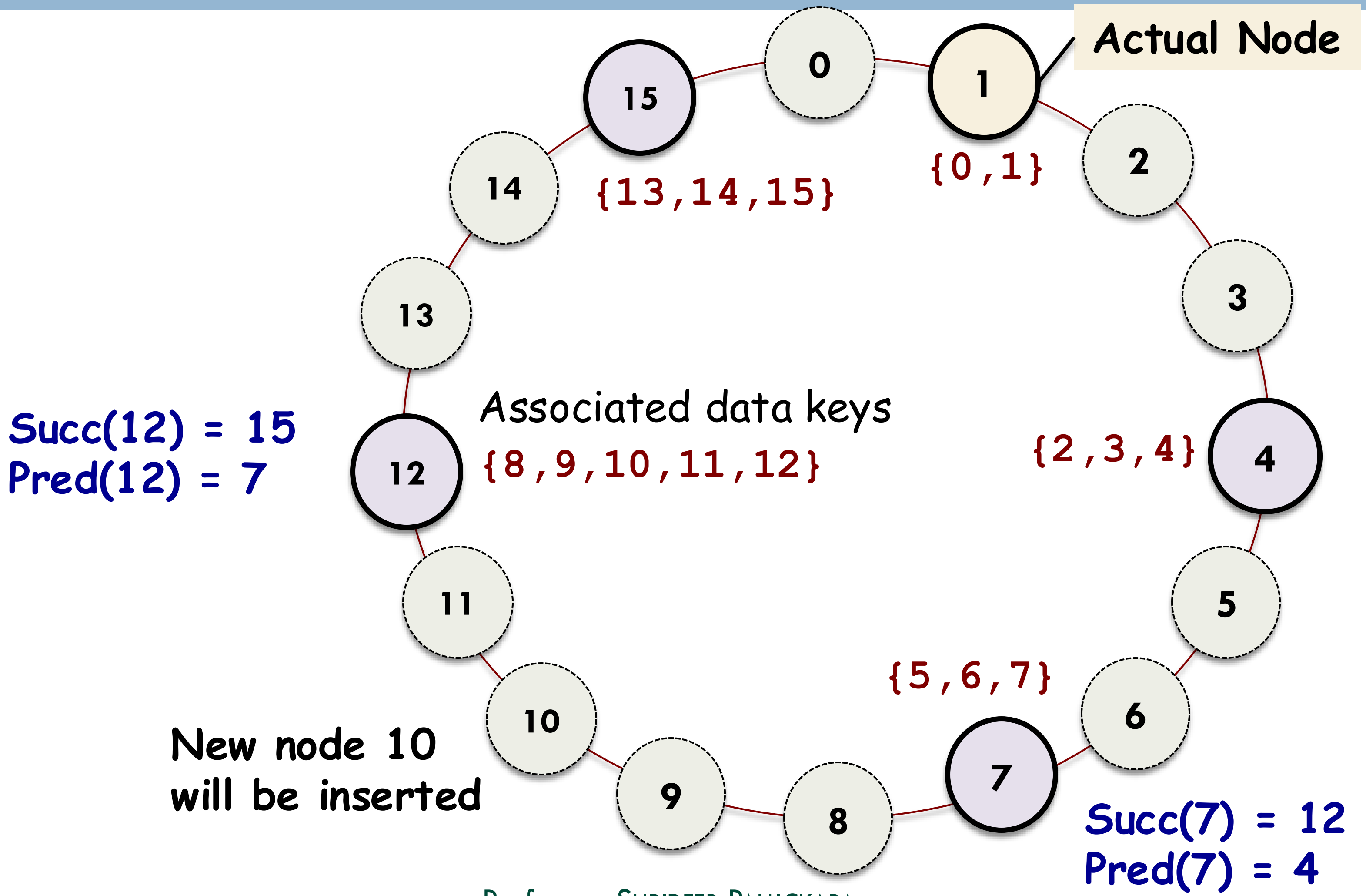
The Chord System

- Assigns IDs to keys and nodes from the same 1-dimensional ID space
- Nodes are organized into a **ring**
- Data item with key **`k`** is mapped to a node with the **smallest** **`id`** $\geq$ **`k`**
  - Also referred to as `successor(`**`k`**`)`

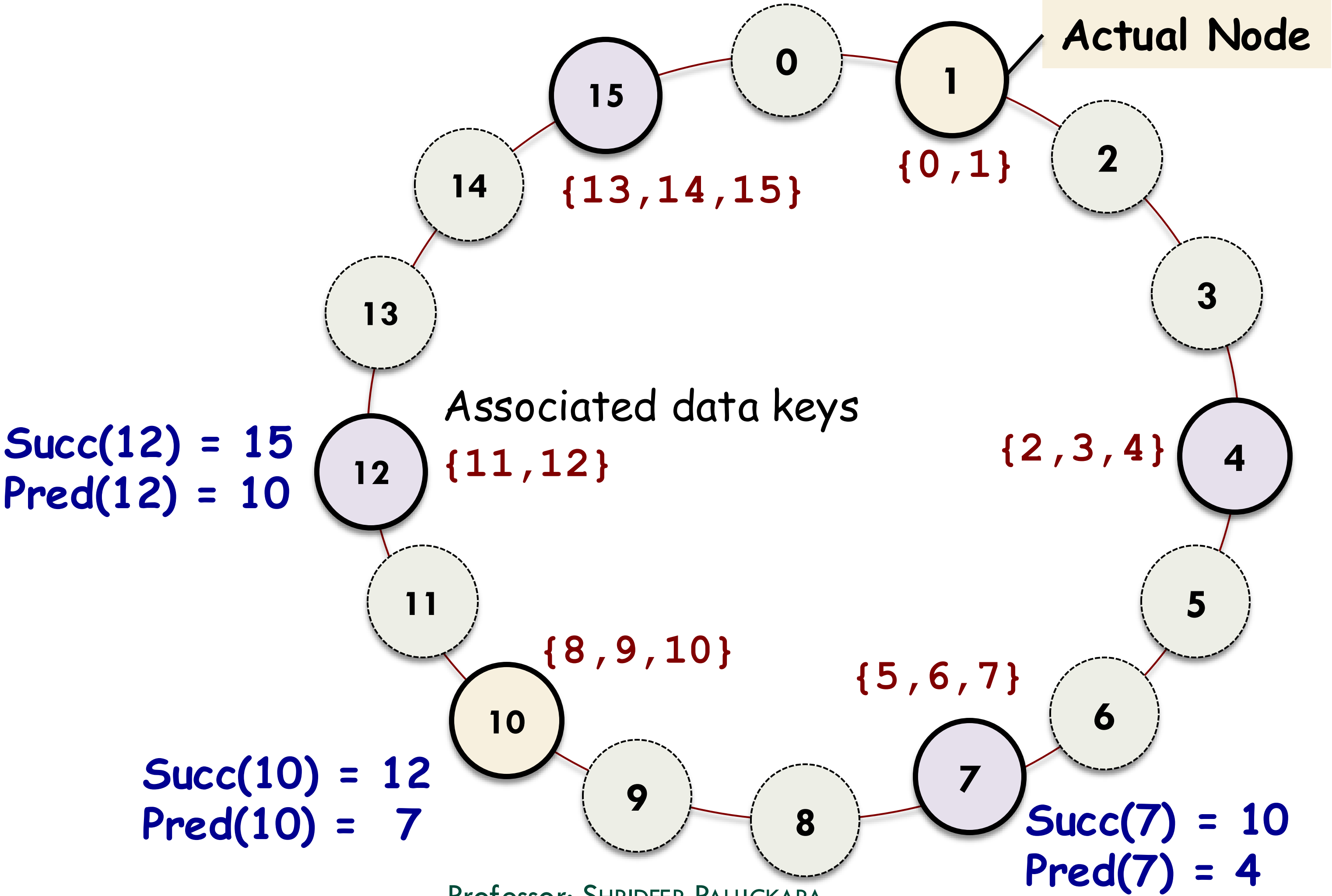
Mapping of data items to nodes in Chord

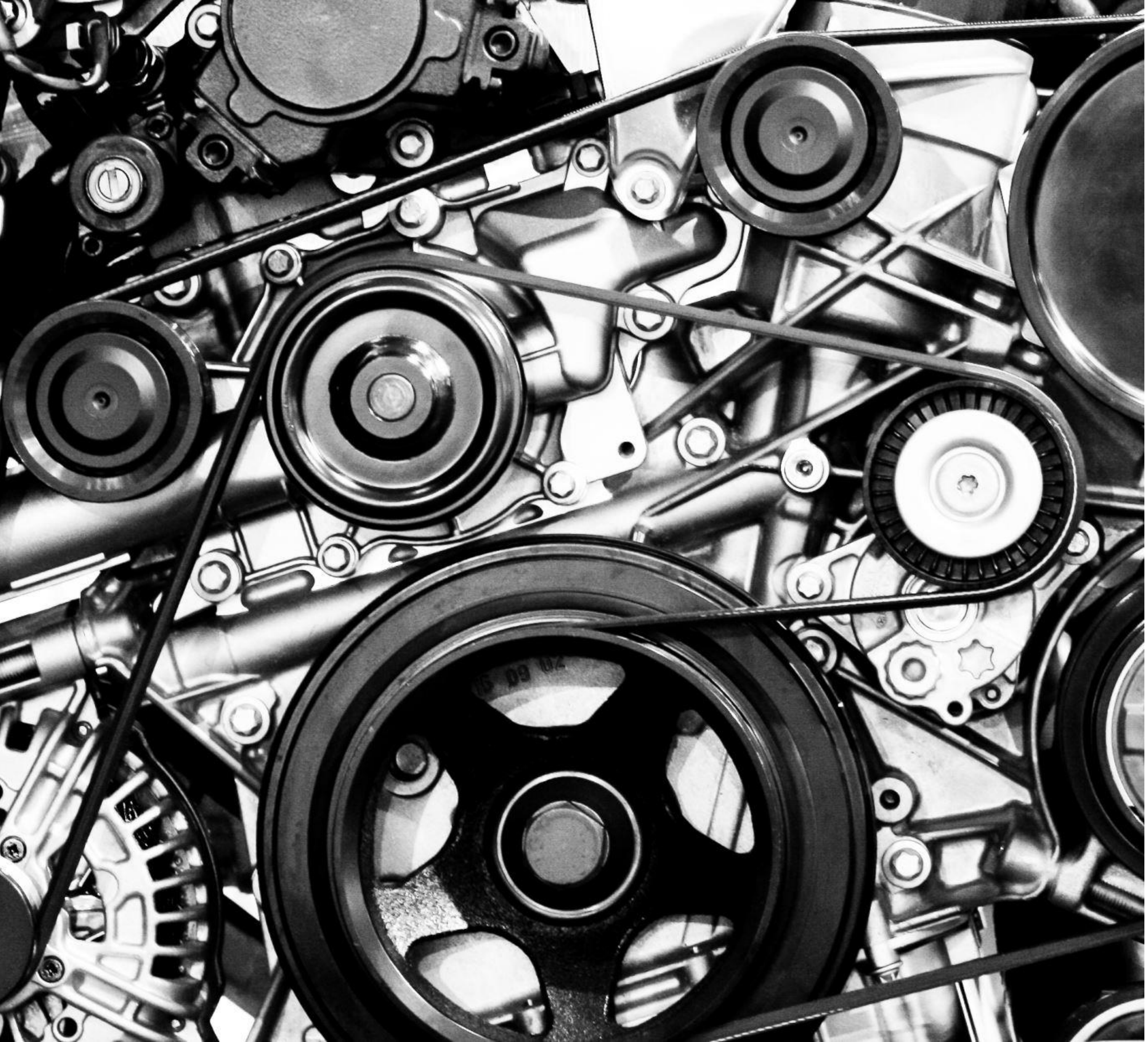


An example of inserting a new node



An example of inserting a new node





Dynamo Partitioning Algorithm

COLORADO STATE UNIVERSITY

A key requirement is that Dynamo must scale incrementally

- *Dynamically partition* data over a set of storage nodes
- Uses **consistent hashing**
  - DHT
  - Data item identified by key
    - Assigned to node responsible for MD5-hash(key)

Basic hashing scheme presents some challenges

- Random position assignment may lead to
  - Non-uniform data and load distribution

- Algorithm **oblivious** to heterogeneity of devices

Dynamo uses a variant of consistent hashing

- Introduces the notion of **virtual nodes**
- Virtual node looks like a real node
- Each node is responsible for <u>more than 1</u> virtual nodes
  - A node is assigned **multiple positions** in the ring

Advantages of virtual nodes

- If a node becomes *unavailable*
  - **Load** handled by failed node, **dispersed** across remaining virtual nodes
- When node becomes available again
  - Accepts roughly the same amount of work from other nodes
- **Number of virtual nodes** are decided based on machine's capacity

DYNAMO REPLICATION

Dynamo replicates data on multiple hosts

- Each data item is replicated at **N** hosts
- Coordinator is *responsible* for nodes that fall in its range
- Additionally, a coordinator *replicates* key at **N**-1 clockwise **successor** nodes

What does this mean?

- Each node is responsible for <u>region between</u>
  - *Itself* and its N^{th} *predecessor*
- List of nodes responsible for a key
  - Preference list
- A node maintains a list of more than N to account for failures
  - Account for virtual nodes
    - Make sure your list contains *different* physical nodes

Dynamo Versioning

Data versioning

- A put() may return *before* it is applied to all replicas
- If there are no failures
  - **Upper bound** on update propagation times
- If there are failures
  - Things take much longer

There are applications at Amazon that tolerate this

- Shopping carts
- Add to Cart can never be forgotten or rejected
- If most recent state of cart unavailable
  - Make changes to the *older* version
  - **Divergent** versions are reconciled later

Dynamo treats each modification as a new, immutable version of the data

- Multiple versions of data present at same time
- Often new versions **subsume** old data
  - *Syntactic* reconciliation
- When an automatic reconciliation is not possible
  - Clients have to do it
  - **Collapse** branches into one
  - Manage your shopping cart

Dynamo uses vector clocks to capture causality

- A vector clock for *each* version of the object
- Two versions of object being compared
  - If $VC_1 <= VC_2$ **for all** indices of the vector clock
    - O_1 occurred before O_2
  - Otherwise, changes are in conflict
    - Need reconciliation

A client must specify which version it is updating

- Pass context from an earlier read operation
  - Context contains vector clock information
- Requests with branches that cannot be reconciled?
  - Returns **all** objects with versioning info in **context**
  - Update done using this context *reconciles* and *collapses* all branches

Execution of get() and put() operations

- Read and write operations involve the first **N** healthy nodes
- During failures, nodes lower in priority are accessed

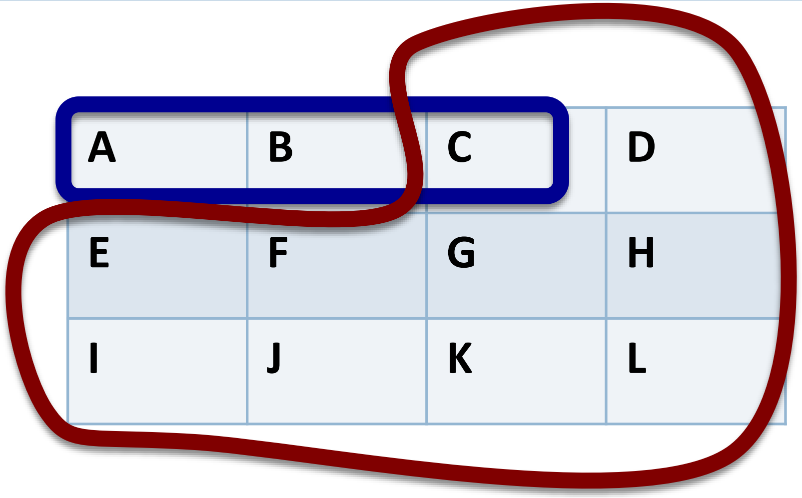
To maintain consistency, Dynamo uses a quorum protocol

- Uses configurable settings for replicas that must participate in
  - Reads
  - Writes

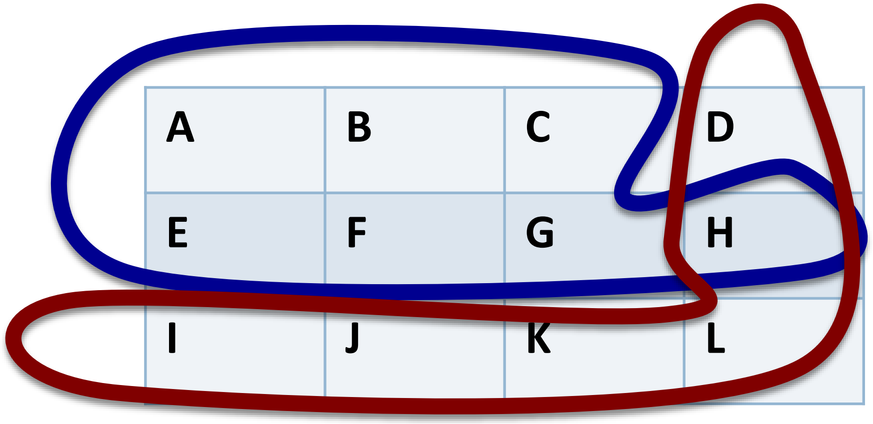
Quorum-based protocols: When there are N replicas

- Read quorum N_R
- To modify a file write-quorum N_W
- $N_R + N_W > N$
  - Prevent **read-write** conflict
- $N_W > N/2$
  - Prevent **write-write** conflict

Quorum-based protocols: Example



N_R=3 N_W=10



N_R=7 N_W=6

Read Quorum:

Write Quorum:

Write-write conflict

Concurrent writes to {A, B, C, E, F, G} and {D, H, I, J, K, L} will be accepted

Upon receiving a put() request for a key

- Coordinator generates a **vector clock** for new version
  - Sends new version to $\mathbf{N}$ highest-ranked reachable nodes
  - If at least $\mathbf{N_W - 1}$ nodes respond: write is successful!

External Discovery: During node adds

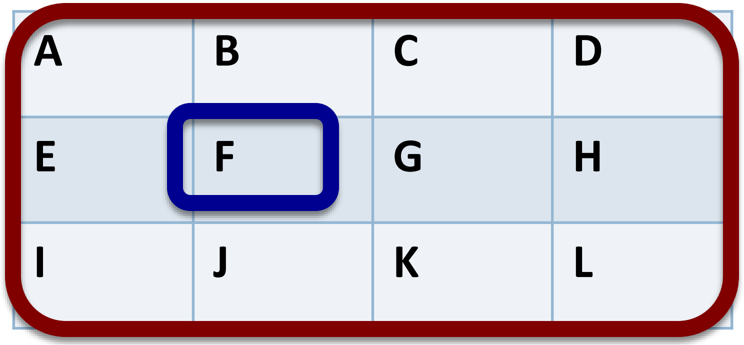
- When **A** and **B** join, it might be a while before they know each other's existence
  - **Logical partitioning**
- Use seed nodes that are known to all nodes
  - All nodes *reconcile* membership with seed

DYNAMO: EXPERIENCES

Popular reconciliation strategies

- Business logic specific
- Timestamp
  - Last write wins
- High performance read engine
  - High read rates
  - Small update rates
    - $N_R=1$ and $N_W=N$

Quorum-based protocols: Example 2



N_R=1 N_W=12

☺

Read Quorum: —
Write Quorum: —

Common configuration of the quorum

- $N_R=2$
- $N_W=2$
- $N=3$

Balancing performance and durability

- Some services not happy with 300 ms SLA
  - Writes tend to be slower than reads

- To cope with this, nodes maintain **object buffer**
  - Main memory
  - *Periodically* written to storage

SANJAY GHEMAWAT, HOWARD GOBIOFF, SHUN-TAK LEUNG:
The Google file system. Proceedings of SOSP 2003: 29-43

THE GOOGLE FILE SYSTEM

Broad brushstroke themes in current extreme scale storage systems

- Voluminous data
- Commodity hardware
- Distributed Data
- Expect failures
- Tune for access by applications
- Optimize for dominant usage
- Tradeoff between consistency and availability

Demand pulls in GFS: I

- Component failures are the **norm**
- Files are huge by traditional standards
- File mutations predominantly through **appends**
  - Not overwrites
- Applications and File system API designed in **lock-step**

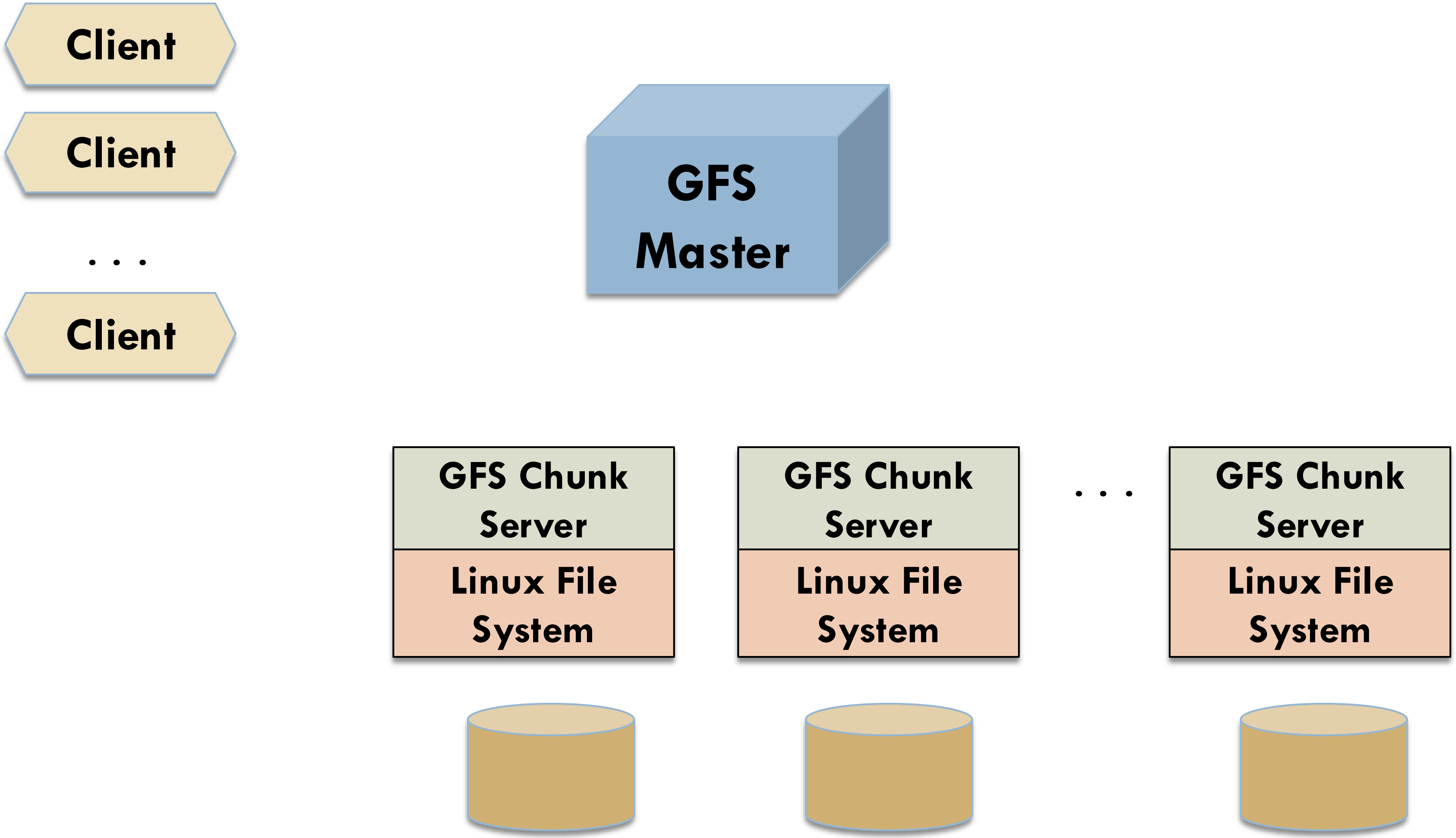
Demand pulls in GFS - II

- Hundreds of producers will **concurrently append** to a file
  - Many-way merging
- High **sustained bandwidth** is more important than low latency

The file system interface

- Does not implement standard APIs such as POSIX
- Supports **create, delete, open, close, read** and **write**
- **snapshot**
  - Create a fast copy of file and directory tree
- **record append**
  - Multiple writers can concurrently append records to the same file
    - Without additional locking

Architecture of GFS



In GFS a file is broken up into fixed-size chunks

- Obvious reason
  - The file is too big

Map-Reduce ↓

- **Set the stage** for computations that operate on this data
  - Parallel I/O
  - I/O seek times are 14 x 10^6 slower than CPU clock cycles

In GFS a file is broken up into fixed-size chunks

- Each chunk has a 64-bit globally unique ID
  - Assigned by the Master
- Chunks are stored by chunk servers
  - On local disks as LINUX files
- Each chunk is **replicated**
  - Default is 3

Master operations

- Manage system **metadata**
- **Leasing** of chunks
- **Garbage collection** of orphaned chunks
- Chunk **migrations**

ALL system metadata is managed by the Master and stored in **main memory**

1. File and chunk namespaces
2. Mapping from files to chunks
3. Location of chunks

} *Logs mutations into a permanent log*

Why have a single Master?

- Vastly **simplifies** design
- Easy to use global knowledge to **reason about**
  - Chunk placements
  - Replication decisions

Communications with the chunk servers

- Periodic communications using **heartbeats**
  - Instructions to the chunk server
  - Collect/retrieve state from the chunk server

Chunk size

- This is fixed at **64 MB**
  - Much larger than typical filesystem block sizes (512 bytes)

- **Lazy space allocation**
  - Stored as plain Linux file
  - Extended only as needed

But why this big?

- **Reduces client interaction** with the master
  - Can cache info for a multi-TB working set
- Reduce network **overhead**
  - With a large chunk, client performs more operations
  - Persistent connections
- Reduce **size of metadata** stored in the master
  - 64 bytes of metadata per 64 MB chunk

Why keep the entire metadata in memory?

- **Speed**
- Master can **scan** its **state** in the background
  - Implement chunk garbage collection
  - Re-replicate if there are failures
  - Chunk migration to balance load and space
- **Add** extra memory to increase file system size

Size of the file system with 1 TB of RAM: Assume file sizes are exact multiples of chunk sizes

- Number of entries = $2^{40}/2^{6}$
- MAXIMUM SIZE of the file system

$$= \text{Number of entries} \times \text{Chunk size}$$

$$= \frac{2^{40}}{2^{6}} \times 2^{6} \times 2^{20}$$

$$= 2^{60} = 1 \text{ EB}$$

The contents of this slide-set are based on the following references

- Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, Werner Vogels: *Dynamo: Amazon's Highly Available Key-value Store*. SOSP 2007: 205-220
- Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung: The Google file system. Proceedings of SOSP 2003: 29-43.