LAB 7 STARTING ADVICE; SCALING APPROACHES

George Porter
May 15, 2025

UC San Diego

ATTRIBUTION

UC San Diego

GETTING STARTED WITH LAB 7

- Review the TA discussion section on Go's HTTP server support
  - Also the Go book's chapter 1
  - Also the Network programming with Go book, chapter 8, specifically starting around page 191 or so
- Review Templates which are used to generate the web page content
  - Network programming with Go book, chapter 9

POSSIBLE FIRST STEPS

- I'd focus on setting up your web server first
- Entry point "/":
  - Render the index template
- Entry point: "/upload":
  - Create a temp dir (os.MkdirTemp) to store the file
  - Redirect back to /
  - Once that's working, write them out to the ContentService

POSSIBLE FIRST STEPS

- Entry point “/content”:
  - Read from the ContentService the requested file and return to the user
  - Will need to set the Content-Type and Content-Length headers. OK to set former to “video/mp4” and latter to the file size, and only support mp4 videos
- At this point you’ve shown you can upload mp4 files, store them in your content service, and read them from your content service

NEXT STEPS

- When a video is uploaded
  - Should create the SQLite DB if needed
  - Insert metadata about that file into the database
- Entry point: “/” (part 2):
  - List the files/videos/times in your DB
- Entry point “/upload” (part 2)
  - Call out to the mp4 file you stored in the temp directory using ffmpeg, generating video segment file content and the mpd file
  - Store those in your content service
- Entry point “/videos”
  - Render the appropriate template

TUESDAY MAY 20TH

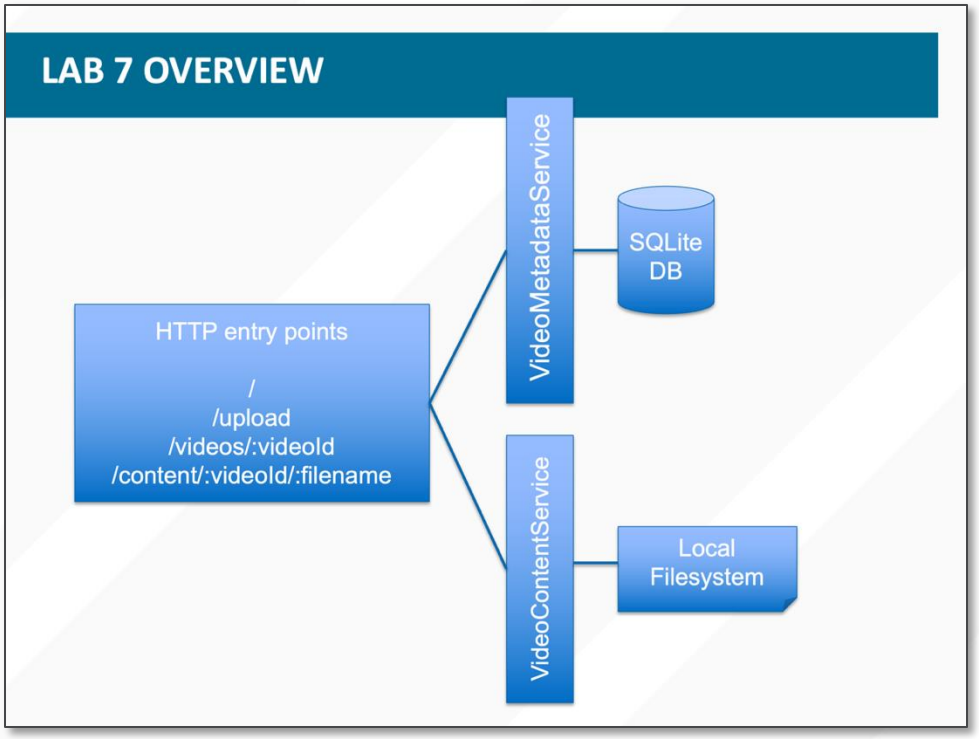
- The CDN lecture will be pre-recorded as a video
  - Possible guest appearance by Tippi and/or Cali??
- Instead, we'll have an in-person Lab 7 "clinic"
  - I and TAs in person to answer questions, talk over approaches
  - Peer-to-peer help and support
    - Can't share code about your lab submission
    - But totally fine to talk in general about solution approaches, testing strategies (no code), and questions / pointers / advice about Go as a language
  - Come and at least get 1.5 hours of your project done + live synchronous support
  - If you don't have a laptop, can print out code and talk it over with teaching staff, or talk over the spec itself with fellow students

SCALING OVERVIEW

UC San Diego

LOCATING ITEMS (AT SCALE) IS A HARD PROBLEM

- Consider our video server:



- Assume (for sake of an example) we want to keep the metadata and video content in memory to reduce latency

LET’S CHOOSE AN AWS INSTANCE TYPE

- General Purpose
- Compute Optimized
- Memory Optimized
- Accelerated Computing
- Storage Optimized
- Instance Features
- Measuring Instance Performance

LET'S PICK THE MEMORY-OPTIMIZED INSTANCE TYPE

Memory Optimized

Memory optimized instances are designed to deliver fast performance for workloads that process large data sets in memory.

R8g | R7g | R7i | R7iz | R7a | R6g | R6i | R6in | R6a | R5 | R5n | R5b | R5a | R4

U7i | High Memory (U-1) | X8g | X2gd | X2idn | X2iedn | X2iezn | X1 | X1e | z1d

Amazon EC2 R8g instances are powered by AWS Graviton4 processors. They deliver the best price performance in Amazon EC2 for memory-intensive workloads.

Features:

- Powered by custom-built AWS Graviton4 processors
- Larger instance sizes with up to 3x more vCPUs and memory than R7g instances
- Features the latest DDR5-5600 memory
- Optimized for Amazon EBS by default
- Instance storage offered via EBS or NVMe SSDs that are physically attached to the host server
- With R8gd instances, local NVMe-based SSDs are physically connected to the host server and provide block-level storage that is coupled to the lifetime of the instance
- Supports Elastic Fabric Adapter (EFA) on r8g.24xlarge, r8g.48xlarge, r8g.metal-24xl, r8g.metal-48xl, r8gd.24xlarge, r8gd.48xlarge, r8gd.metal-24xl, and r8gd.metal-48xl
- Powered by the AWS Nitro System, a combination of dedicated hardware and lightweight hypervisor

MEMORY INSTANCE TYPES

Instance size	vCPU	Memory (GiB)	Instance storage (GB)	Network bandwidth (Gbps)	EBS bandwidth (Gbps)
r8g.medium	1	8	EBS-only	Up to 12.5	Up to 10
r8g.large	2	16	EBS-only	Up to 12.5	Up to 10
r8g.xlarge	4	32	EBS-only	Up to 12.5	Up to 10
r8g.2xlarge	8	64	EBS-only	Up to 15	Up to 10
r8g.4xlarge	16	128	EBS-only	Up to 15	Up to 10
r8g.8xlarge	32	256	EBS-only	15	10
r8g.12xlarge	48	384	EBS-only	22.5	15
r8g.16xlarge	64	512	EBS-only	30	20
r8g.24xlarge	96	768	EBS-only	40	30
r8g.48xlarge	192	1,536	EBS-only	50	40

Instance name ▲	On-Demand hourly rate ▽	vCPU ▽	Memory ▽	Storage ▽	Network performance ▽
r8g.48xlarge	$11.31072	192	1536 GiB	EBS Only	50 Gigabit

HOW MANY VIDEOS CAN FIT INTO R8G.48XLARGE?

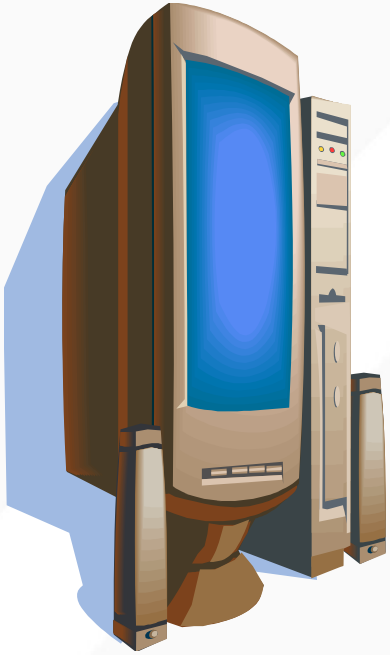
- 1536 GB of RAM
- Storage requirements for reach video?
  - Metadata + Video contents
  - Rough estimate (per hour):
    - **720p (HD):** 800MB - 900MB
    - **1080p (Full HD):** 1.2GB - 1.4GB
    - **4K:** 20GB - 22GB
    - **4K (Ultra HD):** 7GB - 7.2 GB
  - Assuming 1080p (HD), 1.2 GB/hour with 1536 GB of memory possible, then 1,280 hours of video

** Discounting metadata, which is negligible in this setting*

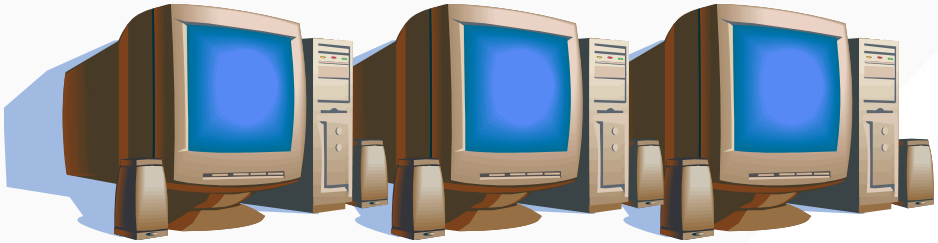
BUT WHAT IF YOU NEED MORE SPACE?

- What if you have more than 1,280 hours of video you want to serve??
- You need *scale*

SCALING



Vertical Scaling
(bigger machines)



Horizontal Scaling
(more machines)

VERTICAL SCALING

- Get a machine with more RAM, more storage, a faster CPU, more CPUs, …
- Advantages:
  - Simple: Single machine abstraction
  - Simple: Only one IP address/hostname to consult
- Disadvantages:
  - Machines only get so big (have so much ram, etc)
  - What if the machine fails?

HORIZONTAL SCALING

- Form a *cluster* of 10, 100, 1000… servers that work together
- Advantages:
  - No one machine has to be very expensive/fancy
  - A failure of one machine doesn't result in everything being lost (maybe, depending…)
- Disadvantages:
  - How to find the data you're looking for??
  - Performance is hard to reason about
    - Recall the Tail at Scale discussion

HORIZONTAL SCALING ISSUES

- Probability of any failure in given period = $1-(1-p)^n$
  - p = probability a machine fails in given period
  - n = number of machines

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

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

THE LOCATION PROBLEM

- Imagine we're looking for `chunk-1-00282.m4s`
- Given a cluster C consisting of *N* servers, how do we locate the specific server C_i responsible for that data item?
- E.g. For a logical storage *service* spread across N machines, which machine has `chunk-1-00282.m4s`? What about `chunk-1-00283.m4s?`

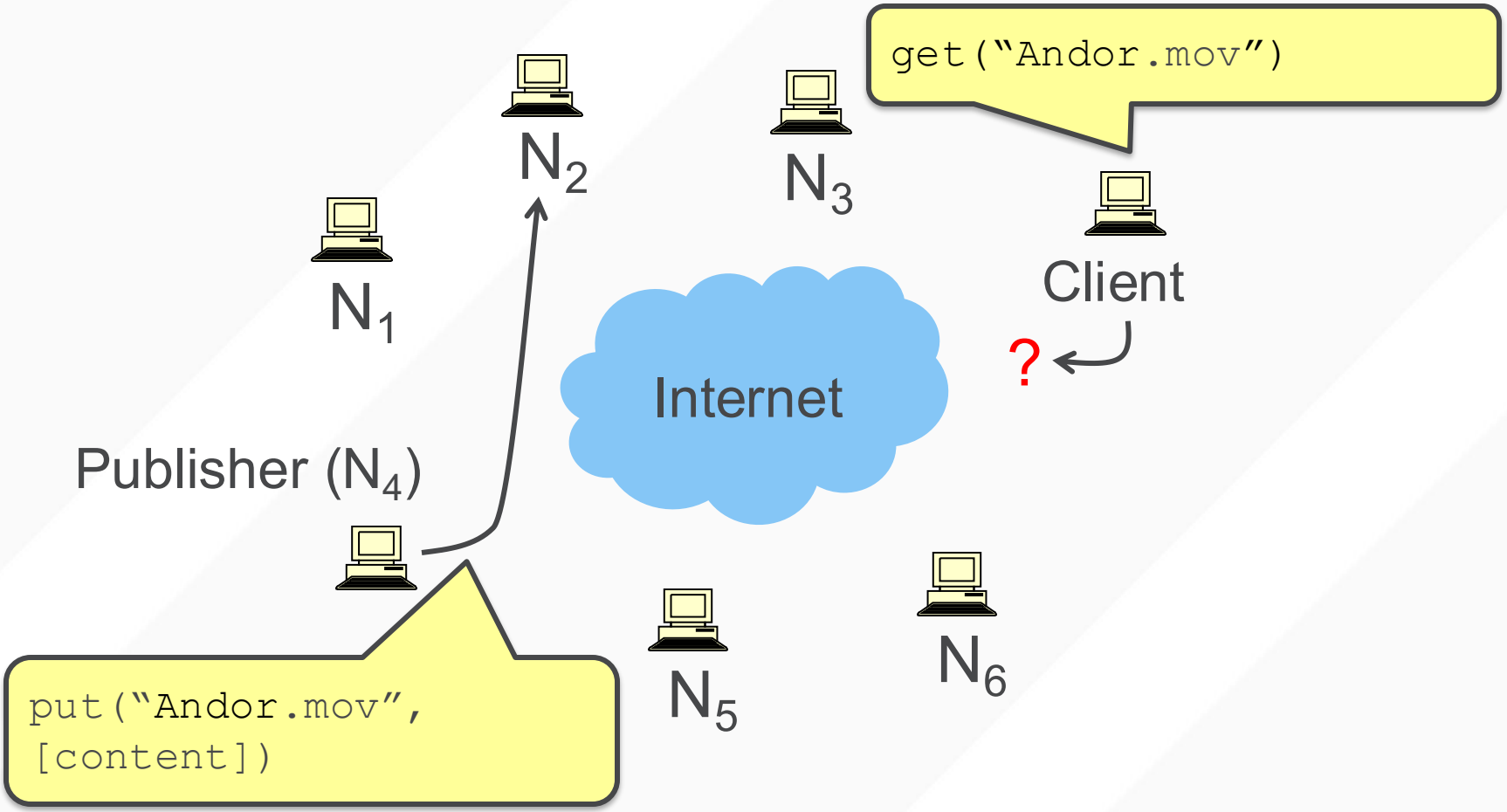
PEER TO PEER NETWORKS

- A distributed set of nodes sharing content, often based on “flat” names, is called a peer to peer network

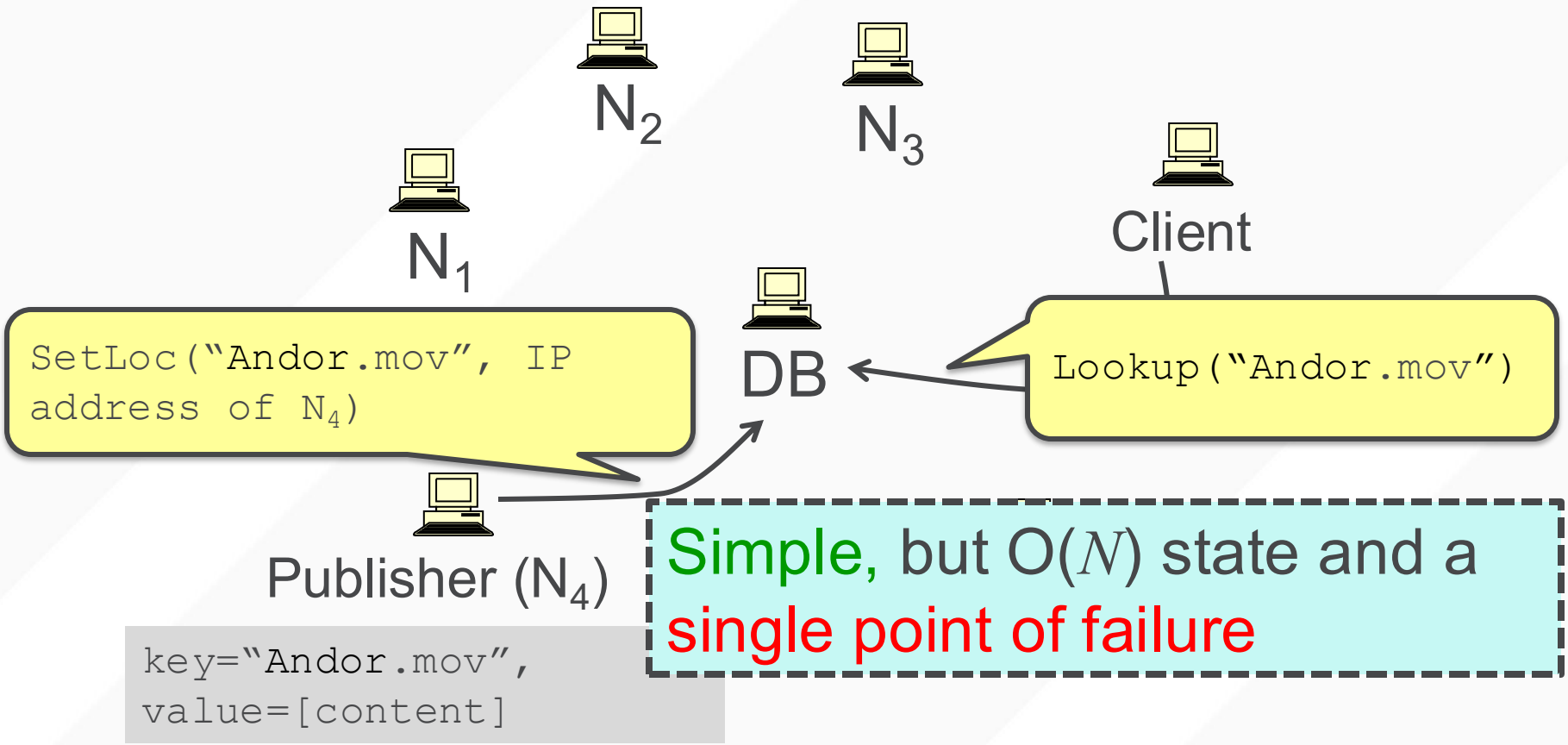
WHAT IS “FLAT” NAMING?

- The name doesn’t give you an indication of where the data is located
- Flat:
  - MAC address: 00:50:56:a3:0d:2a
- Vs hierarchical:
  - IP address: 206.109.2.12/24
  - DNS name: starbase.neosoft.com

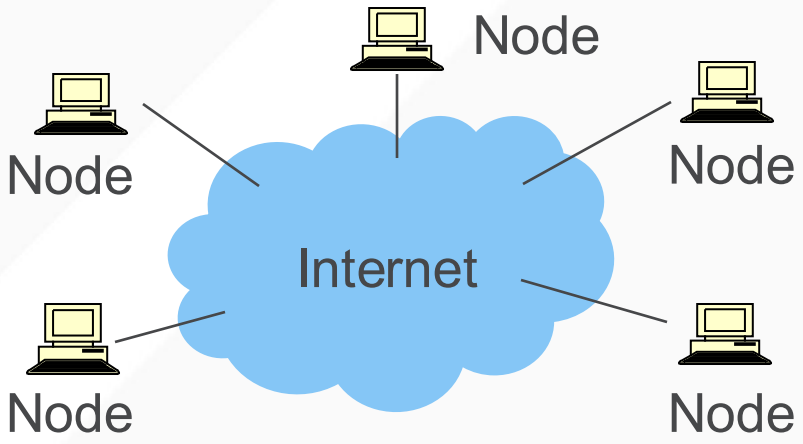
FLAT NAME LOOKUP PROBLEM



CENTRALIZED LOOKUP (NAPSTER)

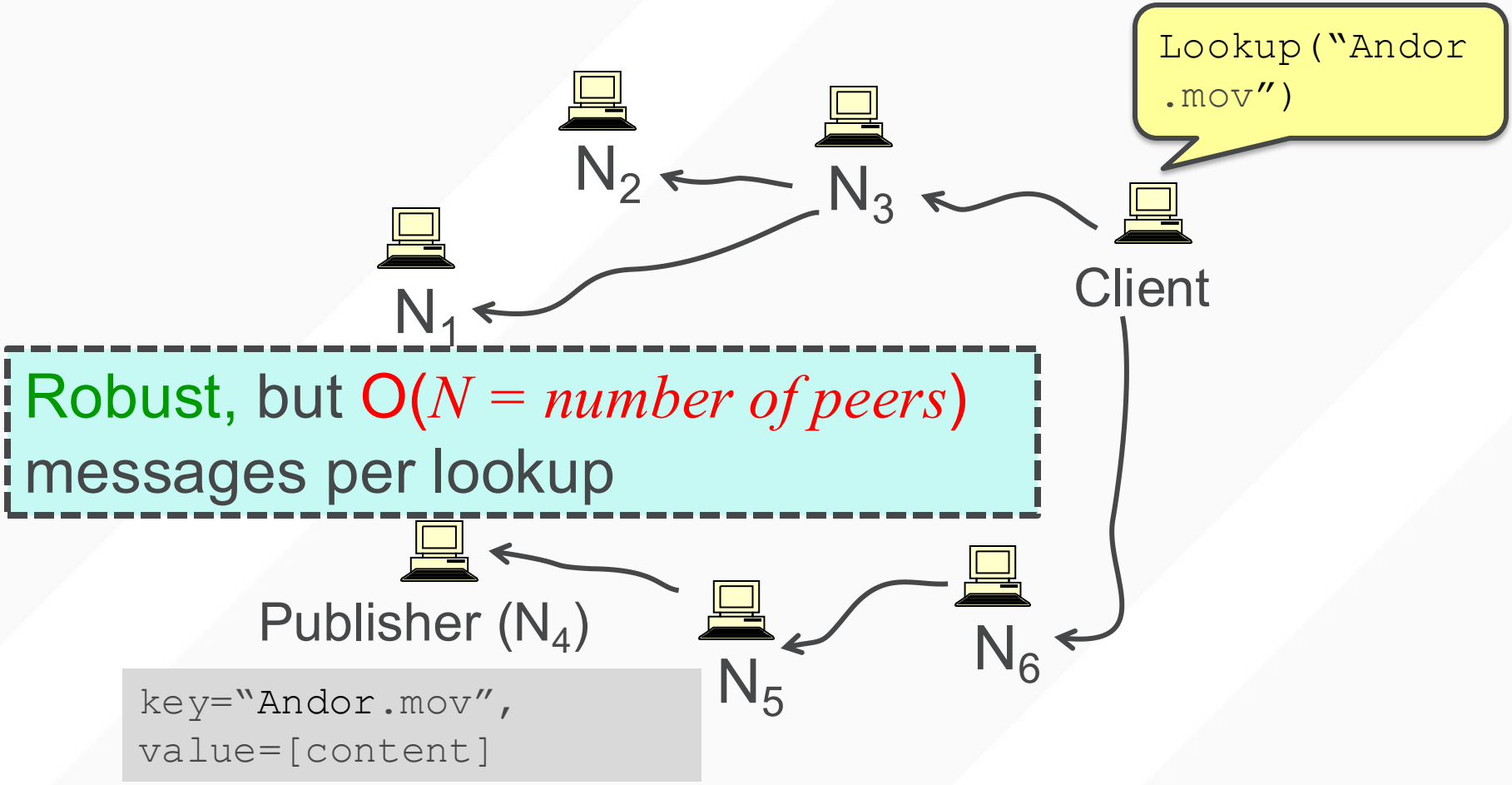


PEER-TO-PEER (P2P) NETWORKS

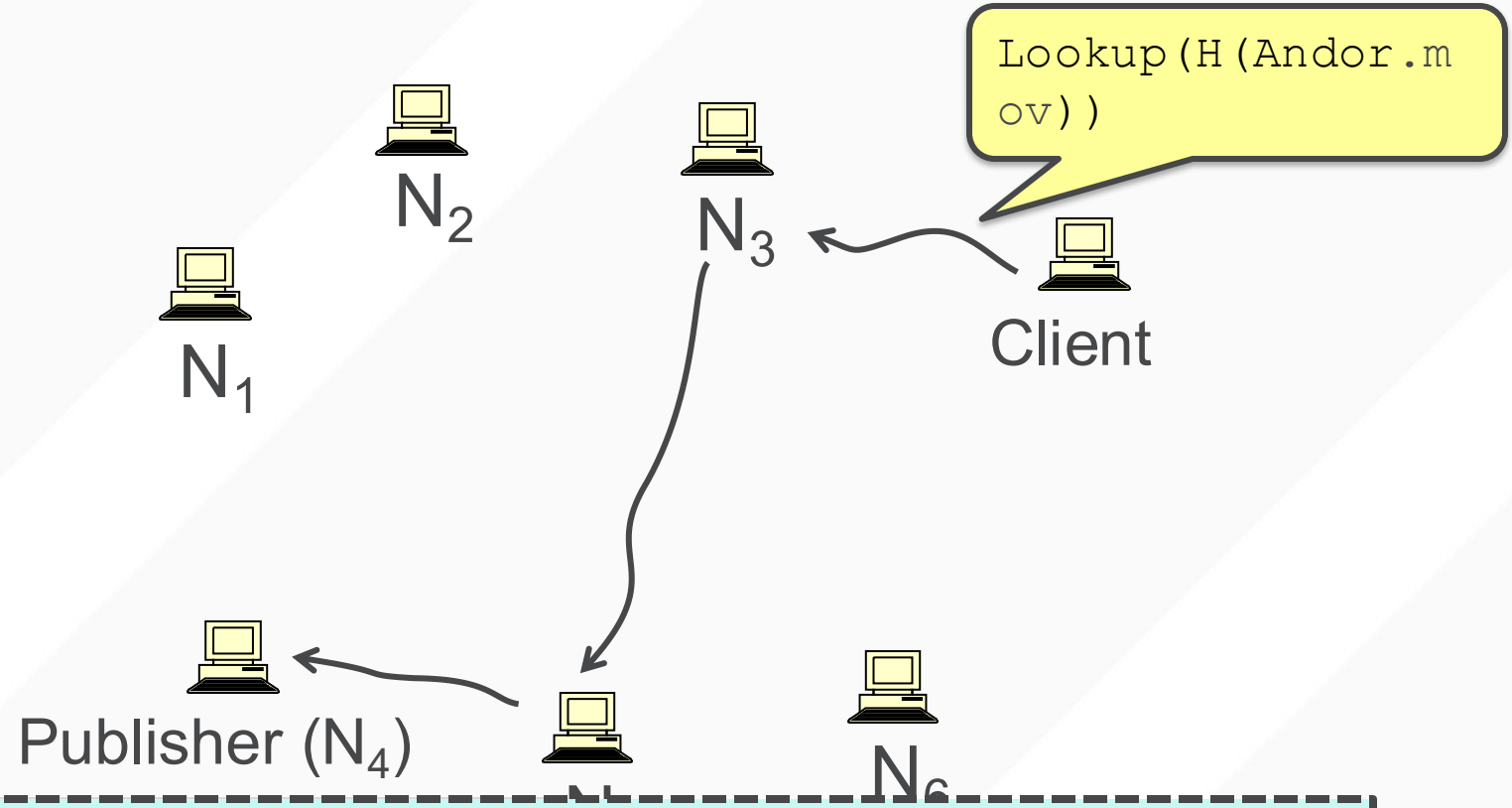


- A **distributed** system architecture:
  - **No centralized control**
  - Nodes are **roughly symmetric** in function
- **Large** number of **unreliable** nodes (could be reliable too)

FLOODED QUERIES (ORIGINAL GNUTELLA)



ROUTED DHT QUERIES (DYNAMODB)



Can we make it robust, reasonable state, reasonable number of hops?

SYSTEMATIC FLAT NAME LOOKUPS VIA DHTS

- Local hash table:

```
key = Hash(name)
put(key, value)
get(key) → value
```

- **Service:** Constant-time insertion and lookup

How can I do (roughly) this across millions of hosts on the Internet or within a giant datacenter application?
Distributed Hash Table (DHT)

WHAT IS A DHT (AND WHY)?

- Distributed Hash Table:

```
key = hash(data)
lookup(key) → IP addr
send-RPC(IP address, put, key, data)
send-RPC(IP address, get, key) → data
```

- **Partitioning data** in truly **large-scale distributed systems**
  - Tuples in a global database engine
  - Video files in TritonTube
  - Files in a P2P file-sharing system

SUMMARY OF IDEA

- We're going to rely on hashing to map keys to servers
  - That way, to find a key (e.g. filename), just hash the name you're looking for and consult just that server!
  - Cool… let's see how that works in practice…

STRAWMAN: MODULO HASHING (E.G. HASHMAP)

- 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?

PROBLEMS WITH MODULO HASHING

h(x) = x + 1 (mod 4)

Add one machine: h(x) = x + 1 (mod 5)

All entries get remapped to new nodes!

→ Need to move objects over the network

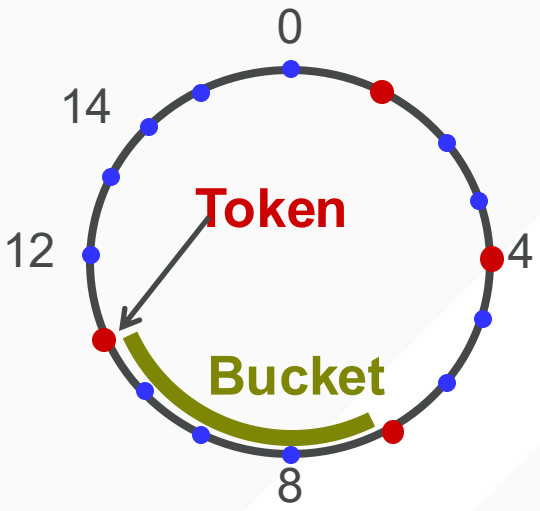
We need a different hashing approach that doesn't change everything when a server comes or goes…

CONSISTENT HASHING [KARGER '97]

- **Key identifier** = hash(key)
- **Node identifier** = hash(server's IP address) or hash(server's hostname) or hash(server's identity)
- Same hash function maps two *different* types of data to the same ID space!

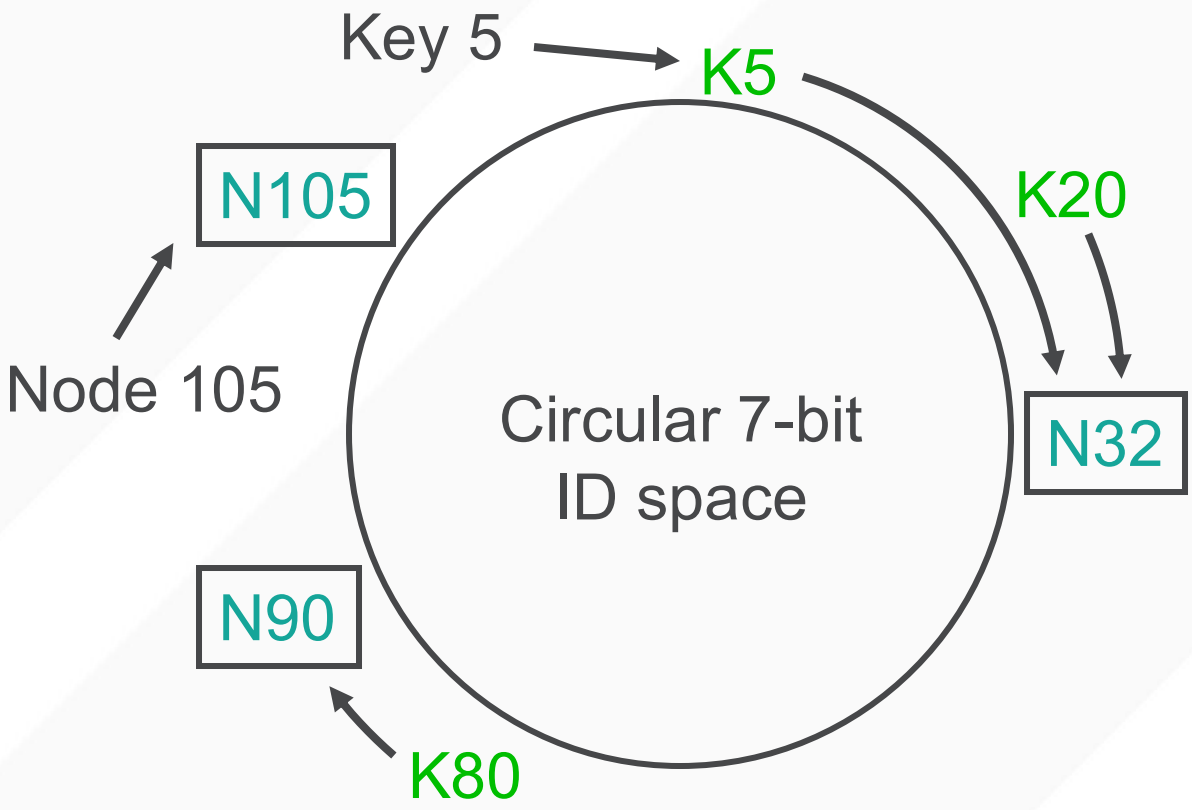
CONSISTENT HASHING

- Assign *n tokens* to random points on mod 2^k circle; hash key size = *k*
- Hash object to random circle position
- Put object in closest clockwise bucket
  - *successor* (key) → bucket



- Desired features –
  - Balance: No bucket has "too many" objects
  - Smoothness: Addition/removal of token minimizes object movements for other buckets

CONSISTENT HASHING [KARGER '97]



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

CONSISTENT HASHING AND LOAD BALANCING

- Each node owns **1/nth** of the ID space in expectation
  - Says nothing of request load per bucket

- If a node fails, its successor takes over bucket
  - **Smoothness goal ✓:** Only localized shift, not O(n)

  - But now successor owns **two** buckets: **2/nth of key space**
    - The failure has **upset the load balance**

VIRTUAL NODES

- **Idea:** Each **physical node** now maintains **v > 1 tokens**
  - Each token corresponds to a ***virtual node***
- Each virtual node owns an expected **$1/(vn)^{th}$** of ID space
- **Upon a physical node's failure, v** successors take over, each now stores **$(v+1)/v \times 1/n^{th}$** of ID space
- **Result: Better load balance** with larger v

UC San Diego