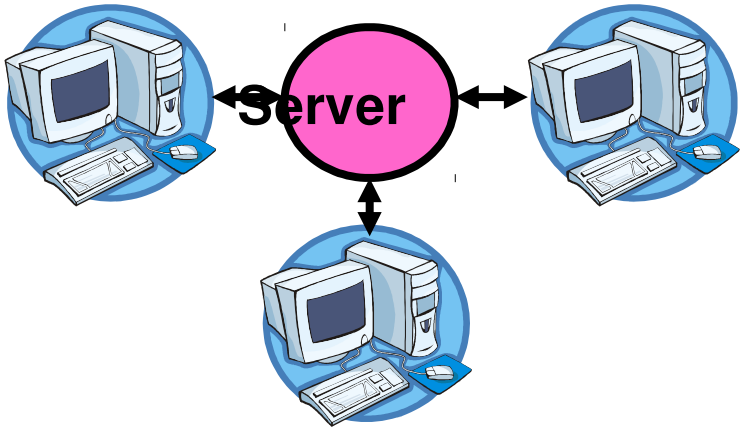
CS162:
Operating Systems and Systems Programming

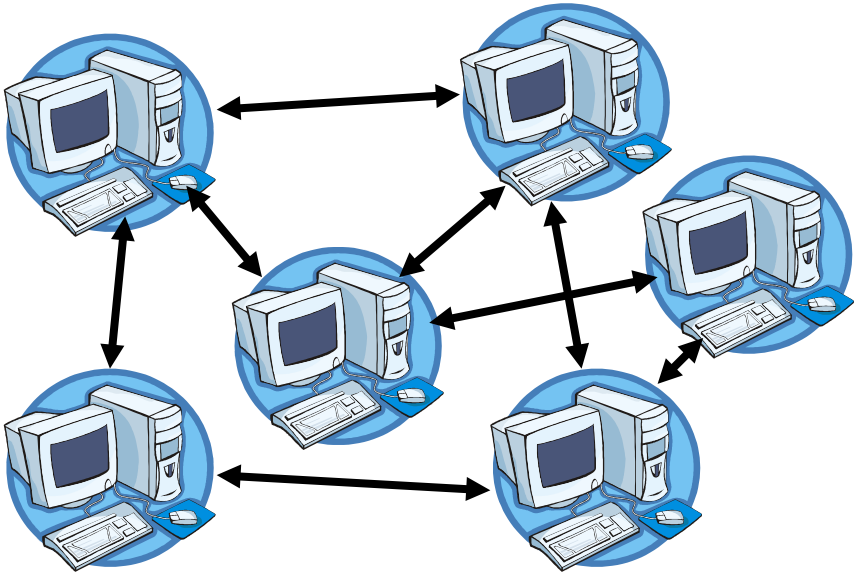
Lecture 17:
Distributed Systems Intro

16 July 2015
Charles Reiss
https://cs162.eecs.berkeley.edu/

Centralized versus Distributed



Client/Server Model



Peer-to-Peer Model

Centralized system: System where major functions performed on one *physical* computer

Distributed system: Physically separate computers working together to perform a single task

Parallel versus Distributed

Distributed – different machines responsible for different parts of task

- Usually no centralized state
- Usually about different responsibilities or redundancy

Parallel – different parts of same task performed on different machines

- Usually about performance

Distributed: Why

Simple, **cheaper** components

Easy to **add** to **incrementally**

Let **multiple users cooperate** (maybe)

- Physical components owned by different users
- Enable **collaboration** between diverse users

Distributed: Promise

Availability

- One machine goes down, *system* up

Durability

- One machine loses data, *system* does not

Security

- Divide security problem into simpler pieces?

Distributed: Reality (sometimes)

Availability

- One machine takes them all down

Durability

- Any machine can lose your data

Security

- More places to break in

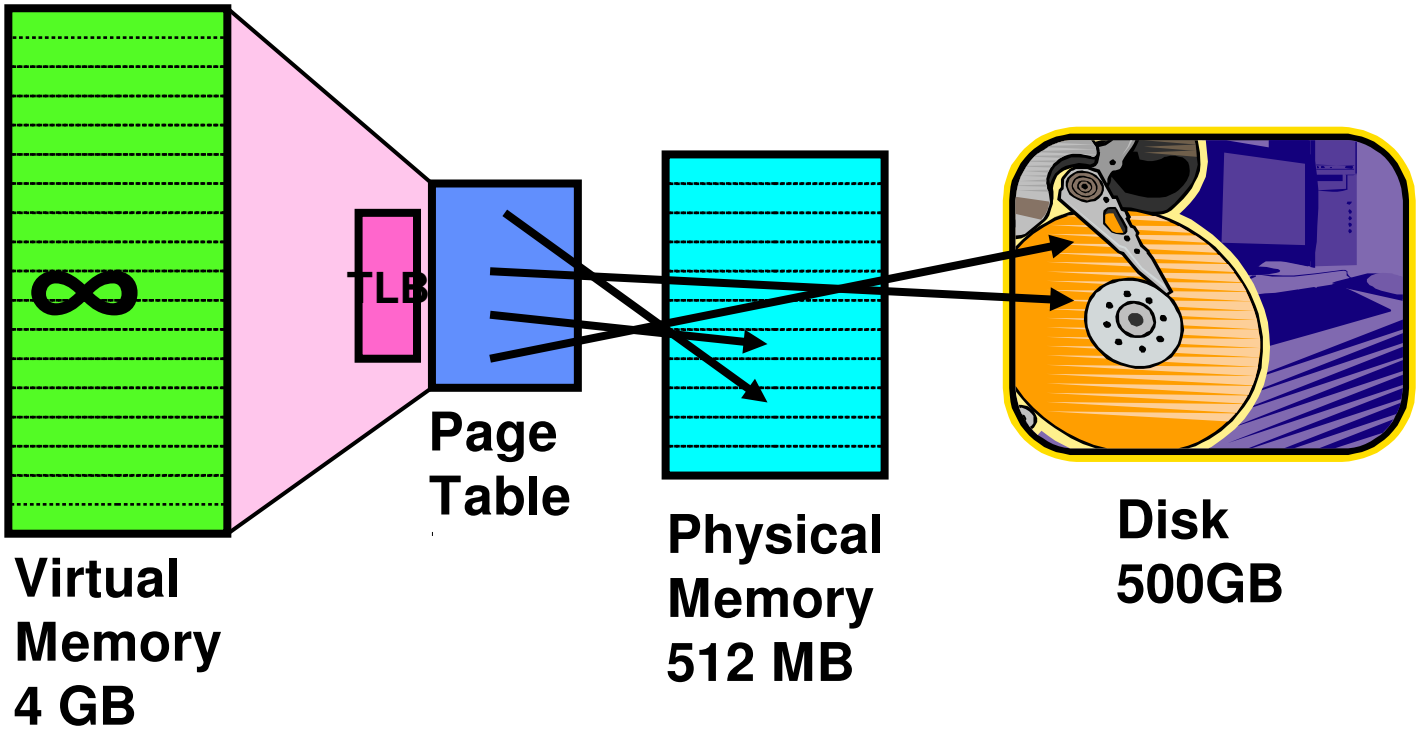
Distributed: Goal

Transparency: Hide distributedness, make system simpler

Kinds:

- *Location* – location of resources invisible/irrelevant
- *Migration* – location of resources changes invisibly
- *Replication* – invisible extra copies of resources (for reliability, performance)
- *Parallelism* – jobs split into pieces, look like on serial task
- *Fault tolerance* – components fail without users knowing

Recall: Illusion of "infinite" memory



How? **Transparent layer of indirection**

- the page table + page fault handlers

The Coordination Problem

Components communicate with network:

- Lets you send **messages** between machines

Need to use messages to **agree on state**

- Problem **does not exist** in centralized system

Protocols

Protocol is **an agreement on how to communicate**

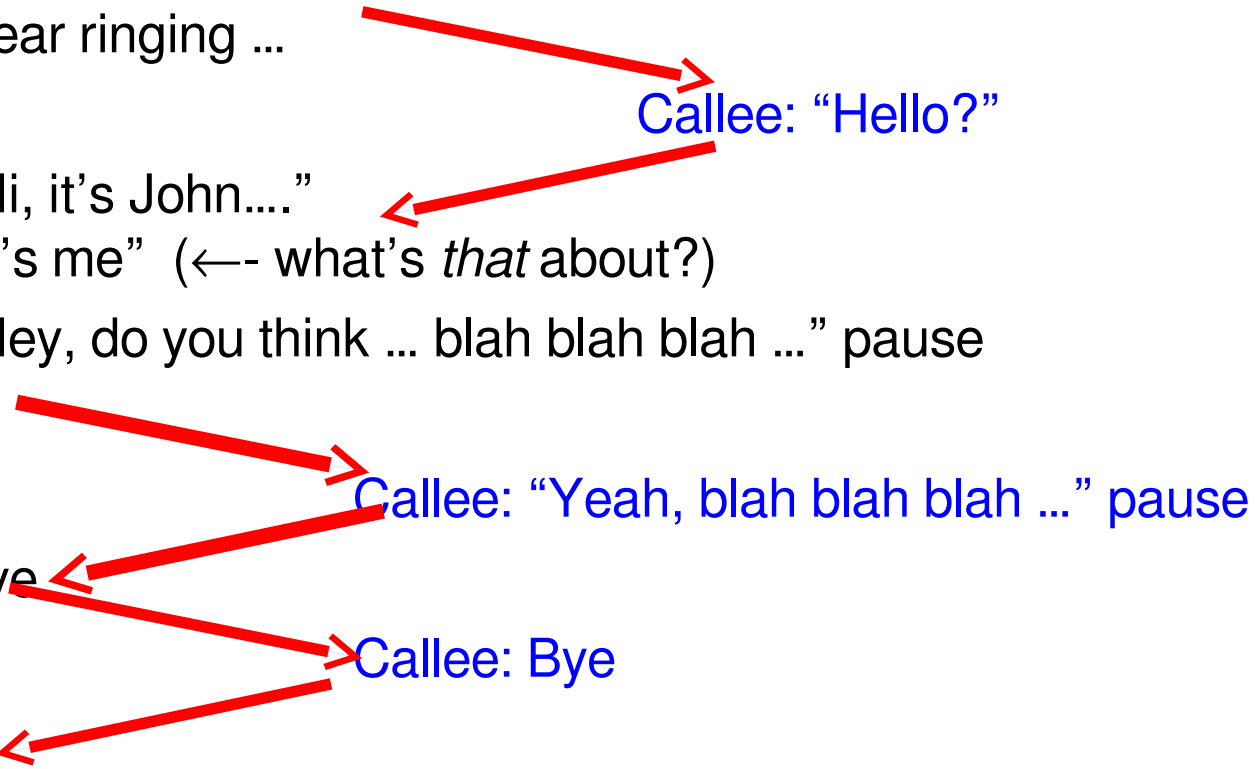
- *Syntax*: "structure" – format, order of messages
- *Semantics:* "meaning" – actions taken when transmitting, receiving, or when timer expires

Formal description: ***state machine***

A distributed system is embodied by a protocol

Human Protocols: Telephone

1. (Pick up / open up the phone)
2. Listen for a dial tone / see that you have service
3. Dial
4. Should hear ringing …
5. Callee: “Hello?”
6. Caller: “Hi, it’s John….”
   Or: “Hi, it’s me” (←- what’s *that* about?)
7. Caller: “Hey, do you think … blah blah blah …” pause
8. Callee: “Yeah, blah blah blah …” pause
9. Caller: Bye
10. Callee: Bye
11. Hang up



Protocols With Humans: Asking a Question

1. Raise your hand

2. Wait to be called on

or

1. Wait for speaker to pause.

2. Speak.

Models for Organizing Distributed Systems

Client/server

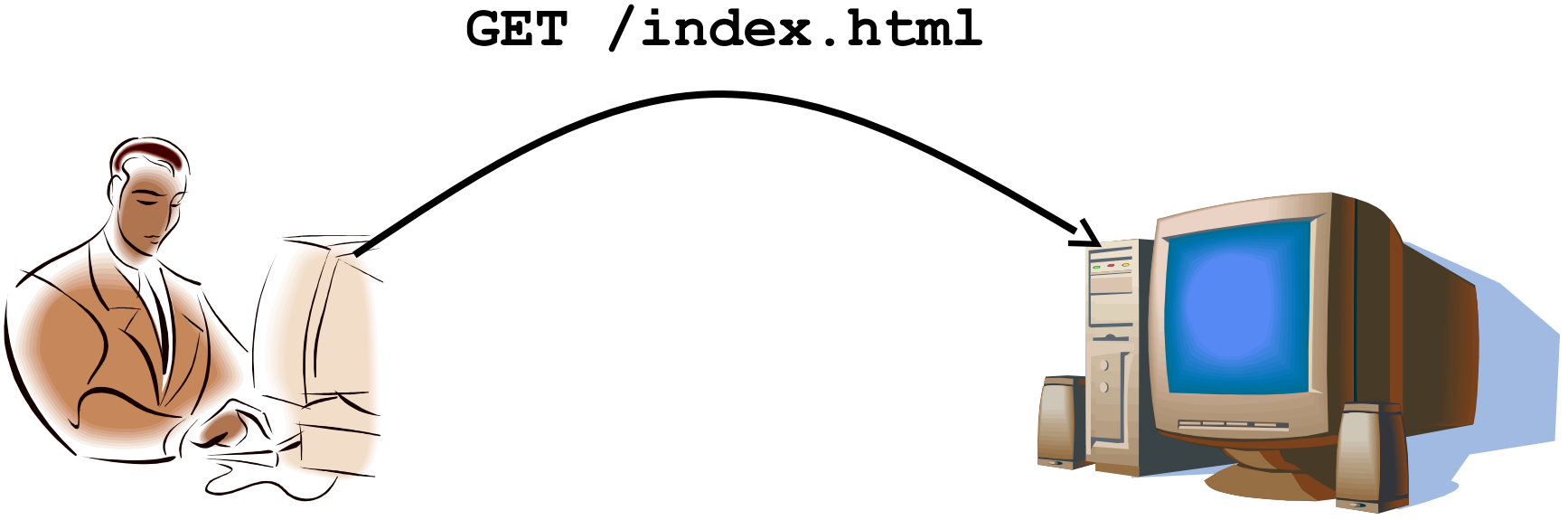
Peer-to-peer

???

Clients and Servers

Client program

- Running on end host
- Requests service
- E.g., Web browser



Client-Server Communication

Client “sometimes on”

- Initiates a request to the server when interested
- E.g., Web browser on your laptop or cell phone
- Doesn’t communicate directly with other clients
- Needs to know the server’s address

Server is “always on”

- Services requests from many client hosts
- E.g., Web server for the www.cnn.com Web site
- Doesn’t initiate contact with the clients
- Needs a fixed, well-known address

Peer-to-Peer Communication

No always-on server at the center of it all

- Hosts can come and go, and change addresses
- Hosts may have a different address each time

Example: peer-to-peer file sharing (e.g., BitTorrent)

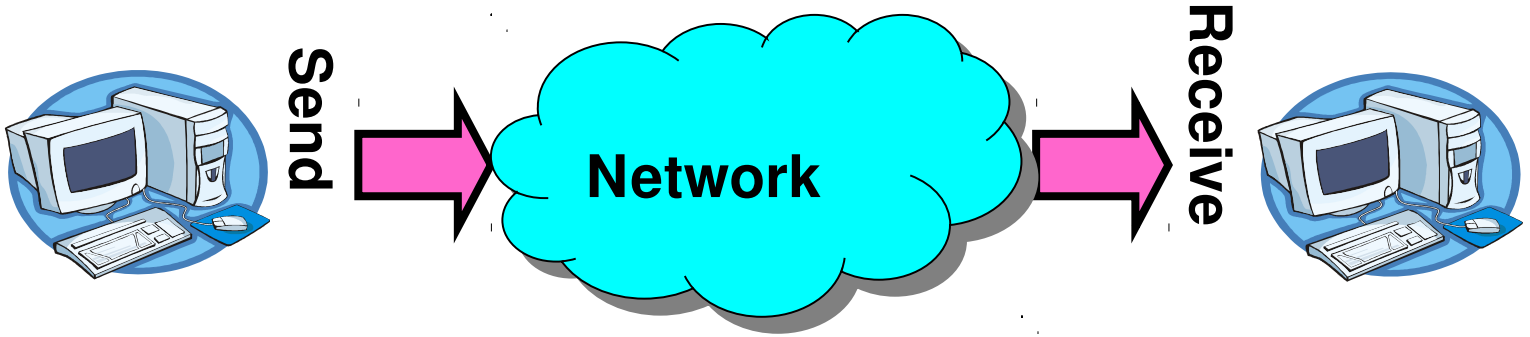
- Any host can request files, send files, query to find where a file is located, respond to queries, and forward queries
- Scalability by harnessing millions of peers
- Each peer acting as both a client and server

The Mailbox Abstraction (1)

Multiple machines, no shared memory

One Abstraction: send/receive messages

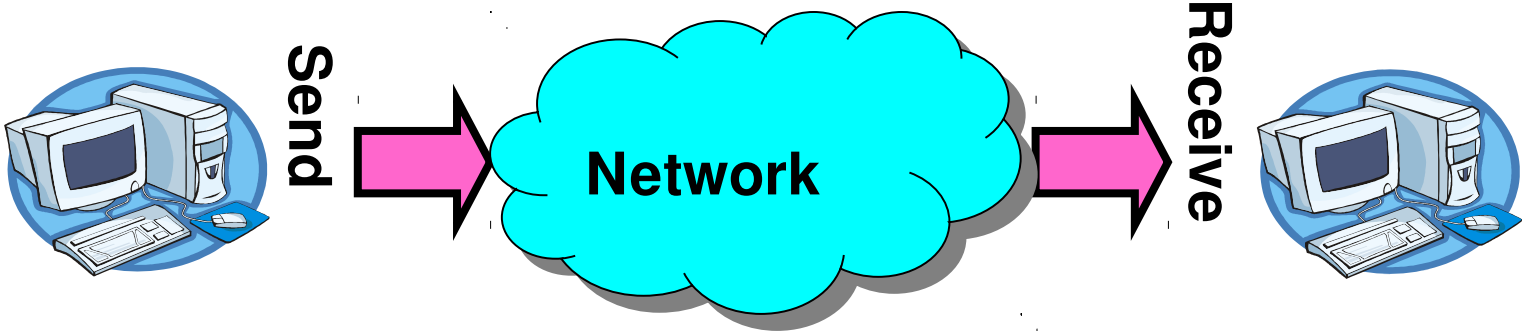
- Already atomic: no receiver gets portion of a message and two receivers cannot get same message



The Mailbox Abstraction (2)

Interface:

- Mailbox (mbox): temporary holding area for messages
  - Includes both destination location and queue
- Send(message,mbox)
  - Send message to remote mailbox identified by mbox
- Receive(buffer,mbox)
  - Wait until mbox has message, copy into buffer, and return
  - If threads sleeping on this mbox, wake up one of them



Using Messages: Send/Receive behavior

When should send(message,mbox) return?

- Wait for sender to get it?

Mailbox provides 1-way communication

- A $\leftrightarrow$ buffer $\leftrightarrow$ T2
- Very similar to producer/consumer

Recall: Producer/Consumer

Shared buffer (queue) – fixed size

- Producer inserts items
- Consumer removes items

Producer/consumer don't need to work in lockstep

Example: C compiler

- preprocessor → compiler → assembler → linker

Messaging for Producer-Consumer Style

Using send/receive for producer-consumer style:

Producer:

```
int msg1[1000];
while(1) {
  prepare message;
  send(msg1,mbox);
}
```

Consumer:

```
int buffer[1000];
while(1) {
  receive(buffer,mbox);
  process message;
}
```

No need for producer/consumer to keep track of space in mailbox: handled by send/receive

- Block when needed

Messaging for Request/Response communication

What about two-way communication?

- Request/Response
  - Read a file stored on a remote machine
  - Request a web page from a remote web server
- Also called: client-server
  - Client <-> requester, Server <-> responder
  - Server provides “service” (file storage) to the client

Messaging for Request/Response communication

Example: File service

Client: (requesting the file)

```
char response[1000];

send("read rutabaga", server_mbox);
receive(response, client_mbox);
```

Request File

Get Response

Server: (responding with the file)

```
char command[1000], answer[1000];

receive(command, server_mbox);
decode command;
read file into answer;
send(answer, client_mbox);
```

Receive Request

Send Response

General's Paradox

Two generals, on separate mountains

- Can only communicate via messengers
- Messengers can be captured

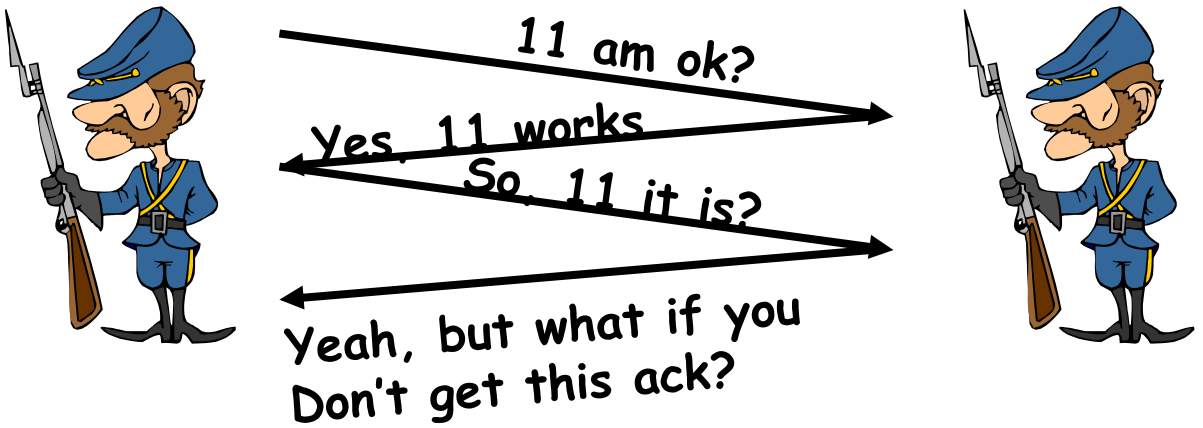
Problem: need to coordinate attack

- If they attack at different times, they all die
- If they attack at same time, they win

General's Paradox

Can messages over an unreliable network be used to guarantee two entities do something simultaneously?

- Remarkably, "no", ***even if all messages get through***



No way to be sure last message gets through!

Logistics

Project 2 Initial Design – **TODAY AT 8PM**

HW 2 Out – Malloc

Break

Distributed Storage

Get data from server elsewhere

Example: your home directories on your course accounts

- Same files on multiple machines

Let's start simple

Key/Value Storage

Key Value Storage

Simple interface

put(key, value); // insert/write “value” associated with “key”

value = get(key); // get/read data associated with “key”

Other operations??

Why Key-Value Storage

Easy to scale:

- handle huge volumes of data, e.g., petabytes
- uniform items – distribute easily/roughly equally across many machines

Simpler consistency properties:

- no relations, etc.

Used sometimes as a simpler but more scalable "database"

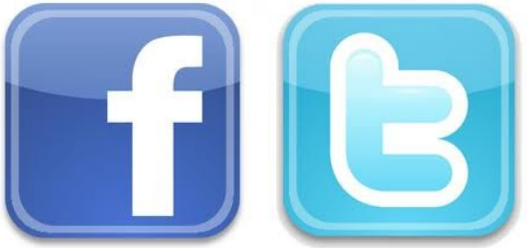
- Can be a building block for fully capable databases

Key Values: Examples

Amazon:

- Key: customerID
- Value: customer profile (e.g., buying history, credit card, ..)

Facebook, Twitter:



- Key: UserID
- Value: user profile (e.g., posting history, photos, friends, …)

iCloud/iTunes:

- Key: Movie/song name
- Value: Movie, Song

Key-Value Stores in the Wild

Amazon

- **DynamoDB**: internal key value store used to power Amazon.com (shopping cart)
- Simple Storage System (S3)

BigTable/HBase/Hypertable: distributed, scalable data storage

Cassandra: “distributed data management system” (developed by Facebook)

Memcached: in-memory key-value store for small chunks of arbitrary data (strings, objects)

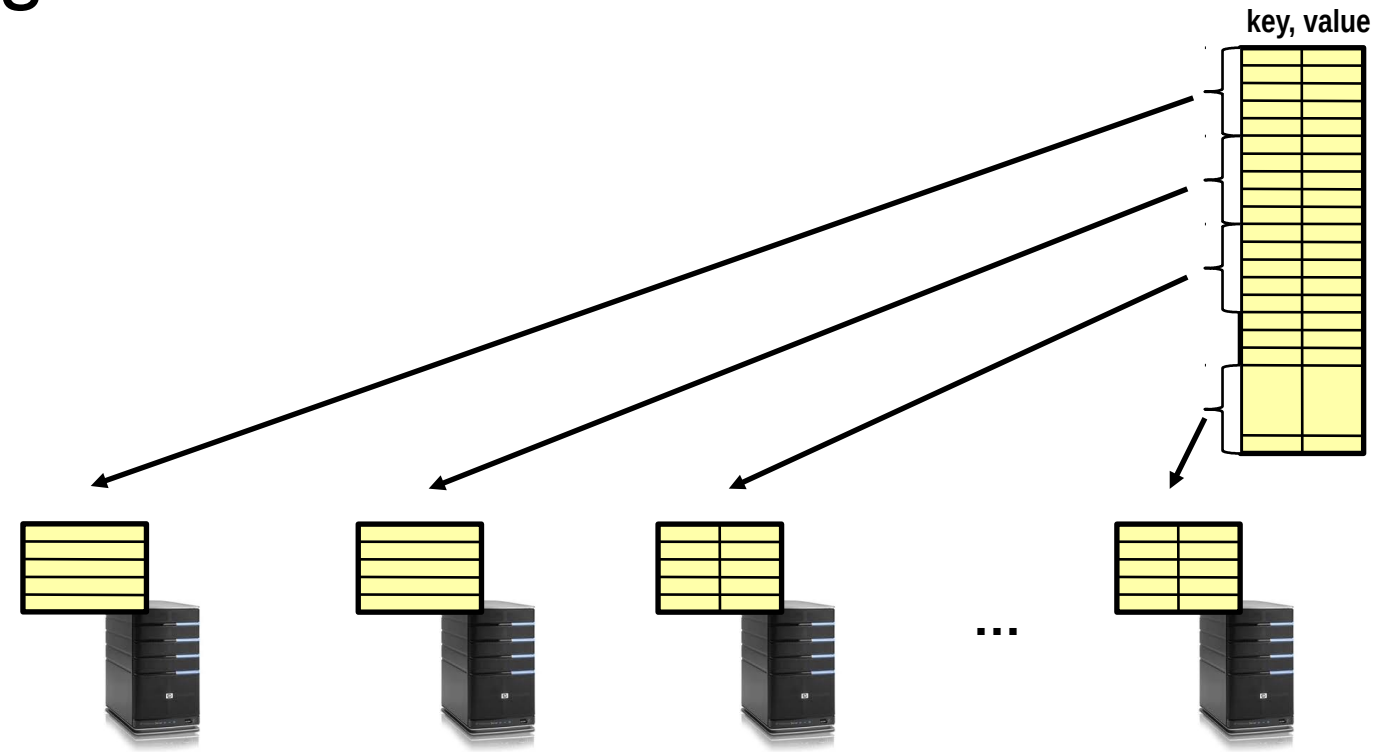
eDonkey/eMule: peer-to-peer sharing system

...

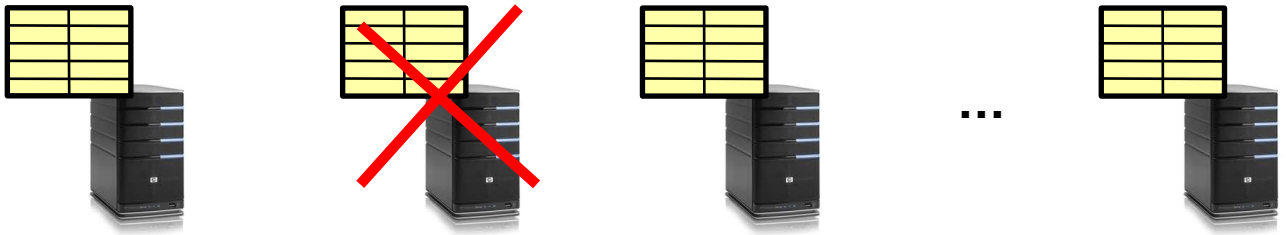
Key-Value Store

Also called Distributed Hash Tables (?) (DHT)

Main idea: partition set of key-values across many machines



Challenges



Fault Tolerance: handle machine failures without losing data and without degradation in performance

Scalability:

- Need to scale to thousands of machines
- Need to allow easy addition of new machines

Consistency: maintain data consistency in face of node failures and message losses

Heterogeneity (if deployed as peer-to-peer systems):

- Latency: 1ms to 1000ms
- Bandwidth: 32Kb/s to 100Mb/s

Key Questions

put(key, value): where do you store a new (key, value) tuple?

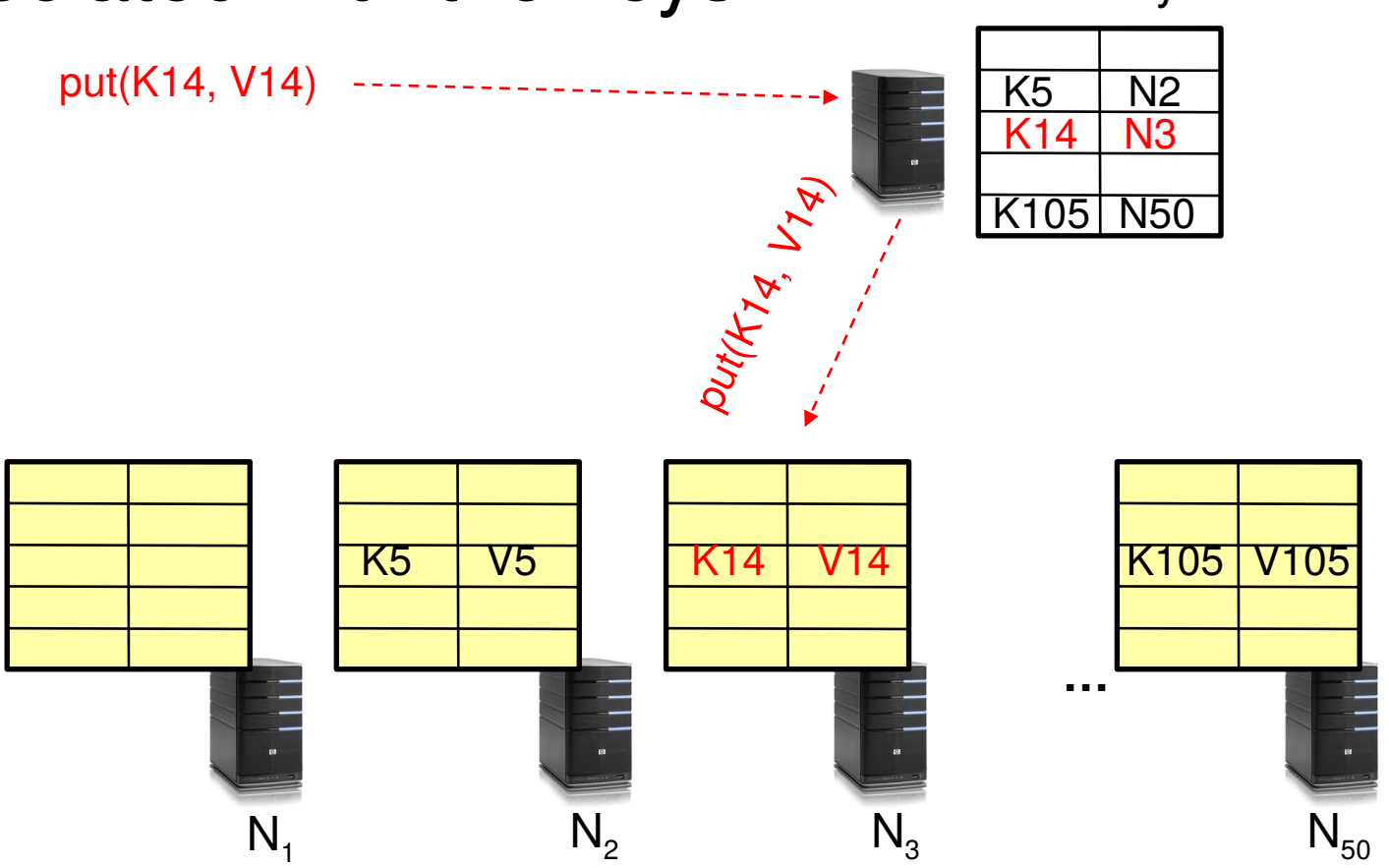
get(key): where is the value associated with a given “key” stored?

And, do the above while providing

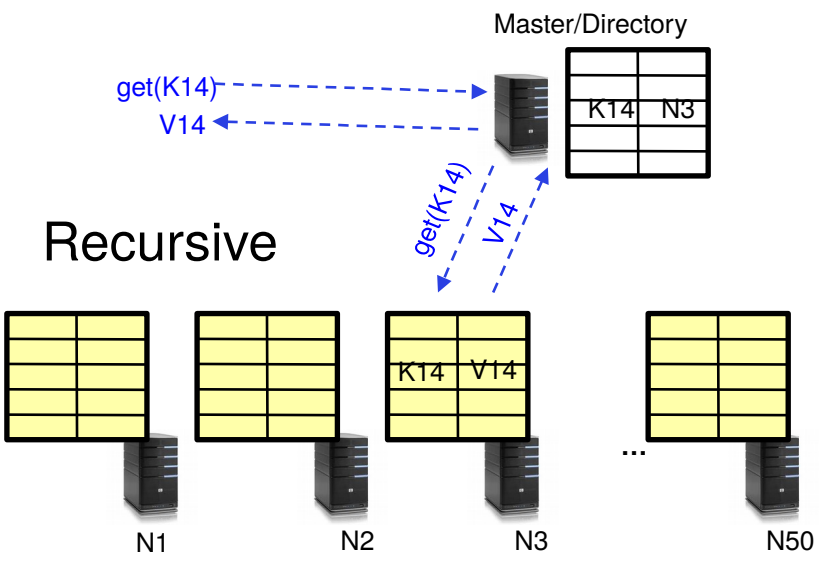
- Fault Tolerance
- Scalability
- Consistency

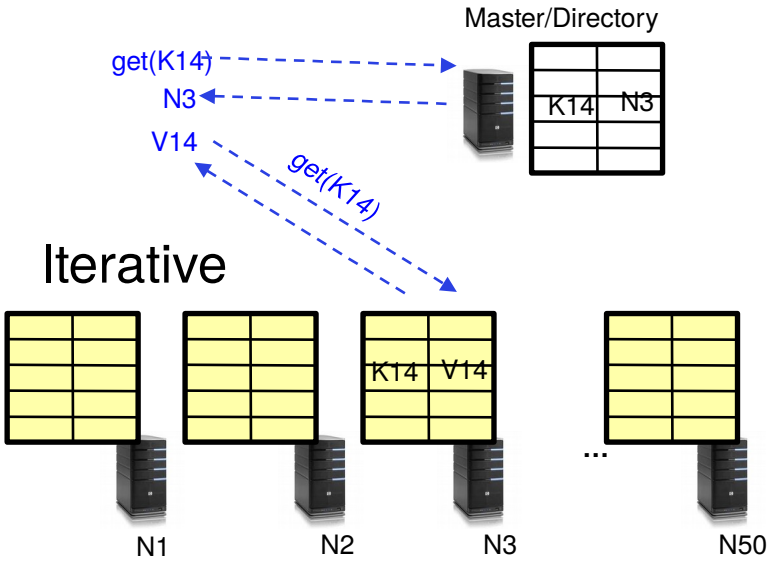
Directory-Based Architecture

Have a node maintain the mapping between keys and the machines (nodes) that store the values associated with the keys



Discussion: Iterative vs. Recursive Query





Recursive Query: Directory delegates

Iterative Query: Client delegates

Discussion: Iterative vs. Recursive Query

Recursive Query:

- Advantages:
  - Faster, as typically master/directory closer to nodes
  - Easier to maintain consistency, as master/directory can serialize puts()/gets()
- Disadvantages: scalability bottleneck, as all “Values” go through master/directory

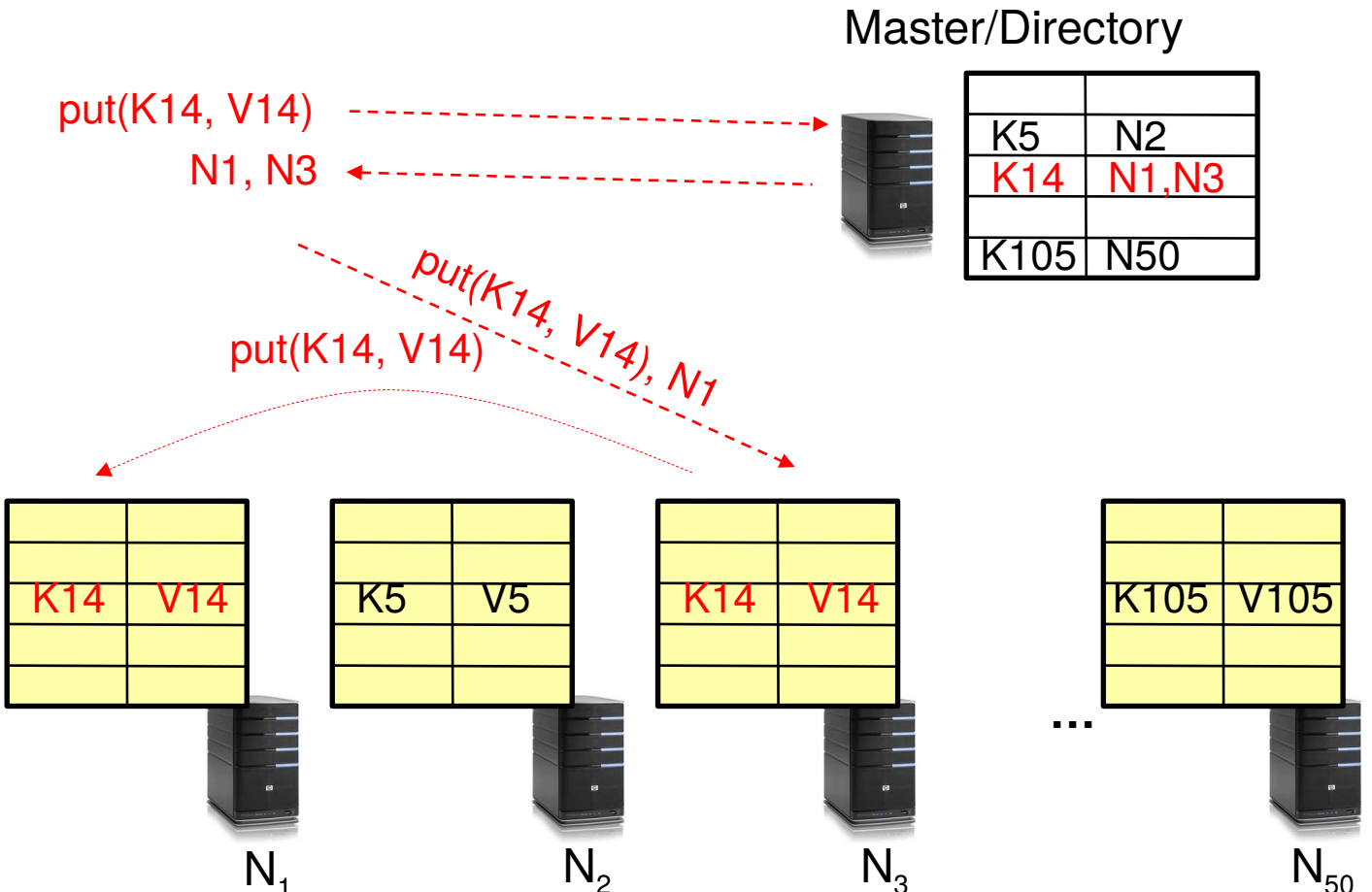
Iterative Query

- Advantages: more scalable
- Disadvantages: slower, harder to enforce data consistency

Fault Tolerance

Replicate value on several nodes

- Usually, place replicas on different racks in a datacenter to guard against rack failures



Scalability

Storage: use more nodes

Number of requests:

- Can serve requests from all nodes on which a value is stored in parallel
- Master can replicate a popular value on more nodes

Master/directory scalability:

- Replicate it
- Partition it, so different keys are served by different masters/directories
  - How do you partition?

Scalability: Load Balancing

Directory keeps track of the storage availability at each node

- Preferentially insert new values on nodes with more storage available

What happens when a new node is added?

- Cannot insert only new values on new node. Why?
- Move values from the heavy loaded nodes to the new node

What happens when a node fails?

- Need to replicate values from fail node to other nodes

Scaling Up Directory

Challenge:

- Directory contains a number of entries equal to number of (key, value) tuples in the system
- Can be tens or hundreds of billions of entries in the system!

Solution: **consistent hashing**

Associate to each node a unique id in an uni-dimensional space 0..2^m-1

- Partition this space across m machines
- Assume keys are in same uni-dimensional space
- Each (Key, Value) is stored at the node with the smallest ID larger than Key

Key to Node Mapping Example

m = 6 / ID space: 0..63

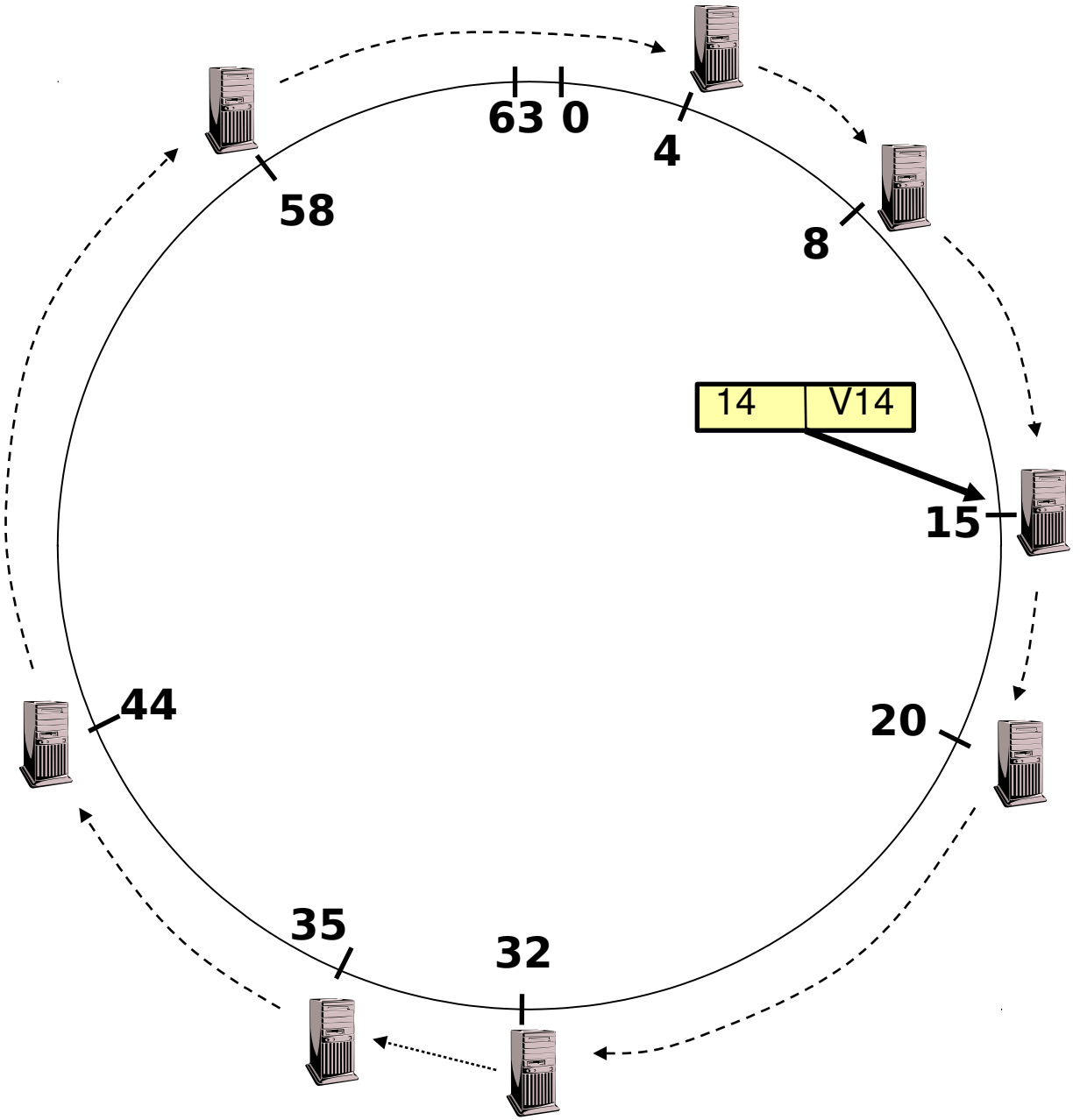
Node 8 maps keys [5,8]

Node 15 maps keys [9,15]

Node 20 maps keys [16, 20]

…

Node 4 maps keys [59, 4]



Lookup in Chord-like system (with Leaf Set)

Assign IDs to nodes

- Map hash values to node with closest ID

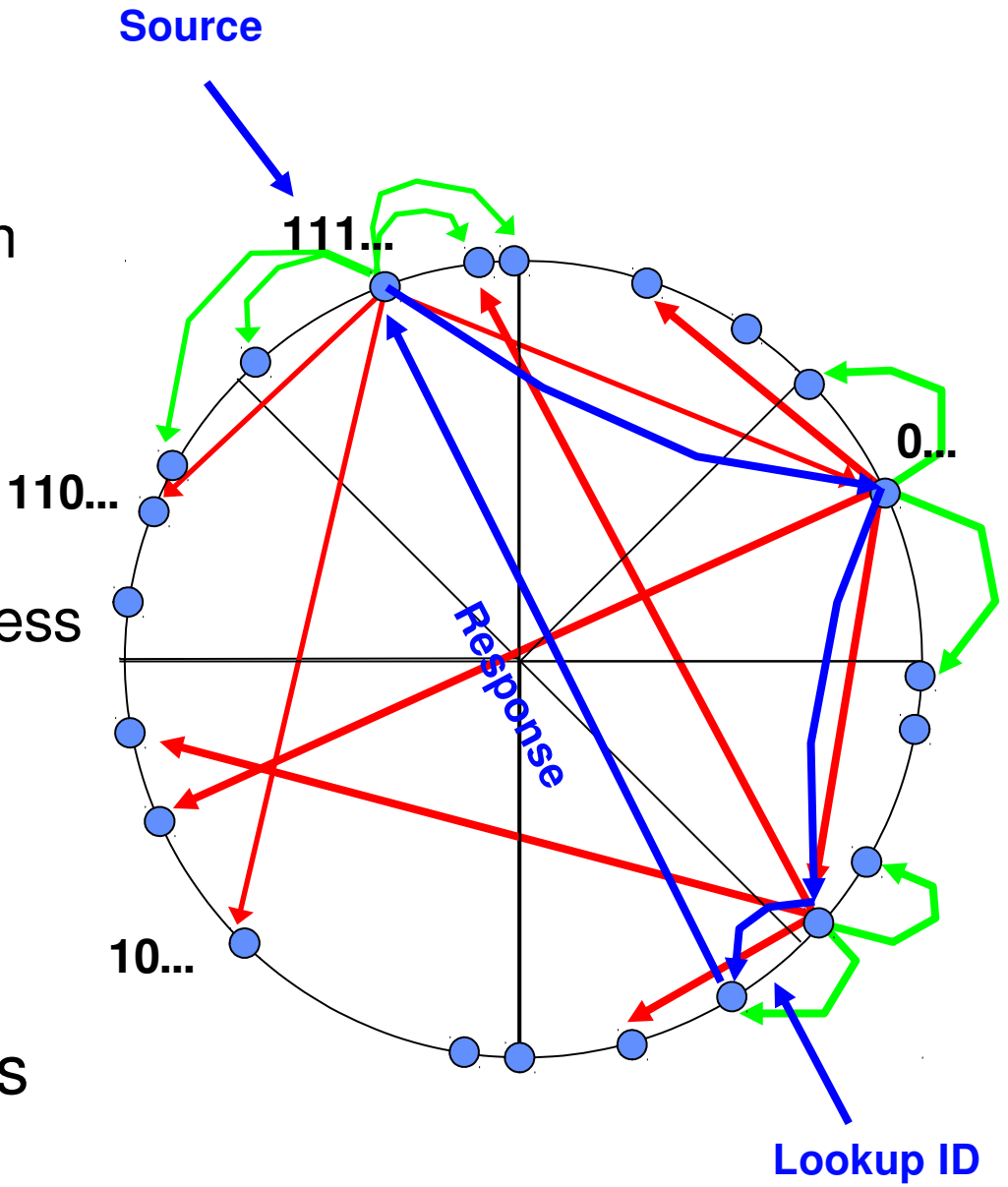
Leaf set is successors and predecessors

- All that's needed for correctness

Routing table matches successively longer prefixes

- Allows efficient lookups

Replicate to "adjacent" nodes



Chord network maintainence

Adding a node:

- New node chooses ID
- New node queries network for "neighbors"
  - This is the new node's **leaf set**
- Node announces to its adjacent nodes
  - They update their **leaf sets**

Fully decentralized

- Can start anywhere
- No decision-maker
- Still works if any node leaves network

"Distributed Hash Tables"

Popular in the early 2000s networking research

Several variants – slightly different kinds of consistent hashing/"leaf sets"

- Chord, CAN, Tapestry, Pastry, Kademlia
- All published 2001-2

Largest (?) in existence: BitTorrent DHT

- Based on Kademlia design

DynamoDB

Amazon's key-value store

Uses consistent hashing (ring)

Central directory (replicated everywhere)

- Not targeting Internet-level scalability like Chord

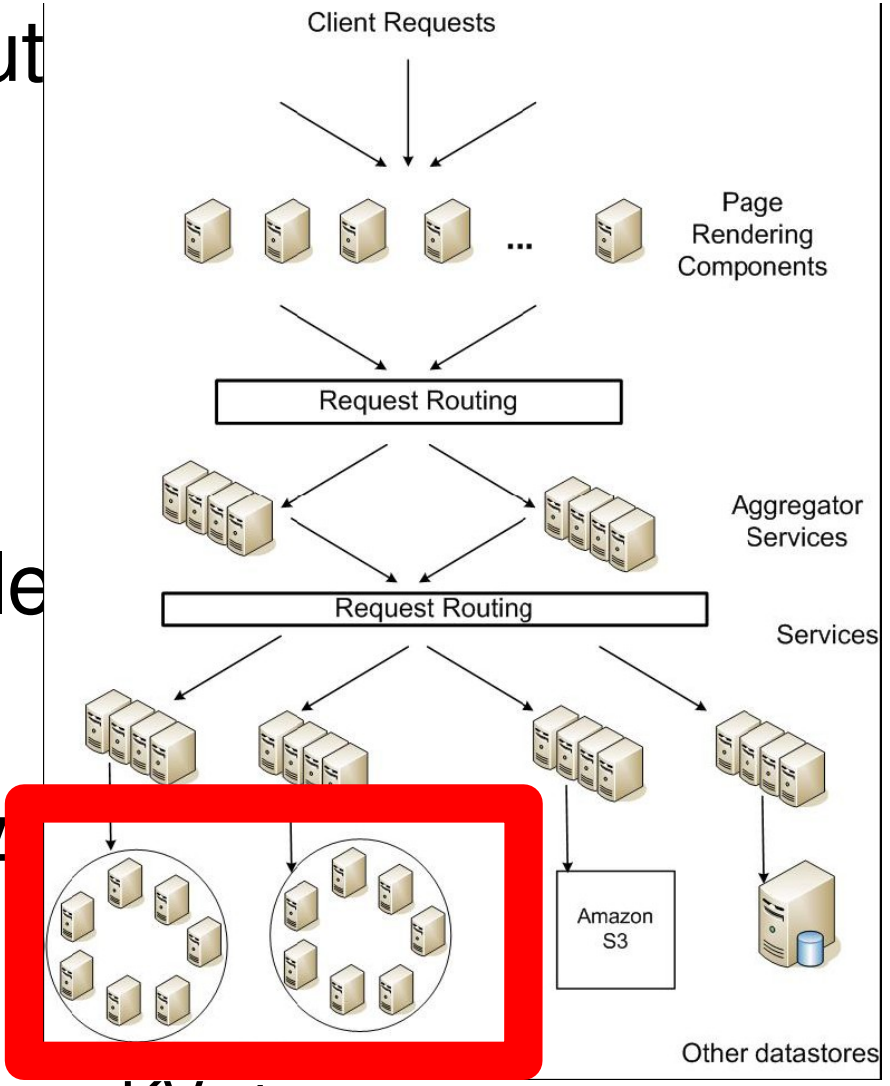
Composing services

We've scaled a hashtable, but that's not enough to do something useful alone...

Example: Amazon

Trees of services with multiple distinct KV stores

Each service has its own SLA (service level agreement)



KV stores instances

SLAs

How fast should you be should you be?

Example goal:

- **99.9th percentile** latency <300 ms

KV store reads:

- Speed of slowest replica with an answer

KV store writes:

- Speed of slowest replica that needs to ACK write

SLA composition

Why a high percentile? [From DynamoDB paper:]

Factor of 10

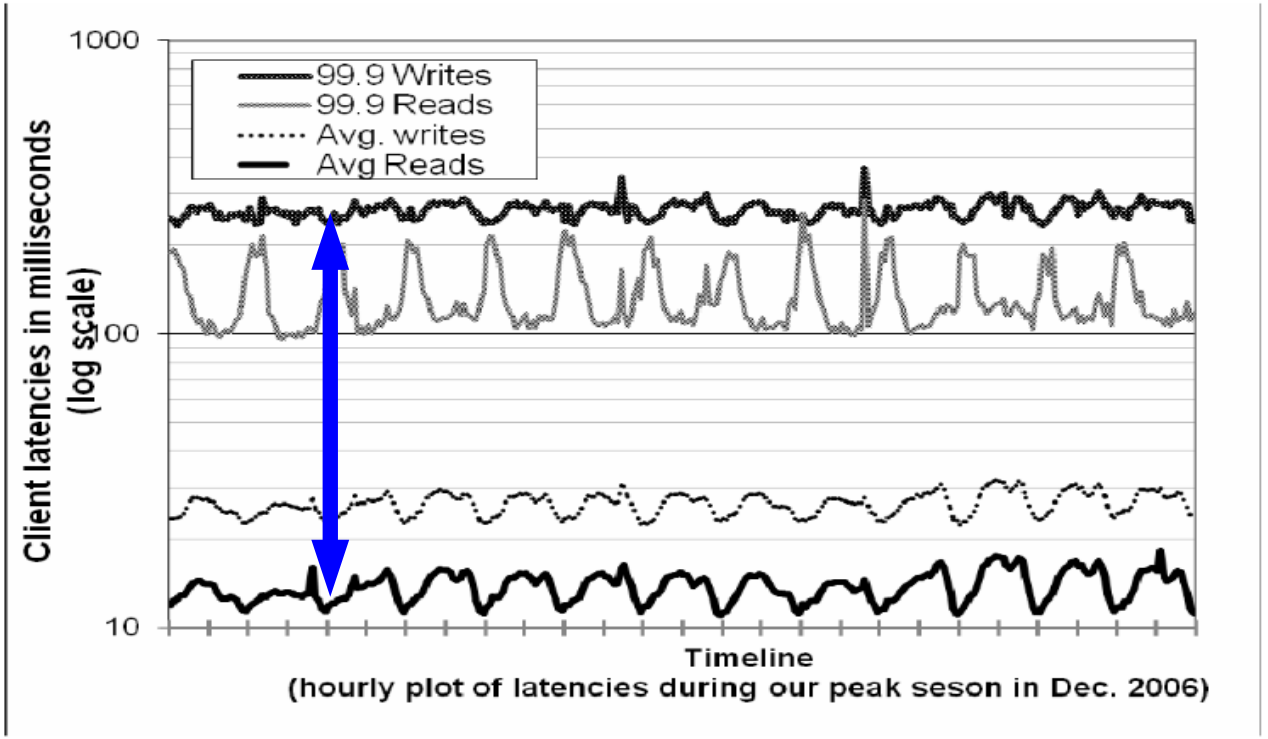


Figure 4: Average and 99.9 percentiles of latencies for read and write requests during our peak request season of December 2006. The intervals between consecutive ticks in the x-axis correspond to 12 hours. Latencies follow a diurnal pattern similar to the request rate and 99.9 percentile latencies are an order of magnitude higher than averages

SLA composition

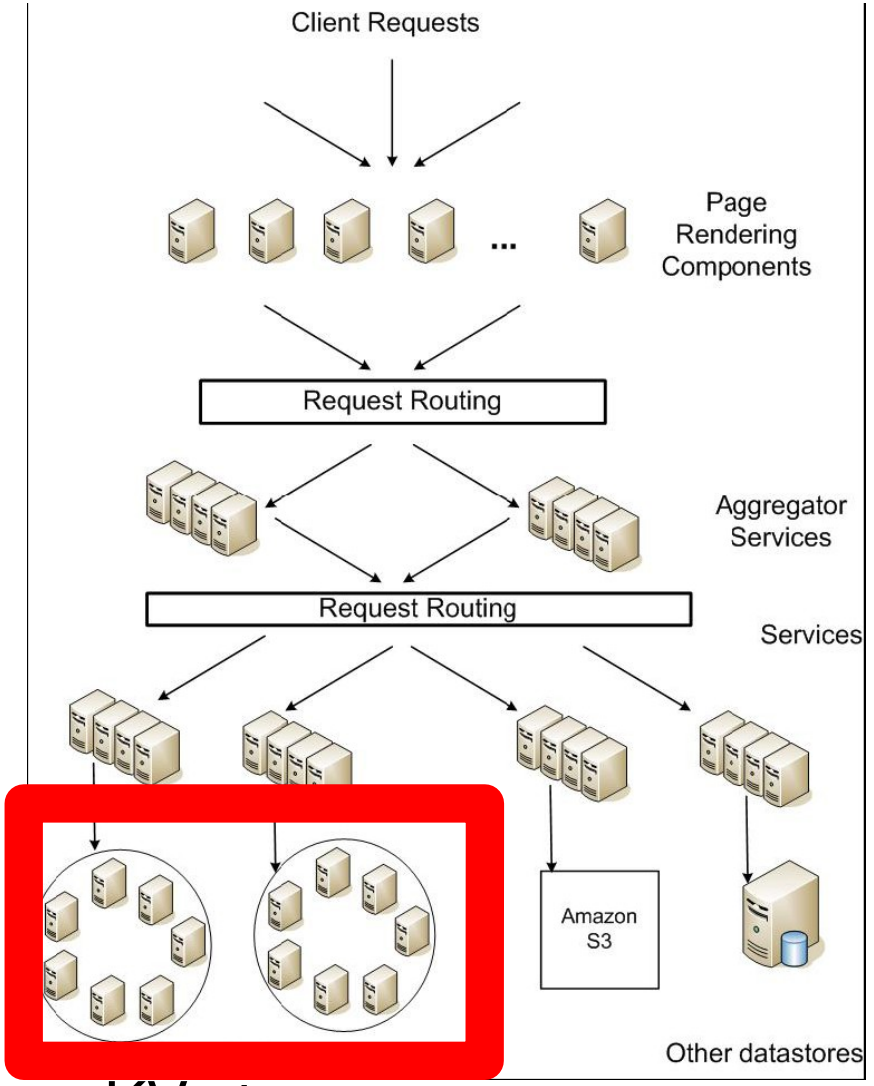
Why a high percentile?

Make lots of requests to KV stores

- Lots of elements on, e.g., Amazon webpage

Slowest time matters

(Why not max?)



Consistency

Need to make sure that a value is replicated correctly

How do you know a value has been replicated on every node?

Wait for acknowledgements from every node

Consistency

What happens if a node fails during replication?

- Pick another node and try again

What happens if a node is slow?

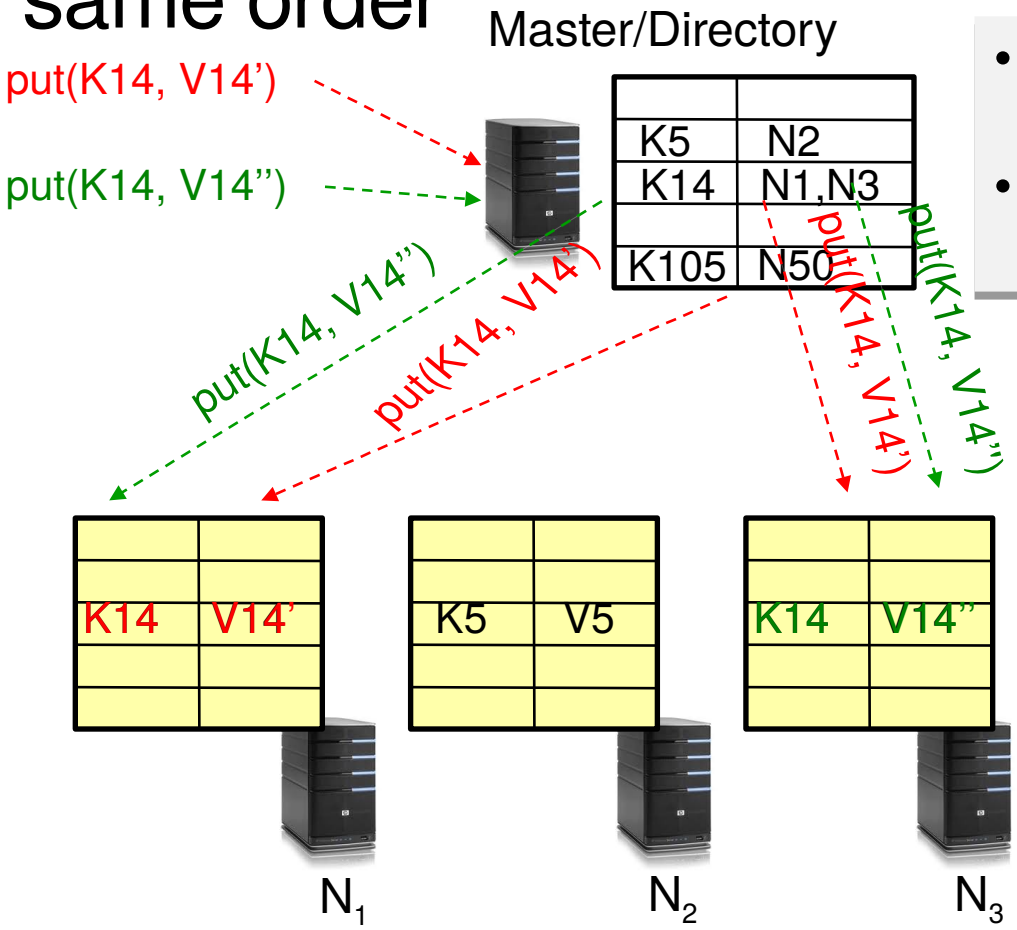
- Slow down the entire put()? Pick another node?

In general, with multiple replicas

- Slow puts and fast gets

Consistency (cont'd)

If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order



- put(K14, V14') and put(K14, V14'') reach N1 and N3 in reverse order
- What does get(K14) return?
  - Undefined!

Quorum Consensus

Improve put() and get() operation performance

Define a replica set of size N

- put() waits for acknowledgements from at least W replicas
- get() waits for responses from at least R replicas
- W+R > N

Why does it work?

- There is at least one node that contains the update

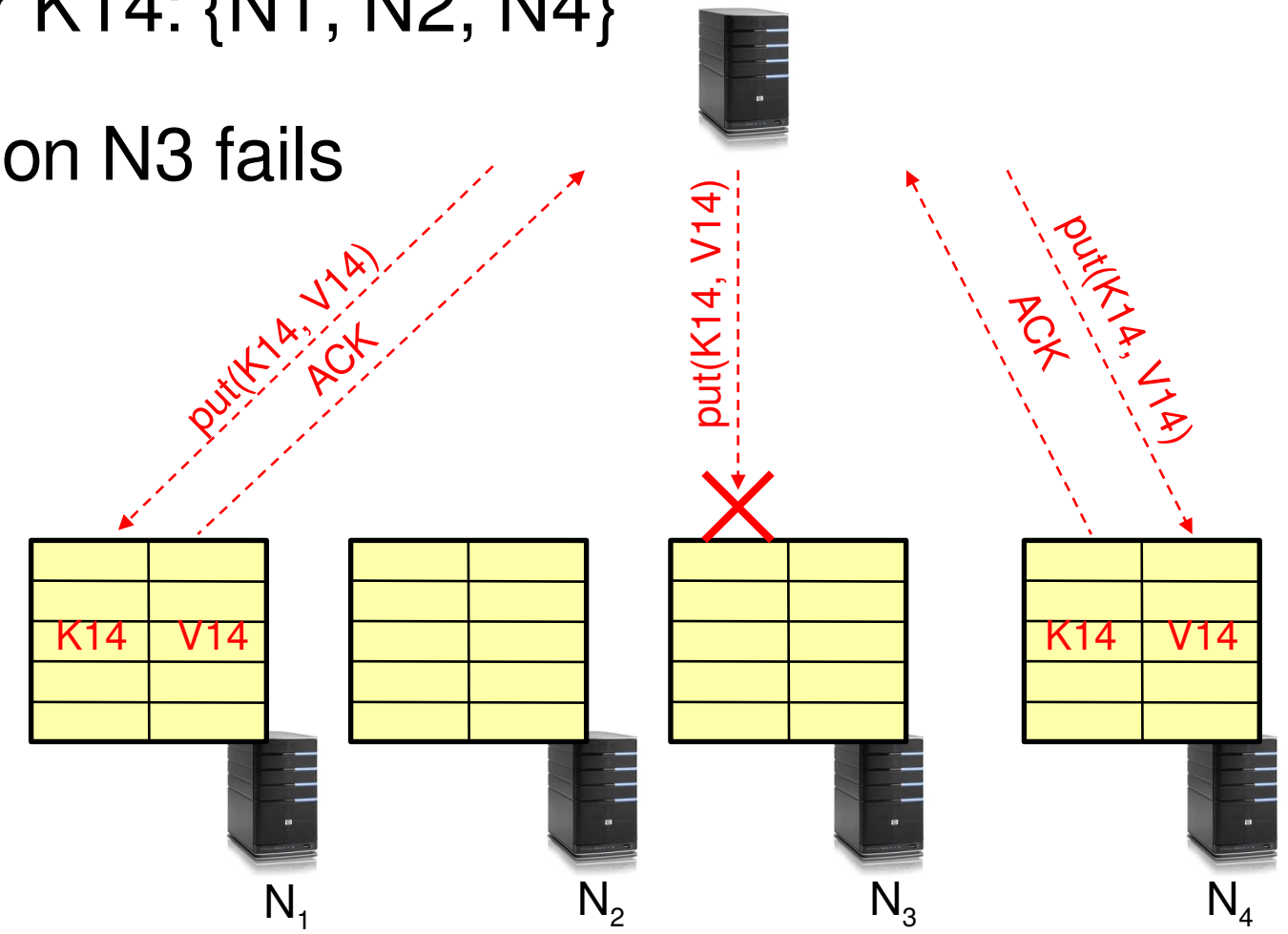
Why might you use W+R > N+1?

Quorum Consensus Example

N=3, W=2, R=2

Replica set for K14: {N1, N2, N4}

Assume put() on N3 fails



Transactions

Closely related to critical sections in manipulating shared data structures

Extend concept of atomic update from memory to atomic update of distributed system's state

Key concept: Transaction

An atomic sequence of actions (reads/writes) on a storage system (or database)

That takes it from one consistent state to another

Typical Structure

Begin a transaction – get transaction id

Do a bunch of updates

- If any fail along the way, roll-back

Commit the transaction

“Classic” Example: Transaction

```
BEGIN;    --BEGIN TRANSACTION
UPDATE accounts SET balance = balance - 100.00 WHERE
  name = 'Alice';
UPDATE branches SET balance = balance - 100.00 WHERE
  name = (SELECT branch_name FROM accounts WHERE name =
  'Alice');
UPDATE accounts SET balance = balance + 100.00 WHERE
  name = 'Bob';
UPDATE branches SET balance = balance + 100.00 WHERE
  name = (SELECT branch_name FROM accounts WHERE name =
  'Bob');


COMMIT;    --COMMIT WORK
```

Transfer $100 from Alice’s account to Bob’s account

The ACID properties of Transactions

Atomicity: all actions in the transaction happen, or none happen

Consistency: transactions maintain data integrity, e.g.,

- Balance cannot be negative
- Cannot reschedule meeting on February 30

Isolation: execution of one transaction is isolated from that of all others; no problems from concurrency

Durability: if a transaction commits, its effects persist despite crashes

Summary: Distributed Systems

Distributed – multiple machines performing different parts of task

- Contrast to parallel: multiple (usually similar) things at the same time

Goal: Transparent – doesn't look like distributed system

Communication with **messages**

- Contrast with threads: shared memory

Summary: Key-Value Stores

One way of building **distributed storage**

- **put(key, value)**
- **get(key)**

Easy to scale

- Spread keys over nodes with **consistent hashing** on keys

Subject of project 3