

The Practical Challenges of Managing Big Data in the Cloud

Amr El Abbadi

University of California,

Santa Barbara

Evolution of computing history

- Main Frame with terminals
- Network of PCs & Workstations.
- Client-Server
- Now, moving forward to
Large cloud.

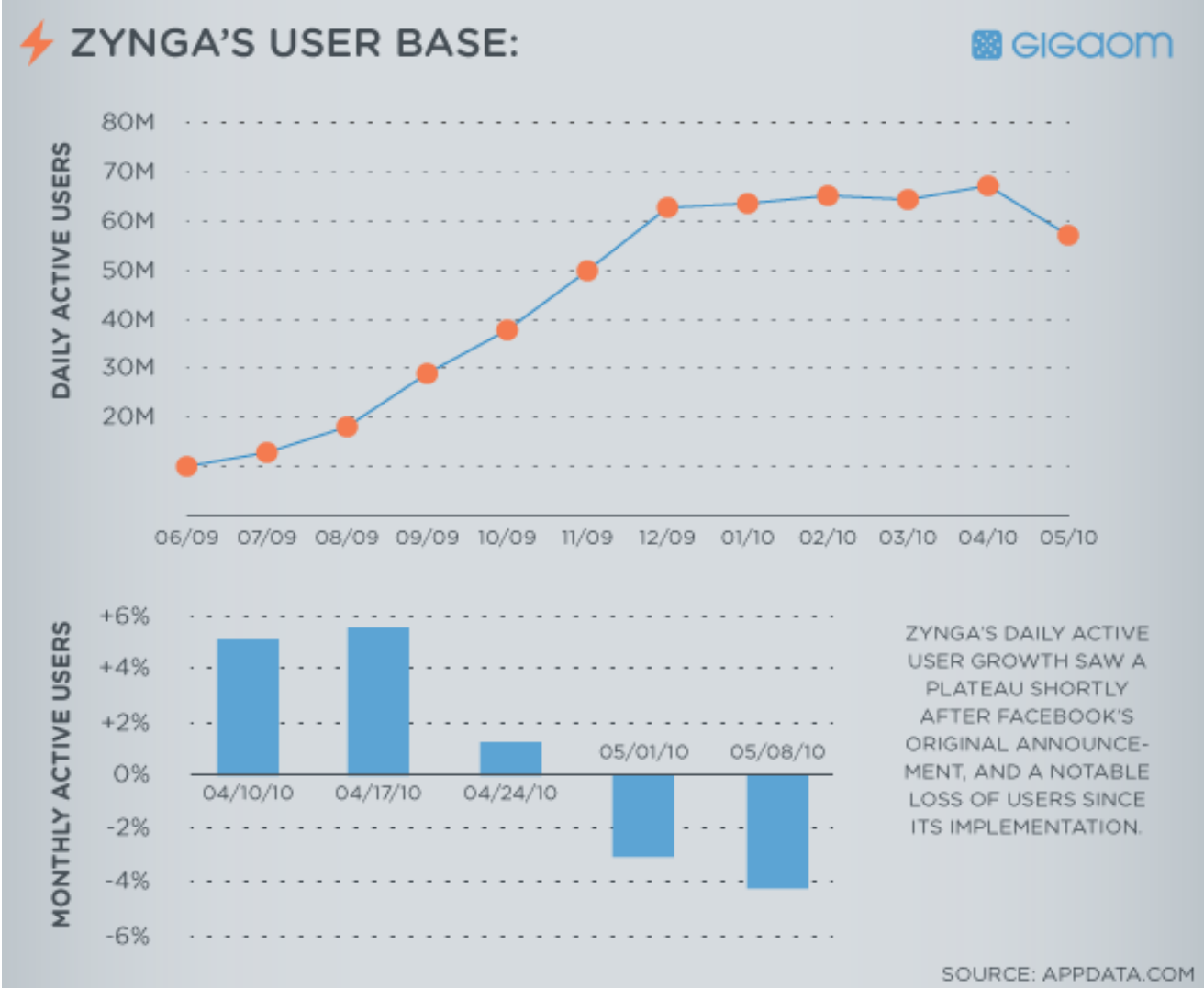
Cloud Computing: Why Now?

- Experience with **very large datacenters**
 - Unprecedented **economies of scale**
 - Transfer of **risk**
- **Technology factors**
 - Pervasive **broadband Internet**
 - Maturity in **Virtualization Technology**
- **Business factors**
 - Economies of **Scale**
 - **Pay-as-you-go** billing model

Scalability in the Cloud



Cloud Reality: Elasticity



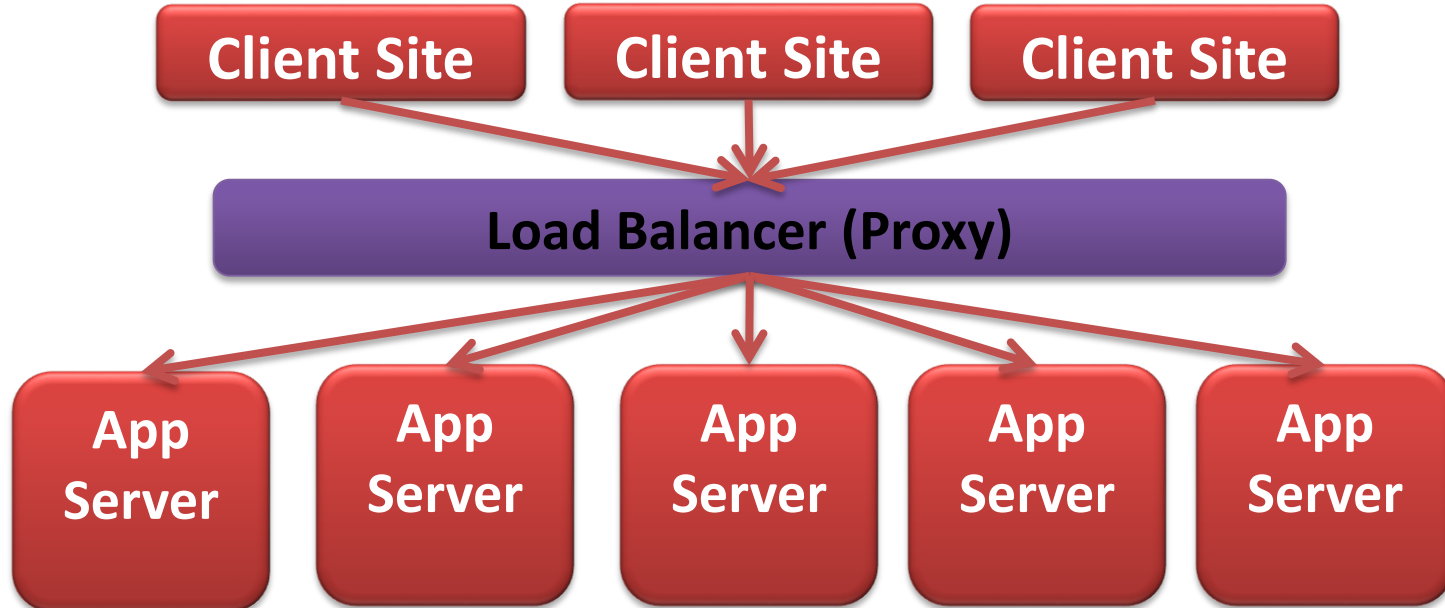
Explosive Data growth

- **Wikipedia** has over **5.4 million English Articles**, and in general more than **40 million** articles in 293 languages..
- **Yahoo!** **Over 1Billion unique user** and over **204M visits/month**
- **Instagram** users have shared over **40 billion photos** and share an average of **95 million photos and videos per day**
- **Facebook:** **1.86Billion** users, on average, the Like and Share Buttons are viewed across almost **10 million websites daily**; **Every 60 seconds: 510,000 comments, 293,000 statuses, and 136,000 photos** are uploaded.
- **YouTube:** **1.3 Billion users**; **300 hours of video** are uploaded every minute; Almost **5 billion videos** are watched on Youtube every single day.

Cloud Properties

- Commodity hardware
- Large Scale
- Elasticity

Elasticity in the Cloud



Why does this work?

- As long as requests are **stateless**, we can add more resources, thus providing:
- Scale
- Elasticity

But, most services need DATA!

- Challenges:

- How to scale with the increasing amounts of data
- Where to store the data
- Accessing data on multiple sites
- Failures

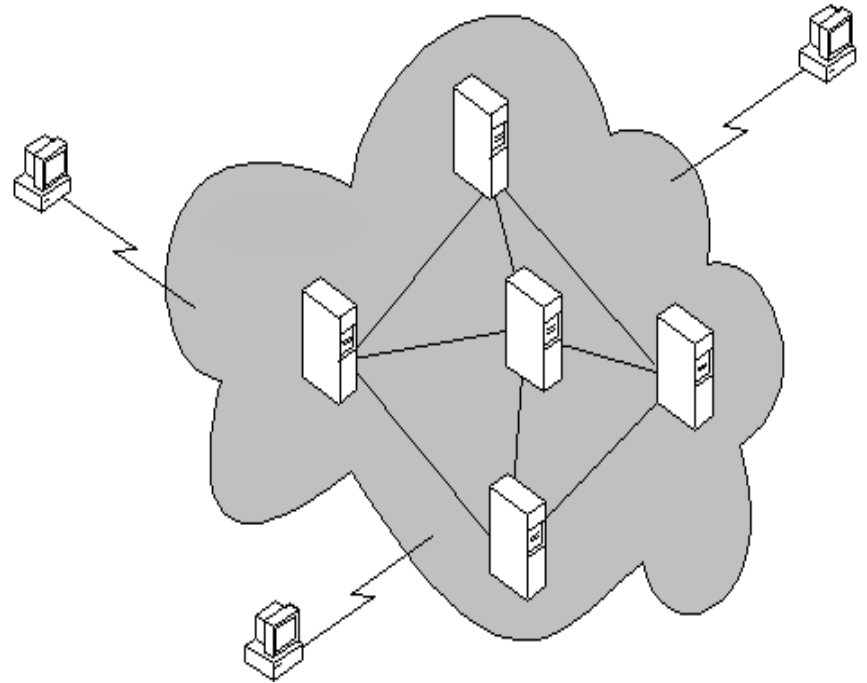
Need

- Fault-tolerance:
 - Replication
- Large scale data:
 - Partition data across multiple servers
- Managing the system state.
- Must understand:
 - Database foundations
 - Distributed systems foundations.

DISTRIBUTED SYSTEMS FOUNDATIONS

Main Characteristics of Distributed Systems

- Independent processors, sites, processes
- Message passing
- No shared memory
- No shared clock
- Independent failure modes



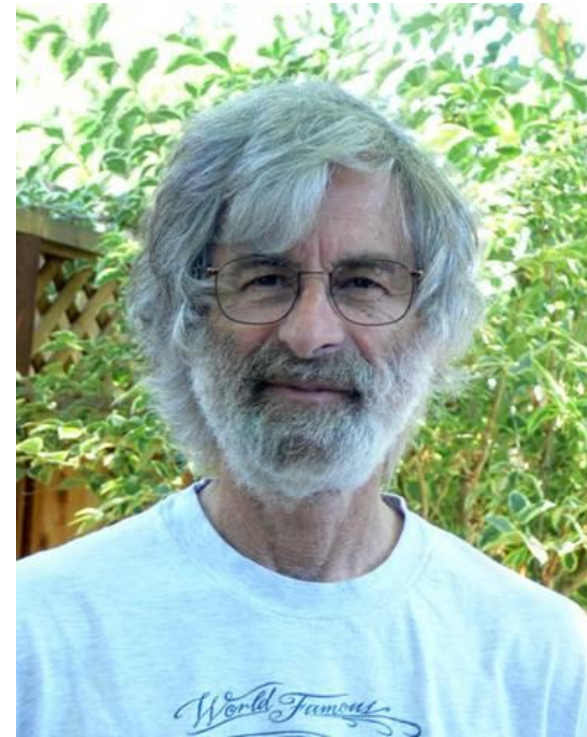
Distributed System Models

- **Synchronous System:** **Known bounds** on **times** for message transmission, processing, bounds on local clock drifts, etc.
 - Can use **timeouts**
- **Asynchronous System:** **No known bounds** on **times** for message transmission, processing, bounds on local clock drifts, etc.
 - More realistic, practical, but **no timeout**.

CAUSALITY AND TIME

What is a Distributed System?

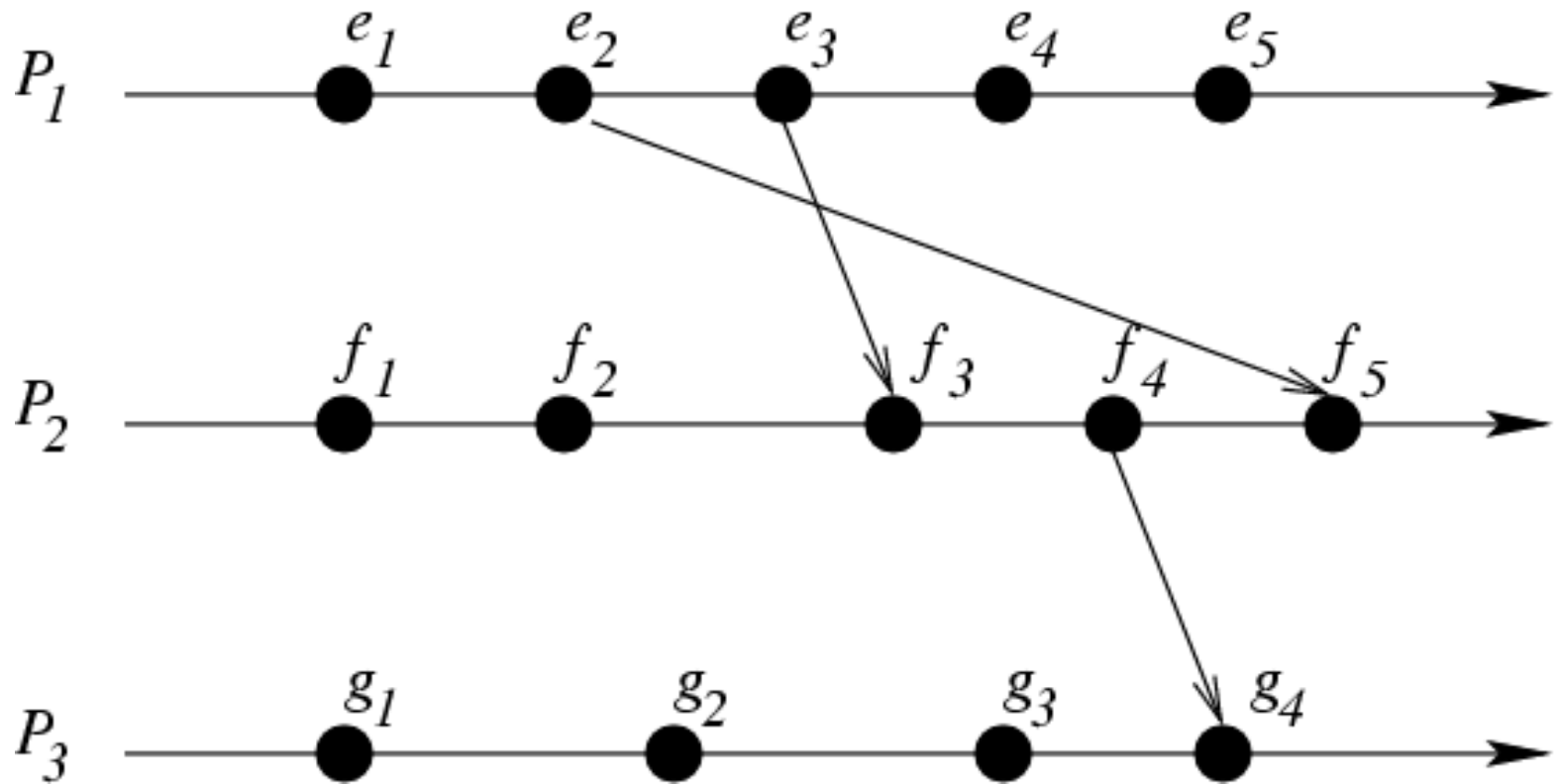
- A simple model of a distributed system proposed by [Lamport](#) in a landmark 1978 paper in the Communications of the ACM: “Time, Clocks and the Ordering of Events in a Distributed System”



What is a Distributed System?

- A set of **processes** that **communicate** using **message passing**.
- A **process** is a sequence of **events**
- 3 kinds of events:
 - **Local** events
 - **Communication events**:
 - **Send** events
 - **Receive** events
- **Local events** on a process for a **total order**.

Example of a Distributed System



Happens Before or Causal Order on Events

- Event e *happens before (causally precedes)* event f , denoted $e \rightarrow f$ if:
 1. The **same** process executes e before f ; or
 2. e is **send(m)** and f is **receive(m)**; or
 3. Exists h so that $e \rightarrow h$ and $h \rightarrow f$
- We define *concurrent*, $e \parallel f$, as:
$$\neg(e \rightarrow f \vee f \rightarrow e)$$

Lamport Logical Clocks

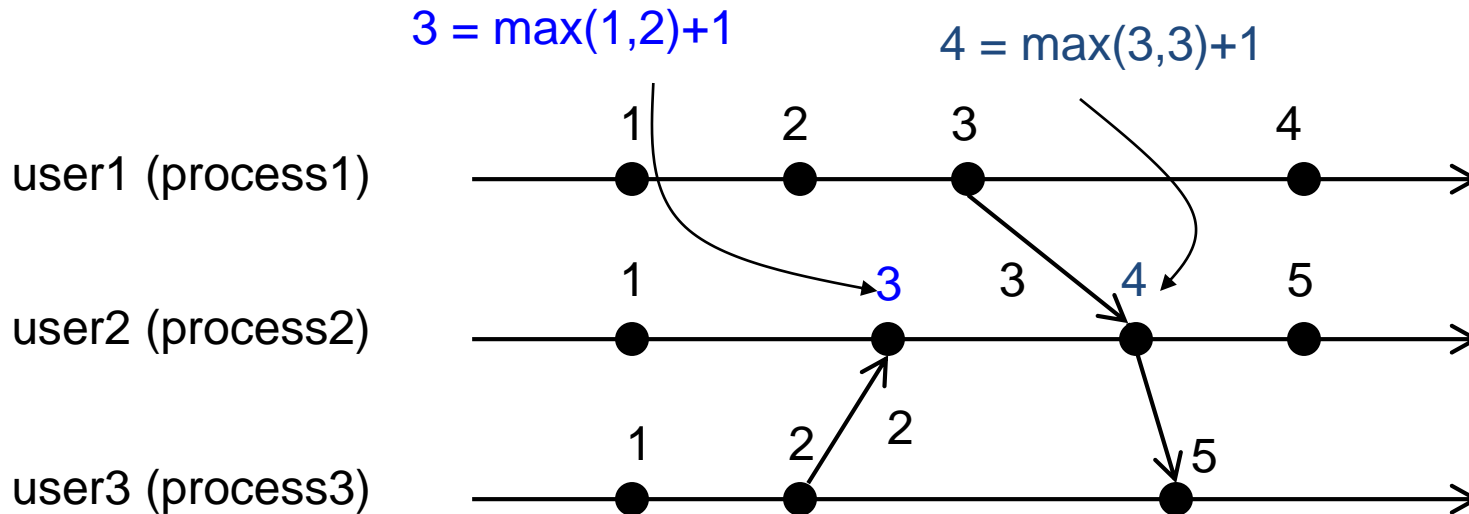
- Assign a “clock” value to each event such that

if $a \rightarrow b$ then

$$\text{clock}(a) < \text{clock}(b)$$

Logical Clocks

- Each event has a single integer as its logical clock
 - Each process has a local counter C
 - Increment C between two events
 - At each “send”, logical clock value V is attached.
 - At each “receive”, $C = \max(C, V) + 1$



Problem: Detecting causal relations

If $L(e) < L(e')$

- Cannot conclude that $e \rightarrow e'$

Checking Lamport timestamps

- Cannot conclude which events are causally related

Solution: use a **vector clock**

Vector Clocks

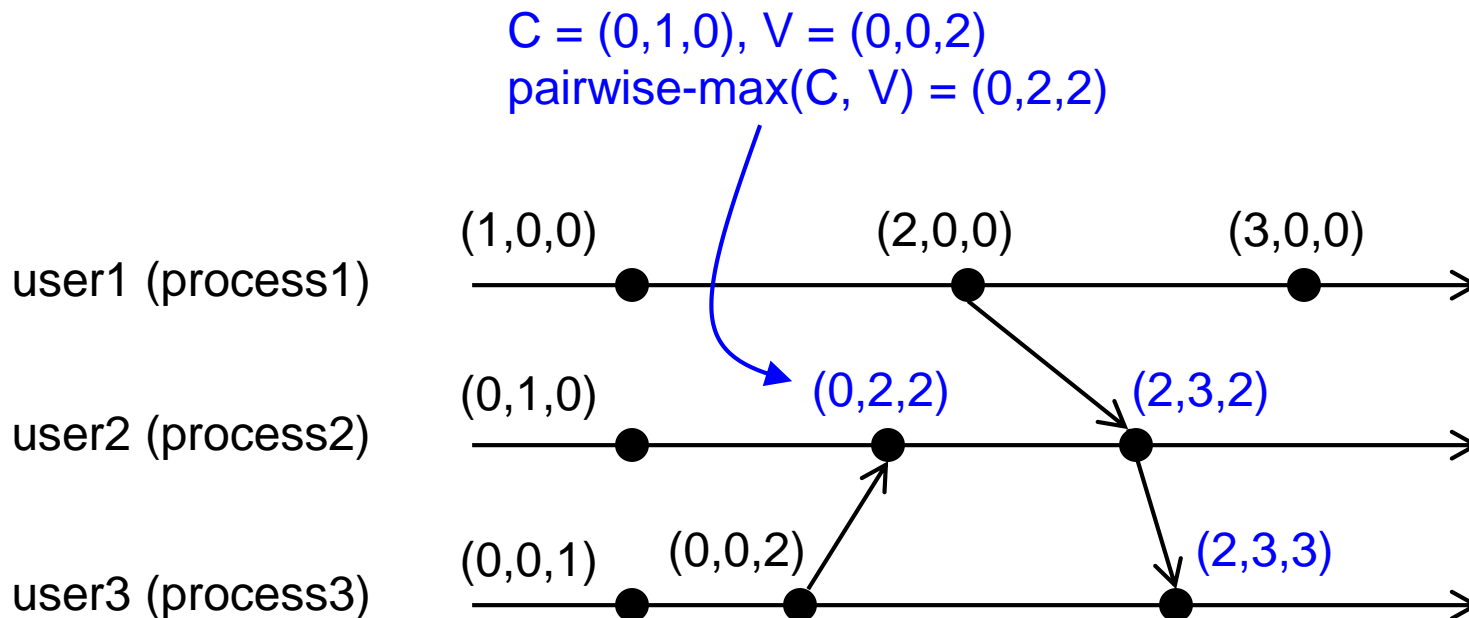
- Developed **independently** by Fidge, Mattern and Schmuck.
- Time is represented by a set of n-dimensional non-negative integer vectors.
- Each process has a clock C_i consisting of a **vector of length n** , the total number of processes $vt[1..n]$. $vt[j]$ is the local logical clock of P_j and describes the logical time progress at process P_j

Vector Clock Protocol

- Timestamp $C(e)$ of event e is the clock value after ticking
- P_i ticks by **incrementing its own component** of its clock: $C_i[i] += 1$
- Each **message is piggybacked** with the local vector u
- Recipient updates local vector **to max of u and local vector v**

Vector Clock Protocol

- Each process i has a local vector C
- Increment $C[i]$ at each “local computation” and “send” event
- When sending a message, vector clock value V is attached to the message. At each “receive” event, $C = \text{pairwise-max}(C, V); C[i]++$;



Comparing vector timestamps

Define

$V \leq V'$ iff $V[i] \leq V'[i]$ for $i = 1 \dots N$

$V < V'$ iff $V \leq V'$ and $\exists i$ such that $V[i] \neq V'[i]$

For any two events e, e'

$e \rightarrow e'$ if and only if $V(e) < V(e')$

Two events are **concurrent** if **neither**

$V(e) < V(e')$ nor $V(e') < V(e)$

MUTUAL EXCLUSION AND QUORUMS

Distributed Mutual Exclusion

- Given a **set of processes** and a **single resource**, develop a protocol to ensure **exclusive access** to the resource by a **single process at a time**.
- This is a **fundamental** operation in operating systems, and is generalized to **locking** in databases.

Distributed Solution (Lamport '78)

- Instead of a central coordinator, **all processes collectively**
- Use **similar approach**:
 - Process sends **request** to **all processes** and put request in local queue.
 - On receipt of request, **process** sends back **reply**.
 - Process **accesses resource**
 - On receipt of **all replies**
 - Own request at **head of queue**
 - Once done, process sends **release** to **all processes**.
 - On receipt of release, **process** removes request

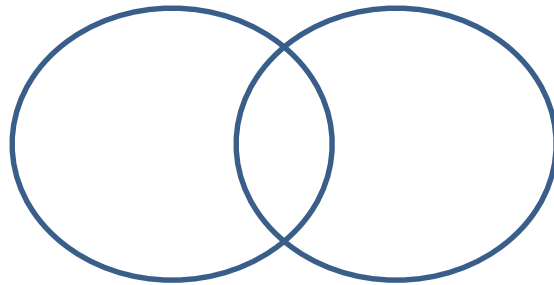
Quorums

- What if there are **failures**?
 - No failure of the resource holder
- Do we need to communicate with **ALL** processes?
- A **quorum** is the minimum number of **votes** that a process has to obtain in order to be allowed access to the shared resource.

Quorums

- Any two requests should have a common process to act as an **arbiter**.
- V_i is called a **quorum**.
- Let process p_i (p_j) request permission from V_i (V_j), then

$$V_i \cap V_j \neq \emptyset$$



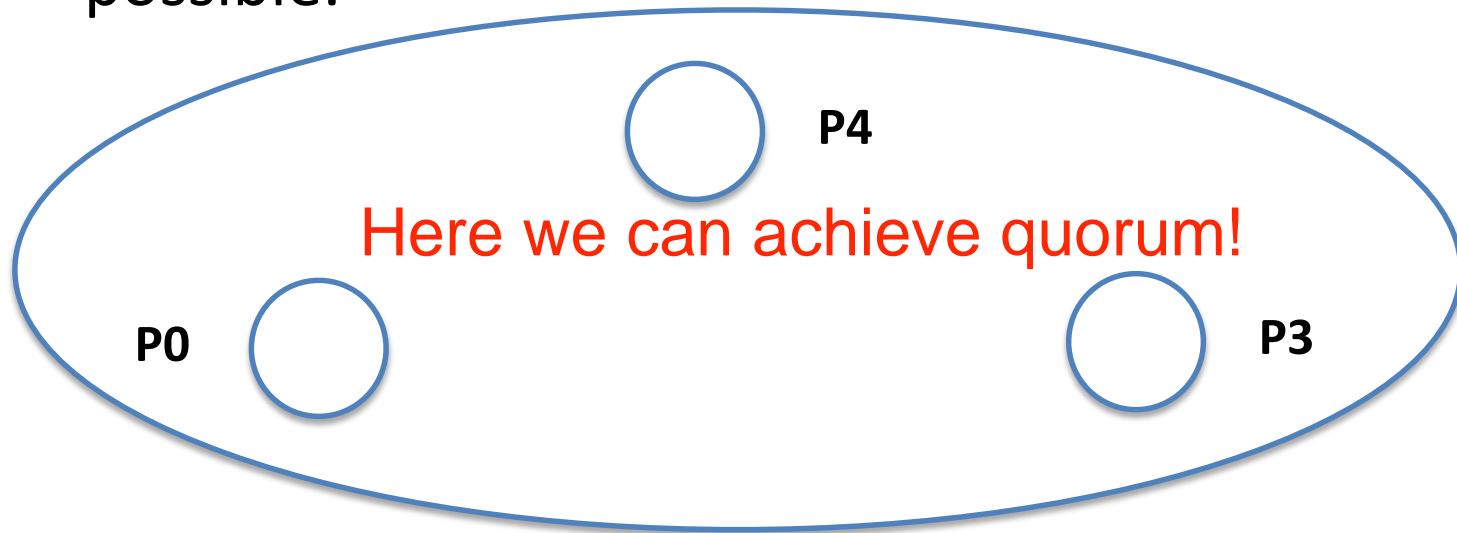
- In general, use **majority**, ie $\lceil (N/2) \rceil$. [Gifford 79]

Quorums

- Main problem: **Deadlock**

Network failure: Partitioning

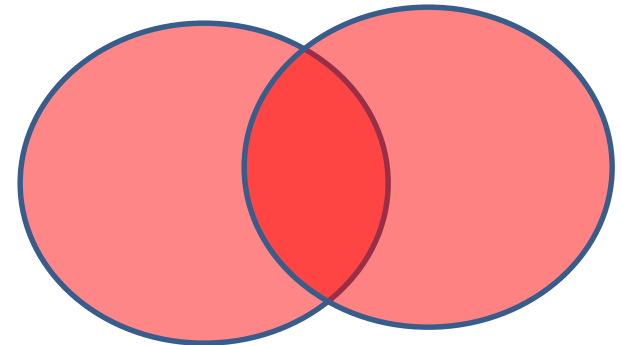
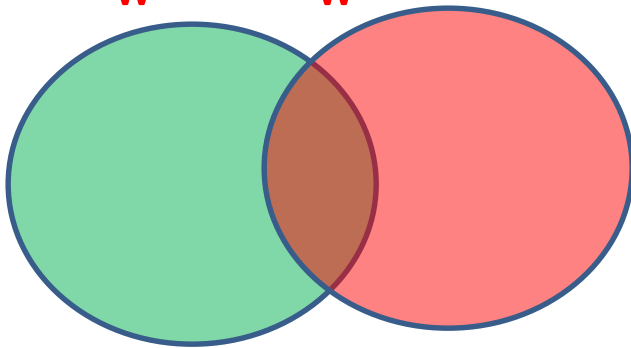
If sites are unreachable, quorum might still be possible.



This partition won't be able to get the resource.

General Quorums

- In a **database context**, we have read and write operations. Hence, **read quorums**, Q_r , and **write quorums**, Q_w .
- Simple generalization:
 - $Q_r \cap Q_w \neq \varnothing$, $Q_r + Q_w > n$ and
 - $Q_w \cap Q_w \neq \varnothing$, $2 Q_w > n$



LEADER ELECTION

Election Algorithms

- Many distributed algorithms need **one process to act as coordinator**
 - Doesn't matter which process does the job, just need to pick one
- **Election algorithms**: technique to pick a unique coordinator (aka *leader election*)
- Types of election algorithms: **Bully and Ring algorithms**

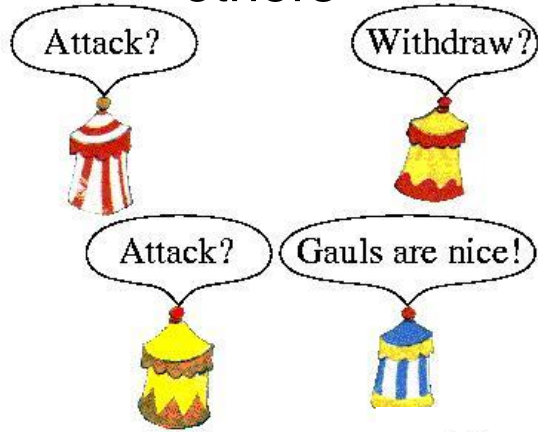
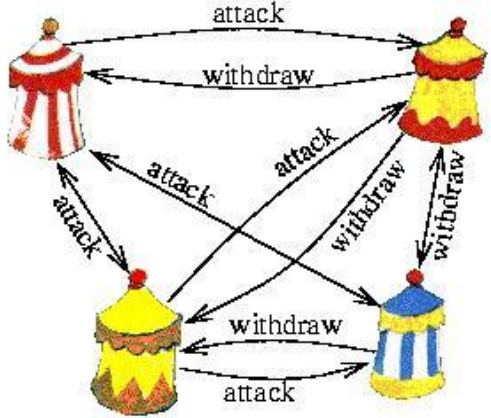
CONSENSUS AND BYZANTINE AGREEMENT

Once upon a time...

Some of them may be traitors who will try to confuse the others



Communicating only by messenger



Generals must agree upon a common battle plan

Consensus or Byzantine Agreement

- Proposed by Lamport, Shostak, and Pease in 1982
- **Malicious** Failures (**byzantine** failures)
- **General** sends an binary value to **n-1 participants** such that:
 1. **Agreement:** All correct participants **agree on same value**
 2. **Validity:** If general is correct, **every participant agrees on the value general sends**

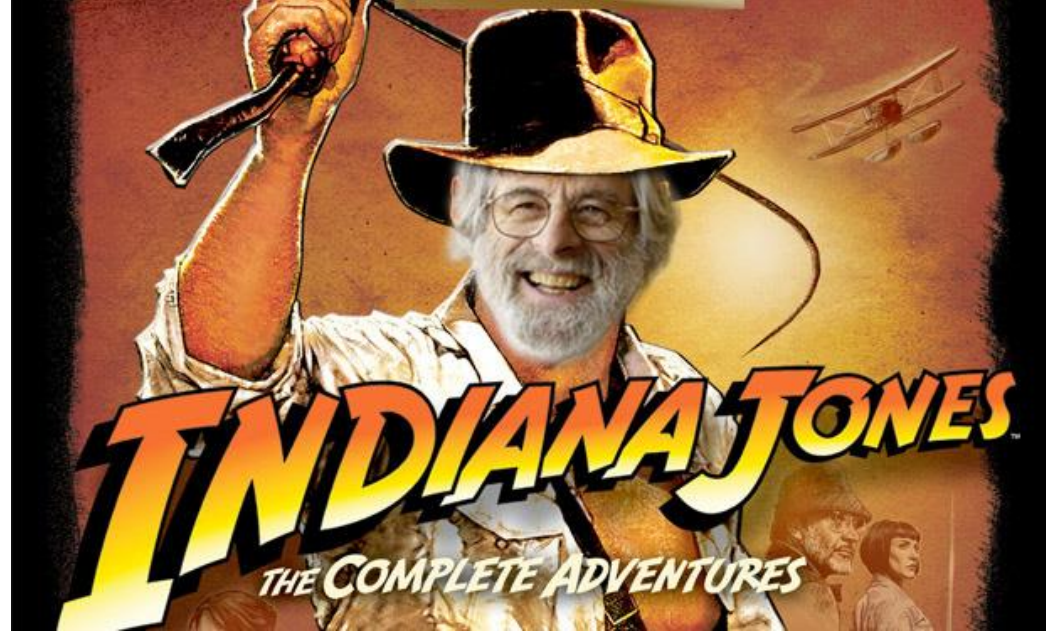
Impossibility of Distributed Consensus with ONE faulty Process

- Fisher, Lynch & Patterson. PODS 83, JACM 85.
- **Asynchronous** system; but reliable NW.
- Process: **Crash failures**. Max **ONE** failure.
- **No asynchronous consensus protocol is totally correct in spite of one fault.**
- **OR: Every “safe” protocol for the asynchronous consensus problem has some execution that does not terminate.**

Impossibility Result for Synchronous Systems

- With only 3 processes, no solution can work with even 1 MALICIOUS failure
- In general, no solution if more than $1/3$ of sites have malicious failures.

Paxos



- **Lamport** the archeologist and the “Part-time Parliament” of **Paxos**:
 - The Part-time Parliament, TOCS 1998
 - Paxos Made Simple, *ACM SIGACT News* 2001.
 - Paxos Made Live, PODC 2007

—.....

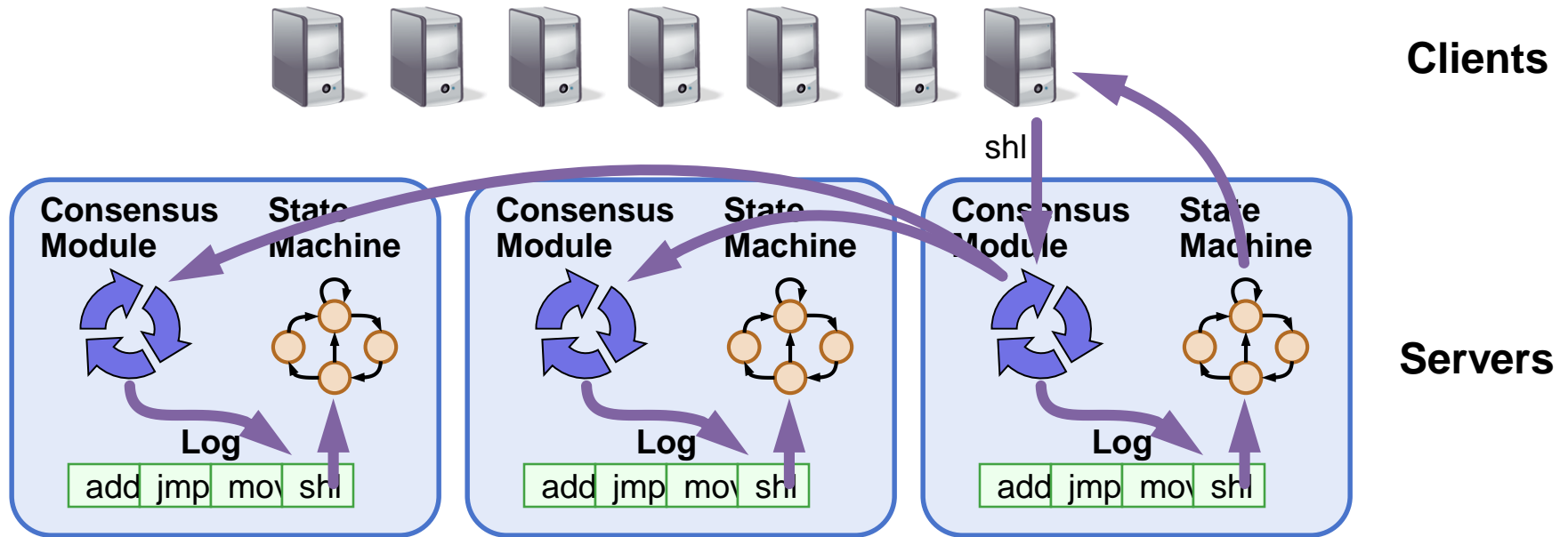
Paxos System Assumptions

- Paxos is an **asynchronous** consensus algorithm
 - Asynchronous networks
- Set of processes is **known a-priori**
- Processes suffer **crash failures**

Paxos Properties

- Paxos guarantees **safety**.
 - Consensus is a **stable property**: once reached it is never violated; the agreed value is not changed.
- Paxos does **not** guaranteed **liveness**.
 - Consensus is reached if “a large enough subnetwork...is non-faulty for a long enough time.”
 - Otherwise Paxos might never terminate.

Goal: Replicated Log



- Replicated log => replicated state machine
 - All servers execute same commands in same order
- Consensus module ensures proper log replication

Overview of the Paxos Algorithm

- **Leader based**: each client has an estimate of who is the current leader
- To order an operation, a client sends it to current leader
- **Quorums** (majority) are used for fault-tolerance.
- Multiple phases or rounds of communication.

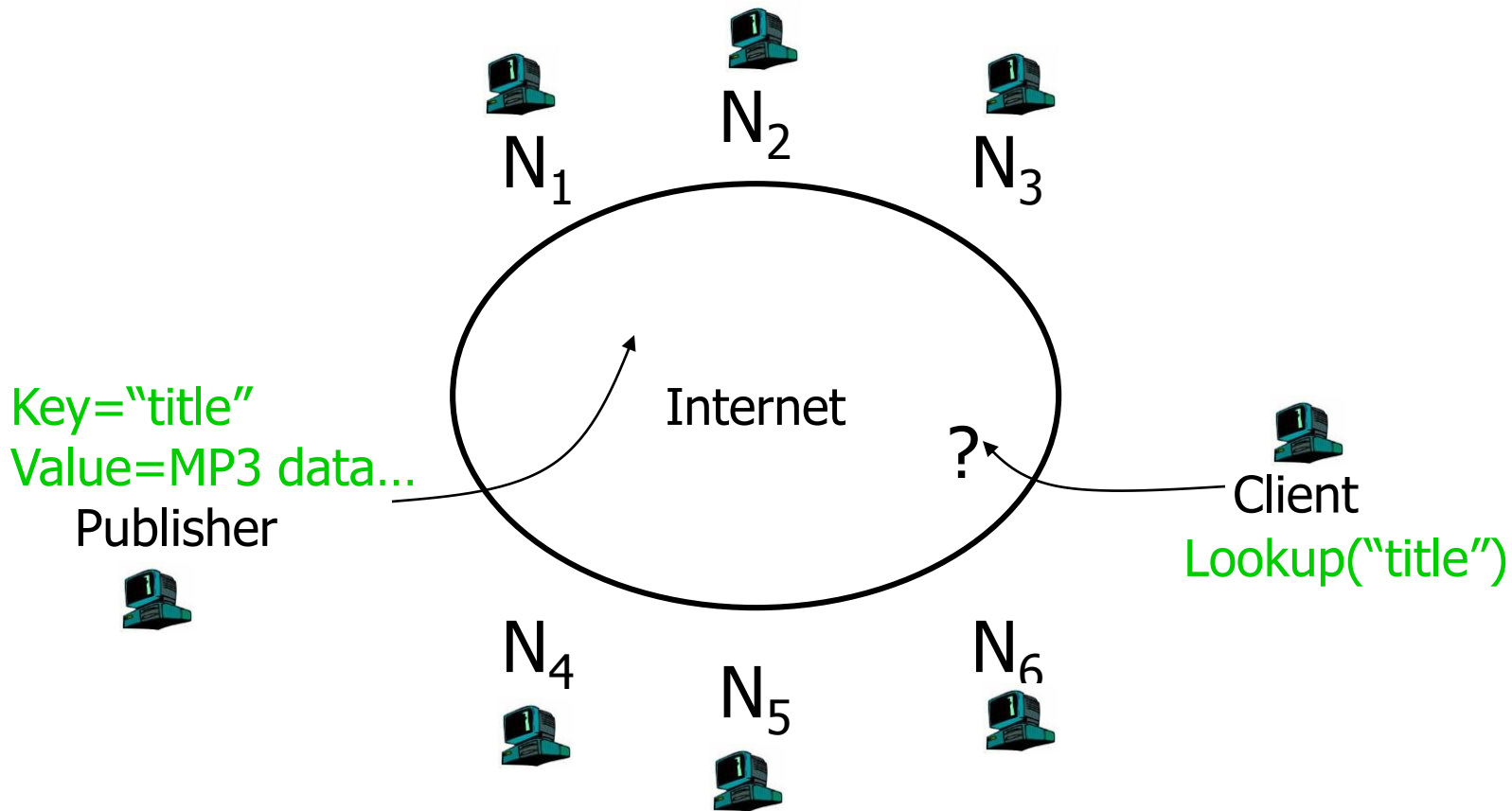
PEER TO PEER AND DISTRIBUTED HASH TABLES

Distributed Hash Tables

Challenge: To design and implement a **robust and scalable distributed system** composed of inexpensive, individually unreliable computers in unrelated administrative domains

- **Goal:** Make billions of objects available to millions of concurrent users
- **Basic Operations:**
 - Insert(key)
 - Lookup(key)

Searching

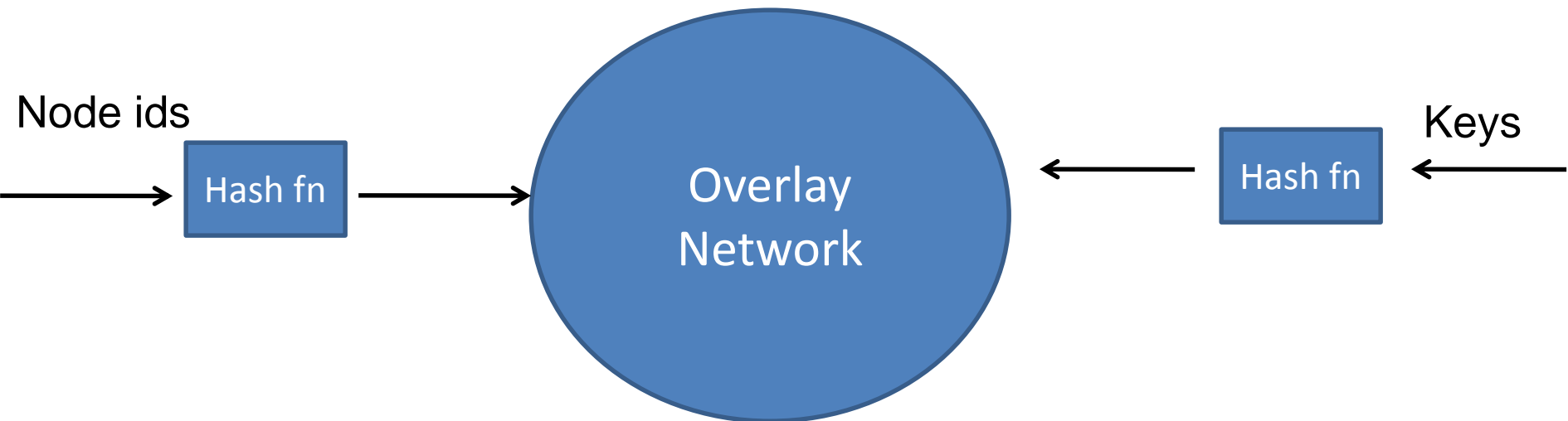


Simple Solution

- First There was **Napster**
 - Centralized server/database for lookup
 - Only file-sharing is peer-to-peer, lookup is not
- Launched in 1999, peaked at 1.5 million simultaneous users, and shut down in July 2001.

Overlay Networks

- A virtual structure imposed over the physical network (e.g., the Internet)
 - A graph, with hosts as nodes, and some edges



Structured vs. Unstructured

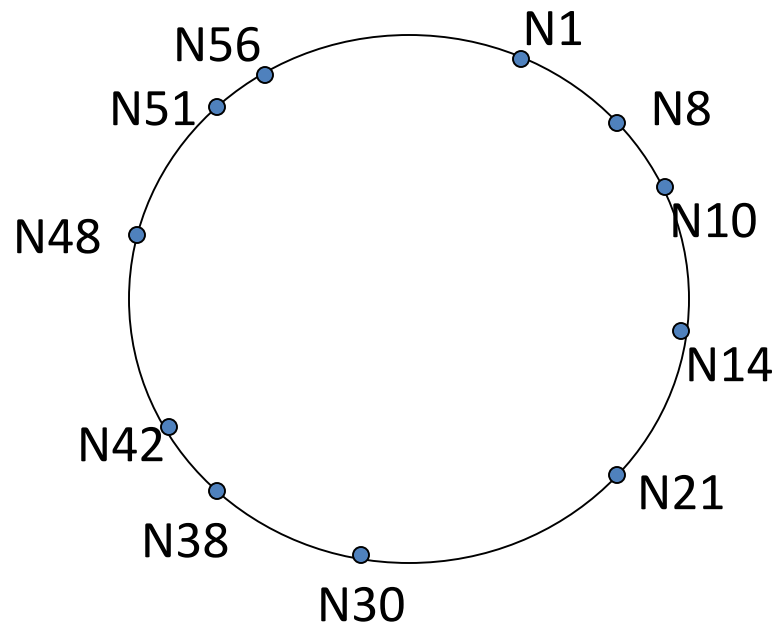
- *Unstructured P2P*
 - There is **no systematic rule** for how edges are chosen,
each node “knows some” other nodes
 - **Any node can store any data** so a searched data might reside at any node
- *Structured overlay:*
 - The **edges** are chosen according to **some rule**
 - **Data** is stored at a **pre-defined place**
 - **Tables** define **next-hop for lookup**

Distributed Hash Tables (DHTs)

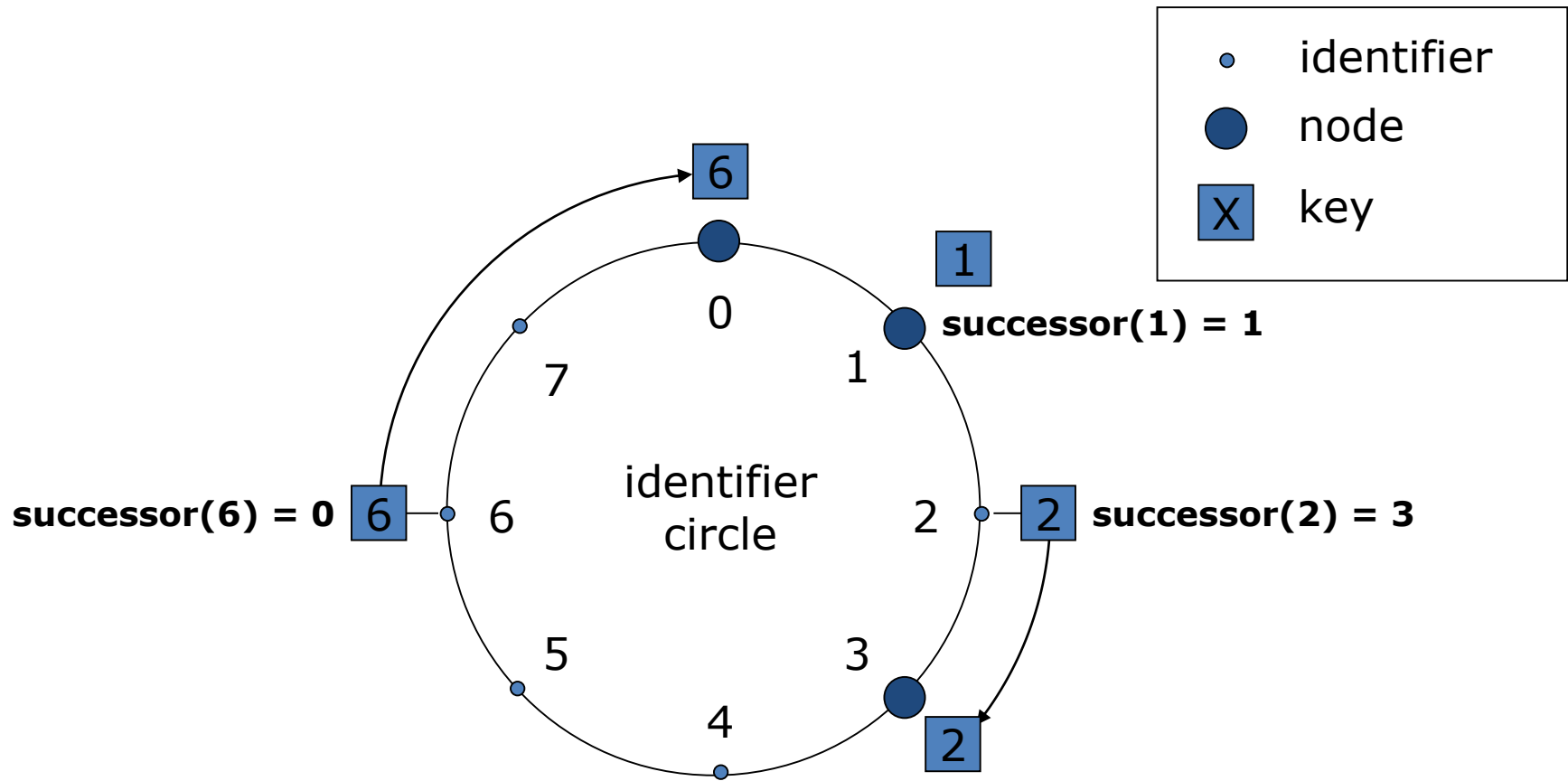
- Nodes store table entries
- **lookup(key)** returns the **location of the node** currently responsible for this **key**
- We will discuss **Chord**, Stoica, Morris, Karger, Kaashoek, and Balakrishnan SIGCOMM 2001
- **Other examples:** **CAN** (Berkeley), **Tapestry** (Berkeley), **Pastry** (Microsoft Cambridge), etc.

Chord Logical Structure (MIT)

- m -bit ID space (2^m IDs), usually $m=160$.
- Nodes organized in a **logical ring** according to their IDs.

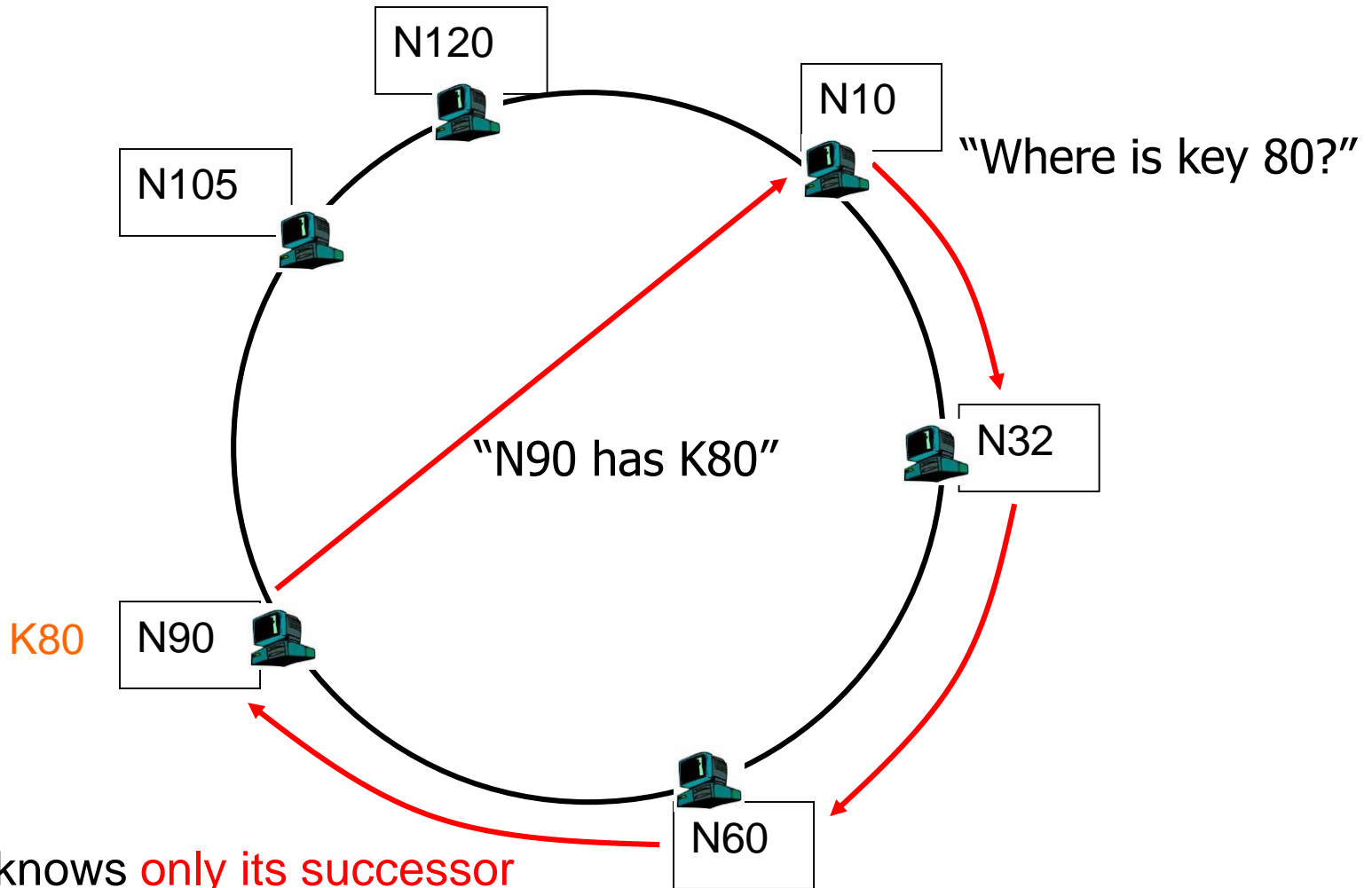


Consistent Hashing - Successor Nodes (Ex: three sites/nodes at 0, 1, 3)



A key is stored at its successor: node with next higher ID

DHT: Chord Basic Lookup



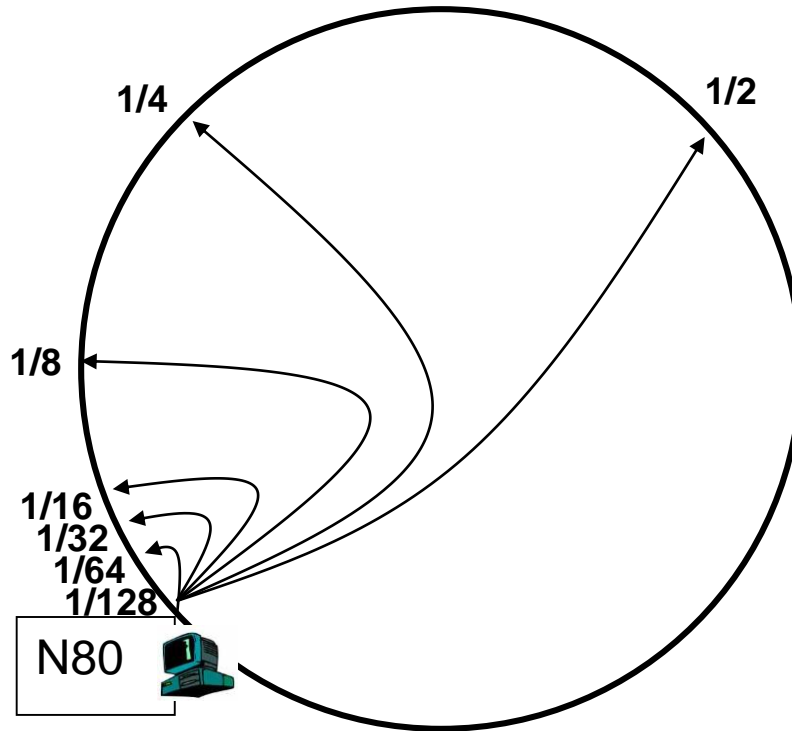
Each **node** knows **only its successor**

- Routing around the circle, one node at a time.

Scalable Key Location – Finger Tables

- To accelerate lookups, Chord maintains additional routing information.
- Each node n' maintains a **routing table** with up to m entries (which is in fact the number of bits in identifiers), called *finger table*.
- The i^{th} entry in the table at node n contains the identity of the *first* node s that succeeds n by at least 2^{i-1} on the identifier circle.
- $s = \text{successor}(n + 2^{i-1})$.
- s is called **the i^{th} finger of node n** , denoted by $n.\text{finger}(i)$

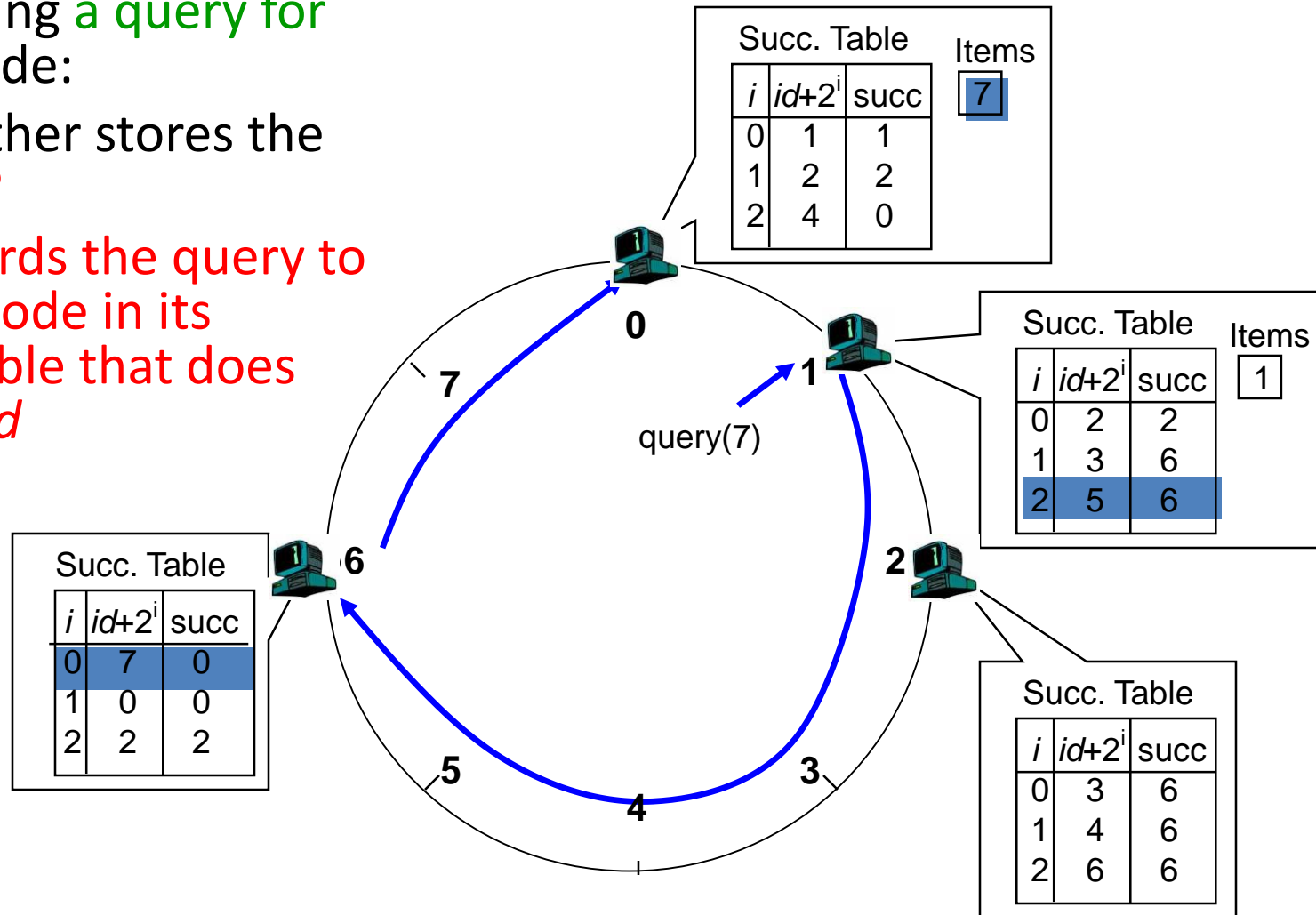
DHT: Chord “Finger Table”



- Entry i in the finger table of node n is the first node that succeeds or equals $n + 2^i$
- In other words, the i^{th} finger points $1/2^{n-i}$ way around the ring

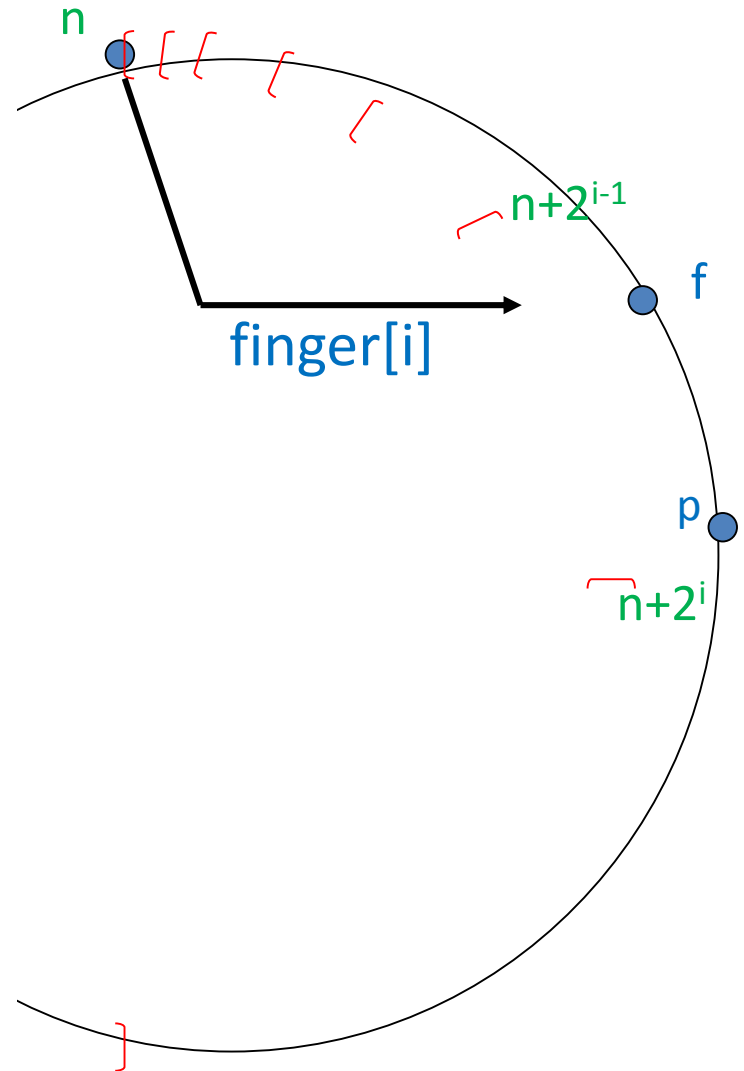
Chord Query Routing

- Upon receiving a query for item id , a node:
- Checks whether stores the item locally?
- If not, forwards the query to the largest node in its successor table that does not exceed id



Routing Time

- Node n looks up a key stored at node p
- p is in n 's i th interval:
 $p \in ((n+2^{i-1}) \bmod 2^m, (n+2^i) \bmod 2^m]$
- n contacts $f = \text{finger}[i]$
 - The interval is not empty so:
 $f \in ((n+2^{i-1}) \bmod 2^m, (n+2^i) \bmod 2^m]$
- f is at least 2^{i-1} away from n
- p is at most 2^{i-1} away from f
- The distance is **halved** at each hop.



Routing Time

- Assuming uniform node distribution around the circle, the number of nodes in the search space is halved at each step:
 - Expected number of steps: $\log N$
- Note that:
 - $m = 160$
 - For 1,000,000 nodes, $\log N = 20$

DATABASE FOUNDATIONS

The Transaction Concept

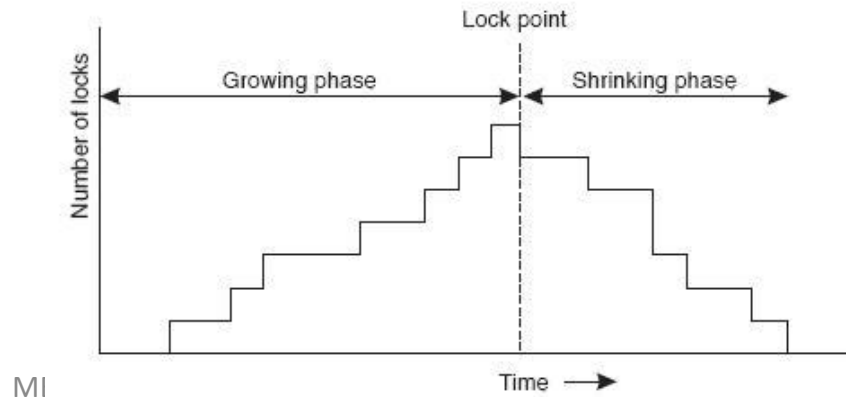
- Transactions were originally developed in the context of DBMS as a **paradigm** to deal with:
 - **Concurrent** access to **shared** data
 - **Failures** of different kinds/types.
- The key problem solved in an elegant manner:
 - Subtle and difficult issue of keeping **data consistent** in the presence of concurrency and failureswhile ensuring **performance, reliability, and availability**.

Preliminaries: A database

- A **database** consists of a set of objects.
- A **transaction** is a set of operations (typically read and write) executed in some partial order.
- Transaction execution must be **atomic**:
 - no interference among transactions.
 - Either **all** its operations are executed **or none**.
- **Concurrency control protocol** ensures that concurrent transactions do **not interfere** with each other.
- **Recovery protocol** ensures the **all or nothing** property.

Concurrency Control

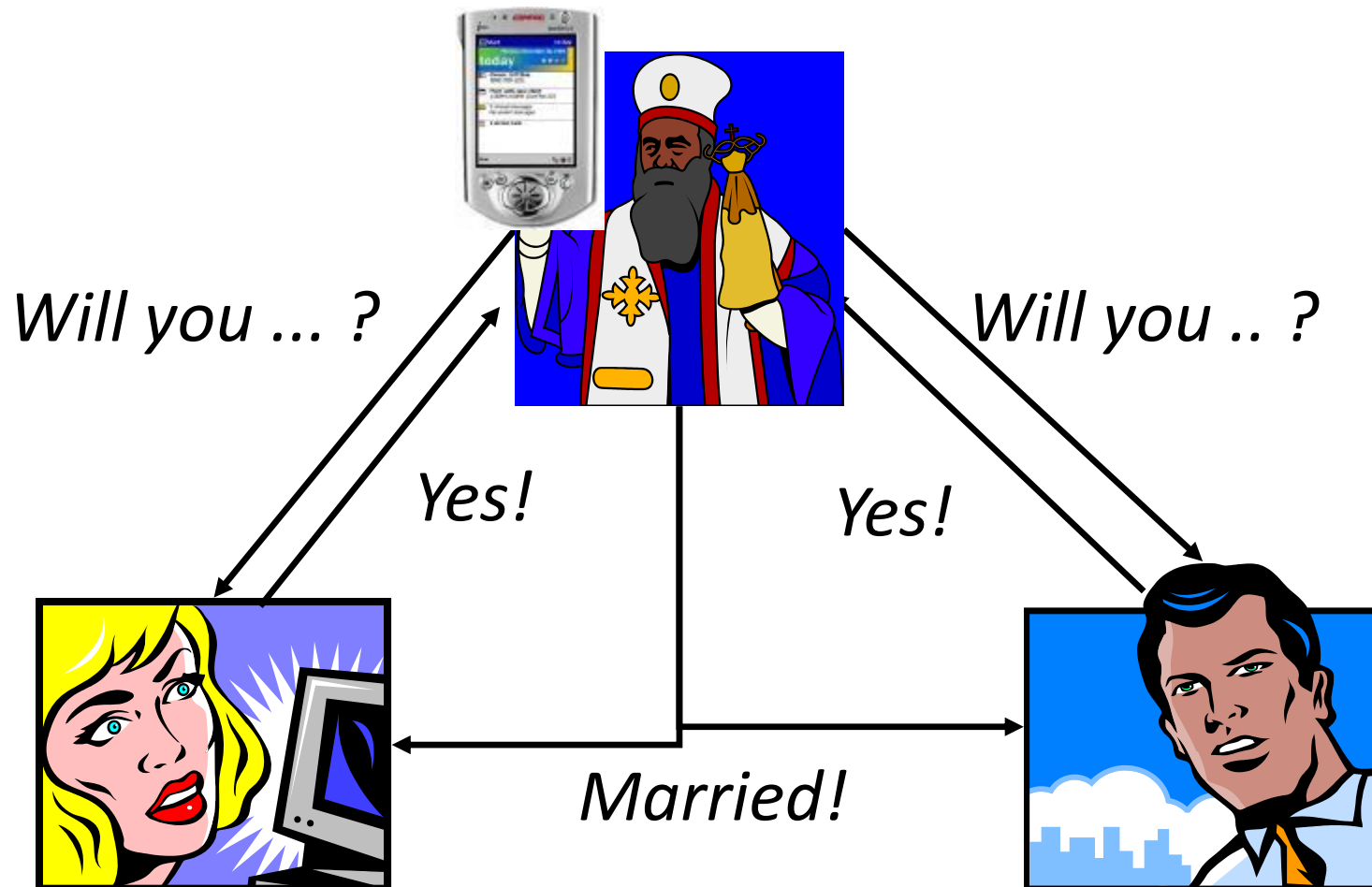
- A history is **serializable** if it is equivalent to a **serial history** over the same set of transactions.
- Different notions of serializability:
 - View Serializability: NP Complete ☹️
 - Conflict Serializability: **H is CSR iff SG(H) is acyclic**
- Two Phase locking.
 - Deadlock
- Optimistic CC



Atomic Commitment

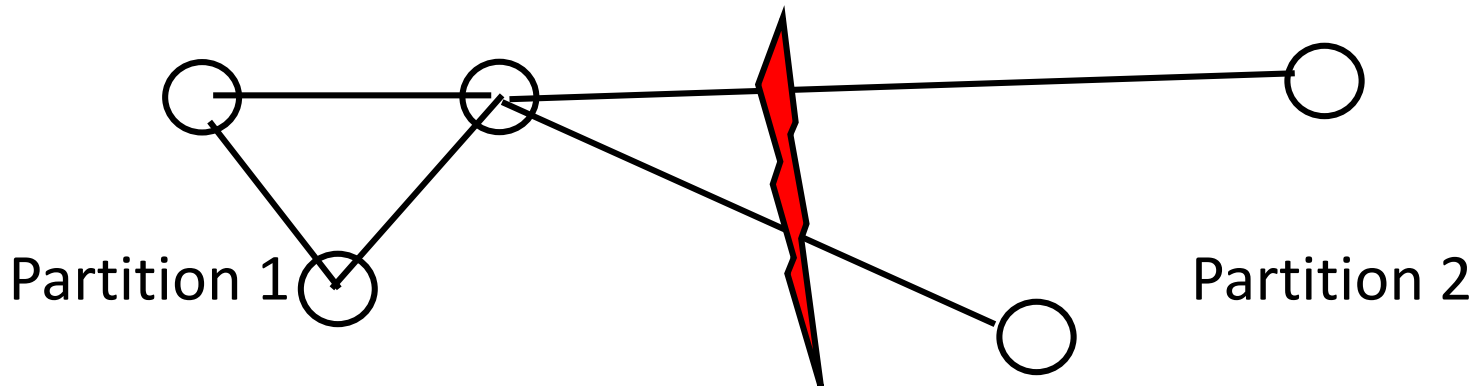
- Distributed handshake protocol known as **two-phase commit (2PC)**:
 - A **coordinator** (the **Transaction Manager**) takes the responsibility of unanimous decision:
 - **COMMIT** or **ABORT**
 - All database servers are the **cohorts** in this protocol and become dependent on the coordinator
- Considers a **synchronous** distributed system.

Idea: Getting Married over the NW



Commit Protocols

- What does a process do if it does not receive a message it is expecting?
 - It **BLOCKS**.
- **2 PC blocks** with site (and NW) failures
- **3PC is non-blocking** with **site** failures only.

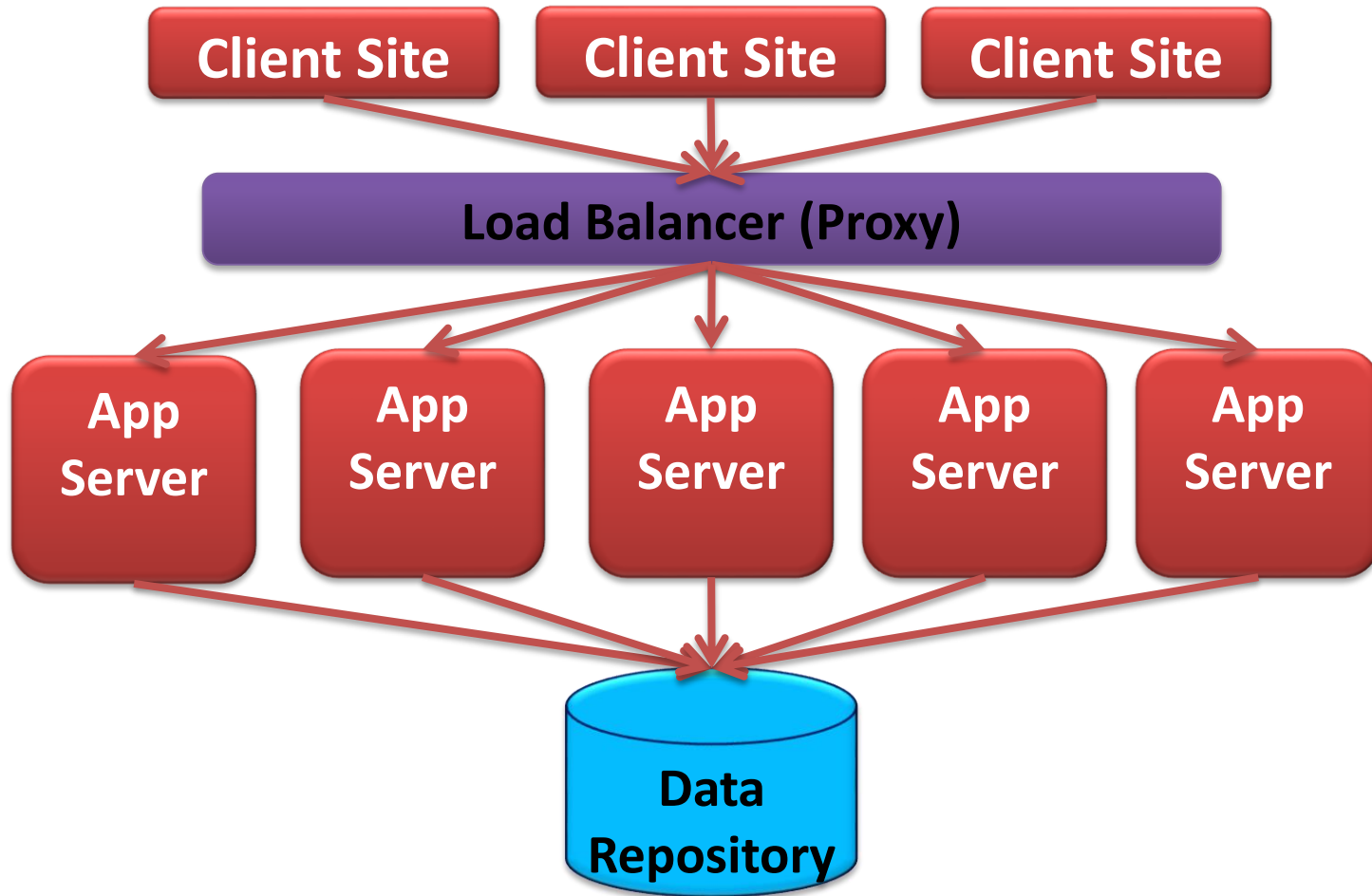


- **3PC blocks** with **partitioning** failures.
- **Theorem [Skeen83]: There is no non-blocking atomic commit protocol in the presence of partitioning failures.**

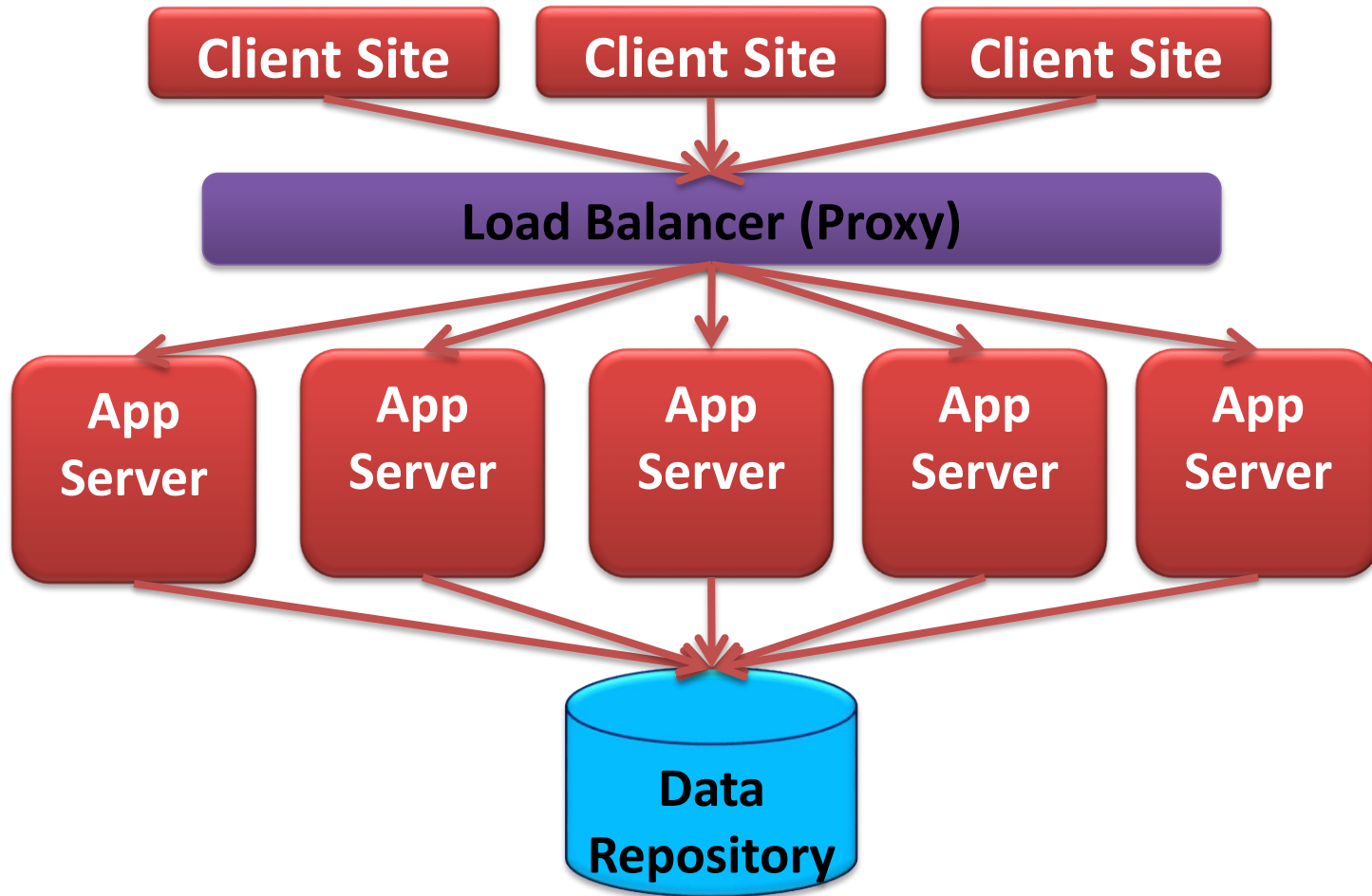
Cloud Reality: The Data Centers



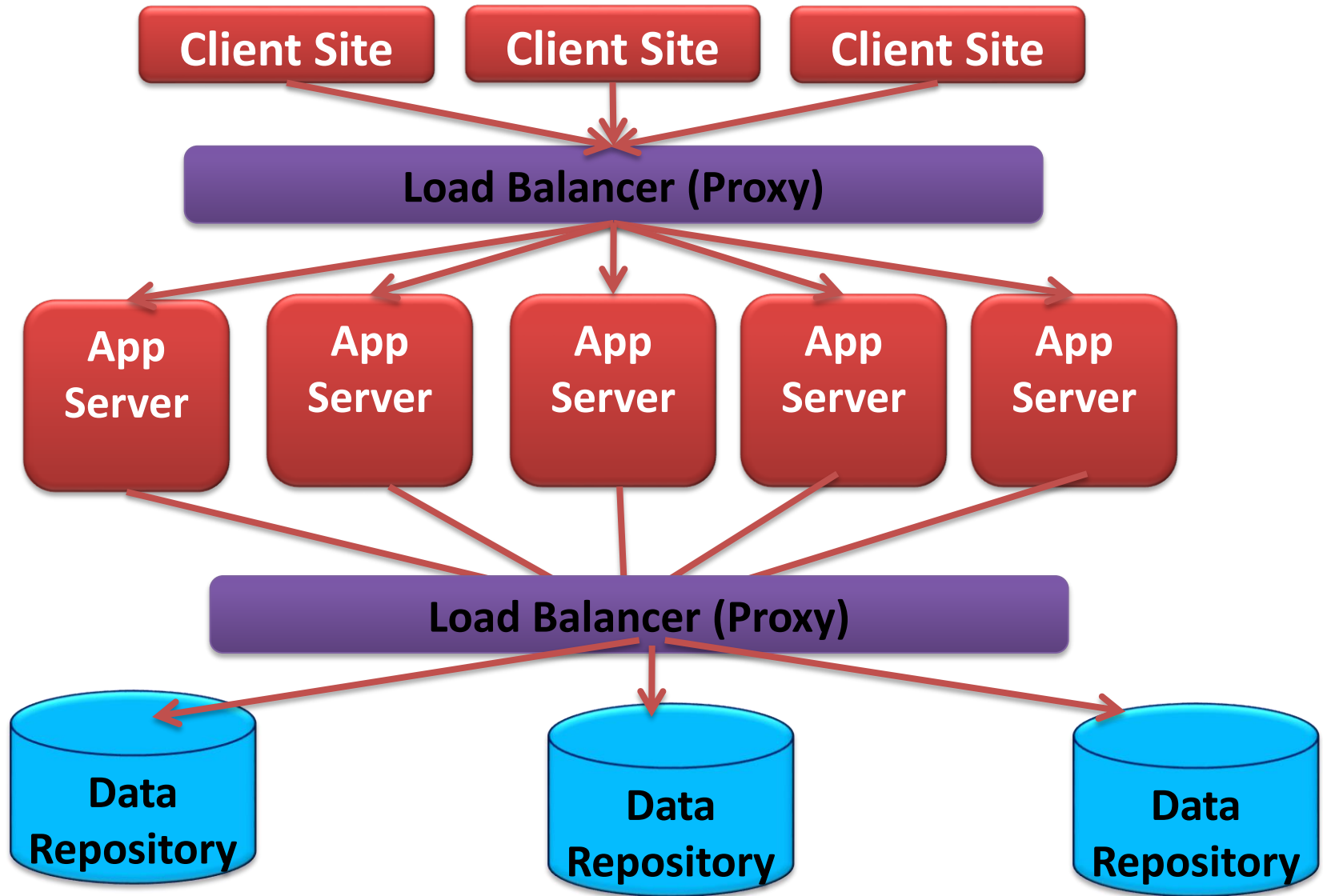
Scaling in the Cloud



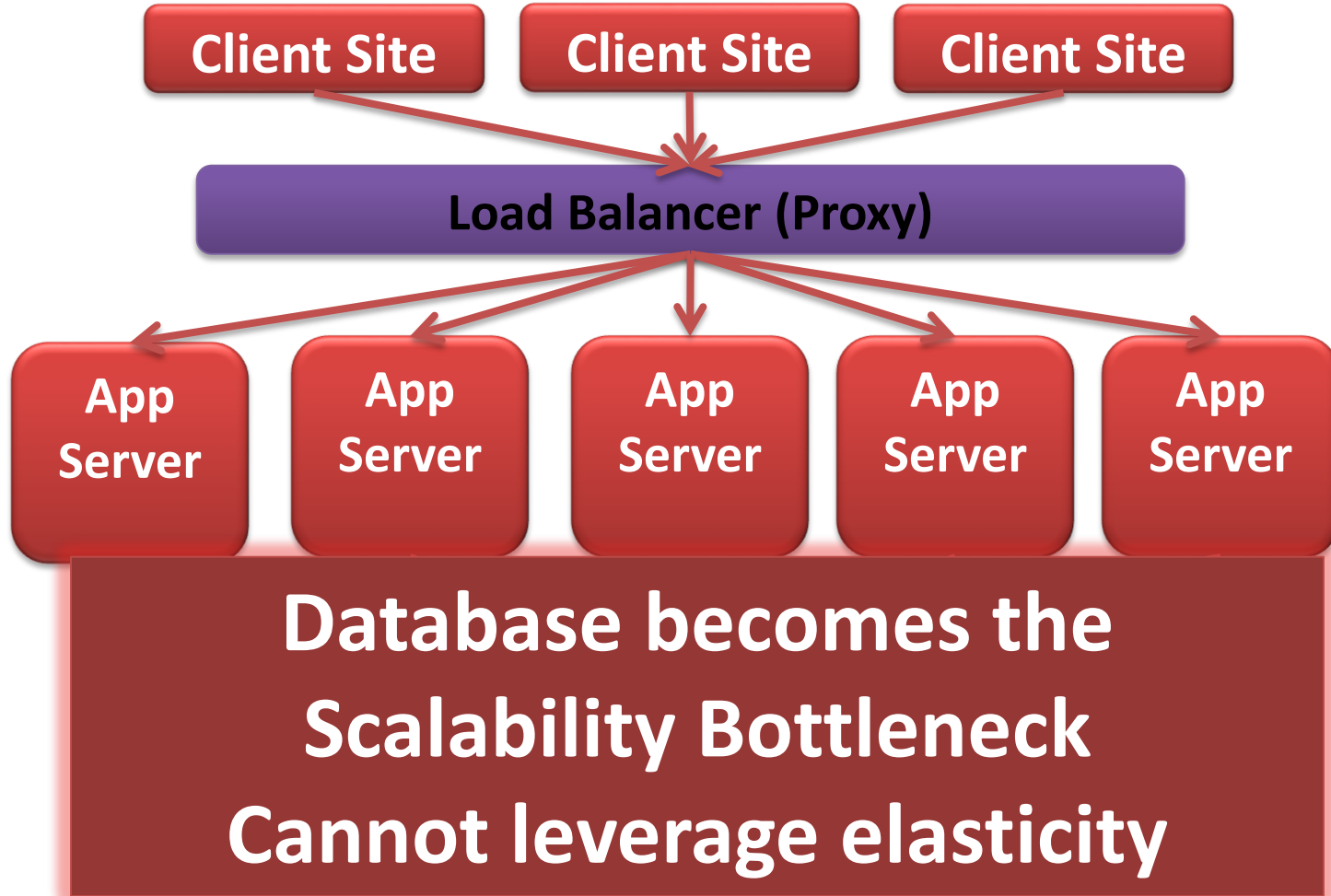
Scaling in the Cloud



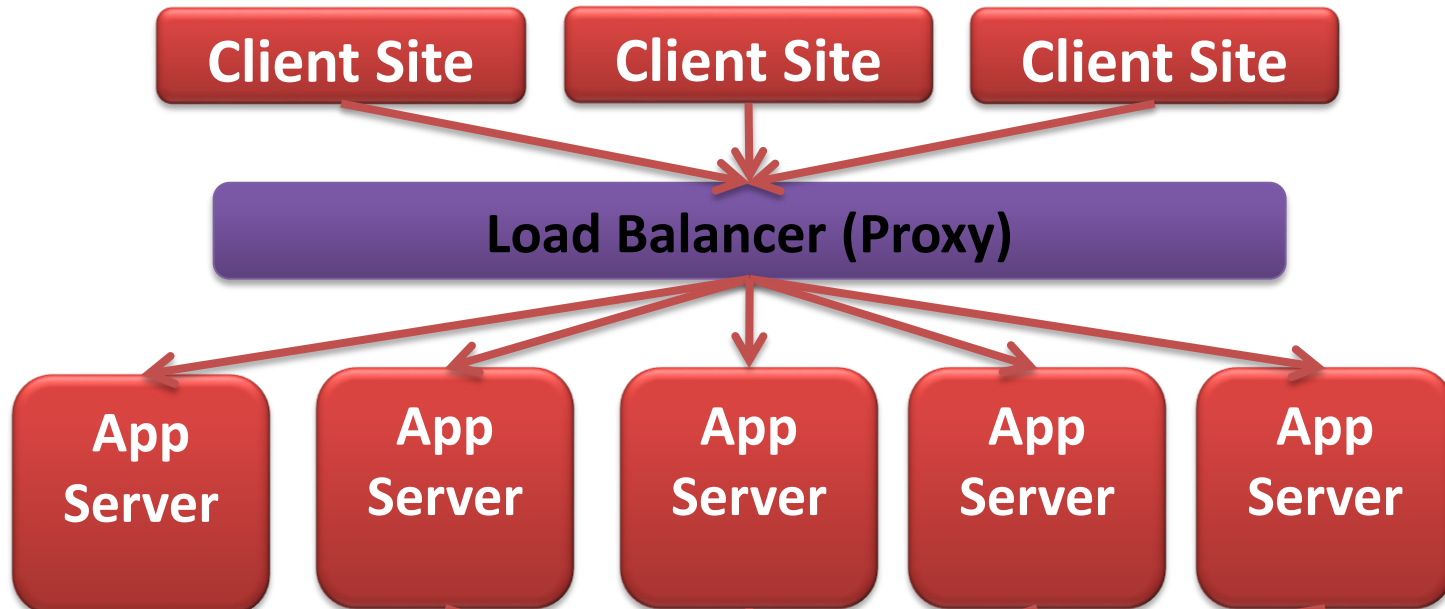
Scaling in the Cloud



Scaling in the Cloud



Scaling in the Cloud



**Scalable and Elastic,
but limited consistency and
operational flexibility**

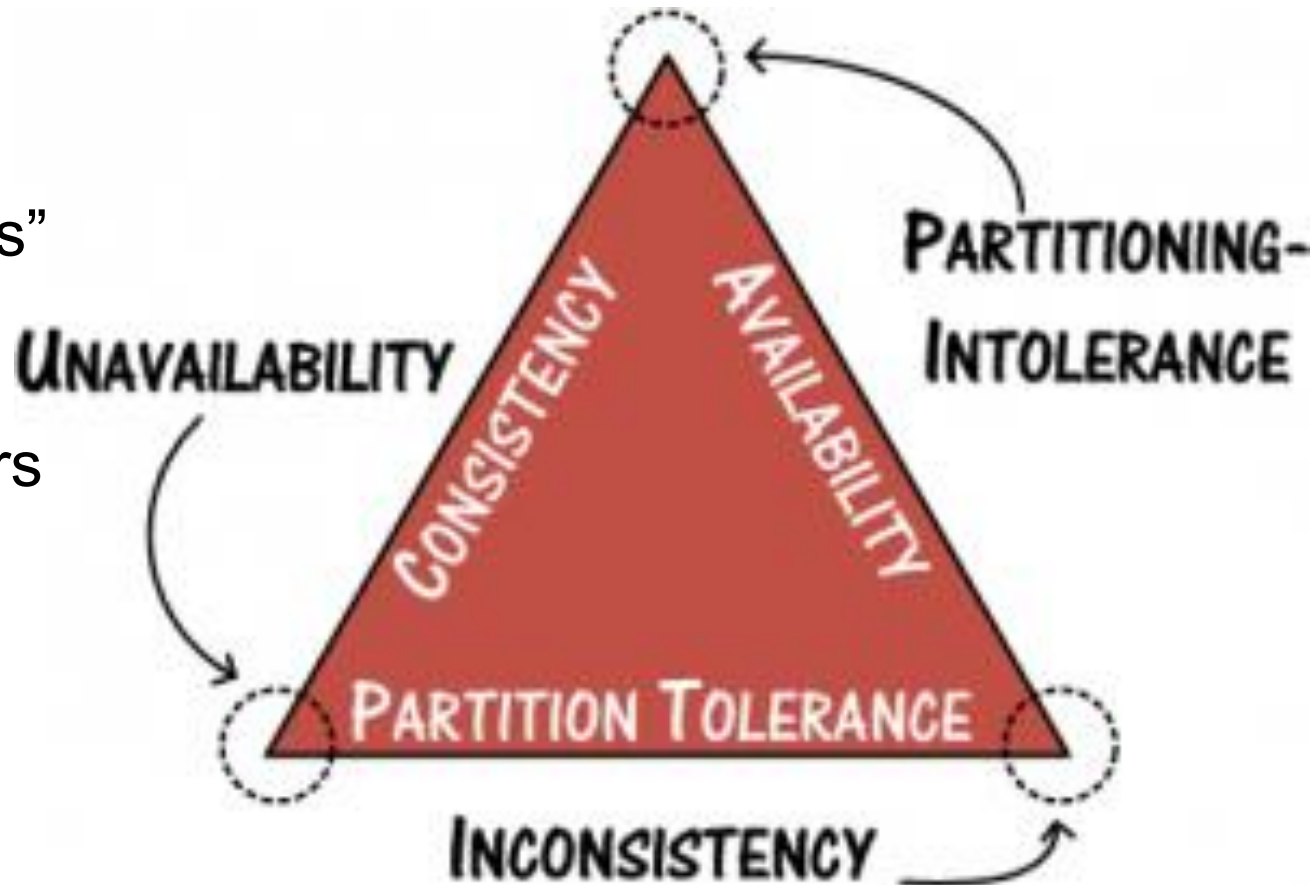
`{name: "mongo", type: "DB"}`

ndra

CAP Theorem (Eric Brewer)

- “Towards Robust Distributed Systems” PODC 2000.

- “CAP Twelve Years Later: How the “Rules” Have Changed” IEEE Computer 2012



The cover of the book 'NOSQL for Dummies' by Tobias Ivarsson. The title 'NOSQL' is in large white letters, with 'for Dummies' below it. The author's name 'Tobias Ivarsson' and 'Hacker @ Neo Technology' are at the bottom left. Contact information for Twitter, email, and website is at the bottom right. The Neo Technology logo is in the top right corner.

neotechnology

NOSQL

for Dummies

Tobias Ivarsson
Hacker @ Neo Technology

twitter: @thobe / #neo4j
email: tobias@neotechnology.com
web: <http://www.neo4j.org/>
web: <http://www.thobe.org/>

Key Value Stores



- Key-Valued data model
 - Key is the **unique identifier**
 - Key is the granularity for consistent access
 - Value can be **structured or unstructured**
- Gained widespread popularity
 - In house: **Bigtable** (Google), **PNUTS** (Yahoo!), **Dynamo** (Amazon)
 - Open source: **HBase**, **Hypertable**, **Cassandra**, **Voldemort**
- Popular choice for the modern breed of web-applications

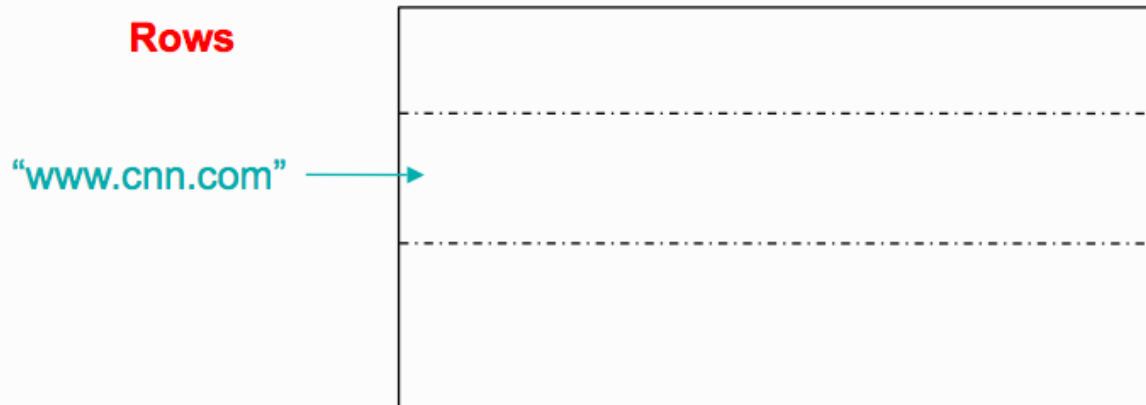
Big Table (Google)



- Data model.
 - Sparse, persistent, multi-dimensional sorted **map** indexed by a **row key**, **column key**, and a **timestamp**.
 - (row: byte[], column: byte[], time: int64) → byte[]
- Scalability and Elasticity: Data is **partitioned** across multiple servers.

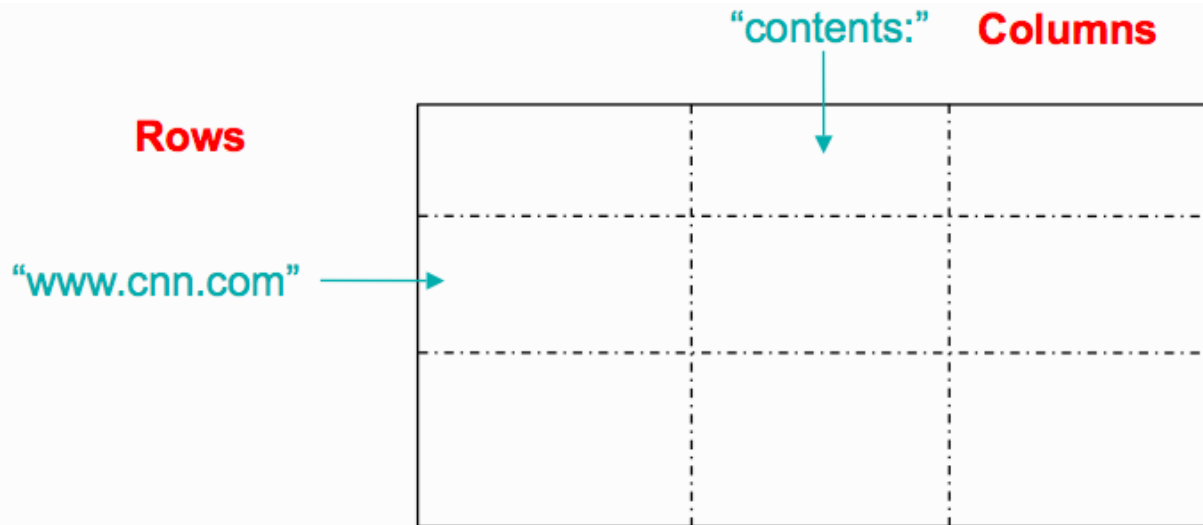
Data Model

- Row Key
 - Arbitrary string
 - Access to data in a row is atomic
 - Ordered lexicographically



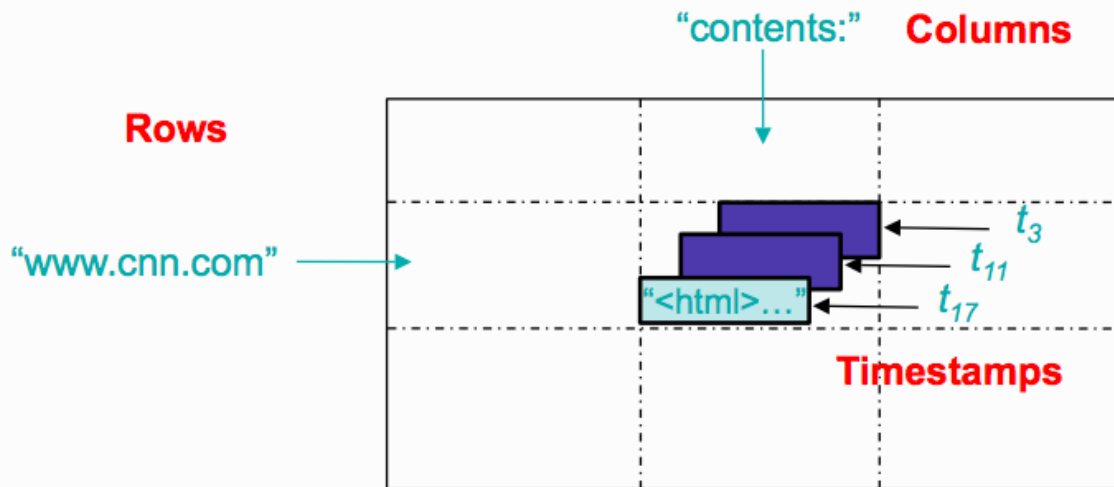
Data Model

- Column
 - Column-level name structure:
 - family: qualifier
 - Column Family is the unit of access control



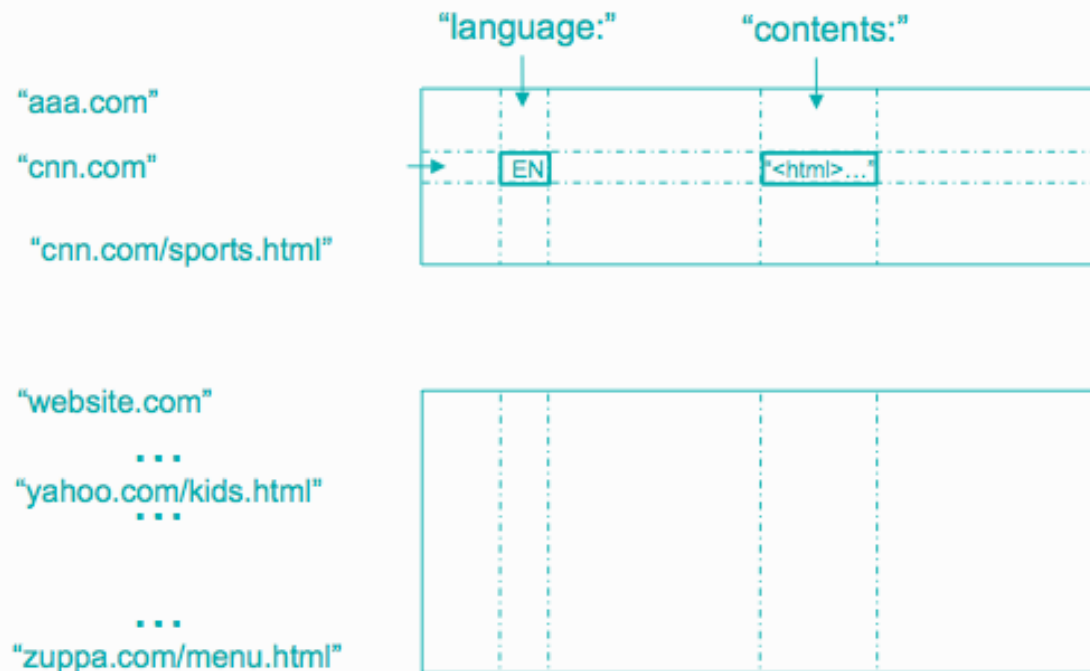
Data Model

- Timestamps
 - Store different versions of data in a cell
 - Lookup options
 - Return most recent K values
 - Return all values



Tablets

- The row range for a **table** is dynamically partitioned
- Each row range is called a **tablet**
- **Tablet is the unit for distribution and load balancing**
 - Each tablet is typically **100-200 MB** in size



Atomicity Guarantees in Key-Value Stores

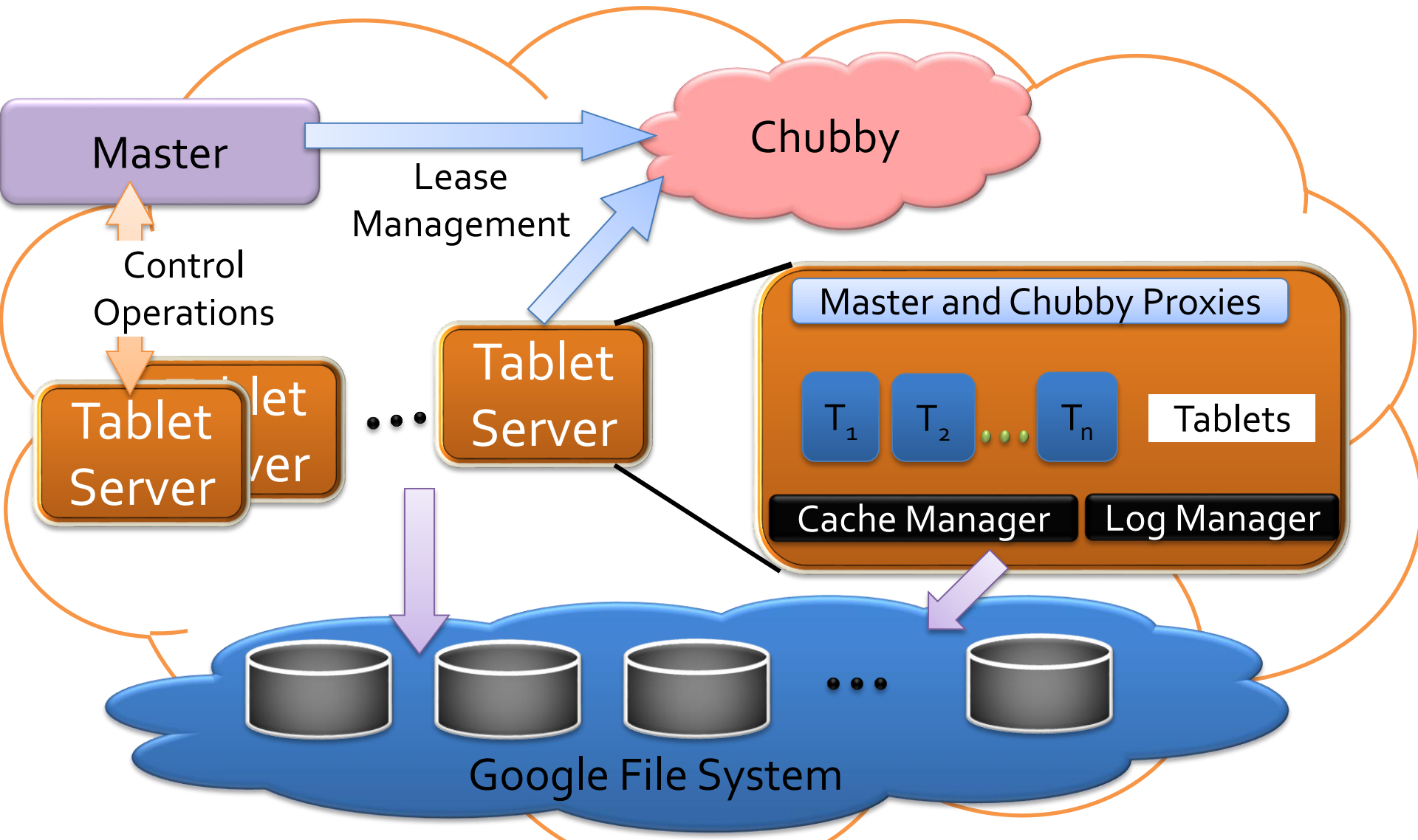
- Simple **PUT** and **GET** operations (+ atomic read-modify-write).
- Every operation on a **single row** and **is atomic**.

Big Table's Building Blocks

A **Bigtable cluster** stores a **number of tables**, each **table** consists of a **set of tablets** and a **tablet** contains a **row range**.

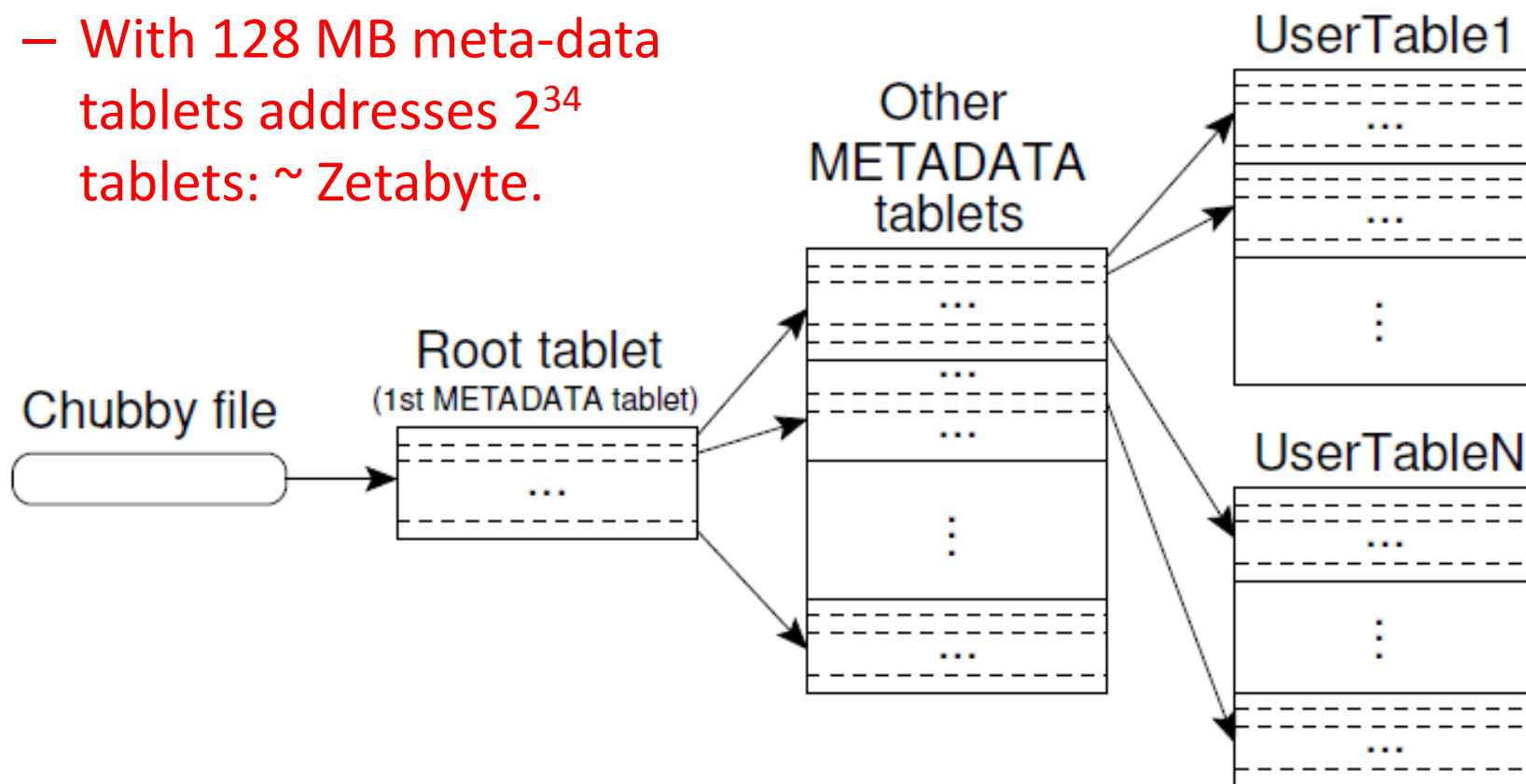
- **Tablet servers**
 - Handles **read and writes** to its tablet and **splits** tablets
 - Each tablet is typically **100-200 MB** in size
- **Master Server**
 - **Assigns** tablets to tablet servers and **Balances** tablet-server load
 - **Detects** the addition and deletion of tablet servers
- **Google File System (GFS)**
 - Highly available distributed file system: **stores** log and data files
- **Chubby**
 - Manage **meta-data**
 - Highly available persistent distributed **lock manager**

Overview of Bigtable Architecture



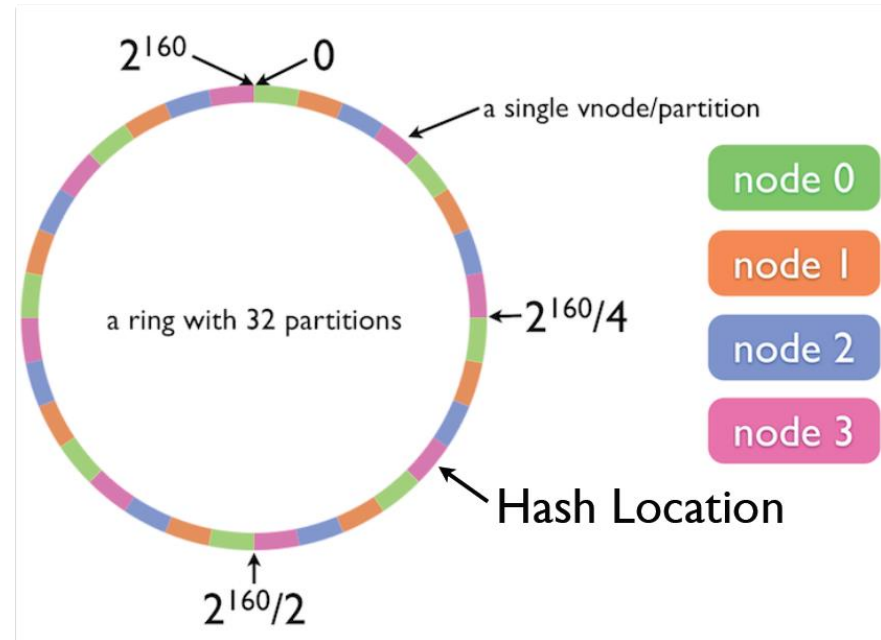
Tablet Location: 3 Level Hierarchy

- Each meta-data row stores ~ 1KB of data,
- With 128 MB meta-data tablets addresses 2^{34} tablets: ~ Zetabyte.



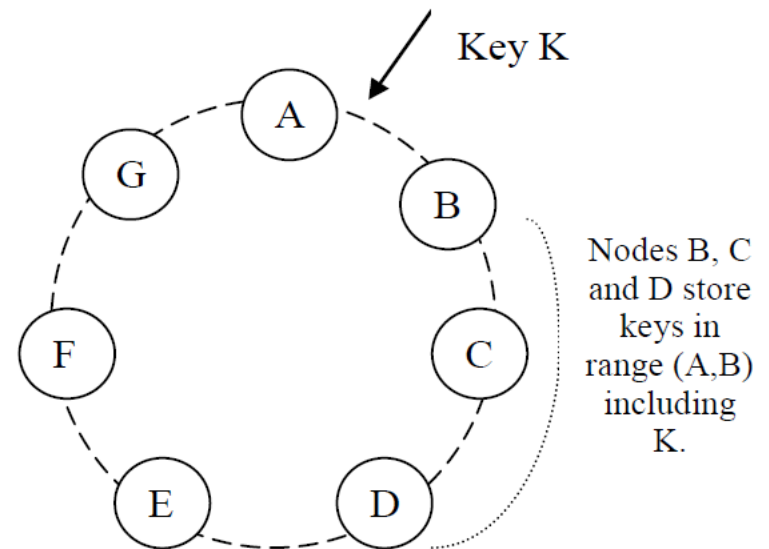
Dynamo (Amazon) and Cassandra (Facebook)

- **Consistent hashing**: the output range of a **hash function** is treated as a fixed circular space or “**ring**” a la Chord.
- “**Virtual Nodes**”: Each node can be **responsible for more than one virtual node** (to deal with non-uniform data and load distribution)



Replication

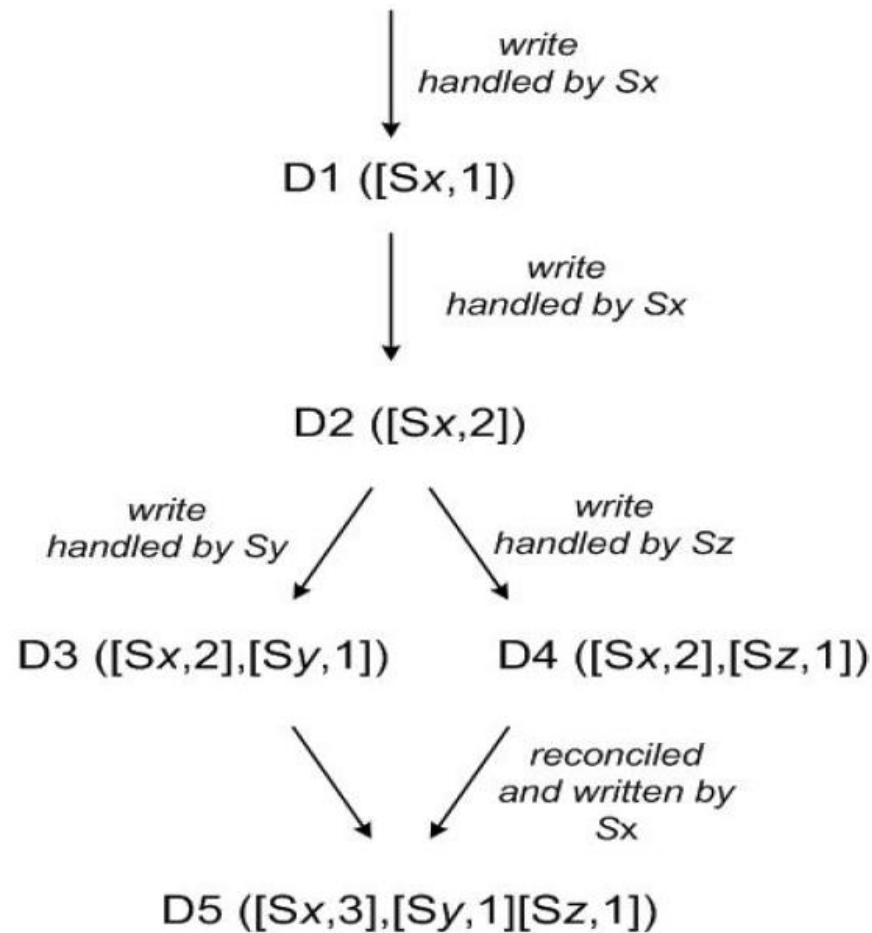
- Each data item is **replicated** at N hosts.
- *preference list*: The list of **nodes that is responsible for storing** a particular key.
- Due to **virtual** nodes, list skips positions to ensure **distinct physical nodes**.
- Due to **failures**, list contains **more than N nodes**.



Sloppy Quorum

- R and W is the **minimum number of nodes** that must participate in a successful **read/write operation**.
- Setting $R + W > N$ yields a **quorum-like system**.
- Operation latency dictated by the slowest of t replicas. For this reason, R and W are usually configured to be **less than N** , to provide **better latency** and **availability**.
- Use **vector clocks** in order to capture **causality** between different versions of same object
- **Application reconciles divergent versions** and collapses into a single new version.

Vector clock example

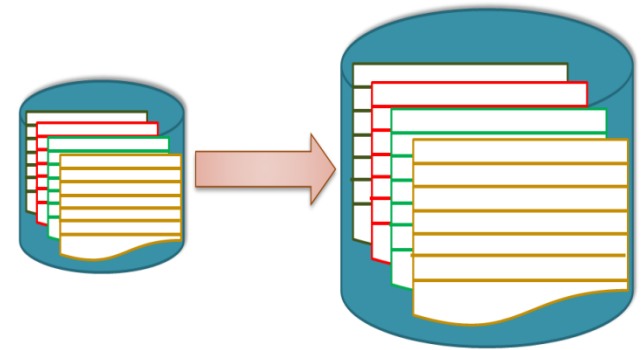


Practical approaches to scalability

Circa Year 2000.

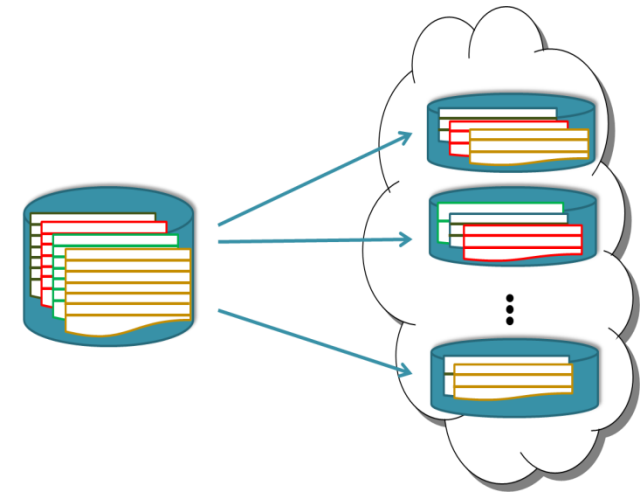
- **Scale-up**

- **Classical enterprise** setting (RDBMS)
- Flexible **ACID transactions**
- Transactions in a single node



- **Scale-out**

- **Cloud friendly** (Key value stores)
- Execution at a single server
 - Limited functionality & guarantees
- No **multi-row** or **multi-step** transactions



What about the Application Programmer?



Sacrificing Consistency

Transaction Solution

```
public void confirm_friend_request(user1, user2)
{
begin_transaction();
    update_friend_list(user1, user2, status.confirmed);
    //Palo Alto
    update_friend_list(user2, user1, status.confirmed);
    //London
end_transaction();
}
```

Sacrificing Consistency

Handle failures

```
confirm_friend_request_A(user1, user2) {  
  try {  
    update_friend_list(user1, user2, status.confirmed);  
  } catch(exception e) {  
    report_error(e);  
    return;  
  }  
  try {  
    update_friend_list(user2, user1, status.confirmed);  
  } catch(exception e) {  
    revert_friend_list(user1, user2);  
    report_error(e);  
    return;  
  }  
}
```

Sacrificing Consistency

Ensuring persistence

```
confirm_friend_request_B(user1, user2) {
  try{
    update_friend_list(user1, user2, status.confirmed);
  } catch(exception e) {
    report_error(e);
    add_to_retry_queue(operation.updatefriendlist, user1, user2,
current_time());
  }

  try {
    update_friend_list(user2, user1, status.confirmed);
  } catch(exception e) {
    report_error(e);
    add_to_retry_queue(operation.updatefriendlist, user2, user1,
current_time());
  }
}
```

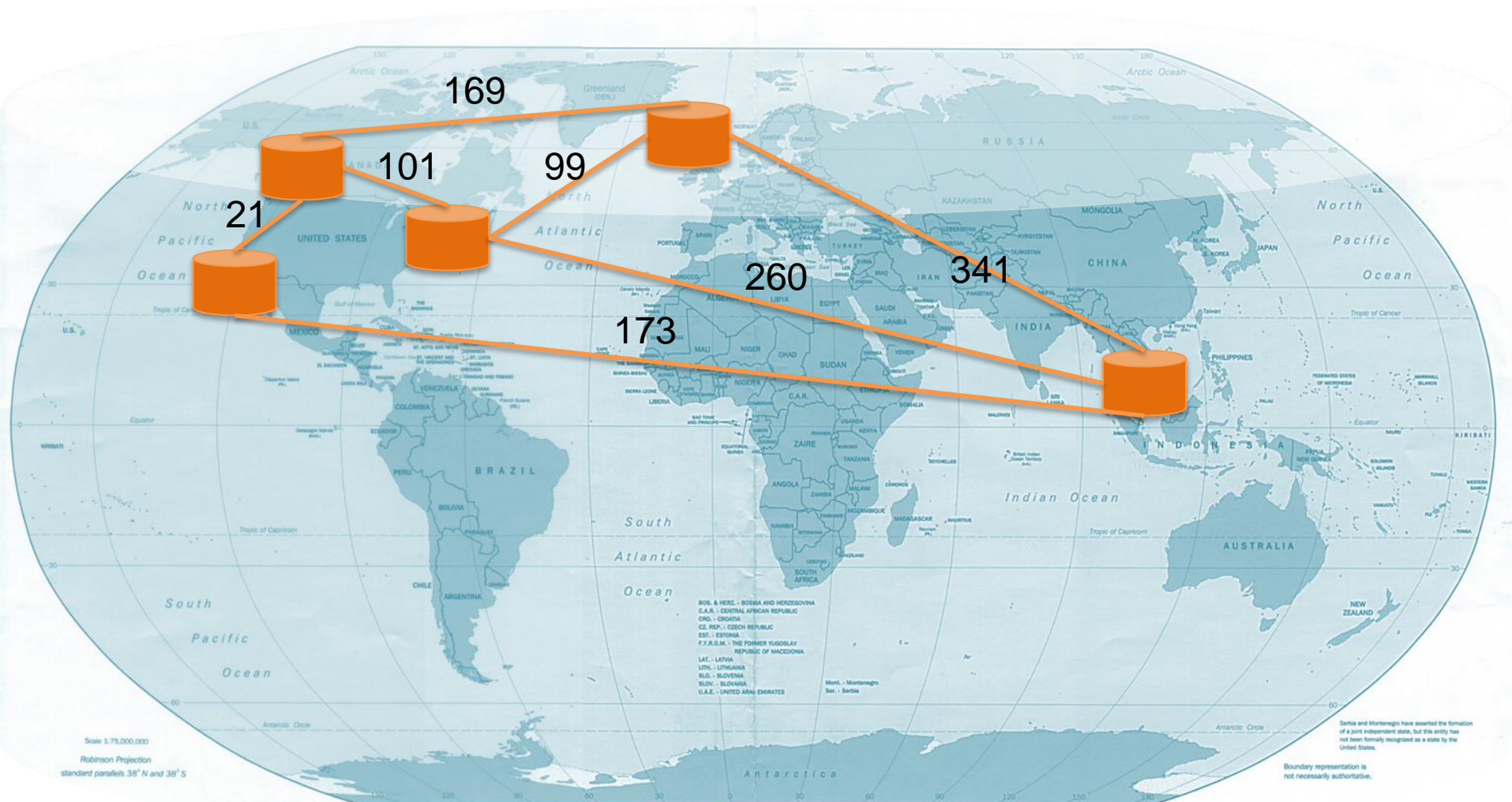
Sacrificing Consistency

Stale reads

```
/* get_friends() method has to reconcile results because there may be data inconsistency due to a change that was applied from the message queue to a subsequent change by the user. In this case, status are all conflicts are merged and it is up to app developer what to do. */
public list get_friends(user1){
    list friends = get_friends_list(user1);
    foreach (friend in friends){
        if(friend.status == friendstatus.confirmed){ //no conflict
            actual_friends.add(friend);
        }else if((friend.status &= friendstatus.confirmed) and !(friend.status &= friendstatus.deleted) ){ //assume friend is confirmed as long as it wasn't also deleted
            friend.status = friendstatus.confirmed;
            actual_friends.add(friend);
        }else{ //assume deleted
            update_friends_list(user1, friend, status.confirmed);
        }
    }
}
return actual_friends;
```

It gets too complicated with Eventual Consistency

Communication Overhead



Concluding Remarks



- The Cloud is the inevitable choice for hosting **Big Data Ecosystems**.
- Big Data Challenges:
 - Volume, Velocity and Variety
 - and PriVacy
- Cloud challenges:
 - Distribution and Consistency
 - Fault-tolerance Efficiency Privacy

