Data Serving Systems in Cloud Computing Platforms

Sudipto Das

eXtreme Computing Group (XCG),
Microsoft Research (MSR)

Day 1 Morning Session

Acknowledgements

- Prof. Divy Agrawal and Prof. Amr El Abbadi
  - Much of the material presented is joint work with them on tutorials, a book, and my PhD work
- Phil Bernstein
  - For the material on "Rethinking eventual consistency"
- Prof. Aoying Zhou for inviting me
- Summer school organizers
  - Xiaoling Wang, Han Lu
  - For all the help in making this last-minute draft possible

Outline of the Lecture (Day 1)

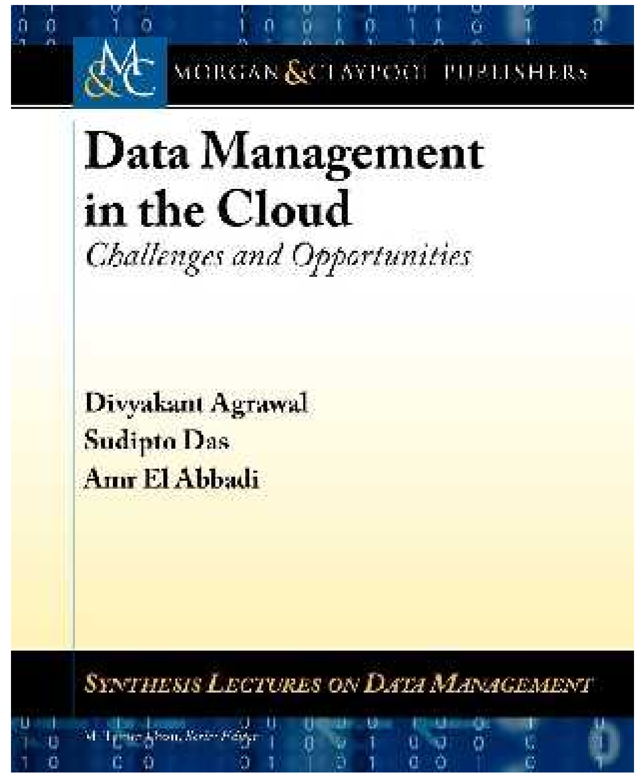
- Morning Session
  - Foundations of Database, P2P, and Distributed Systems
  - Key Value Stores – Design Principles
  - Key Value Stores – A survey of systems
- Afternoon Session
  - CAP and Rethinking Eventual Consistency
  - Scale-out Transaction Processing – Design Principles

Outline of the Lecture (Day 2)

- Morning Session
  - Transactions on co-located data – A survey of systems
  - Transactions on distributed data – A survey of systems
- Afternoon Session
  - Multi-tenant database systems – Design Principles
  - Database Elasticity
  - Performance Management in Multi-tenant Database Systems

Material Covered

- Structure loosely follows our book
- Many slides adapted from presentations from authors or relevant papers



Web replacing Desktop

Paradigm shift in Infrastructure

Windows Azure

rackspace HOSTING

amazon web services

NetApp

hadoop

Eucalyptus Systems

elastra THE ENTERPRISE CLOUD COMPANY

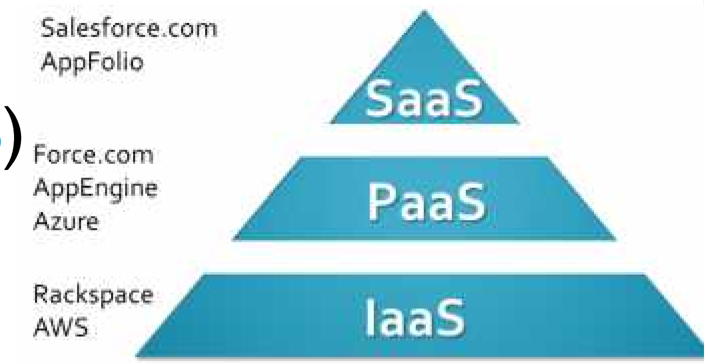
salesforce

cloudera

RIGHTSCALE

Cloud Computing

- Computing **infrastructure** and **solutions** delivered as a **service**
  - Industry worth USD150 billion by 2014*
- Contributors to success
  - Economies of scale
  - Elasticity and pay-per-use pricing
- Popular paradigms
  - Infrastructure as a Service (IaaS)
  - Platform as a Service (PaaS)
  - Software as a Service (SaaS)



*http://www.crn.com/news/channel-programs/225700984/cloud-computing-services-market-to-near-150-billion-in-2014.htm

Cloud Computing: History

> “If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as the telephone system is a public utility... The computer utility could become the basis of a new and important industry.”

—John McCarthy, speaking at the MIT Centennial in 1961[2]

Cloud Computing: Why Now?

- Experience with very large datacenters
  - Unprecedented economies of scale
  - Transfer of risk
- Technology factors
  - Pervasive broadband Internet and smartphones
  - Maturity in virtualization technology
- Business factors
  - Minimal capital expenditure
  - Pay-as-you-go billing model

Databases for Cloud Platforms

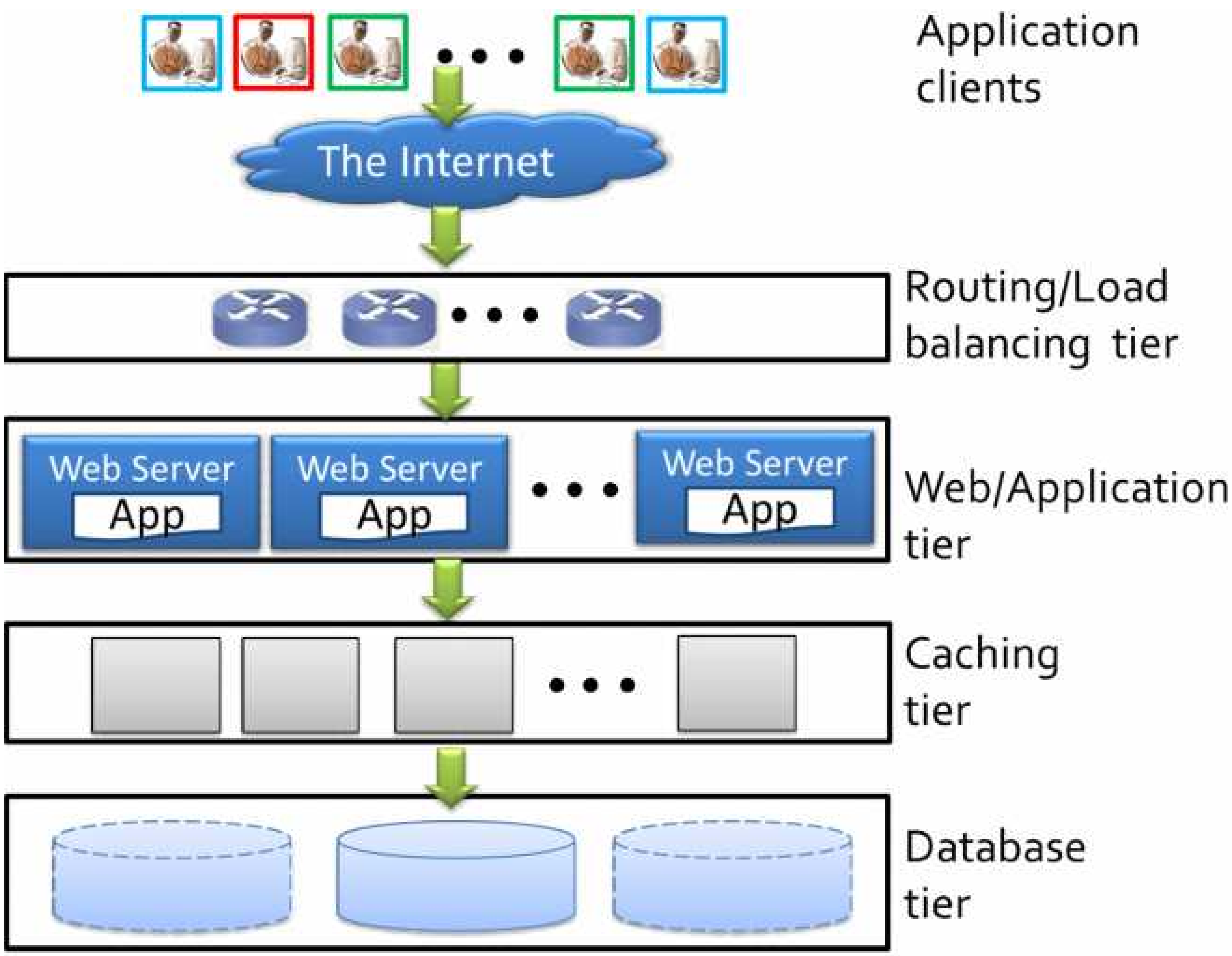
- **Data** is central to applications
- **DBMSs** are mission critical component in cloud software stack
  - Manage petabytes of data, drive revenue
  - Serve a variety of applications **(multitenancy)**
- Data needs for cloud applications
  - **OLTP systems**: store and serve data
  - **Data analysis systems**: decision support, intelligence

Application Landscape

- Social gaming
- Rich content and mash-ups
- Managed applications
- Cloud application platforms

Data Serving in the Cloud

What do I mean by "**data serving**"?



Challenges

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

FOUNDATIONS OF DATABASE, P2P, AND DISTRIBUTED SYSTEMS

Outline

- Transaction Processing Systems
  - Concurrency control
  - Recovery
- Distributed Systems
  - Logical times and Clocks
  - Leader election
  - The consensus problem
- P2P Systems
  - Consistent hashing & DHTs

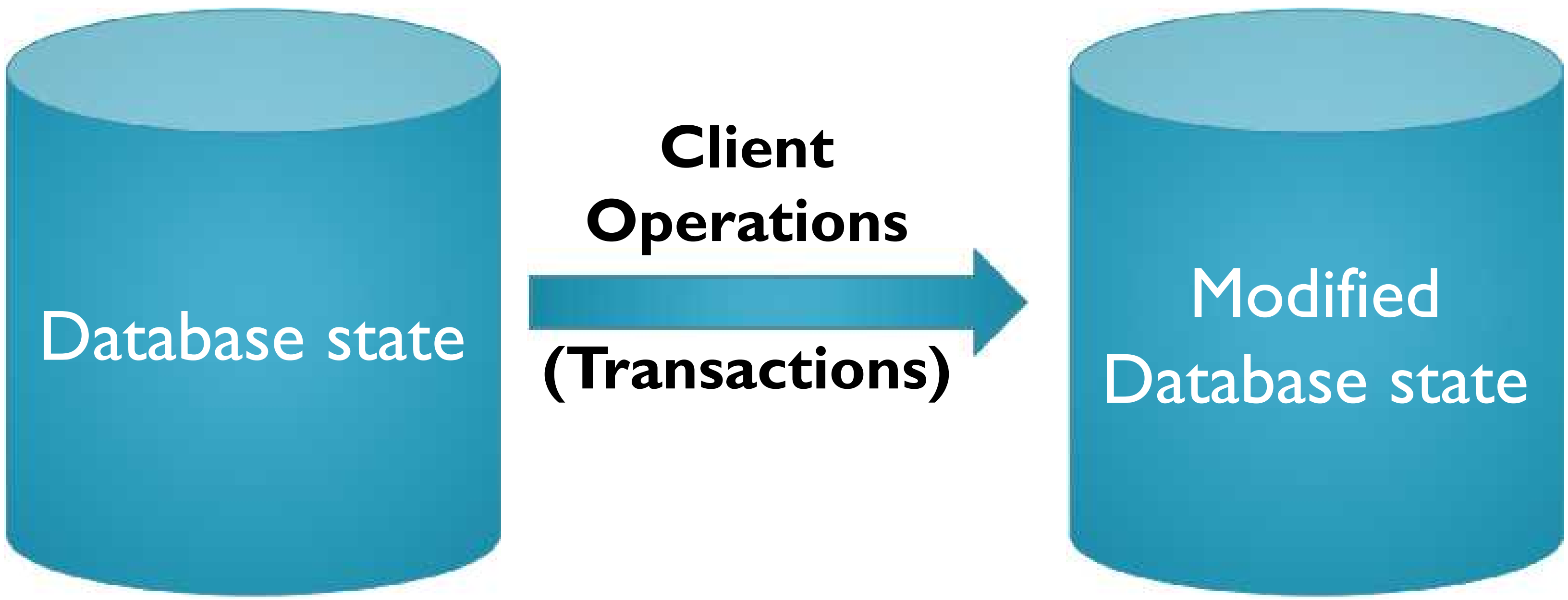
CONCEPTS OF TRANSACTION PROCESSING IN RDBMS

Data Management Evolution

- RDBMS (Relational Data Base Management Systems) became highly successful:
  - Widely adopted by both large and small business entities
- Enterprises became increasingly reliant on databases
- Primarily used for day-to-day operations:
  - Banking operations
  - Retail operations
  - Travel industry

Data Management Evolution

- Typically:
  - Database modeled the state of the application
  - Client operations were applied to update the state.



The OLTP Paradigm

- On-line Transaction Processing:
  - Database state is up-to-date at all times
- Significant challenges:
  - Multiple users/clients need to be supported
  - Handle hardware and software failures
- Emergence of what is now commonly referred to as:
  - The Transaction Concept

The Transaction Concept

- Multiple online users
  - Gives rise to the **concurrency problem**
- Component unreliability:
  - Gives rise to the **failure problem**
- Problems in the context of managing persistent data
  - Online transaction processing system (OLTP)

OLTP Example: Debit/Credit

```
void main ( ) {
  EXEC SQL BEGIN DECLARE SECTION
    int BAL, AID, amount;
  EXEC SQL END DECLARE SECTION;

  scanf (“%d %d”, &AID, &amount);  /* USER INPUT */

  EXEC SQL Select Balance into :BAL From Account
           Where Account_Id = :AID;  /* READ FROM DB */

  BAL = BAL + amount; /* update BALANCE  in memory*/

 EXEC SQL Update Account
     Set Balance = :b Where Account_Id = :AID;  /* WRITE INTO DB*/
  EXEC SQL Commit Work;
}
```

OLTP Example: A Social App

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

The Transaction Concept

- Developed 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 failures

  while ensuring performance, reliability, and availability

Preliminaries: Transactions

- A **transaction** is a set of operations executed in some partial order

- A transaction is assumed to be correct, i.e., if executed alone on a consistent database, it transforms it into another consistent state

- Example: $r_1[x]\ r_1[y]\ w_1[x]\ w_1[y]$ is an example of a transaction t_1 that transfers some amount of money from account x to account y

Correctness Requirements: ACID

- ATOMICITY:
  - All-or-none property of user programs
- CONSISTENCY
  - User program is a consistent unit of execution
- ISOLATION
  - User programs are isolated from the side-effects of other user programs
- DURABILITY:
  - Effects of user programs are persistent forever

Concurrency Control and Correctness

- Goal:
  - A technique/algorithm/scheduler that prevents incorrect or bad execution.
- Develop the notion of correctness – or characterize what does correct execution means

Serial History

- A history H is serial if for any two transactions T_i and T_j in H, all operations of T_i are ordered in H before all operations of T_j or vice-versa

- Example:
  - $r_1(x)\ r_1(z)\ w_1(x)\ c_1\ r_2(x)\ w_2(y)\ c_2\ r_3(z)\ w_3(y)\ w_3(z)\ c_3$

General Idea for Correctness

- Equivalence of two histories H_1 and H_2.
- Use this notion of equivalence to accept all histories which are "equivalent" to some serial history as being correct.
- How to establish this equivalence notion?

Serializability

A history is **serializable** if it is equivalent to a serial history over the same set of transactions.

Conflicts and Serializability

- Operations on different objects do not conflict.
- Reads on the same object do not conflict:
  - $R_1[x]\ R_2[x] = R_2[x]\ R_1[x]$
- Operations on the same object, and at least one of them is write **conflict**:
  - $R_1[x]$ and $W_2[x]$, or
  - $W_1[x]$ and $W_2[x]$

Concurrency Control Variants

- Locking
- Timestamp Ordering
- Optimistic Concurrency Control

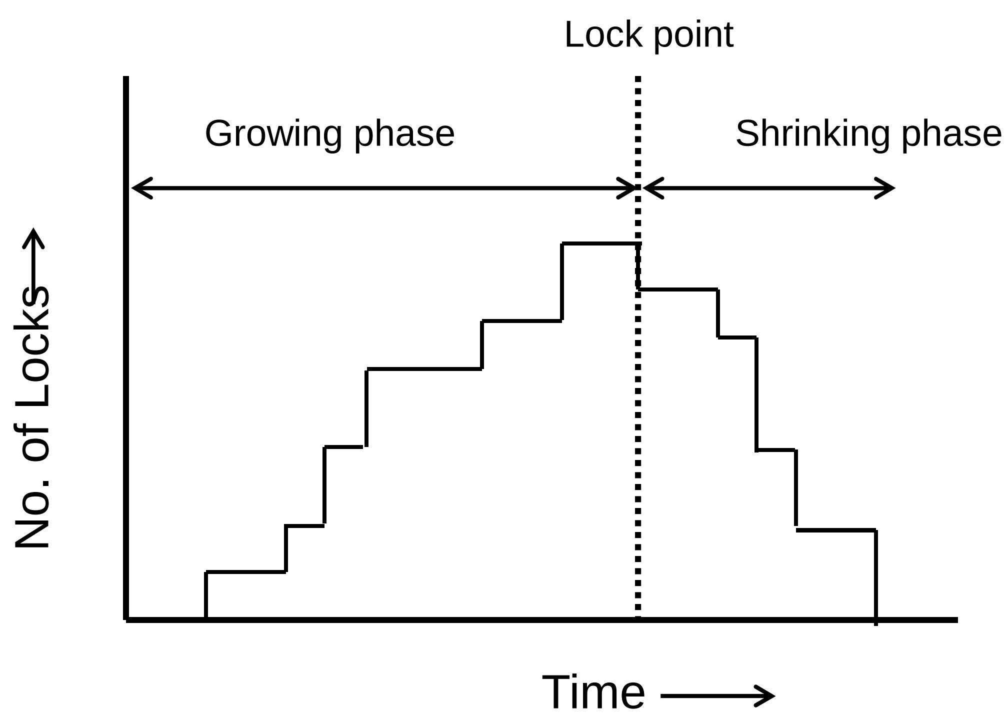
Locking Protocol

- For each step the scheduler **requests a lock** on behalf of the step's transaction.
- Each lock is requested in a specific **mode**
  - **read or write**
- If the data item is not locked in an **incompatible** **mode** the lock is **granted**;
- Otherwise there is a **lock conflict** and the transaction becomes **blocked** (suffers a **lock wait**) until the current lock holder **releases the lock**.

Two Phase Locking Protocol

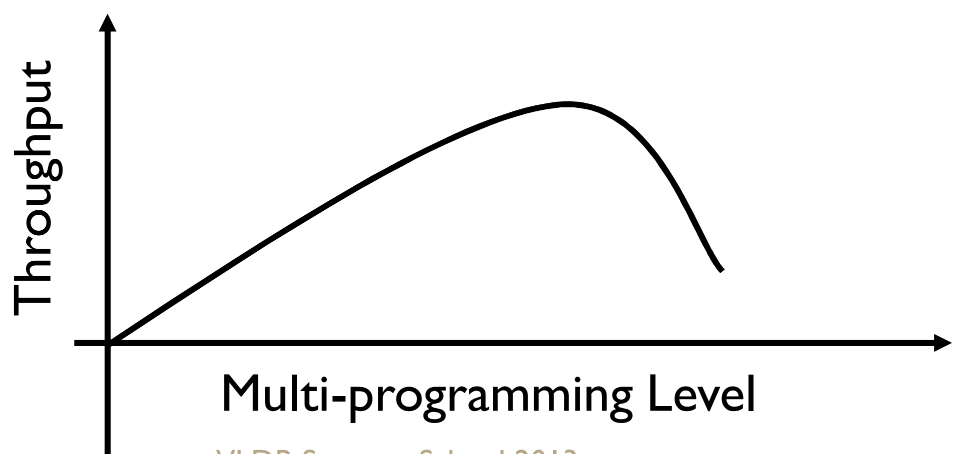
- The 2PL protocol:
  1. On $p_i[x]$, if $pl_i[x]$ conflicts delay it otherwise set $pl_i[x]$.
  2. Once the scheduler has set $pl_i[x]$ it may not release it until the Data Manager has acknowledged processing of $p_i[x]$.
  3. **Once the scheduler has released a lock for a transaction, it may not subsequently obtain any more locks for that transaction (on any data item).**

Two Phase Locking Protocol



Locking Performance

- In a multiprogramming system, resource contention arises over memory, processors, I/O channels, etc.
- In a locking system, data contention arises due to queues , which form due to conflicting operations.
- Locking can cause thrashing



Timestamp Ordering

- Associate with each transaction a **timestamp**.
- The CC protocol orders conflicting operations according to timestamp order.
  - If $p_i[x]$ and $q_j[x]$ are conflicting operations, then p_i is executed before q_j if $time(t_i) < time(t_j)$.
- Every object maintains: *max_read* and *max_write*.
- Read: if $time(t_i)$ < max_write
  - reject read
- Write: if $time(t_i)$ < max_read or $time(t_i)$ < max_write
  - reject write.
- Update max_read and max_write appropriately

Simple Optimistic CC

- Locking may block resources for long periods
- Simple Certification Approach:
  - Immediately execute all operations of t_1.
  - At commit, check if any active transaction has executed a conflicting operation, if so, abort t_1.
  - Proof Idea: if $t_1 \rightarrow t_2$ then t_1 certified before t_2.
- Most famous is optimistic concurrency control protocol by Kung and Robinson.

Kung and Robinson's OCC

- Transactions execute in 3 phases:
  - Read phase: unrestricted reading of any object, writes are local
  - Validation phase: ensure that no conflicts occurred.
  - Write phase: after successful validation, write values in db.
- Validation of transaction t_1:
  - Check all concurrent transactions t_2, i.e., the write phase of t_2 overlaps with read phase of t_1:
    - if readset (t_1) overlaps with writeset (t_2) then **abort** t_1.
- Further optimizations have been explored.

Pragmatic Considerations

- 2PL very popular but imposes significant constraints:
  - High synchronization overhead
  - Not enough concurrency
  - Read-only transactions blocked
- Multi-version Databases:
  - Read-only transaction incur no synchronization
  - Some flexibility in scheduling write operations
- In practice:
  - Multi-version concurrency control
  - Weaker forms of isolation
    - Snapshot isolation weaker than serializable isolation
    - Read committed

Recovery

- When a transaction aborts, the system must wipe out all its effects:
  - on data: use before images
  - on transactions: cascading aborts.
- Consider:

  $w_1[x,2]\ \ r_2[x]\ \ w_2[y,3]\ c_2\ \ a_1$
- What do we do? Semantic dilemma!
- Solution: Only allow recoverable histories.
- A history is **recoverable** if whenever t_j reads-x-from t_i, $c_i < c_j$.

Goal of Crash Recovery

Failure-resilience:

- **redo** recovery for committed transactions
- **undo** recovery for uncommitted transaction

Failure model:

- soft (no damage to secondary storage)
- fail-stop captures most (server) software failures

Requirements:

- fast restart for high availability
    - MTTF / (MTTF + MTTR)
- low overhead during normal operation
- simplicity, testability, very high confidence in correctness

Examples

- Server fails once a month, recovery takes 2 hours

  → 720/722 = 0.997

  i.e., server availability is 99.7%

  server is down 26 hours per year
- Server fails every 48 hours, but can recover within 30 sec

  → 172800/172830 = 0.9998

  i.e., server availability is 99.98%

  server is down less than 2 hours per year
- Fast recovery is essential, not just long uptime!

Actions During Normal Operation

All of the following actions are "tagged" with unique, monotonically increasing **sequence numbers**

Transaction actions:

- *begin (t)*
- *commit (t)*
- *abort (t)*

Data actions:

- *read (pageno, t)*
- *write (pageno, t)*

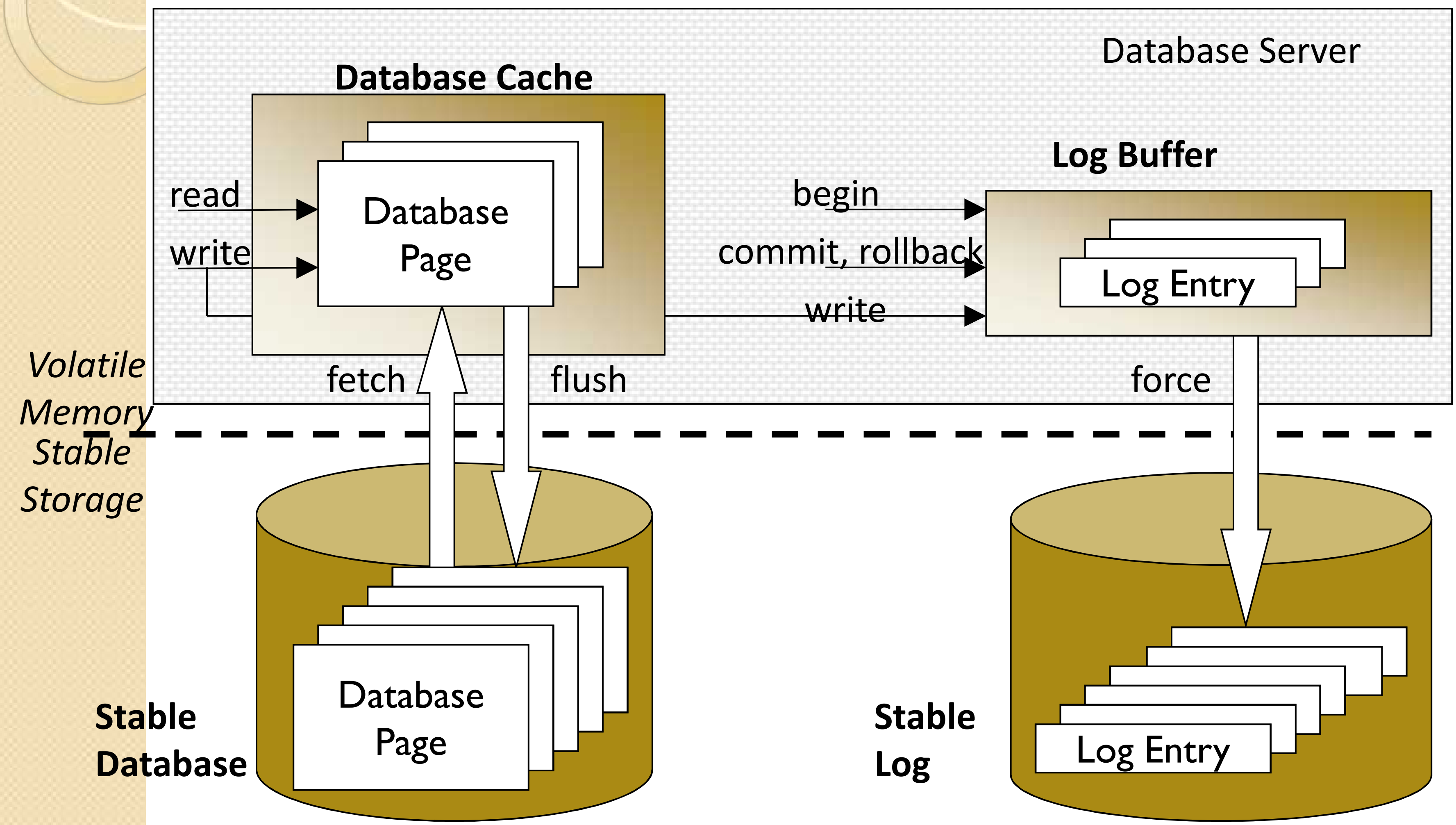
Caching actions:

- *fetch (pageno)*
- *flush (pageno)*

Log actions:

- *force ( )*

Overview of System Architecture



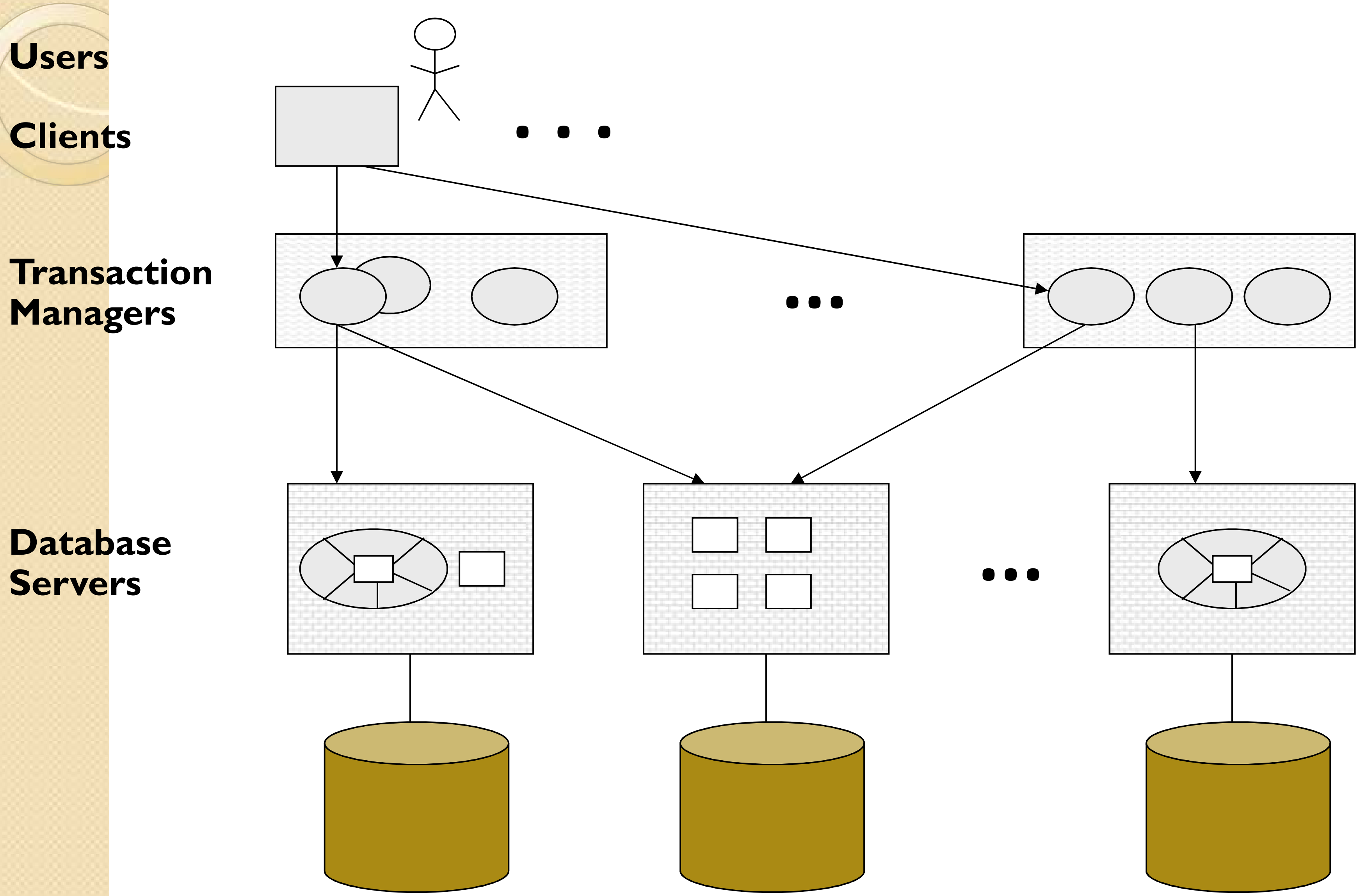
Logging Rules

- During normal operation, a recovery algorithm satisfies
  - the **redo logging rule**
    - if for every committed transaction T, all data actions of T are in the stable log or the stable database
  - the **undo logging rule**
    - if for every data action p of an uncommitted transaction T, the presence of p in the stable database implies that p is in the stable log

Centralized Recovery

- We need to recover disk failures during transaction execution so as to ensure the all or nothing property.
- 3 Approaches:
  - Shadow paging: 2 copies of database.
  - Before images: store on disk log of before values and update database immediately. If failure occurs and transaction has not committed restore db based on log.
  - After images: Perform updates in a log of after images. If transaction commits, install values in db from log.

Distributed Transactions



Concurrency Control Protocols

- Any of the centralized solutions can be extended for the distributed setting:
  - Distributed Two-phase Locking
  - Distributed Timestamp Ordering
  - Distributed Optimistic Protocols

- Every database server runs the same instance of the protocol.

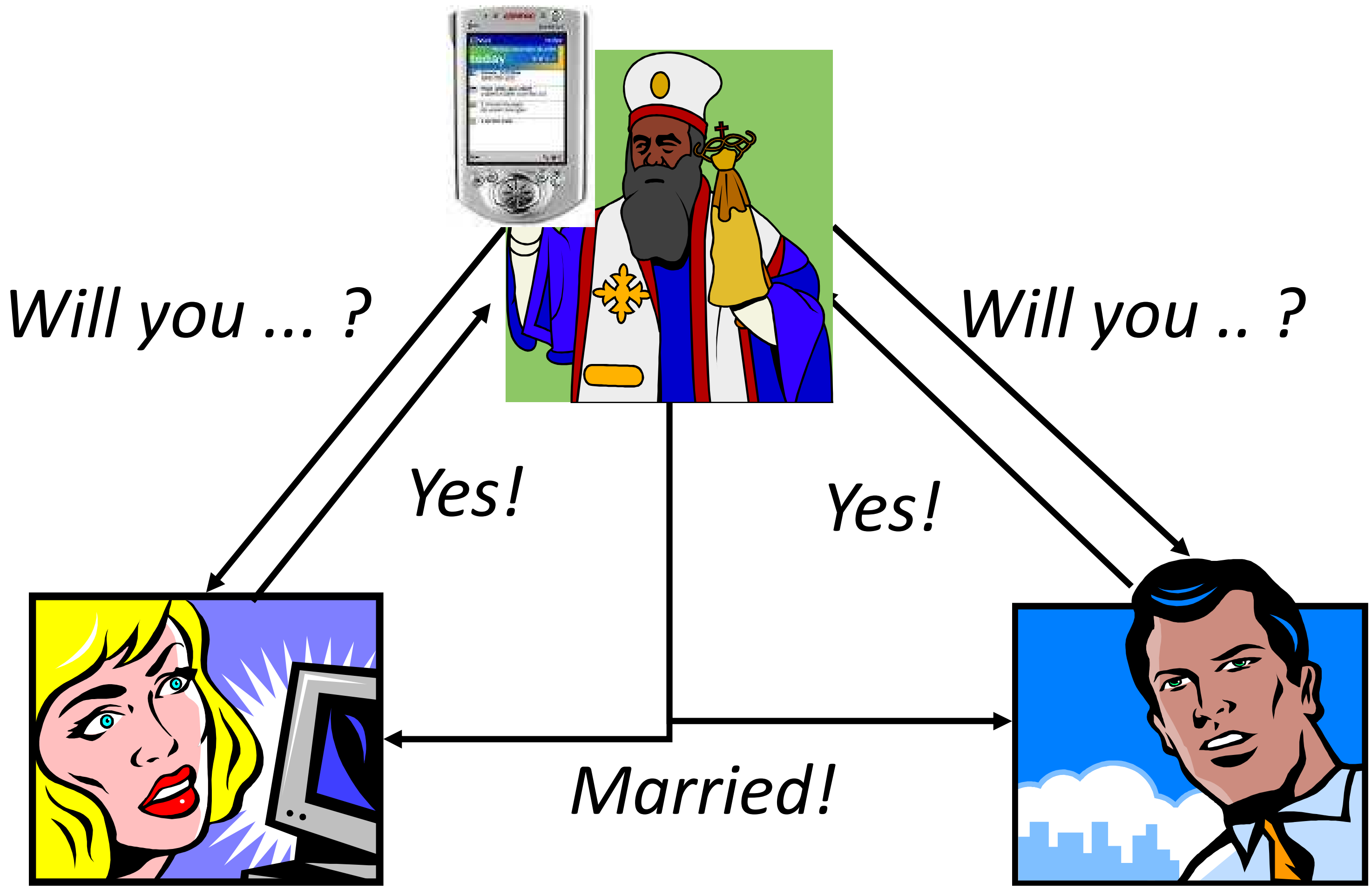
Distributed Transaction Commit

- **Fundamental Problem**:
  - Transaction operates on multiple servers (**resource managers**)
  - Global commit needs unanimous local commits of all **participants (agents)**
- Distributed system may fail partially:
  - Server Crashes
  - Network failures
- Potential danger of inconsistent decision:
  - A Transaction commits at some servers
  - But is aborted at some other servers

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

Getting Married over the Network



Atomic Commitment

- At commit time, the coordinator requests votes from all participants.
- Atomic commitment requires:
  - All processes reach same decision
  - Commit only if *all* processes vote Yes.
  - If there are no failures and all processes vote Yes, decision will be commit.

Two Phase Commit (2PC)

- Coordinator
  - send vote-request
  - Collect votes. If all Yes, then Commit, else Abort.
  - Send decision

- Participant
  - receive vote-request
  - send Yes or No
  - receive decision

Failures and Blocking

- What does a process do if it does not receive a message it is expecting? I.e., on *timeout*?

- 3 cases:
  - participant waiting for vote-request → abort
  - coordinator waiting for vote → abort
  - participant waiting for decision → **uncertain**

- Note: coordinator never uncertain

Termination Protocol

- Can participant find help from other participants?
- Send to all participants: ``Help! What is decision?"
  - if any participant has committed or aborted

    ⟶ send commit or abort decision.
  - If a participant has not yet voted

    ⟶ abort and send abort decision.
  - If all participants voted Yes

    ⟶ all live participants uncertain
- Transaction **BLOCKED**! ☹

Distributed Commit

- Cause of significant complexity
- Failures of another site cause local data to become unavailable
- Most commercial database provide 2PC but in practice 2PC not used

Transaction in Distributed DBMSs

- Significant overhead in managing correct executions
- Reliance on a global synchronization mechanism
- Limits scalability
- Impacts fault-tolerance and data availability
- Logistics
  - Sacrifices autonomy → significant hurdle in large enterprises
- Combination of all these factors made distributed databases less practical

CONCEPTS FROM DISTRIBUTED SYSTEMS

Distributed System Models

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

Outline

- Concept of logical timing in distributed systems
- Quorums
- Leader Election
- Consensus and Paxos

What is a Distributed System?

- A simple model of a distributed system proposed by Lamport in a landmark 1978 paper:
- "Time, Clocks and the Ordering of Events in a Distributed System" Communications of the ACM

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
  - Send events
  - Receive events
- Local events on a process for a total order

Happens Before Order on Events

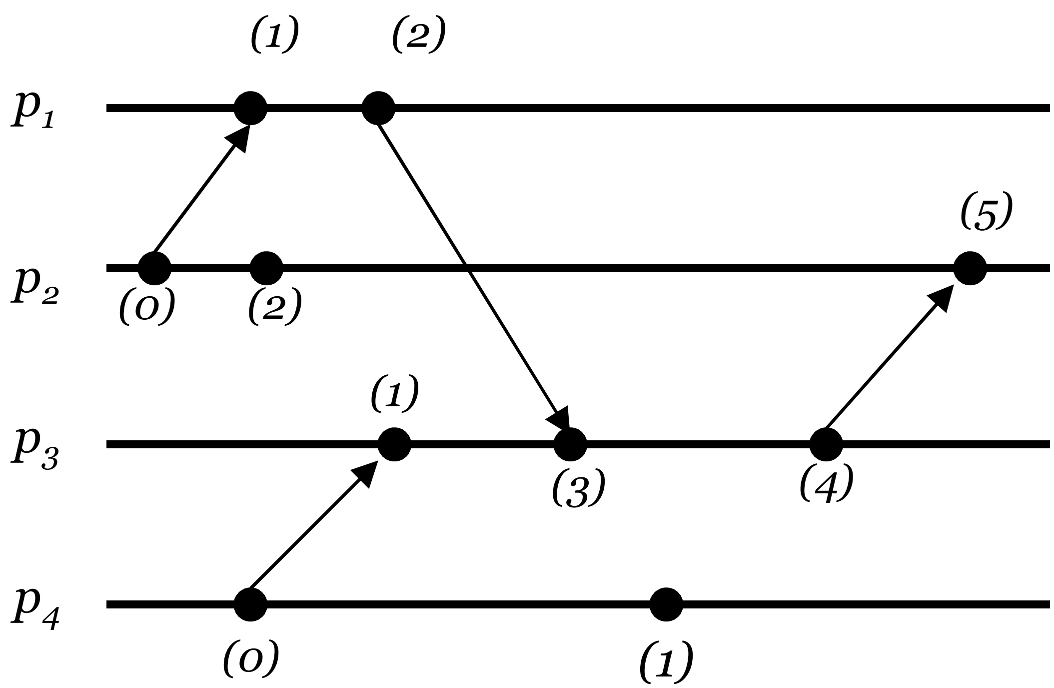
- Event e *happens before (causally precedes)* event *f*, denoted $e \rightarrow f$ if:
  1. The same process executes e before *f* ; or
  2. e is send(*m*) and *f* is receive(*m*); or
  3. Exists *h* so that $e \rightarrow h$ and $h \rightarrow f$
- We define *concurrent*, $e \parallel f$, as:

$$\neg(e \rightarrow f \lor f \rightarrow e)$$

Lamport Logical Clocks

- Assign "clock" value to each event
  - if ***a* ì *b*** then **clock(*a*) < clock(*b*)**
- Assign each process a clock "counter".
  - Clock must be incremented between any two events in the same process
  - Each message carries the sender's clock value
- When a message arrives set local clock to:
  - max(local value, message timestamp + 1)

Example of a Logical Clock



Vector clocks

1. Vector initialized to 0 at each process

   $V_i[j] = 0$ for $i, j = 1, \ldots, N$

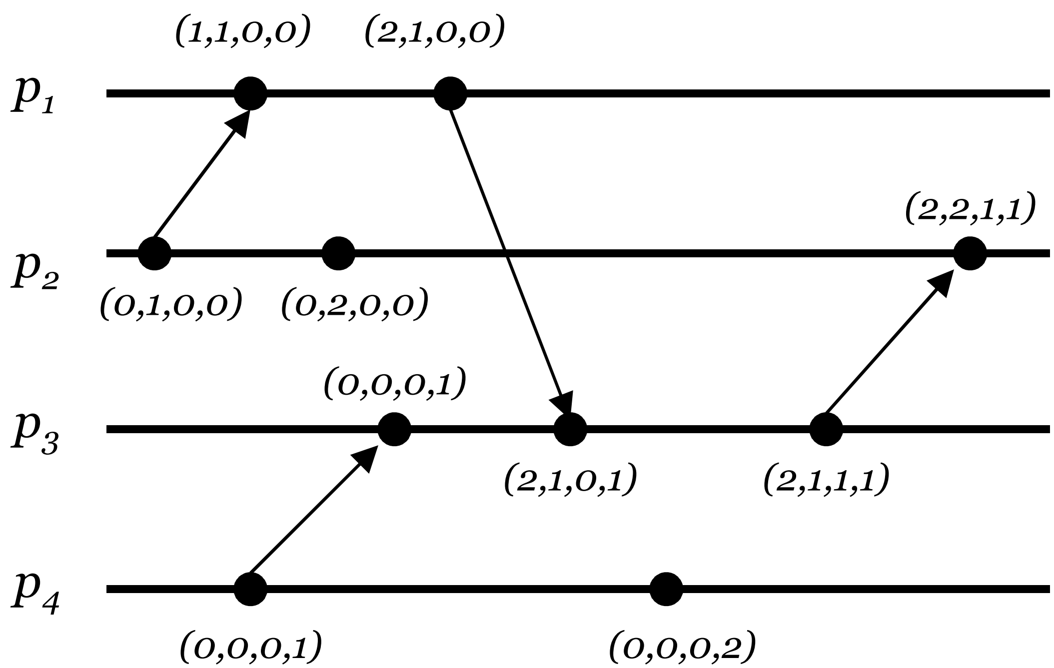
2. Process increments its element of the vector in local vector before event:

   $V_i[i] = V_i[i] + 1$

3. Piggyback V_i with every message sent from process P_i
4. When P_j receives message, compares vectors element by element and sets local vector to higher of two values

   $V_j[i] = \max(V_i[i], V_j[i])$ for $i = 1, \ldots, N$

Example of a Vector Clock

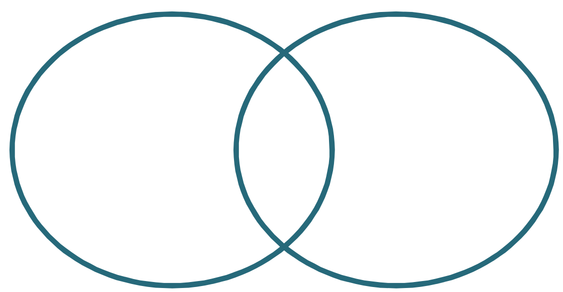


Quorums

- Many distributed actions need to contact multiple servers
- What if there are failures?
  - Do we need to communicate with ALL processes?
- A **quorum** is the minimum number of votes needed for a distributed operation
- Any two requests should have a common process to act as an arbitrator.
- Let process p_i (p_j)request permission from V_i (V_j), then
  - $V_i \cap V_j \neq \phi$.
- V_i is called a quorum.

Quorums

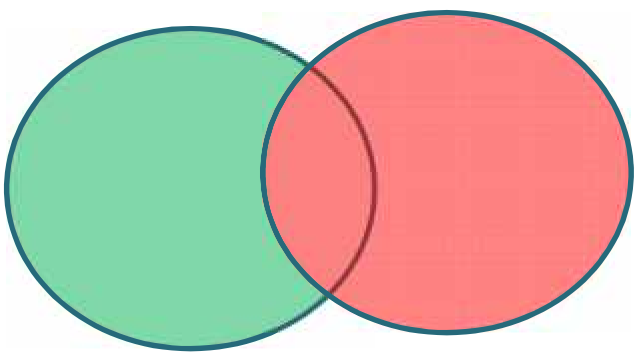
- Given n processes: $2|V_i| > n$, ie,

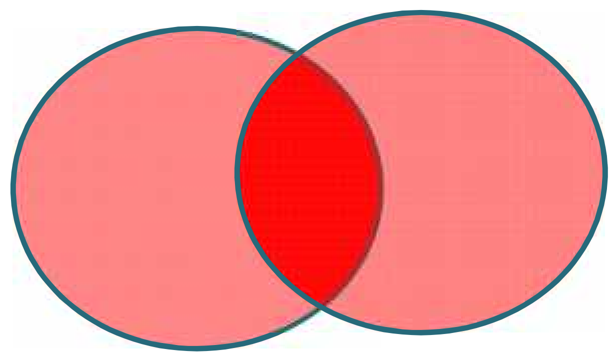


- In general, majority, ie $\lceil (n/2) \rceil$. [Gifford 79]

General Quorums

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





Leader Election

- 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

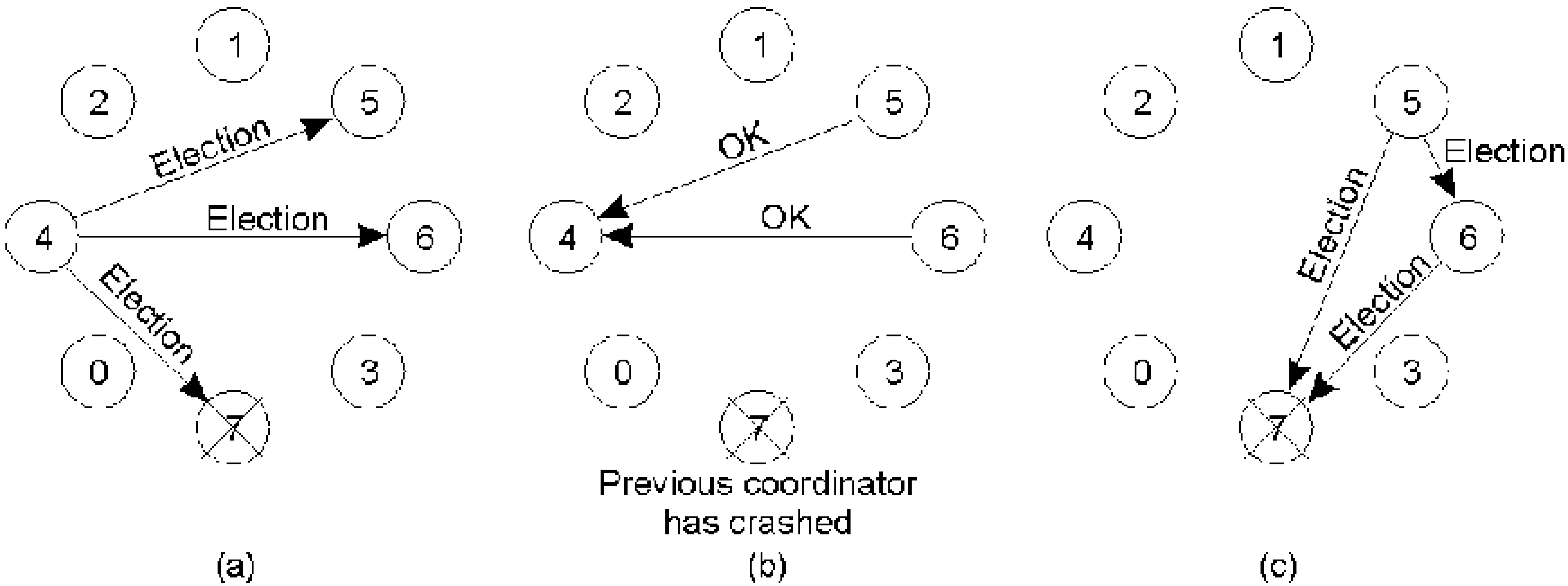
Bully Algorithm

- Each process has a unique numerical ID
- Processes know Ids and address of all other process
- Communication is assumed reliable
- *Key Idea*: select process with highest ID
- Process initiates election if it just recovered from failure or if coordinator failed
- 3 message types: *election*, *OK*, *I won*
- Processes can initiate elections simultaneously
  - Need consistent result

Bully Algorithm Details

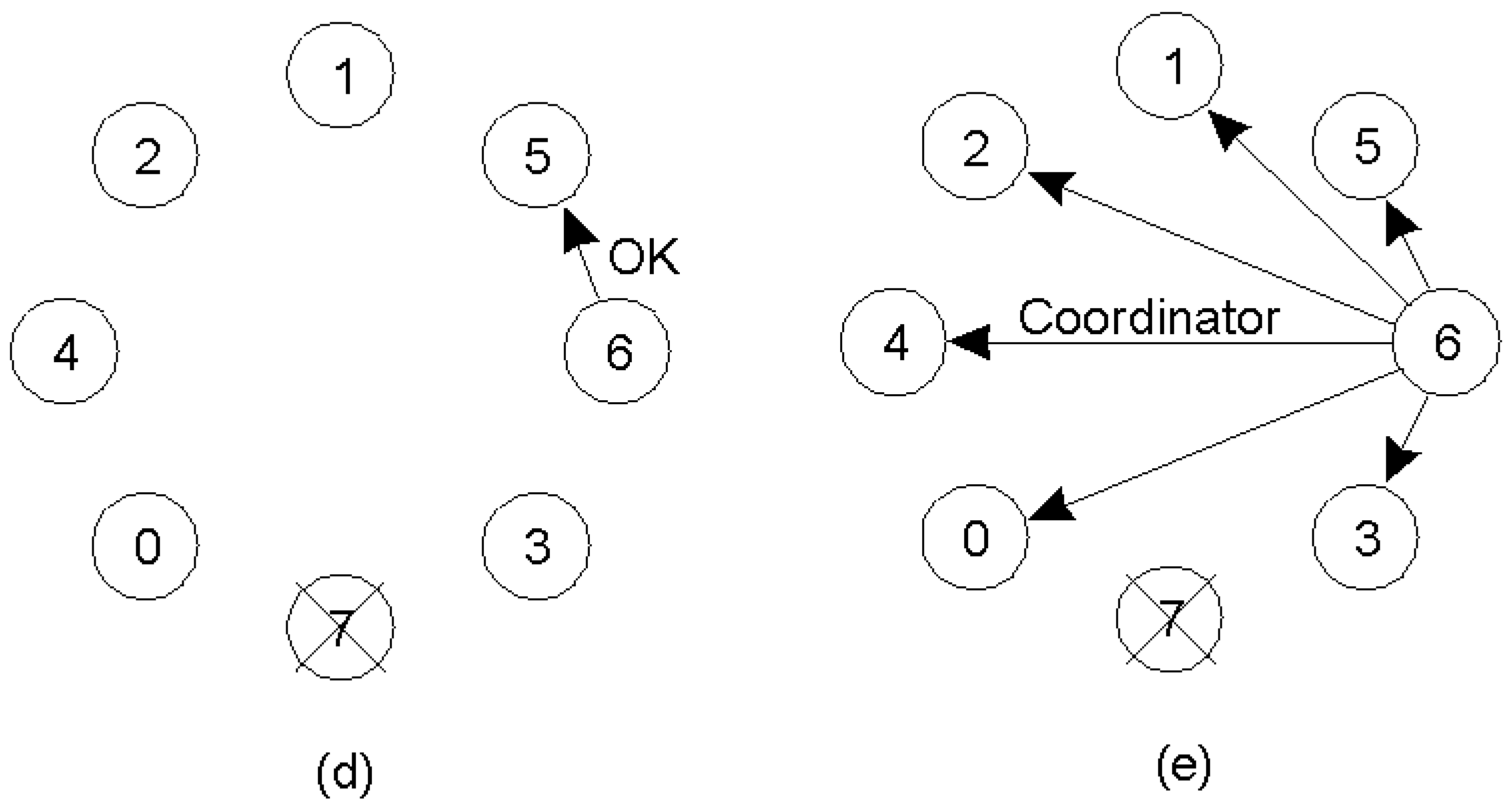
- Any process *P* can initiate an election
- *P* **sends** *Election* messages to all process **with higher Ids** and awaits *OK* messages
- If **no** *OK* messages, *P* **becomes coordinator** & sends *I won* to all process with lower Ids
- If it receives *OK*, it **drops out & waits** for *I won*
- If a process **receives** *Election* msg, it returns *OK* and **starts an election**
- If a process **receives** *I won* then **sender is coordinator**

Bully Algorithm Example



a) Process 4 holds an election
b) Process 5 and 6 respond, telling 4 to stop
c) Now 5 and 6 each hold an election

Bully Algorithm Example



d) Process 6 tells 5 to stop
e) Process 6 wins and tells everyone

Consensus

- **Consensus** requires agreement among a number of processes for a single data value
- Processes may fail or be unreliable
- Properties of consensus
  - Termination
    - Every correct process decides some value
  - Validity
    - If all correct processes propose the same value *v*, then all correct processes decide *v*
  - Integrity
    - Every correct process decides at most one value, and if it decides some value *v*, then *v* must have been proposed by some process
  - Agreement
    - Every correct process must agree on the same value

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 Made Moderately Complex, (Cornell) 2011.
  - ....

The Paxos Algorithm

- Leader based: each process has an estimate of who is the current leader
- To order an operation, a process sends it to current leader
- The leader sequences the operation and launches a Consensus algorithm to fix the agreement

Thanks to Idit Keidar for slides

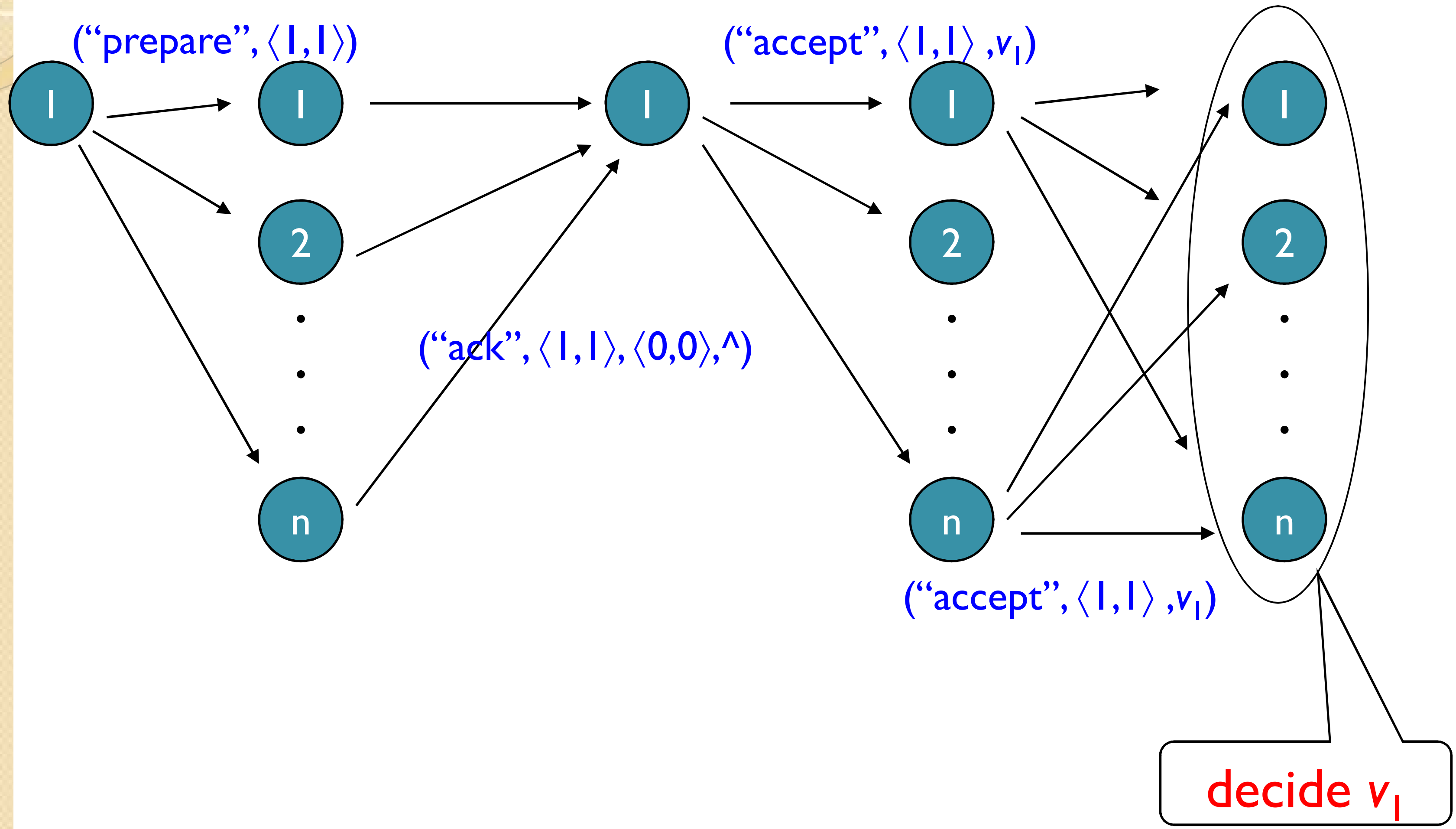
The Consensus Algorithm Structure

- Two phases
- Leader contacts a majority in each phase
- There may be multiple concurrent leaders
- *Ballots* distinguish among values proposed by different leaders
  - Unique, locally monotonically increasing
  - Processes respond only to leader with highest ballot seen so far

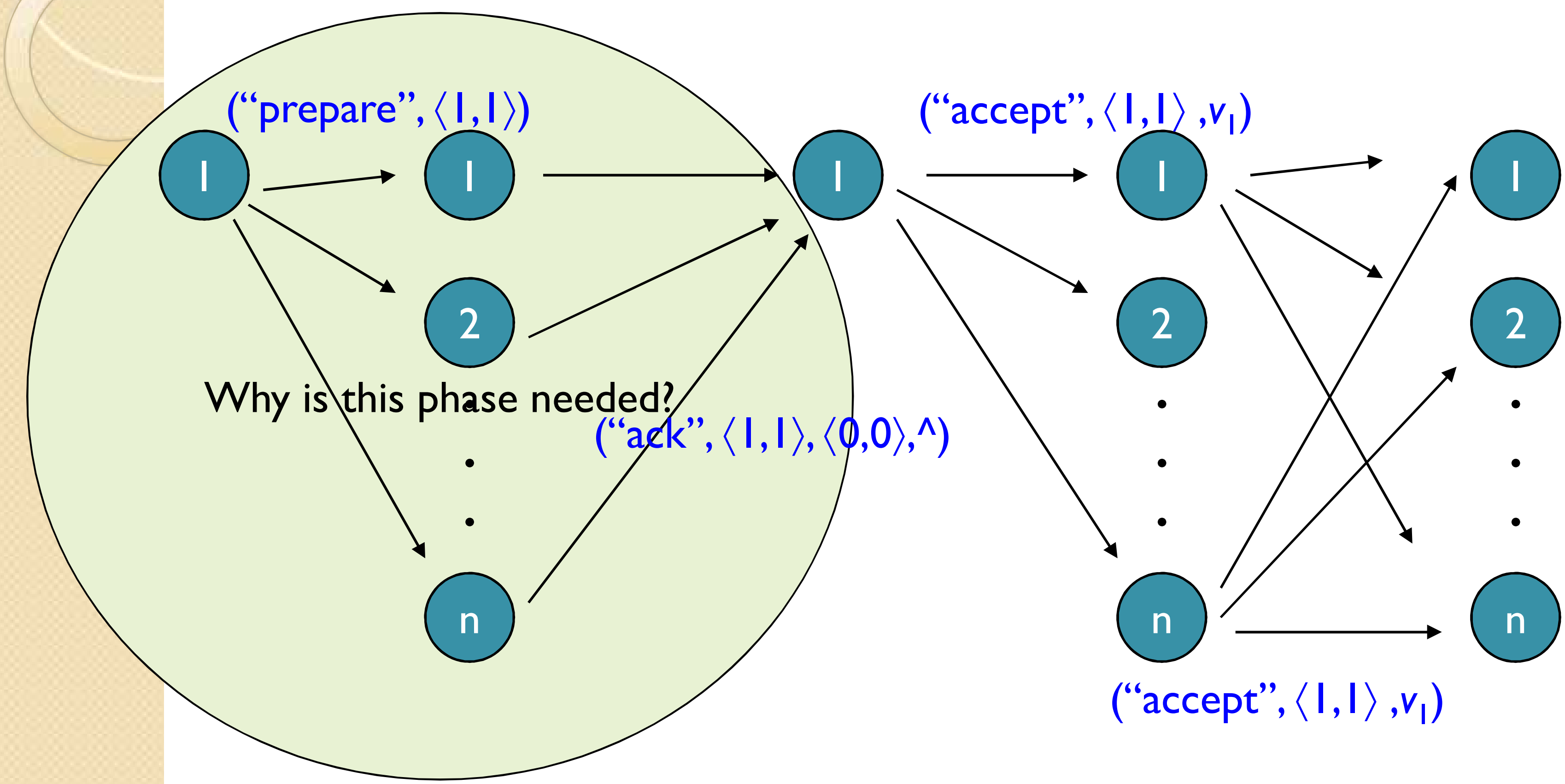
The Two Phases of Paxos

- Phase 1: prepare
  - If you *believe you are the leader*
    - Choose new unique ballot number
    - Learn outcome of all smaller ballots from majority
- Phase 2: accept
  - Leader *proposes* a value with its ballot number
  - Leader gets majority to *accept* its proposal
    - A value accepted by a majority can be decided

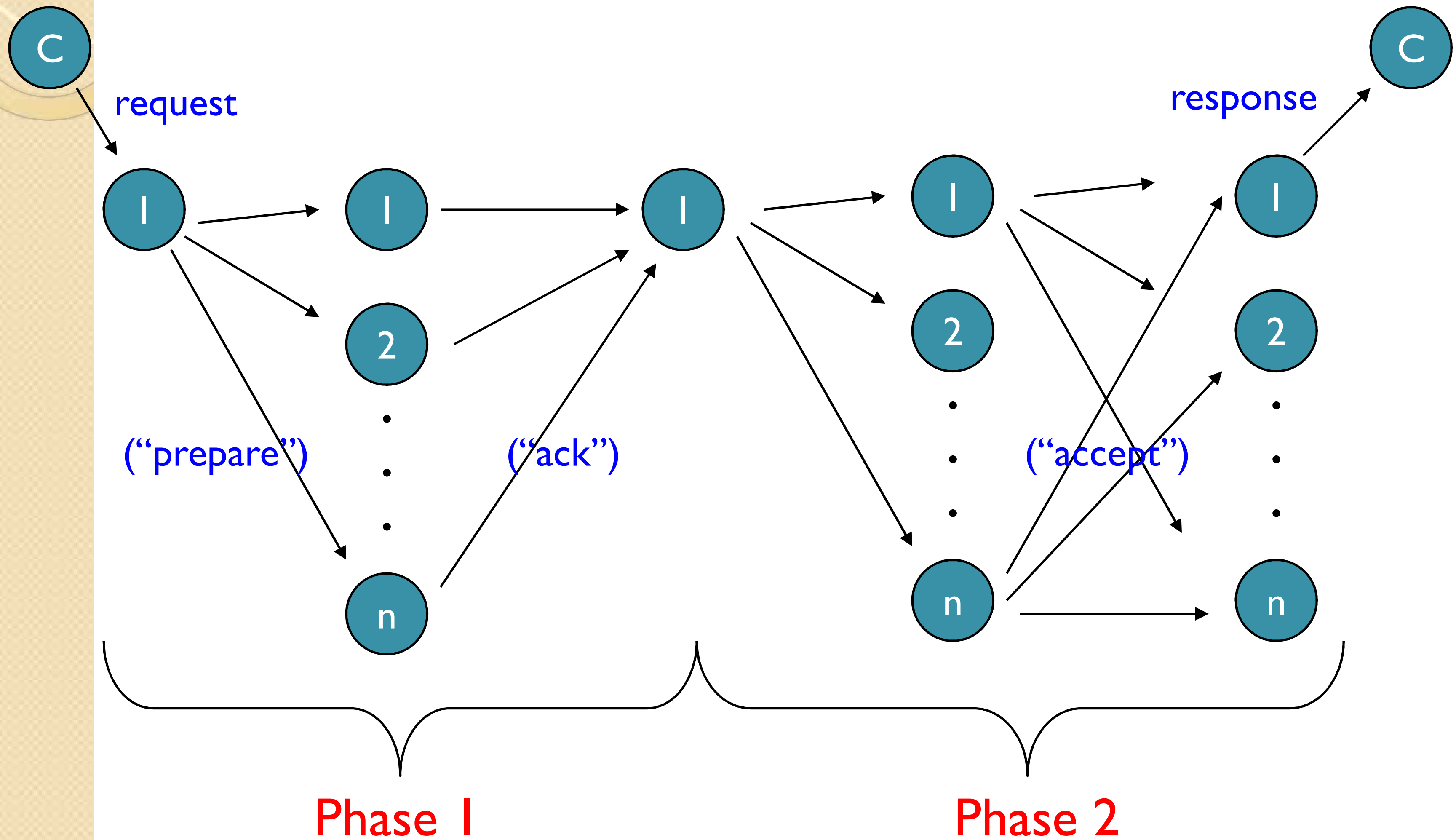
In Failure-Free Execution



Performance?



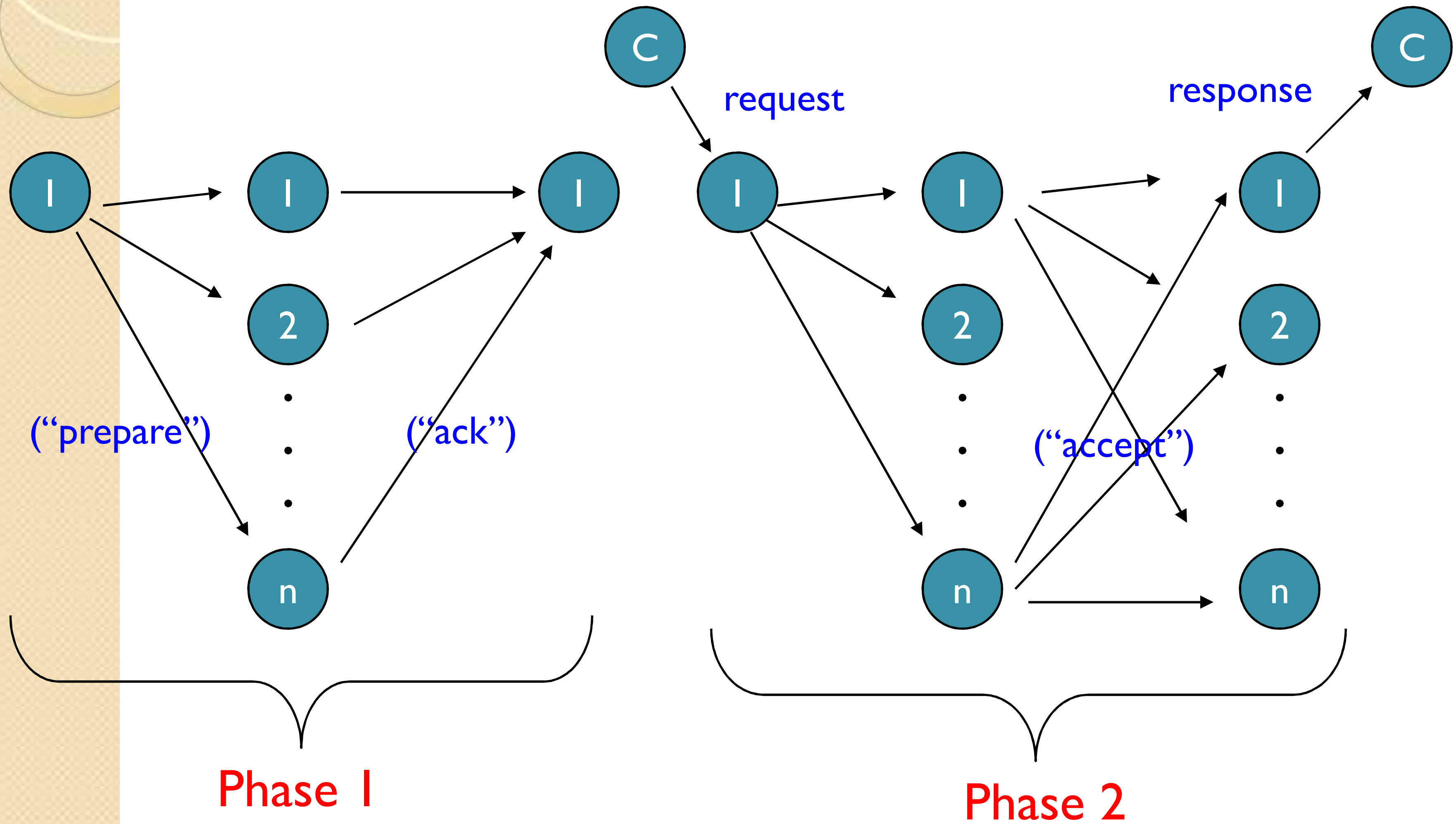
Failure-Free Execution



Observation

- In Phase 1, no consensus values are sent:
  - Leader chooses largest unique ballot number
  - Gets a majority to "vote" for this ballot number
  - Learns the outcome of all smaller ballots
- In Phase 2, leader proposes its own initial value or latest value it learned in Phase 1

Failure free execution



Optimization

- Run Phase 1 only when the leader changes
  - Phase 1 is called "view change" or "recovery mode"
  - Phase 2 is the "normal mode"
- Each message includes BallotNum (from the last Phase 1) and ReqNum
- Respond only to messages with the "right" BallotNum

Summary

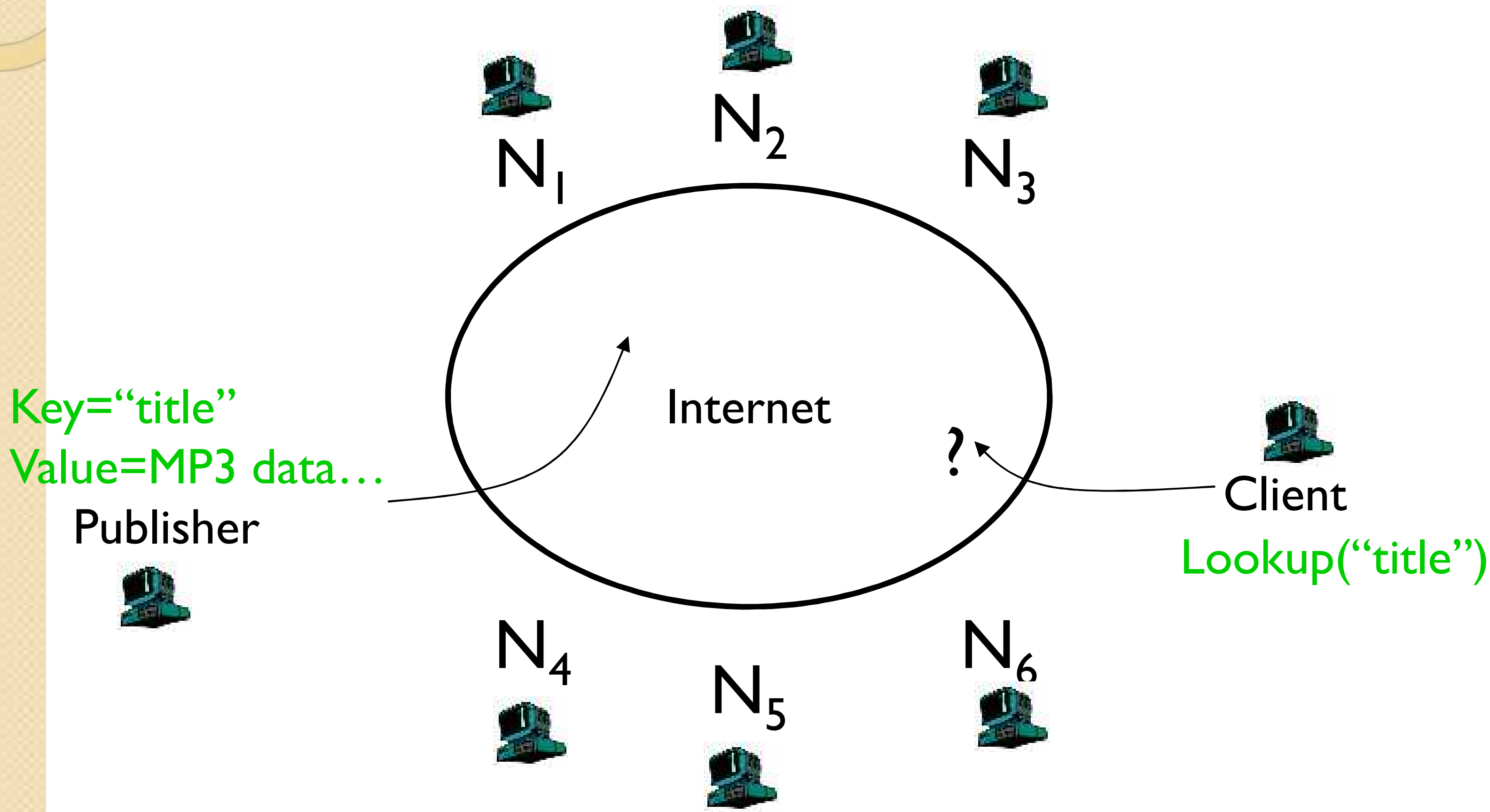
- Concept of logical timing in distributed systems
  - Lamport Clocks
  - Vector Clocks
- Quorums
- Leader Election
- Consensus and Paxos

P2P SYSTEMS

Searching for distributed data

- Goal: Make billions of objects available to millions of concurrent users
  - e.g., music files
- Need a distributed data structure to keep track of objects on different sires.
  - map object to locations
- Basic Operations:
  - Insert(key)
  - Lookup(key)

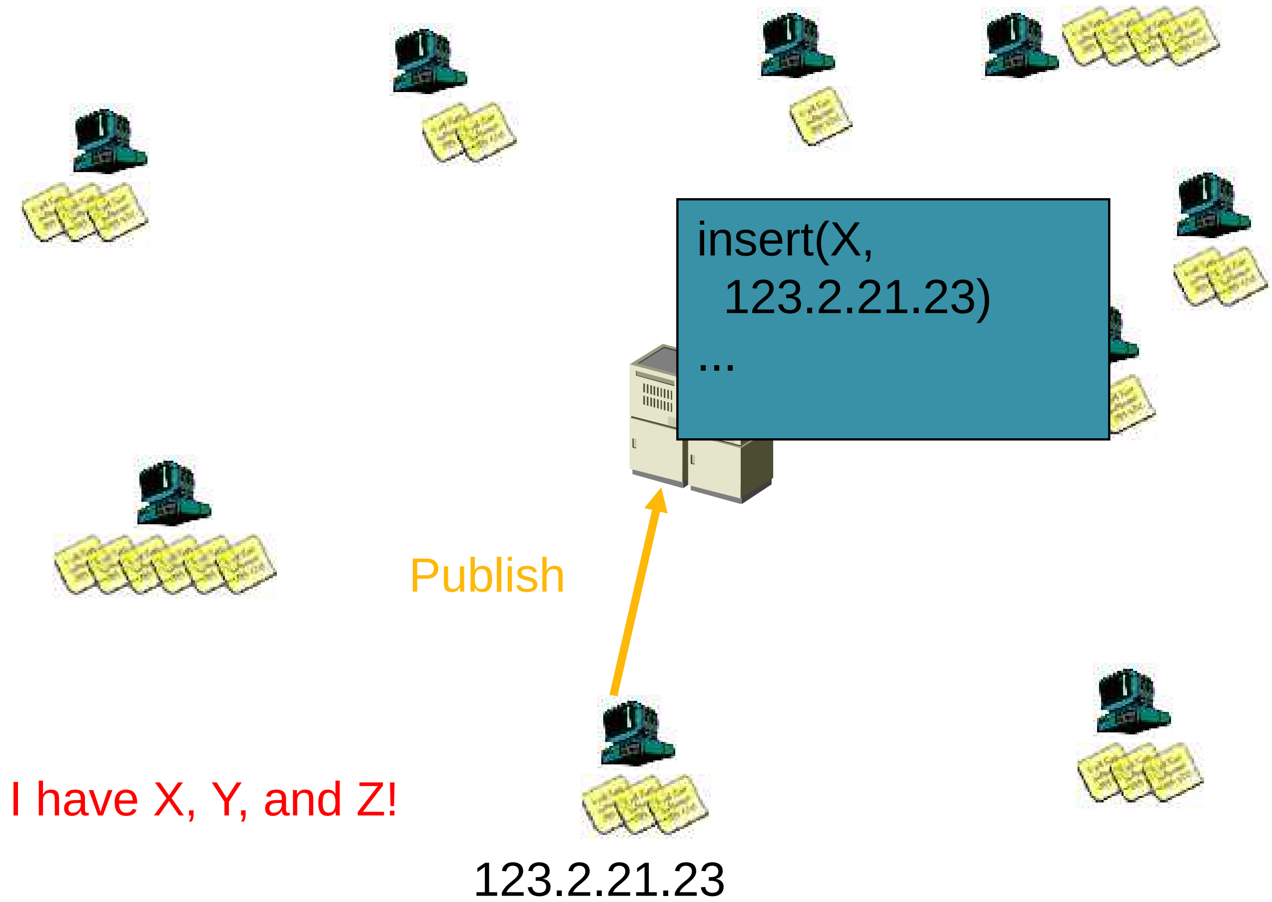
Searching


N1
N2
N3
Internet
Key="title"
Value=MP3 data…
Publisher
?
Client
Lookup("title")
N4
N5
N6

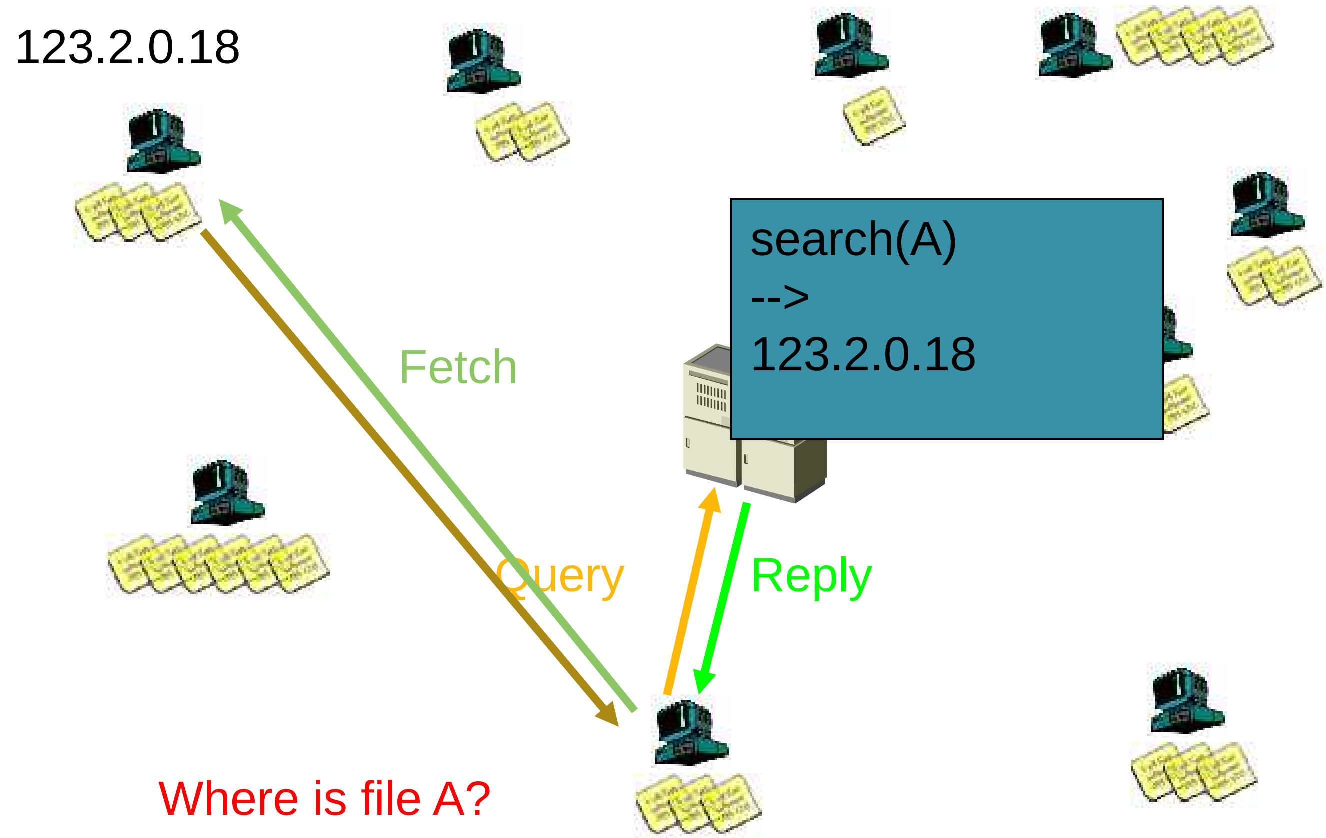
Simple Solution

- 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.

Napster: Publish



Napster: Search

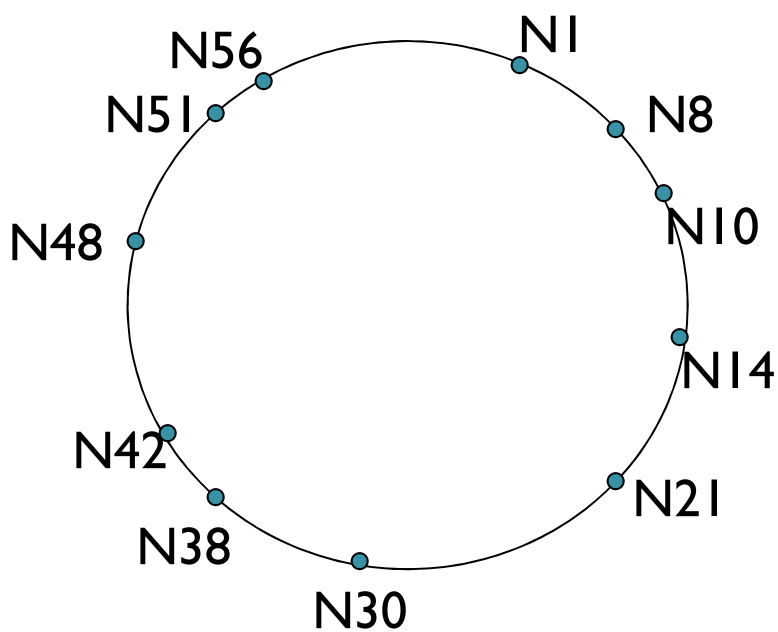


Distributed Hash Tables (DHTs)

- 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)

Chord Logical Structure (MIT)

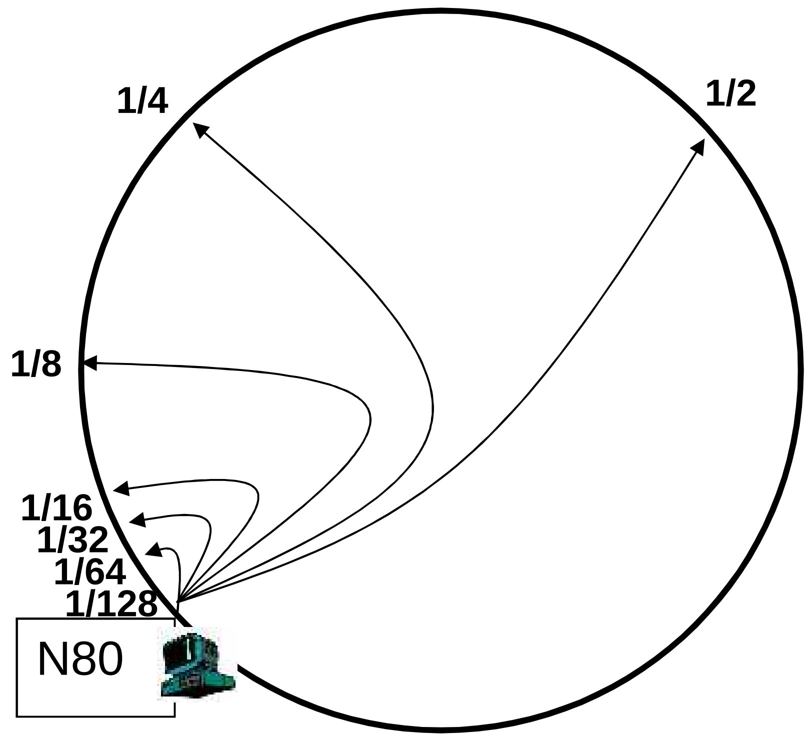
- *m*-bit ID space (2^m IDs), usually m=160.
- Nodes organized in a logical ring according to their IDs.



Consistent Hashing Guarantees

- For any set of N nodes and K keys:
  - A node is responsible for at most $(1 + \varepsilon)K/N$ keys
  - When an $(N + 1)$st node joins or leaves, responsibility for $O(K/N)$ keys changes hands
- For the scheme described above, $\varepsilon = O(\log N)$
- ε can be reduced to an arbitrarily small constant by having each node run $(\log N)$ *virtual nodes*, each with its own identifier.

Finger Table

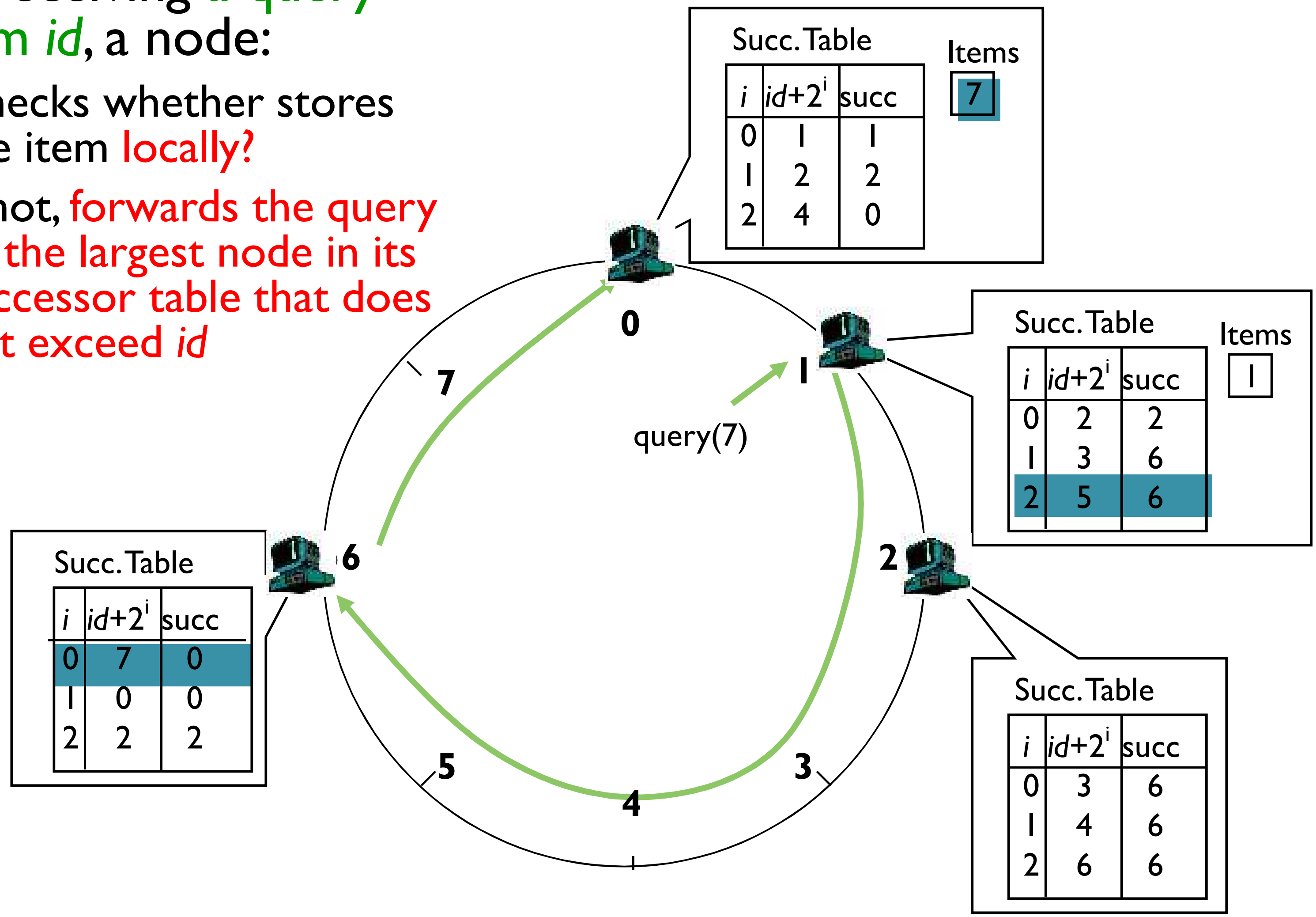


- Entry *i* in the finger table of node *n* is the first node that succeeds or equals $n + 2^i$
- In other words, the i^{th} finger points $1/2^{n-i}$ way around the ring

DHT: Chord 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*



P2P Lessons

- Decentralized architecture
- Avoid centralization
- Flooding can work.
- Logical overlay structures provide strong performance guarantees.
- Churn a problem.
- Useful in many distributed contexts

KEY VALUE STORES

Overview

- Design Choices and their Implications
- Common Key-value Store examples
  - Bigtable
  - PNUTs
  - Dynamo
- Discussion

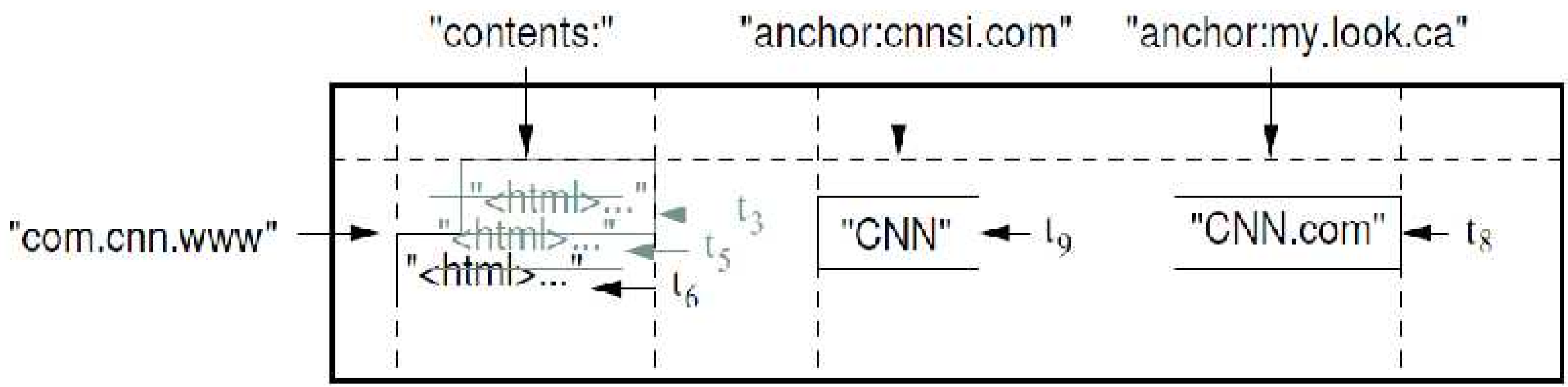
Key Value Stores

- Gained widespread popularity
  - In house: **Bigtable** (Google), **PNUTS** (Yahoo!), **Dynamo** (Amazon)
  - Open source: **HBase, Hypertable, Cassandra, Voldemort**
- Challenges
  - Request routing
  - Cluster management
  - Fault-tolerance and data replication

Key Value Data Models

- Data model
  - Key is the unique identifier
  - Key-value is the granularity for consistent access
  - Value can be structured or unstructured
- Bigtable
  - Sparse multidimensional sorted map
    - Tables comprise of column families
    - Values indexed by the key, column family, column, and timestamp
- PNUTS
  - Flat column structure
- Dynamo
  - Un-interpreted string of bytes (blob)

BigTable Visual Illustration



WebTable Example: URLs are row keys, various aspects of web pages as column names, eg, "contents" stores contents of webpages versions indexed by timestamp.

Different Design Goals

- Bigtable (Google):
  - Scale-out for single-key access and range scans
  - Support for crawl and indexing infrastructure
- PNUTS (Yahoo!):
  - Geographic replication for high read availability
  - Support for geographically distributed clients
- Dynamo (Amazon):
  - High write availability
  - Support for shopping carts (e-commerce)

Request Routing

- **To determine which storage unit has a record**:
  - Hierarchical approach
    - **Bigtable** (Range partitioned)
      - B+-tree stores mapping of key ranges to servers
  - Explicit storing of mapping
    - **PNUTS** (Range or hash partitioned)
      - *Tablet controller* stores interval mapping of partitions to servers
      - *Routing* layer responsible for request routing
  - Distributed Hash Table approach
    - **Dynamo** (Hash partitioned)
      - Consistent hashing a la Chord.

Cluster Management

- **Monitoring nodes, failures, recovery and load balancing**
- Centralized, master based
  - **Bigtable**
    - Master contacts Chubby for node recovery.
  - **PNUTS**
    - Tablet controller
- Decentralized, gossip based
  - **Dynamo**
    - Sloppy quorums

Fault-tolerance and Data Replication

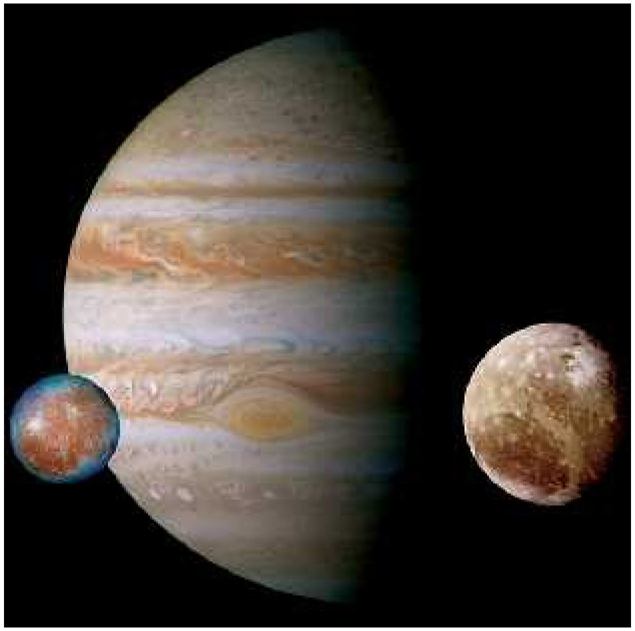
- Modular Shared Storage Design
  - **Bigtable**
    - Fault-tolerant storage: Google File System (GFS)
    - Strong replica consistency
- Explicit Replication
  - **PNUTS**
    - Reliable pub/sub system: Yahoo! Message Broker (YMB)
    - Single object timeline consistency for replicas
    - Per-record master for fine-grain control of locality of writes
  - Dynamo
    - Asynchronous replication using quorums
    - Eventual consistency
      - Divergent versions reconciled by application using vector clocks

Design Principles

What have we learned from Key-value stores?

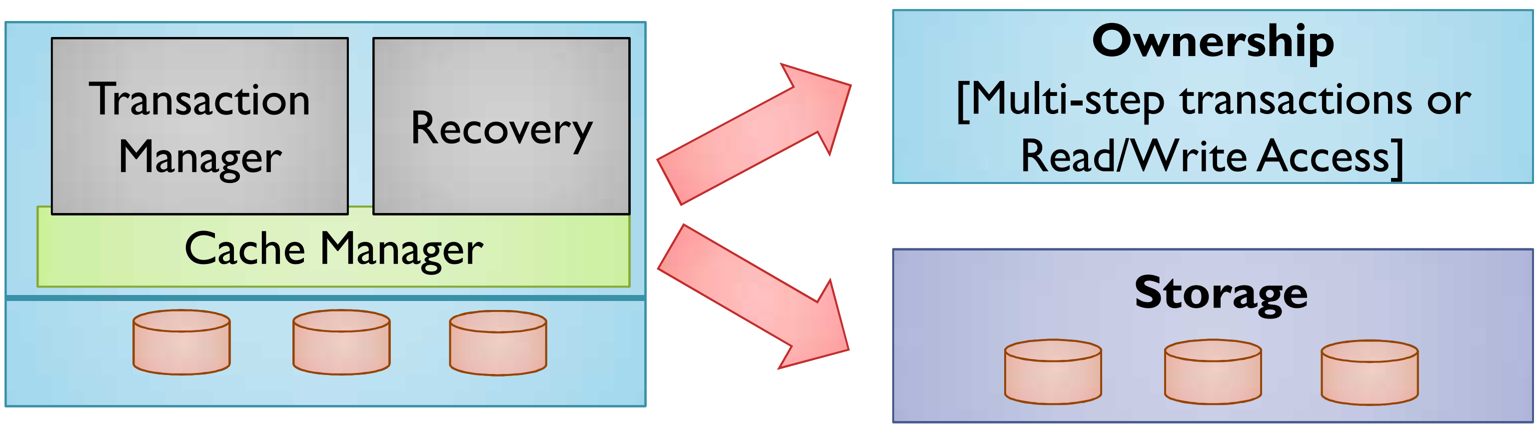
Design Principles

- **Separate System** and **Application** State
  - **System metadata** is critical but small
  - **Application data** has varying needs
  - Separation allows use of different class of protocols



Design Principles

- **Decouple Ownership from Data Storage**
  - Ownership is exclusive read/write access to data
  - Decoupling allows lightweight ownership migration

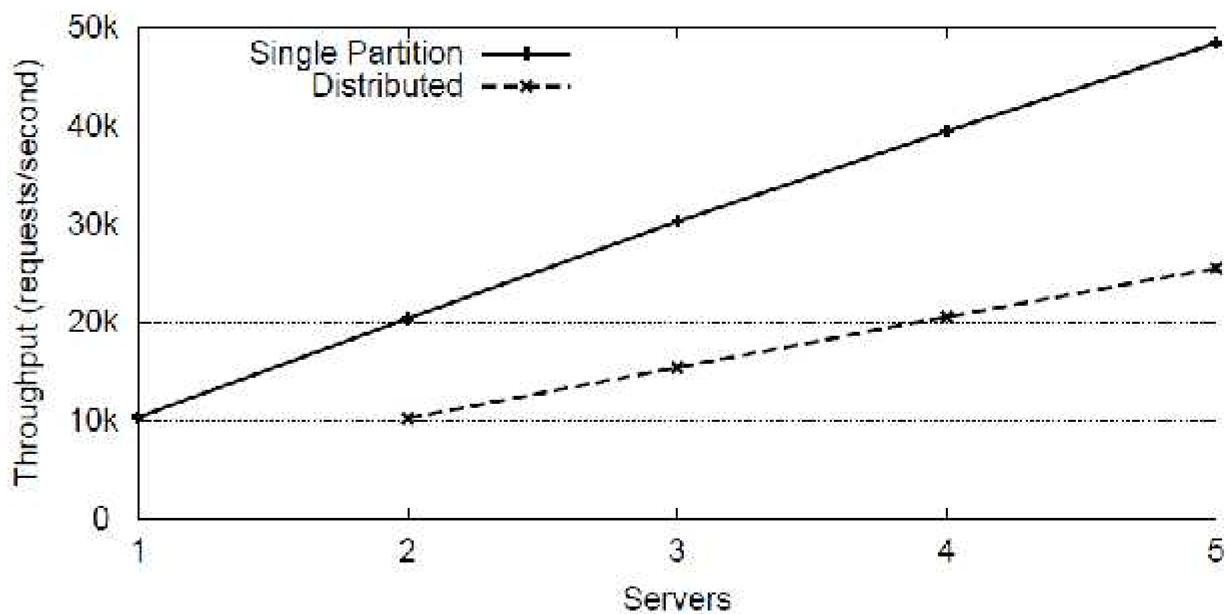


Classical DBMSs

Decoupled ownership and Storage

Design Principles

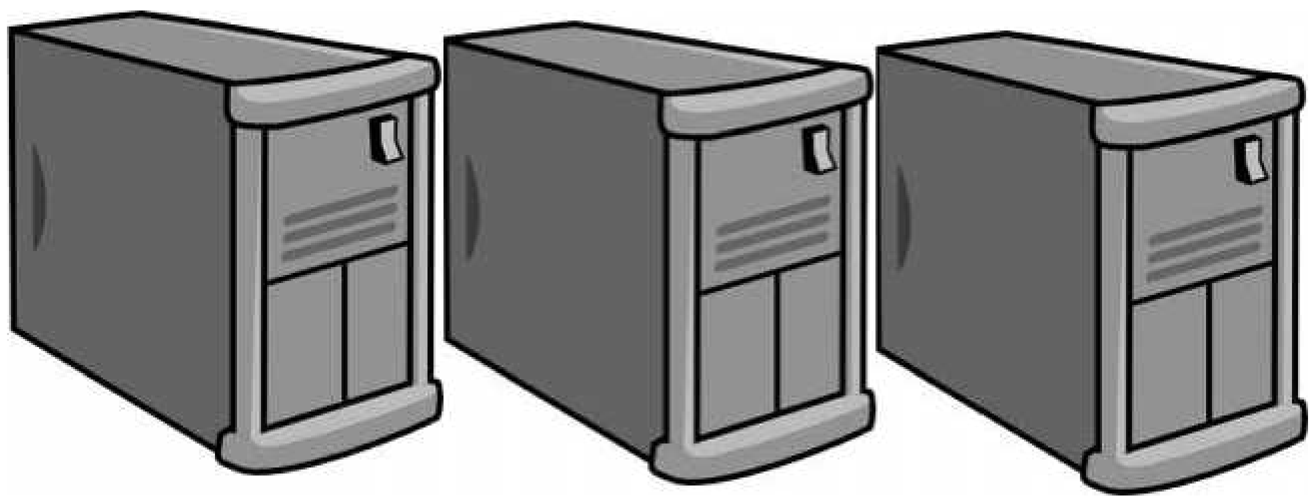
- **Limit most interactions to a single node**
  - Allows **horizontal scaling**
  - **Graceful degradation** during failures
  - No distributed synchronization



Thanks: Curino et al VLDB 2010

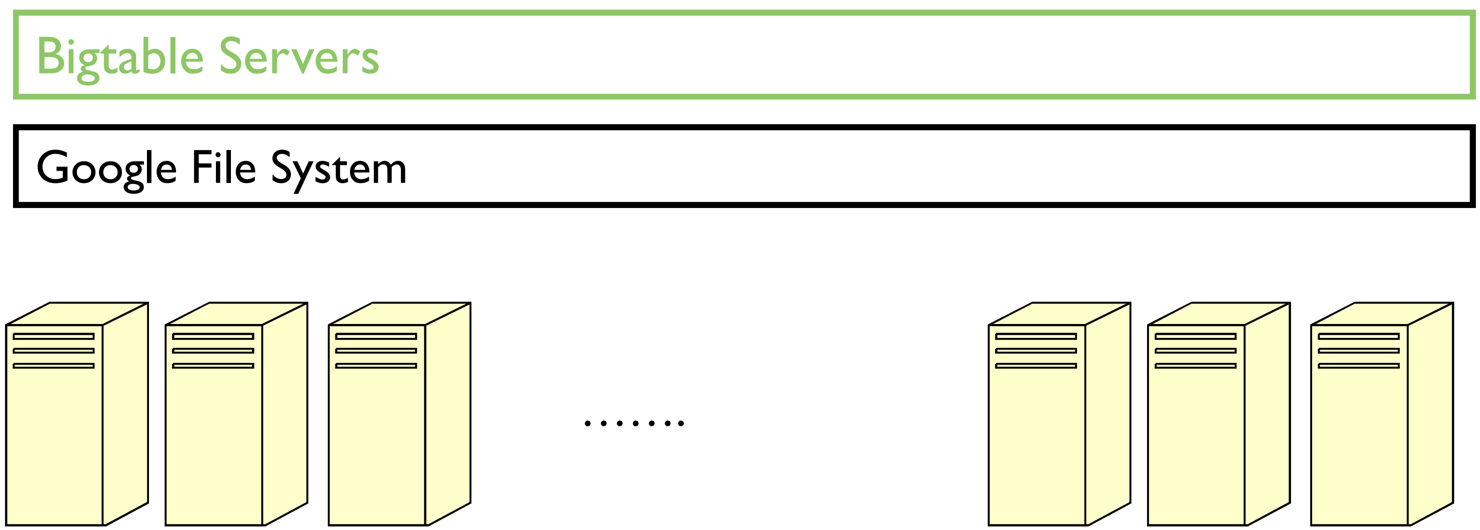
Design Principles

- **Limited distributed synchronization is practical**
  - Maintenance of metadata
  - Provide strong guarantees only for data that needs it



Bigtable

- Shared-nothing architecture consisting of thousands of nodes (commodity PC).



Bigtable

- Data model (a schema).
  - A sparse, distributed persistent multi-dimensional sorted map
- Data is partitioned across the nodes seamlessly.
- The map is indexed by a row key, column key, and a timestamp.
- Output value in the map is an un-interpreted array of bytes.
  - (row: byte[ ], column: byte[ ], time: int64) → byte[ ]

Column Families

- Column keys are grouped into sets called column families (*nested tables*).
- A column family must be created before data can be stored in a column key.
- A unit of storage co-location.
- Hundreds of static column families.
- Syntax is *family:qualifier*:
  - Language:English
  - Language:German

Bigtable API

- Implements interfaces to:
  - create and delete tables and column families,
  - modify cluster, table, and column family metadata such as access control rights,
  - Write or delete values in Bigtable,
  - Lookup values from individual rows,
  - Iterate over a subset of the data in a table,
  - Atomic R-M-W sequences on data in a single row key (No support for TXN across multiple rows).

Example

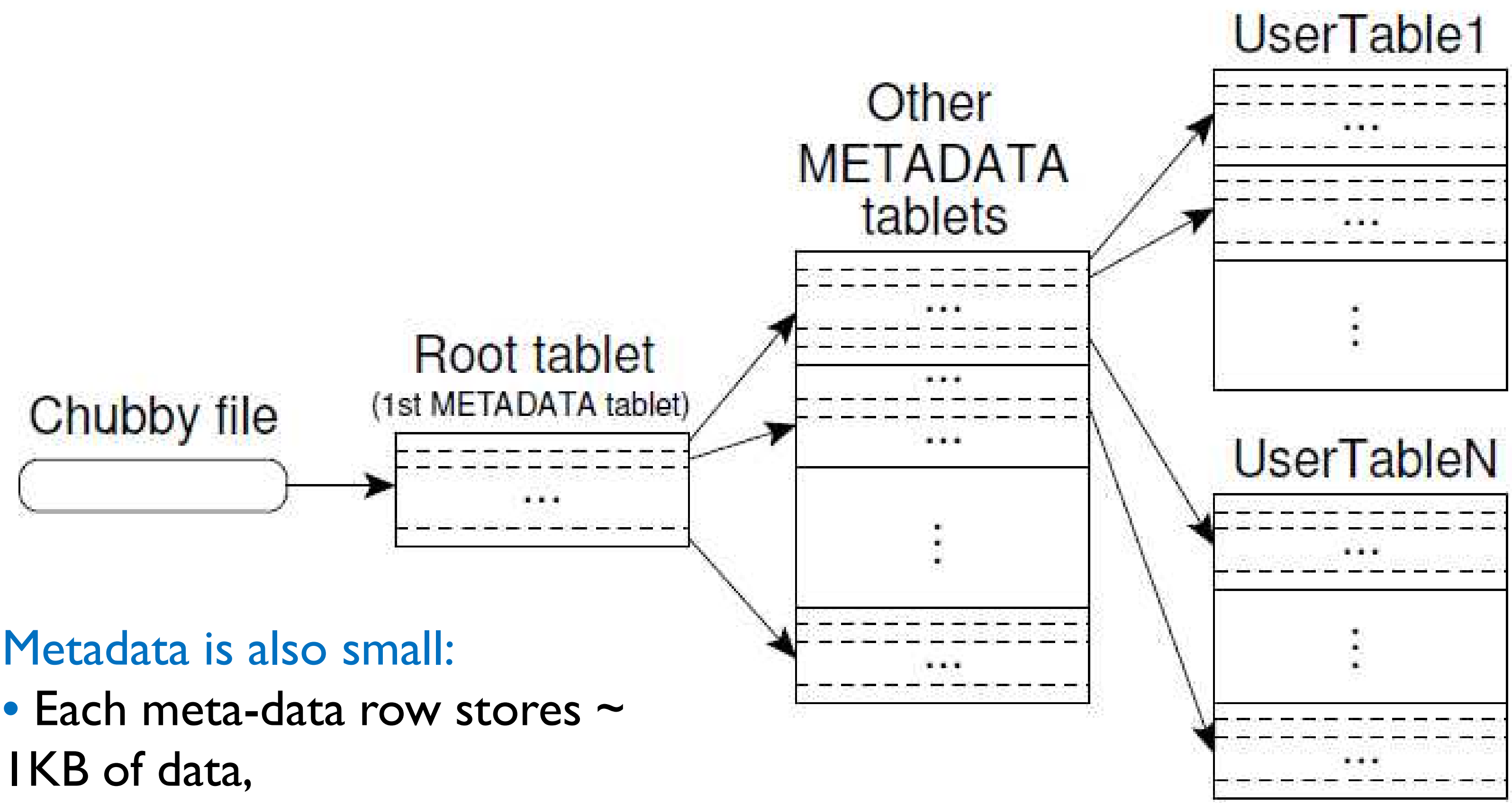
```
// open the table
Table *T =OpenOrDie("/bigtable/web/webtable");

//write a new anchor and delete an old anchor
RowMutation R1(T, "www.cnn.com");
R1.set("anchor:www.c-span.org", "cnn");
R1.delete("anchor:www.abc.com");
Operation &op;
APPLY(&op, &R1);
```

Bigtable's Building Blocks

- Google File System (GFS)
  - Highly available distributed file system that stores log and data files
- SSTable
  - Stores Bigtable data by providing a persistent immutable mapping from keys to values .
- Chubby
  - Highly available persistent distributed lock manager.

Routing: 3 Level Hierarchy



Metadata is also small:

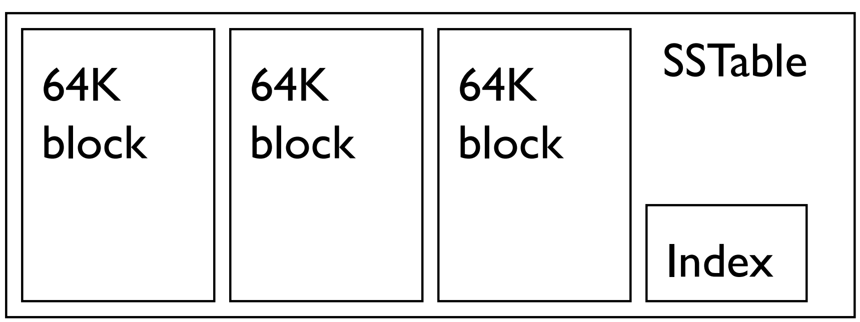
- Each meta-data row stores ~ 1KB of data,
- With 128 MB meta-data tablets, we can address 2^{34} tablets.
- Approaches a Zetabyte (10^6 Petabytes).

Chubby

- A persistent and distributed lock service.
- Consists of 5 active replicas:
  - One replica is elected master and serves requests
  - Live: as long as majority available
  - Paxos is used to keep copies consistent
- Maintains strictly consistent namespaces
  - Small files, which are used as locks
  - Reads and writes are atomic.

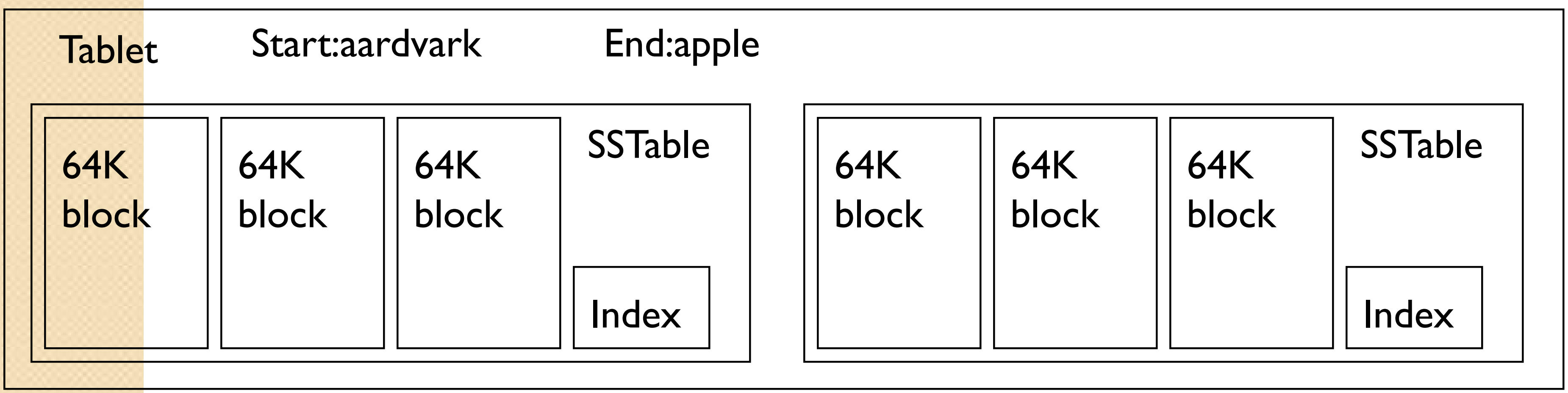
SSTable

- A file format used to store Bigtable data:
  - Stores and retrieves key/data pairs.
  - Supports iterating over key/value pairs given a selection predicate (exact and range).
  - Each **SSTable** contains a sequence of blocks + a block index (loaded in memory on opening)
  - Lookup: use in-memory index to locate block
- An SSTable is stored in GFS.



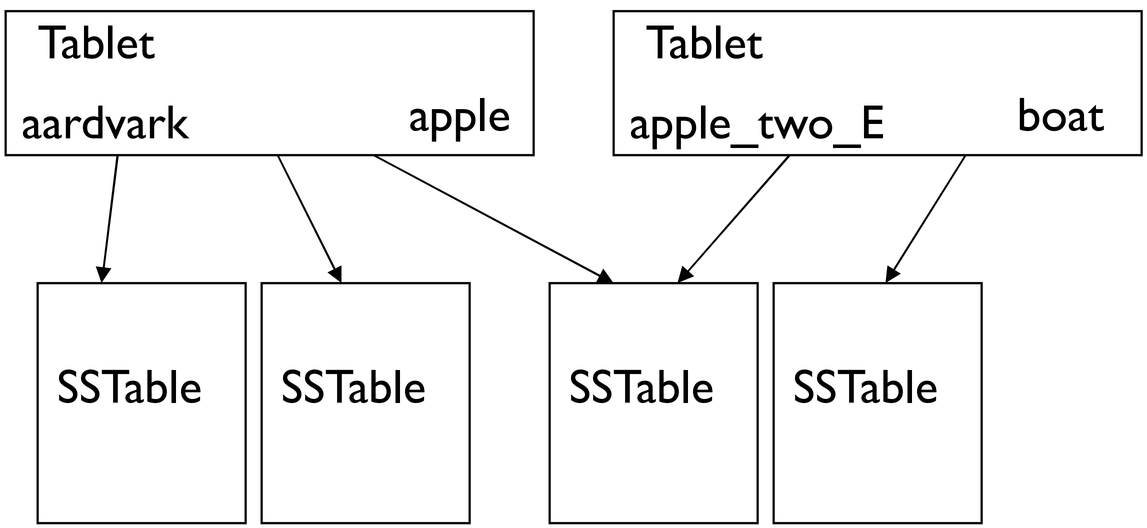
Tablets in Bigtable

- Bigtable maintains data in lexicographic order by row key, range partitioned into tablets
- A tablet is represented a set of SSTable files.
- Tablet is unit of distribution and load-balancing.



Tables

- A table is dynamically partitioned into tablets
- SSTables can be shared
- Tablets do not overlap, SSTables can overlap



Bigtable's 3 Major Components

1. A Bigtable library linked into every client.
2. One Master Server responsible for:
   - Assigning tablets to tablet servers,
   - Detecting the addition and deletion of tablet servers,
   - Balancing tablet-server load,
3. Many tablet servers:
   - Each manages ten to a thousand tablets.
   - Handles read and writes to its tablet and splits tablets.
   - Tablet servers are added and removed dynamically.

- Client communicates directly with tablet servers for reads/writes (not thru master).
- Bigtable cluster stores a number of tables, each table consists of a set of tablets and a tablet contains a row range

Bigtable and Chubby

- Bigtable uses Chubby to keep track of tablet servers:
  - Ensure there is at most one active master at a time,
  - Store the bootstrap location of Bigtable data (Root tablet),
  - Discover tablet servers and finalize tablet server deaths,
  - Store Bigtable schema information (column family info.),
  - Store access control list.

- If Chubby becomes unavailable for an extended period of time, Bigtable becomes unavailable.

Tablet Operations

- Tablet recovery:
  - Server reads metadata from METADATA table
  - Server reads indices of SSTables into memory and reconstructs memtable
- Write operation:
  - A log record is generated to the commit log file of redo records
  - Once the write commits, its contents are inserted into the memtable.
- Read operation:
  - Server ensures client has privileges for the read operation (Chubby),
  - Read is performed on a merged view of (a) the SSTables that constitute the tablet, and (b) the memtable.

Highlights of Bigtable

- Separate storage layer from data management.
- Restrict activity to one server.
- Key-value store with column families.
- Fault-tolerance achieved through:
  - Chubby
  - GFS
- Master-based approach for server/tablet management

PNUTS Overview

- Massively parallel and geographically distributed database system.
- Data organized as **hashed** or **ordered** tables.
- Low latency for concurrent updates and queries
- Novel per-read consistency
- Centrally managed, geographically distributed, automated load-balancing and failover

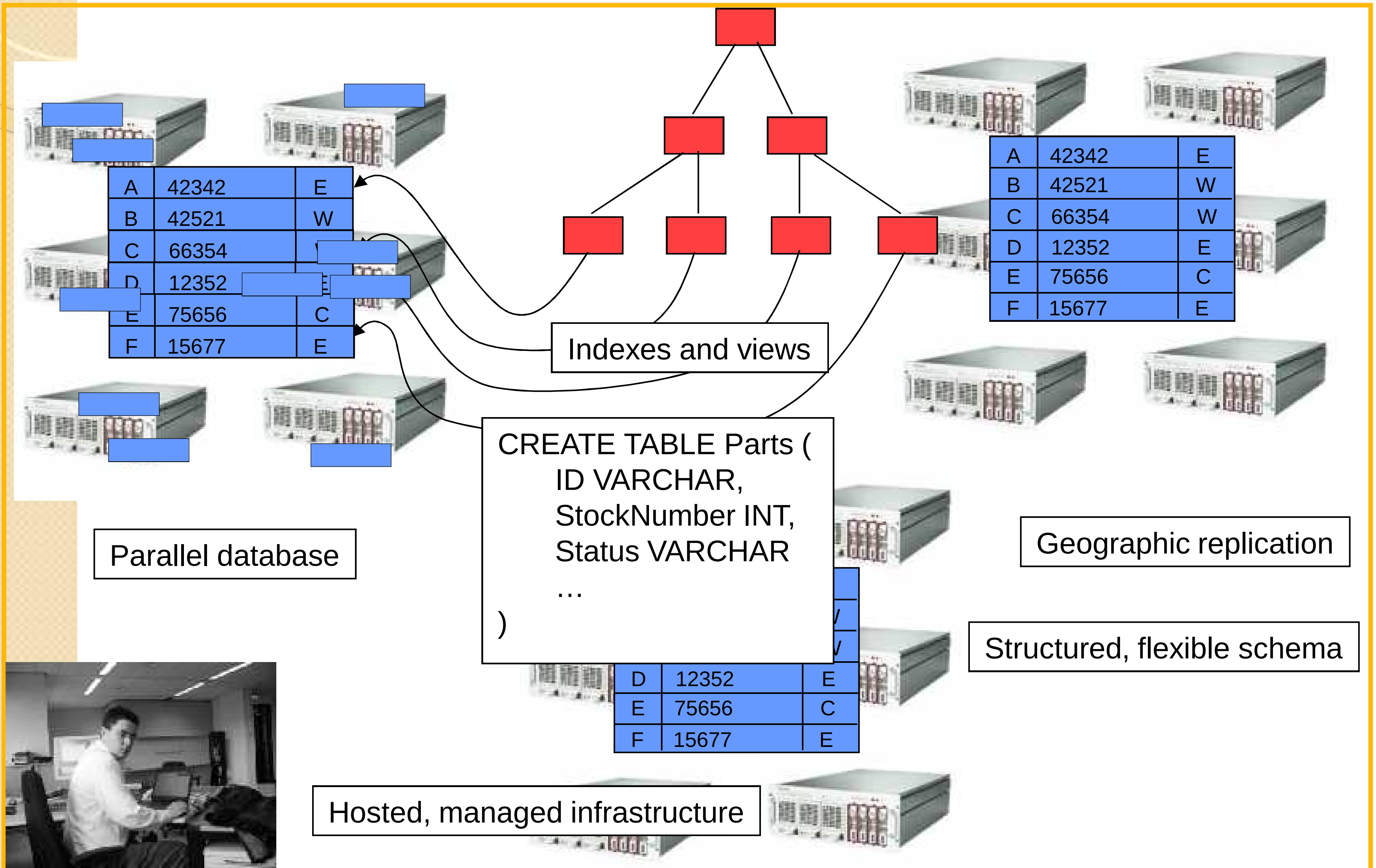
PNUTS Overview

- Data Model:
  - Simple relational model—really key-value store.
  - Single-table scans with predicates
- Fault-tolerance:
  - Redundancy at multiple levels: data, meta-data etc.
  - Leverages relaxed consistency for high availability: reads & writes despite failures
- Pub/Sub Message System:
  - Yahoo! Message Broker for asynchronous updates

PNUTS Overview

- Record-level Mastering:
  - Asynchronous operations to enable record-level mastering
- Hosting:
  - Centrally managed database service
  - Shared among many applications

PNUTS Architecture



Data Model

- Table of records with attributes
- "BLOB" is a valid data-type (exclude image/audio etc.)
- Flexible schema:
  - Attributes can be added dynamically
  - (No mention of dropping attributes ☹)
  - Records not required to have values for all attributes (i.e., integrity constraints minimal)

Query model

- Per-record operations
  - Get
  - Set
  - Delete
- Multi-record operations
  - Multi-get
  - Scan
  - Get-range
- Caveats:
  - No referential integrity
  - No complex operations: joins, group-by, etc.

Asynchronous replication

Consistency Model

- Hide the complexity of data replication
- Between the two extremes:
  - One-copy serializability, and
  - Eventual consistency
- Key assumption:
  - Applications manipulate one record at a time
- Per-record **time-line consistency**:
  - All replicas of a record preserve the update order

Implementation

- A read returns a consistent version
- One replica designated as master (per record)
- All updates forwarded to that master
- Master designation adaptive, replica with most of writes becomes master
- A sequence number
- Only one version of record/replica

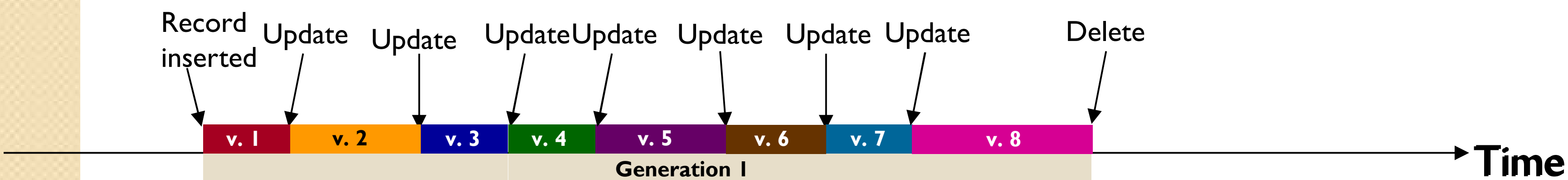
API Calls

- Read-Any:
  - Returns (possibly) a stale version of the record
- Read-Critical (required-version):
  - Version ≥ required-version
- Read-latest:
  - Executed at the master
- Write:
  - ACID guarantees with a single write operation
- TestAndSet (required-version):
  - Performs write if and only if the presented version = required-version

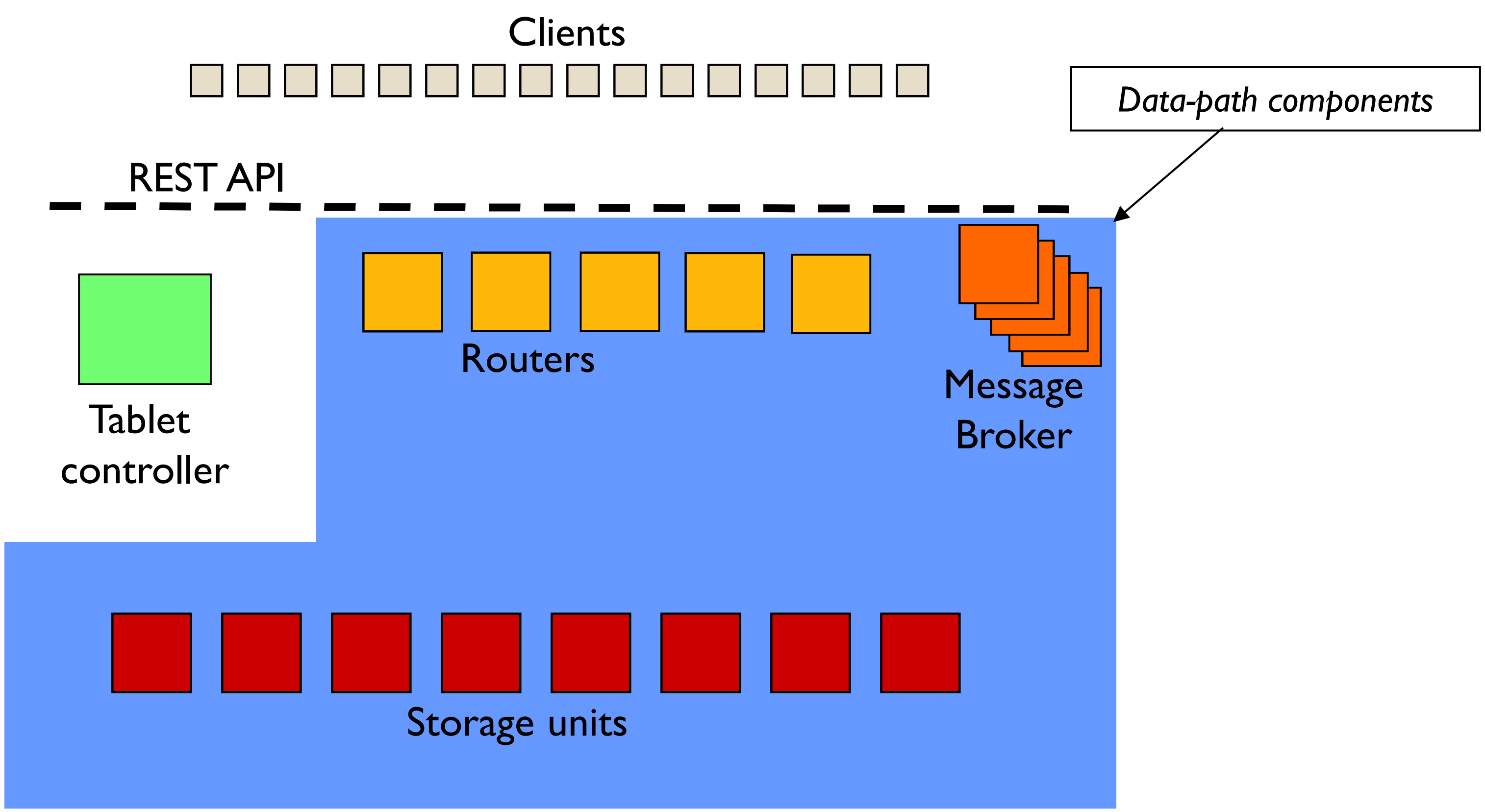
➔ Synchronizes concurrent writers, optimistically

Consistency model

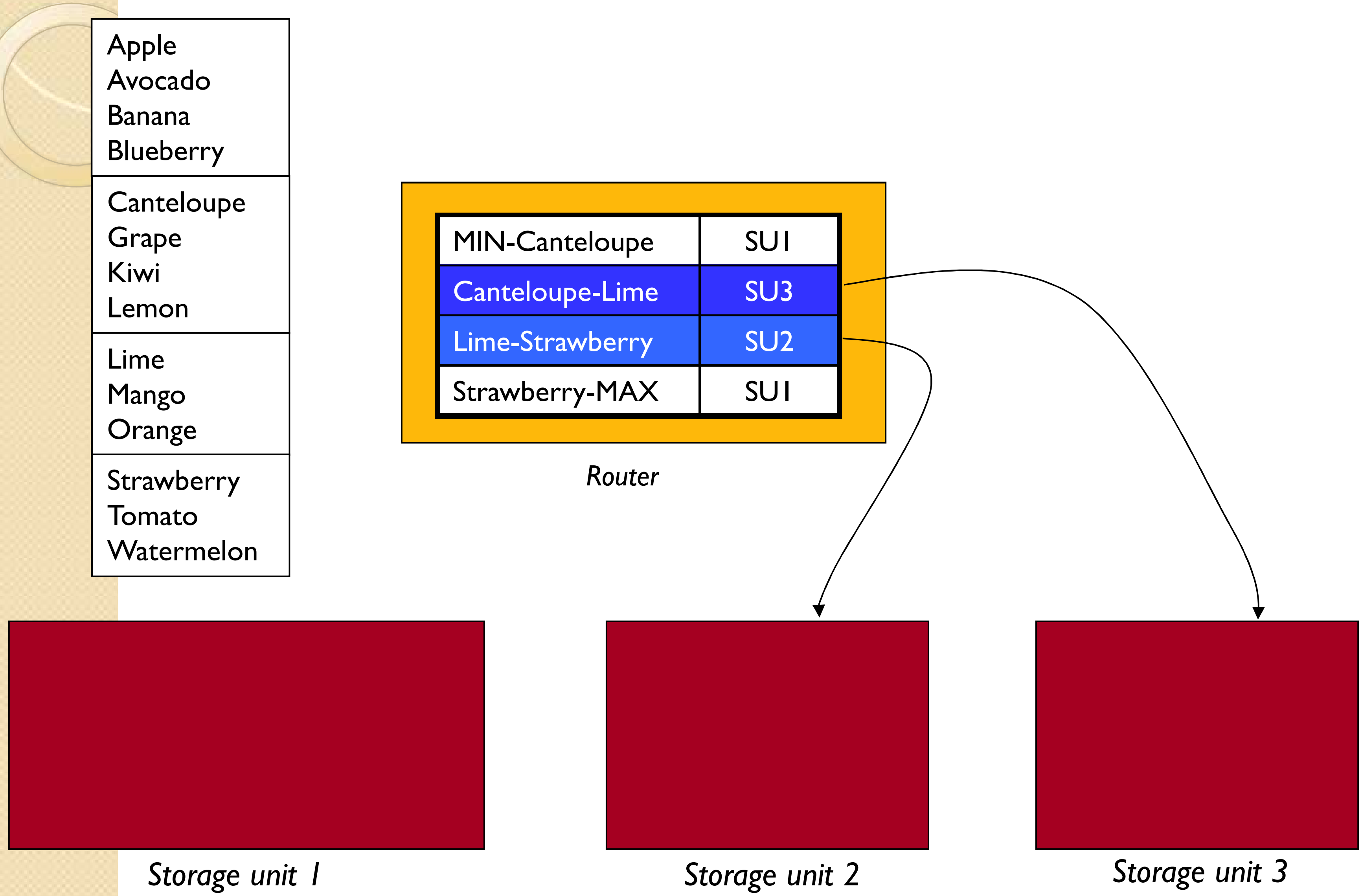
- Goal: make it easier for applications to reason about updates and cope with asynchrony
- What happens to a record with primary key "Brian"?



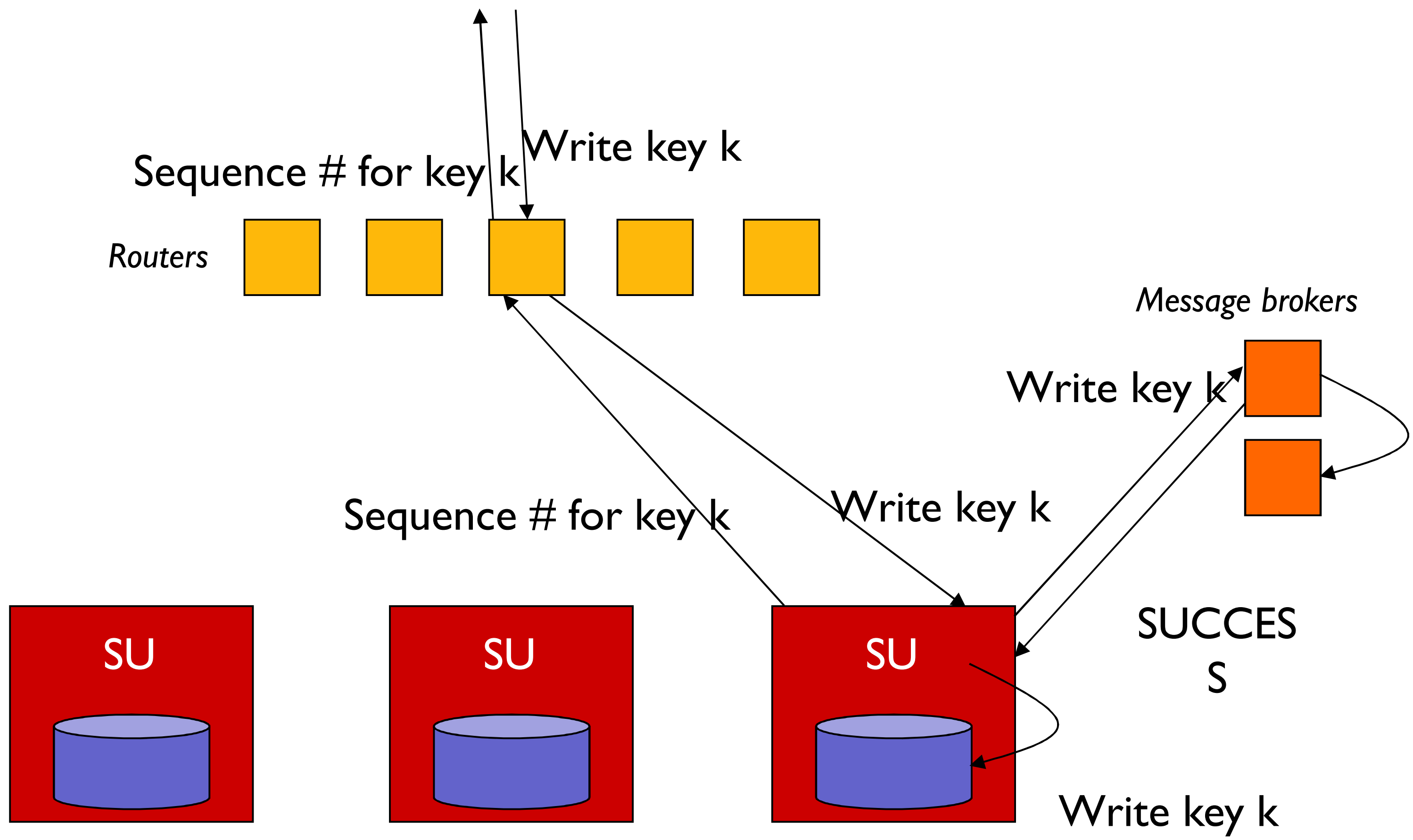
Detailed architecture



Request Routing



Updates



Write key k
Sequence # for key k
Routers
Message brokers
Write key k
Write key k
Sequence # for key k
SU
SU
SU
SUCCES
S
Write key k

Highlights of PNUTS Approach

- Shared nothing architecture
- Multiple datacenter for geographic distribution
- Time-line consistency and access to stale data.
- Use a publish-subscribe system for reliable fault-tolerant communication
- Replication with record-based master.

Dynamo Design Rationale

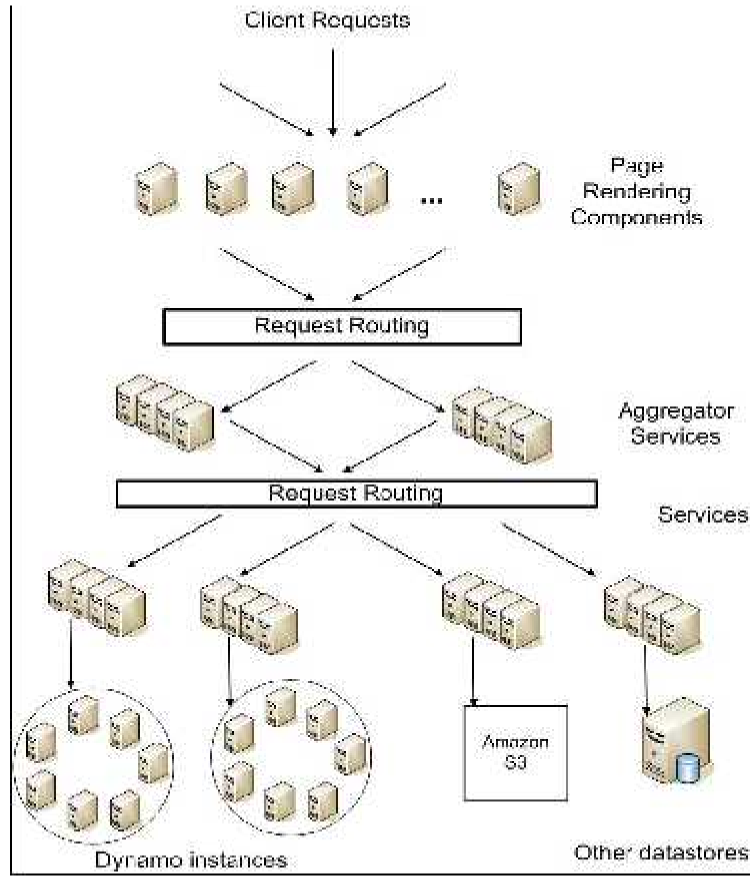
- Most services need key-based access:
  - Best-seller lists, shopping carts, customer preferences, session management, sales rank, product catalog, and so on.
- Prevalent application design based on RDBMS technology will be catastrophic.
- **Dynamo therefore provides primary-key only interface.**

Dynamo Design Overview

- Data partitioning using consistent hashing
- Data replication
- Consistency via version vectors
- Replica synchronization via quorum protocol
- Gossip-based failure-detection and membership protocol

Striving for Application Performance

- Application can deliver its functionality in a bounded time
- Example SLA: service guaranteeing that it will provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second.



Service-oriented architecture of Amazon's platform

Design Notes

- Optimistic/Asynchronous Replication:
  - Leads to update conflicts
  - Hence, need conflict resolution: eventual consistency
- When to resolve conflicts?
  - Traditionally at the time of Write ➔ risk of aborts
  - Reads simple
- Dynamo approach:
  - Always writeable
  - Conflict resolution **complexity** at Reads
- Who resolves the conflict:
  - Data store: limited choices; **syntactic**: last write wins.
  - Application: **semantic**: case-by case

Design Notes

- Incremental scalability:
  - Scale out: One storage node at a time
- Symmetry:
  - Peer-based design
  - Principle of equal responsibility
- Decentralization: decentralized Peer to Peer.
- Heterogeneity: in infrastructure.

Summary of techniques

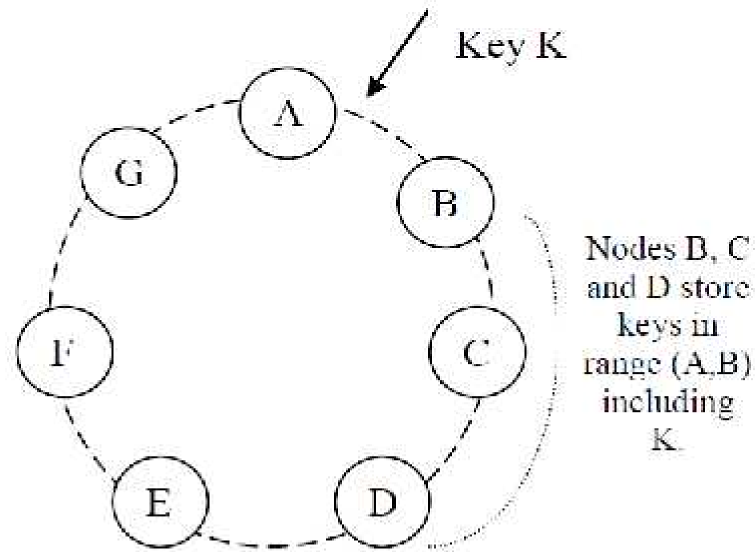
Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

System Interface

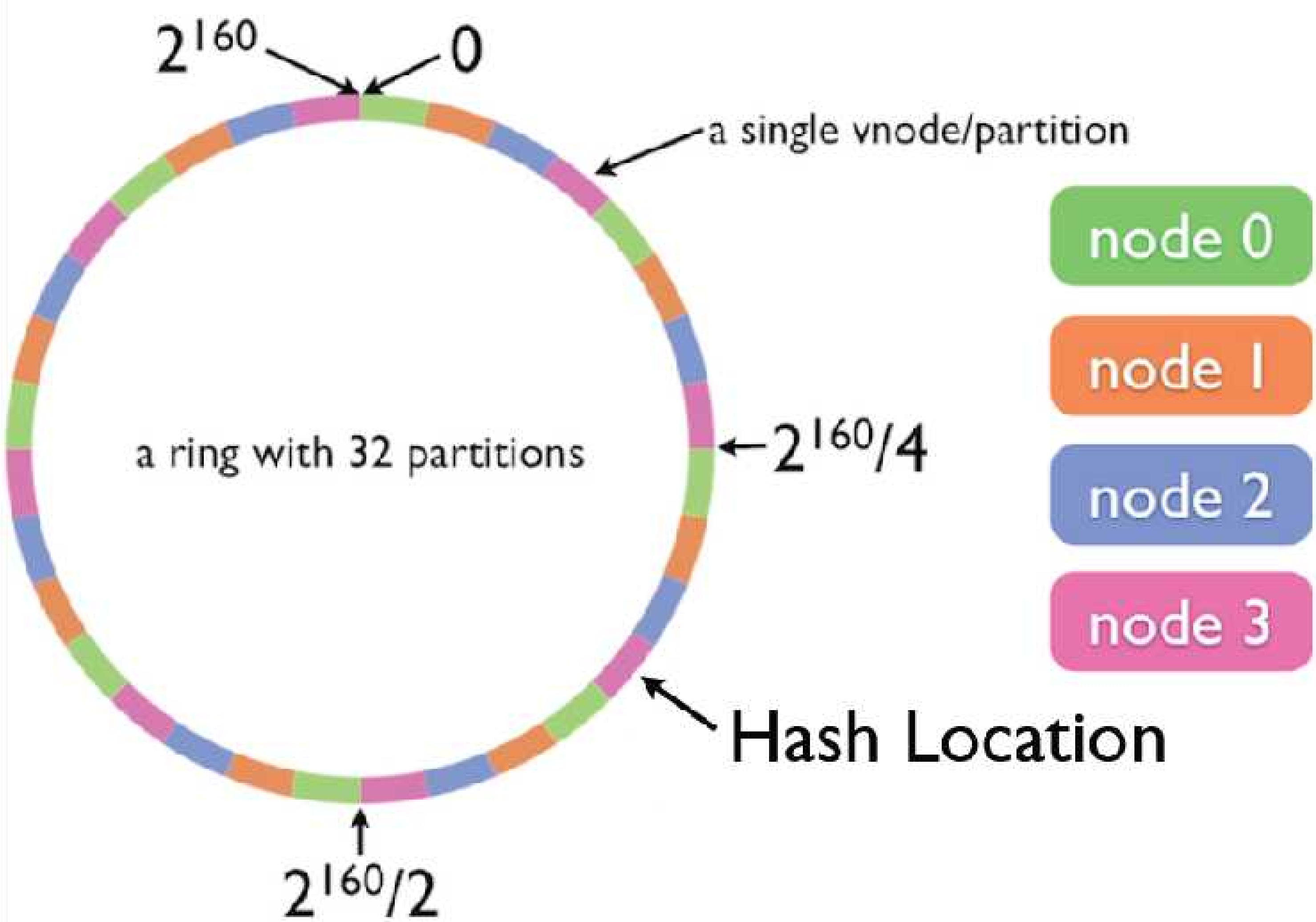
- Two basic operations:
  - Get(key):
    - Locates replicas
    - Returns the object + context (encodes meta data including version)
  - Put(key, context, object):
    - Writes the replicas to the disk
    - Context: version (vector timestamp)
- Hash(key) ➔ 128-bit identifier

Data Partitioning and Routing

- Consistent hashing: the output range of a hash functi is treated as a fixed circular space or "ring" a la Chord.
- "Virtual Nodes": Each noc can be responsible for more than one virtual node (to dea with non-uniform data and lc distribution)

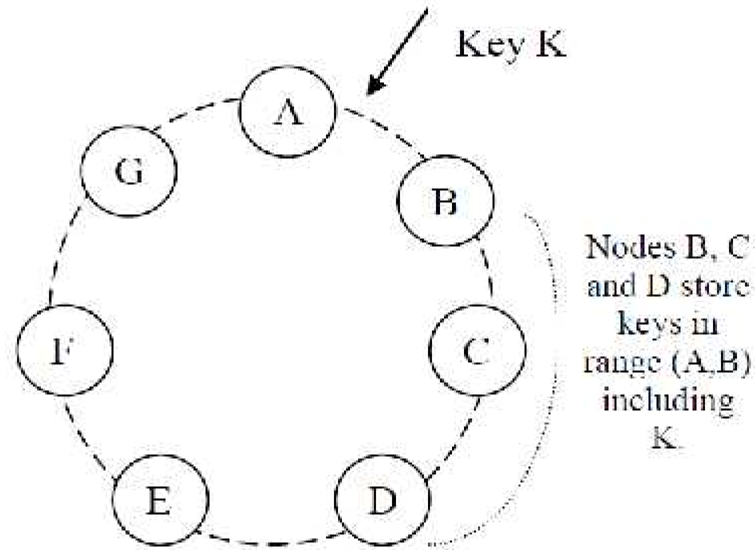


Virtual Nodes

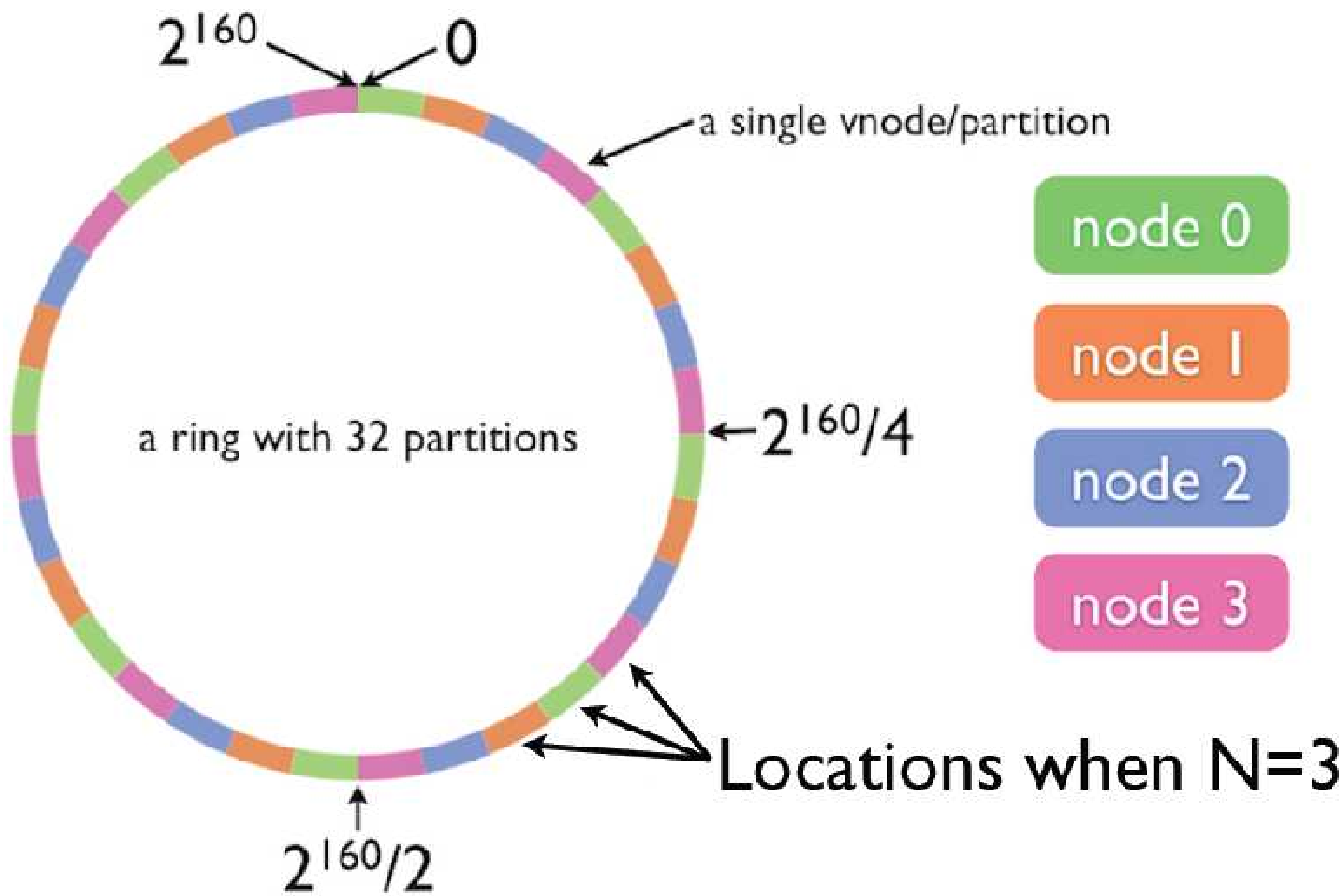


Replication

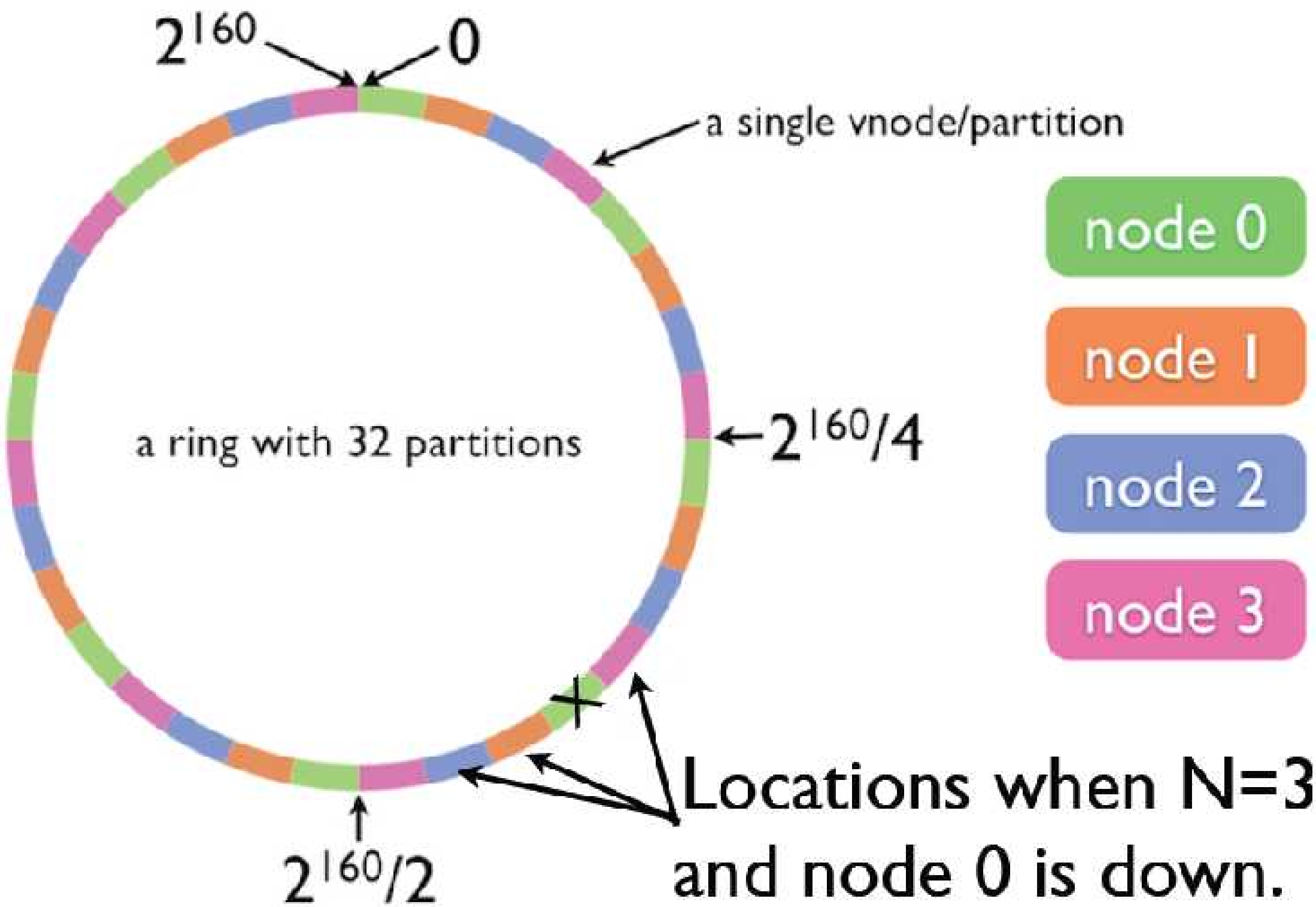
- Each data item is replicated at N hosts.
- *preference list*: The list of nodes that is responsible for storing a particular key.
- Some fine-tuning to account for virtual nodes



Replication



Replication



Data Versioning

- A put() call may return to its caller before the update has been applied at all the replicas
- A get() call may return many versions of the same object.
- Challenge: an object may have distinct versions
- Solution: use vector clocks in order to capture causality between different versions of same object.

Vector Clock

- A vector clock is a list of (node, counter) pairs.
- Every version of every object is associated with one vector clock.
- If the all counters on the first object's clock are less-than-or-equal to all of the counters in the second clock, then the first is an ancestor of the second and can be forgotten.
- Application reconciles divergent versions and collapses into a single new version.

Routing requests

- Route request through a generic load balancer that will select a node based on load information.
- Use a partition-aware client library that routes requests directly to relevant node.
- A gossip protocol propagates membership changes. Each node contacts a peer chosen at random every second and the two nodes reconcile their membership change histories.

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.
- In this model, the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas. For this reason, R and W are usually configured to be less than N, to provide better latency and availability.

Discussion

- Three different approaches to designing scalable data stores
- Many open-source variants inspired by these designs
  - HBase, Cassandra, Voldemort, Riak, …
- Main memory object stores are another form of key-value store
  - Memcached, Redis, …

References

- Leslie Lamport: Time, Clocks, and the Ordering of Events in a Distributed System. Commun. ACM 21(7): 558-565 (1978)
- Mani Chandy, Leslie Lamport: Distributed Snapshots: Determining Global States of Distributed Systems ACM Trans. Comput. Syst. 3(1): 63-75 (1985)
- Gene T. J. Wuu, Arthur J. Bernstein: Efficient Solutions to the Replicated Log and Dictionart Problems. PODC 1984: 233-242
- Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in P2P systems. In Communications of the ACM, February 2003
- Reliable Distributed Computing with the Isis Toolkit. K. Birman and R. van Renesse, eds. IEEE Computer Society Press, 1994.

References

- Leslie Lamport, Robert E. Shostak, Marshall C. Pease: The Byzantine Generals Problem. ACM Trans. Program. Lang. Syst. 4(3): 382-401 (1982)
- Leslie Lamport: The Part-Time Parliament. ACM Trans. Comput. Syst. 16(2): 133-169 (1998)
- Michael J. Fischer, Nancy A. Lynch, Mike Paterson: Impossibility of Distributed Consensus with One Faulty Process. PODS 1983: 1-7
- Eric A. Brewer. Towards robust distributed systems. (Invited Talk)Principles of Distributed Computing, July 2000.

References

- Chang, Dean, Ghemawat, Hsieh, Wallach, Burrows, Chandra, Fikes, Gruber: Bigtable: A Distributed Storage System for Structured Data. OSDI 2006
- The Google File System: Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. Symp on Operating Systems Princ 2003.
- GFS: Evolution on Fast-Forward: Kirk McKusick, Sean Quinlan Communications of the ACM 2010.
- Cooper, Ramakrishnan, Srivastava, Silberstein, Bohannon, Jacobsen, Puz, Weaver, Yerneni: PNUTS: Yahoo!'s hosted data serving platform. VLDB 2008.
- DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, Vogels: Dynamo: amazon's highly available key-value store. SOSP 2007
- Cooper, Silberstein, Tam, Ramakrishnan, Sears: Benchmarking cloud serving systems with YCSB. SoCC 2010