Προχωρημένη Κατανεμημένη Υπολογιστική

HY623

Διδάσκων –
Δημήτριος Κατσαρός

@ Τμ. ΗΜΜΥ
Πανεπιστήμιο Θεσσαλίας

Διάλεξη 2η

MapReduce solutions: k-means clustering

The k-means Clustering Algorithm
Input : Data points D, Number of clusters k
Step 1: Initialize k centroids randomly
Step 2: Associate each data point in D with the nearest centroid. This will divide the data points into k clusters.
Step 3: Recalculate the position of centroids.
Repeat steps 2 and 3 until there are no more changes in the membership of the data points
Output : Data points with cluster memberships

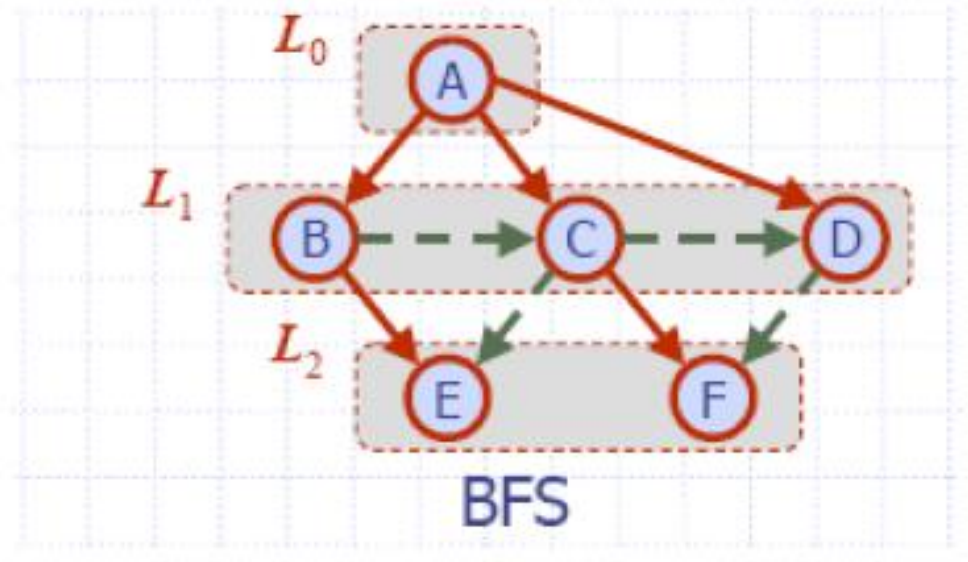
k-means::Map
Input: Data points D, number of clusters k and centroids
1: for each data point d ∈ D do
2: Assign d to the closest centroid
Output: centroids with associated data points

---

k-means::Reduce
Input: Centroids with associated data points
1: Compute the new centroids by calculating the average of data points in cluster
2: Write the global centroids to the disk
Output: New centroids

MapReduce solutions: BFS traversal

- Algorithm:
  - Input: Simple Connected directed graph with 'n' vertices and the node to be searched.
  - Output: if node is found "**Yes**" is printed and the corresponding path is displayed
    else "No" is printed.



MapReduce solutions: BFS traversal

- Graph is represented as adjacency list.
  - Key: Node ID
  - Value: EDGES|DISTANCE_FROM_SOURCE|COLOR|
    - where EDGES is a comma delimited list of the ids of the nodes that are connected to this node. in the beginning, we do not know the distance and will use Integer.MAX_VALUE for marking "unknown". color tells us whether or not we've seen the node before, so this starts off as white.
  - Eg:

Key	Value
1	2,5|0|GRAY|
2	1,3,4,5|Integer.MAX_VALUE|WHITE|
3	2,4|Integer.MAX_VALUE|WHITE|
4	2,3,5|Integer.MAX_VALUE|WHITE|
5	1,2,4|Integer.MAX_VALUE|WHITE|

MapReduce solutions: BFS traversal

- Map Function:
  - For each gray node, the mappers emit a new gray node, with distance = distance + 1. they also then emit the input gray node, but colored black. (once a node has been exploded, we're done with it.) mappers also emit all non-gray nodes, with no change. so, the output of the first map iteration would be

1	2,5|0|BLACK|
2	NULL|1|GRAY|
5	NULL|1|GRAY|
2	1,3,4,5|Integer.MAX_VALUE|WHITE|
3	2,4|Integer.MAX_VALUE|WHITE|
4	2,3,5|Integer.MAX_VALUE|WHITE|
5	1,2,4|Integer.MAX_VALUE|WHITE|

Note: When the mappers "explode" the gray nodes and create a new node for each edge, they do not know what to write for the edges of this new node - so they leave it blank

MapReduce solutions: BFS traversal

- Reduce Function:
  - Reducers, receives all data for a given key - in this case it means that they receive the data for all "copies" of each node.
  - for example, the reducer that receives the data for key = 2 gets the following list of values :

    2 NULL|1|GRAY|

    2 1,3,4,5|Integer.MAX_VALUE|WHITE|
  - the reducers job is to take all this data and construct a new node using
    - the non-null list of edges
    - the minimum distance
    - the darkest color

MapReduce solutions: BFS traversal

- Iterations
  - using this logic the output from our first iteration will be :

```
1    2,5,|0|BLACK
2    1,3,4,5,|1|GRAY
3    2,4,|Integer.MAX_VALUE|WHITE
4    2,3,5,|Integer.MAX_VALUE|WHITE
5    1,2,4,|1|GRAY
```

  - the second iteration uses this as the input and outputs :

```
1    2,5,|0|BLACK
2    1,3,4,5,|1|BLACK
3    2,4,|2|GRAY
4    2,3,5,|2|GRAY
5    1,2,4,|1|BLACK
```

  - and the third iteration outputs:

```
1    2,5,|0|BLACK
2    1,3,4,5,|1|BLACK
3    2,4,|2|BLACK
4    2,3,5,|2|BLACK
5    1,2,4,|1|BLACK
```

  - subsequent iterations will continue to print out the same output.

MapReduce solutions: BFS traversal

- When should we terminate search?
  - Case1:if all the vertices are visited i.e colored black.(Here Output is “No”)
  - Case2:When mapper finds the destined node with color gray(i.e when mapper visits destined node for first time).(Here Output is “Yes”)
  - In above cases crux is usage of shared variables between mapper/reducer and main program.

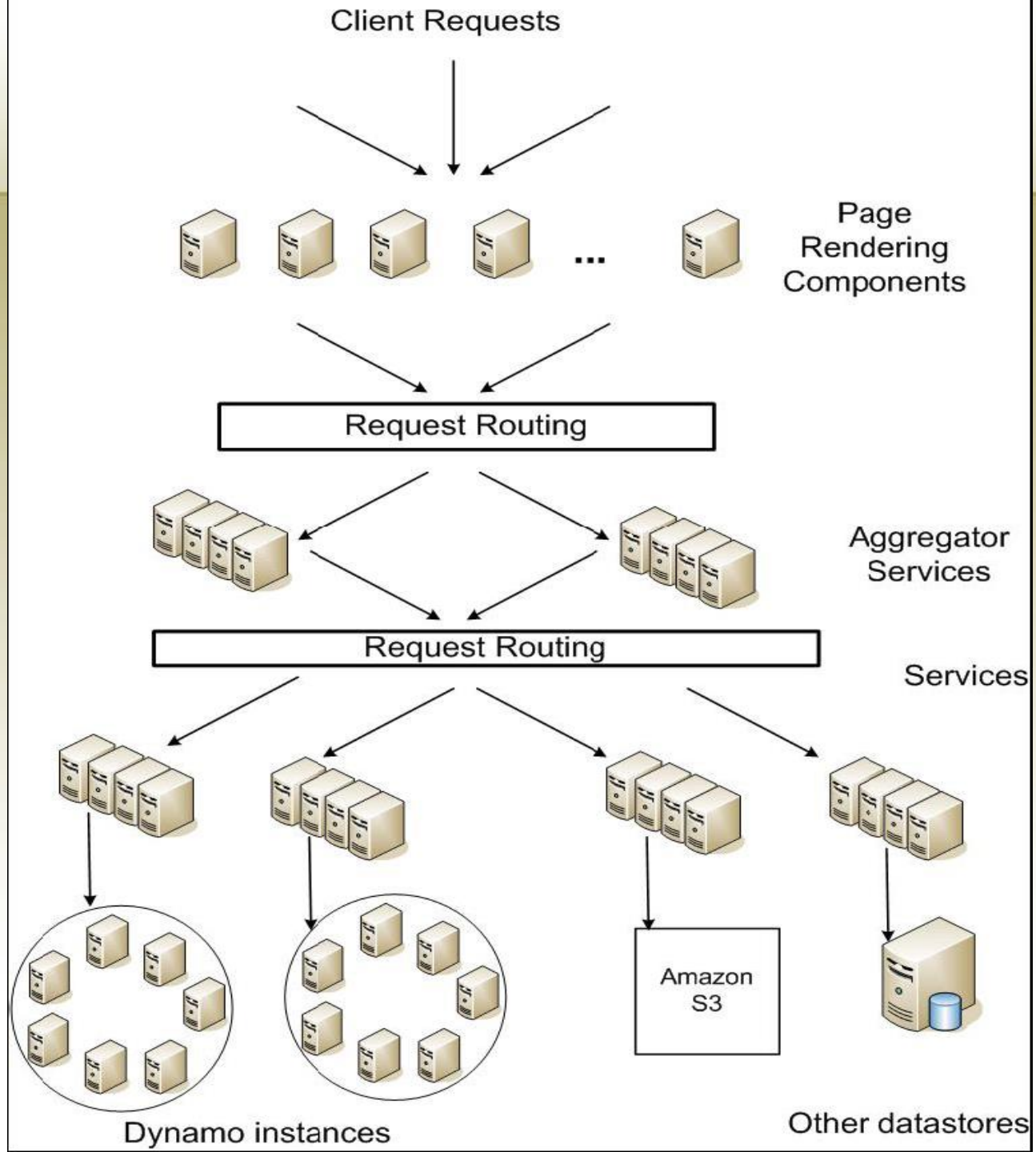
MapReduce solutions: BFS traversal

- Shared Variables are not supported in hadoop.

  We Addressed this issue by serializing mapper's object to HDFS and deserializing those objects in main program.

Dynamo

Amazon's Highly Available Key-value Store


Client Requests
...
Page Rendering Components
Request Routing
Aggregator Services
Request Routing
Services
Amazon S3
Dynamo instances
Other datastores

Amazon eCommerce platform

Scale

- > 10M customers at peak times
- > 10K servers (distributed around the world)
- 3M checkout operations per day (peak season)

Problem: Reliability/availability at massive scale:

- Slightest outage has significant financial consequences and impacts customer trust

Amazon eCommerce platform - Requirements

Key requirements:

- Data/service availability the key issue.
- "Always writeable" data-store
- Low latency – delivered to (almost all) clients/users
  - Example SLA: provide a 300ms response time, for 99.9% of requests, for a peak load of 500requests/sec.
  - Why not average/median?

Architectural requirements

- Incremental scalability
- Symmetry
- Ability to run on a heterogeneous platform

Data Access Model

- Data stored as (key, object) pairs:
  - Interface *put(key, object), get(key)*
    - 'identifier' (key) generated as a hash for object
  - Objects: Opaque
  - Application examples: shopping carts, customer preferences, session management, sales rank, product catalog, S3

Further assumptions:

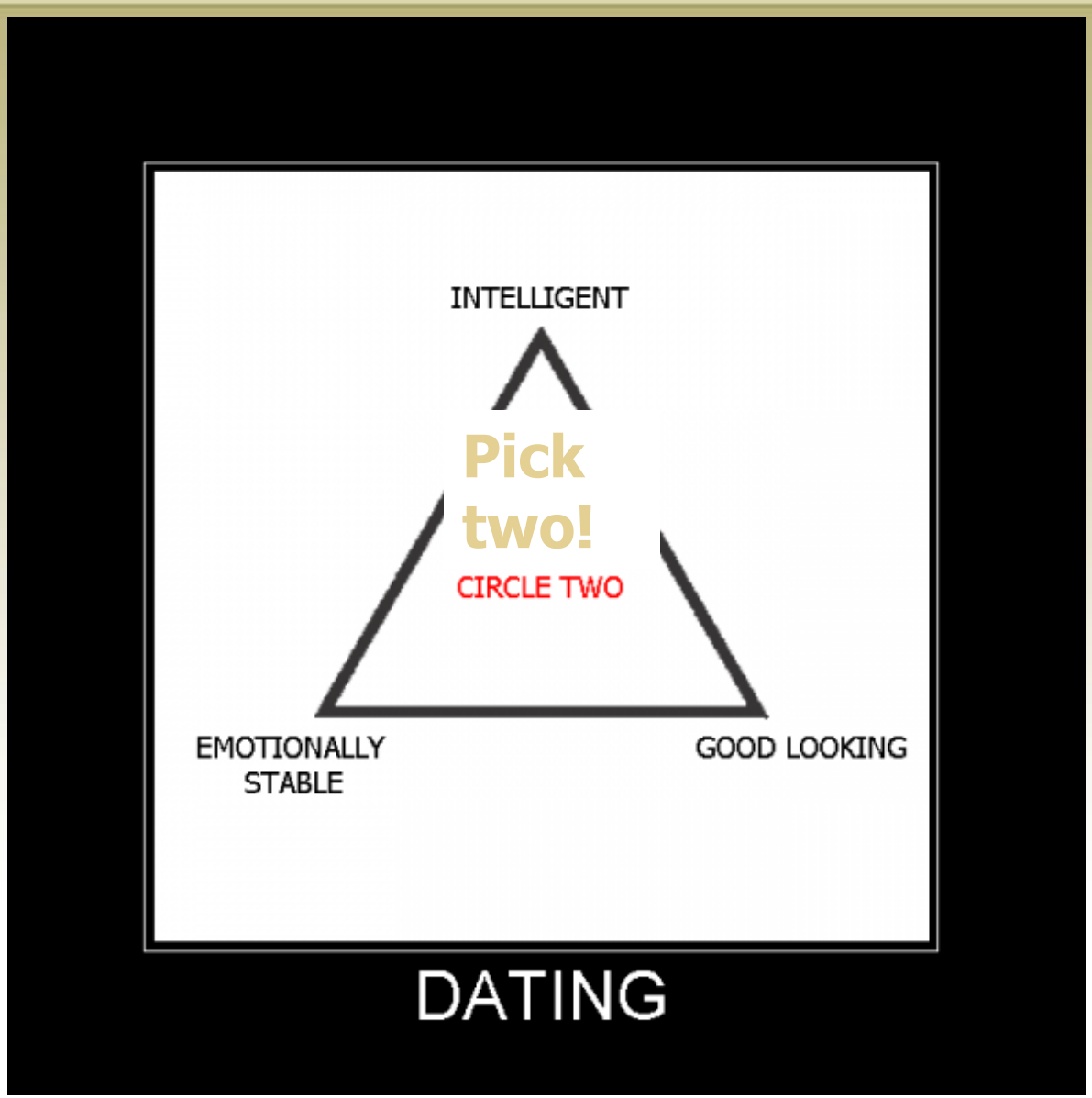
- Relatively small objects (<1MB)
- Operations do not span multiple objects
- Friendly (cooperative) environment,
- One Dynamo instance per service → 100s hosts/service

A database?

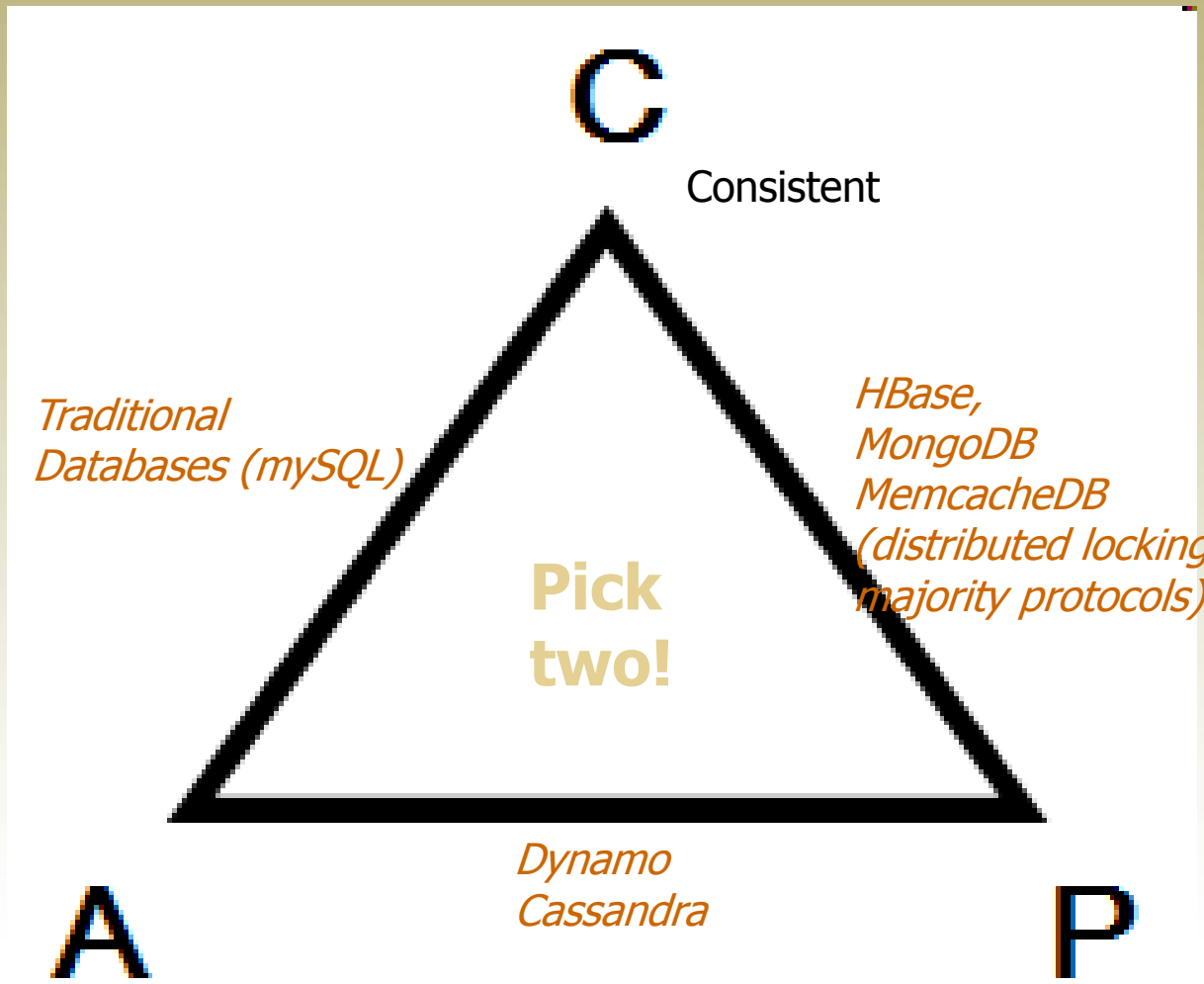
Concerns here: Consistent + Available + Partition tolerant

- Can one get all three?

Dating conjecture



CAP theorem


C
Consistent
Traditional
Databases (mySQL)
HBase,
MongoDB
MemcacheDB
(distributed locking,
majority protocols)
Pick
two!
Dynamo
Cassandra
A
Availabile
P
Partition tolerant

Key ideas

Requirements:

- High data availability; always writeable data store

Solution idea

- Multiple replicas …
- … but avoid synchronous replica coordination …

  (used by solutions that provide strong consistency).
  - Tradeoff: Consistency ←→ Availability
- .. and use 'weak consistency' models to improve availability
  - The problems this introduces: when to resolve possible conflicts and who should solve them
    - **When**: at read time (allows providing an "always writeable" data store)
    - **Who**: *the application* or the data store

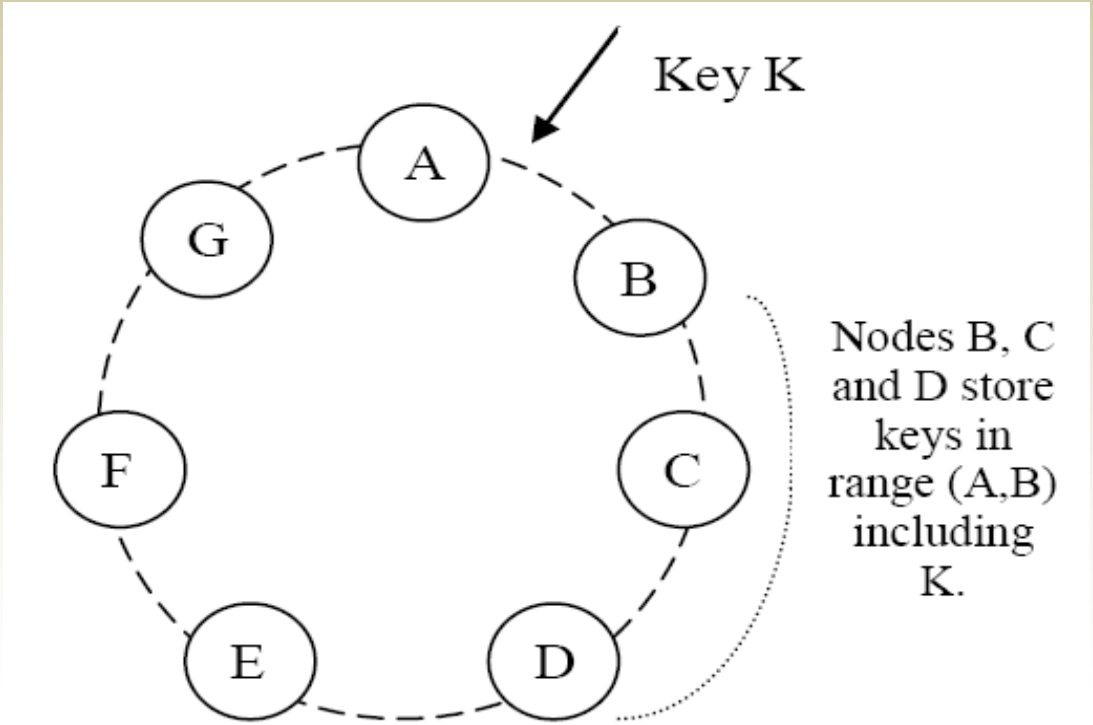
Key technical issues

- Partitioning the key/data space
- High availability for writes
- Handling temporary failures
- Recovering from permanent failures
- Membership and failure detection

Problem	Technique	Advantage
Partitioning	• Consistent hashing	Incremental scalability
High availability for writes	• Eventual consistency • Vector clocks with reconciliation during reads	Availability
Handling temporary failures	• 'Sloppy' quorum protocol 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	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Partition Algorithm: Consistent hashing

- Each data item is replicated at N hosts (successors)



Quorum systems

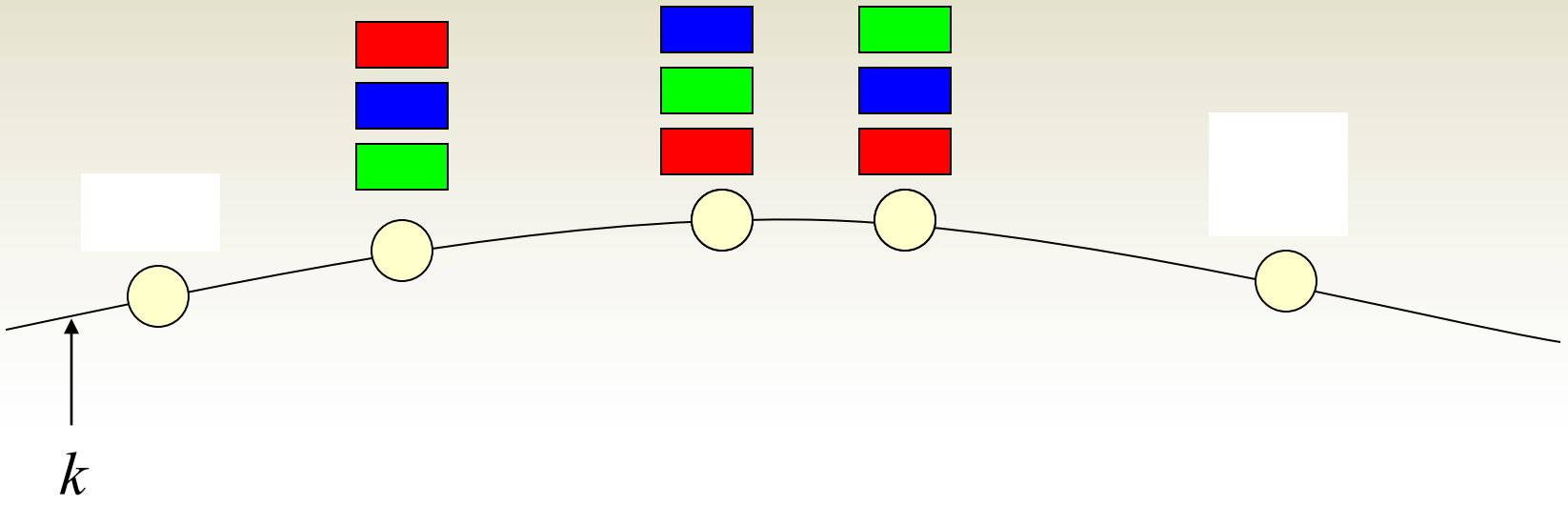
- Multiple replicas to provide durability …
  - … but avoid synchronous replica coordination …

Traditional quorum system:

- R/W: the minimum number of nodes that must participate in a successful read/write operation.
- Problem: the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas.
  - To provide better latency R and W are usually configured to be less than N.
- R + W > N yields a quorum-like system.
  - 'Sloppy quorum' in Dynamo

Why replication is tricky

- The claim: Dynamo will replicate each data item on N successors
  - A pair (k,v) is stored by the node closest to k and replicated on N successors of that node



Data versioning

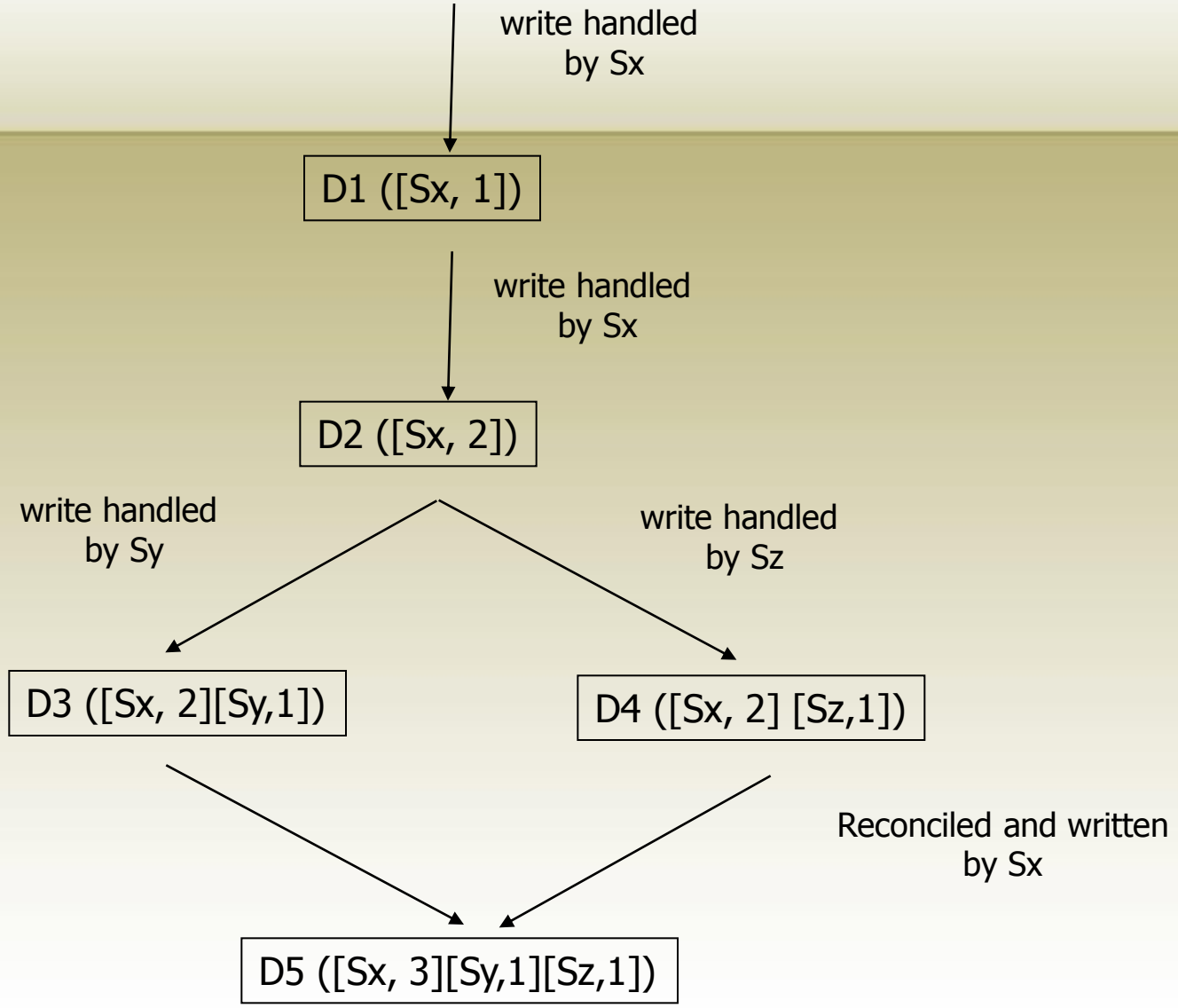
- Multiple replicas …
  - … but avoid synchronous replica coordination …
- The problems this introduces:
  - when to resolve possible conflicts?
    - at read time (allows providing an "always writeable" data store)
    - A *put()* call may return to its caller before the update has been applied at all the replicas
    - A *get()* call may return different versions of the same object.
  - who should solve them
    - the application → use vector clocks to capture causality ordering between different versions of the same object.
    - the data store → use logical clocks or physical time

Vector Clocks

- Each version of each object has one associated vector clock.
  - list of (node, counter) pairs.

Reconciliation:

- If the counters on the first object's clock are less-than-or-equal than all of the nodes in the second clock, then the first is a direct ancestor of the second (and can be ignored).
- Otherwise: application-level reconciliation


write handled by Sx
D1 ([Sx, 1])
write handled by Sx
D2 ([Sx, 2])
write handled by Sy
write handled by Sz
D3 ([Sx, 2][Sy,1])
D4 ([Sx, 2] [Sz,1])
Reconciled and written by Sx
D5 ([Sx, 3][Sy,1][Sz,1])

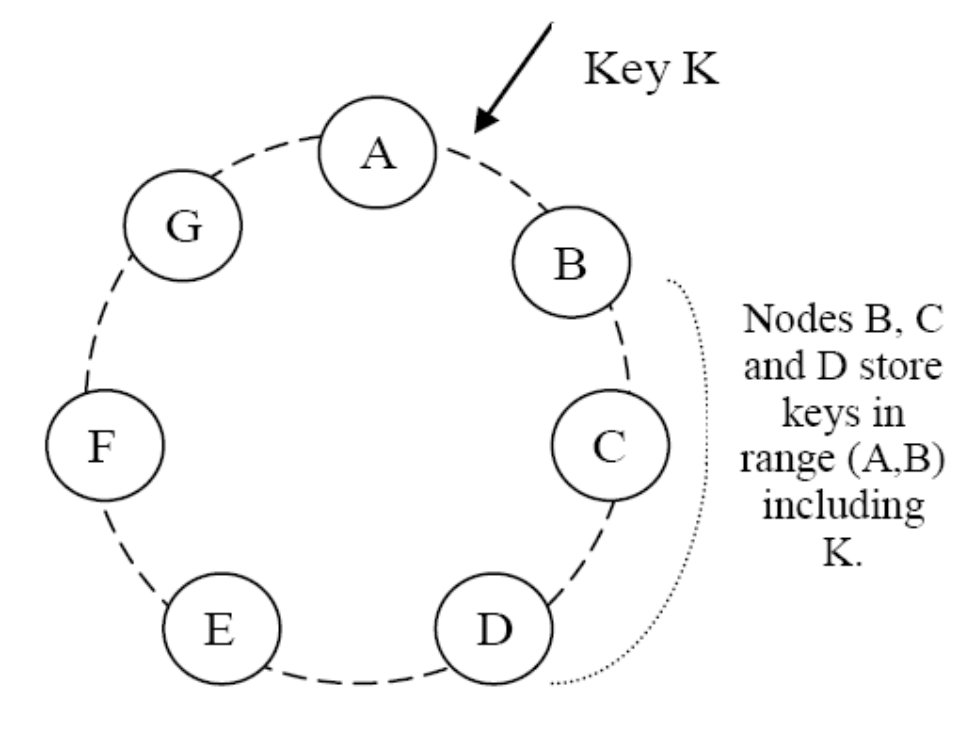
Problem	Technique	Advantage
Partitioning	• Consistent hashing	Incremental scalability
High availability for writes	• Eventual consistency • Vector clocks with reconciliation during reads	
Handling temporary failures	• 'Sloppy' quorum protocol and hinted handoff	Provides high availability and durability 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	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Other techniques

- Node synchronization:
  - Hinted handoff
  - Merkle hash tree.

"Hinted handoff"

- Assume replications factor N = 3. When A is temporarily down or unreachable during a write, send replica to D.
- D is hinted that the replica belongs to A and it will deliver to A when A is recovered.
- Again: "always writeable"

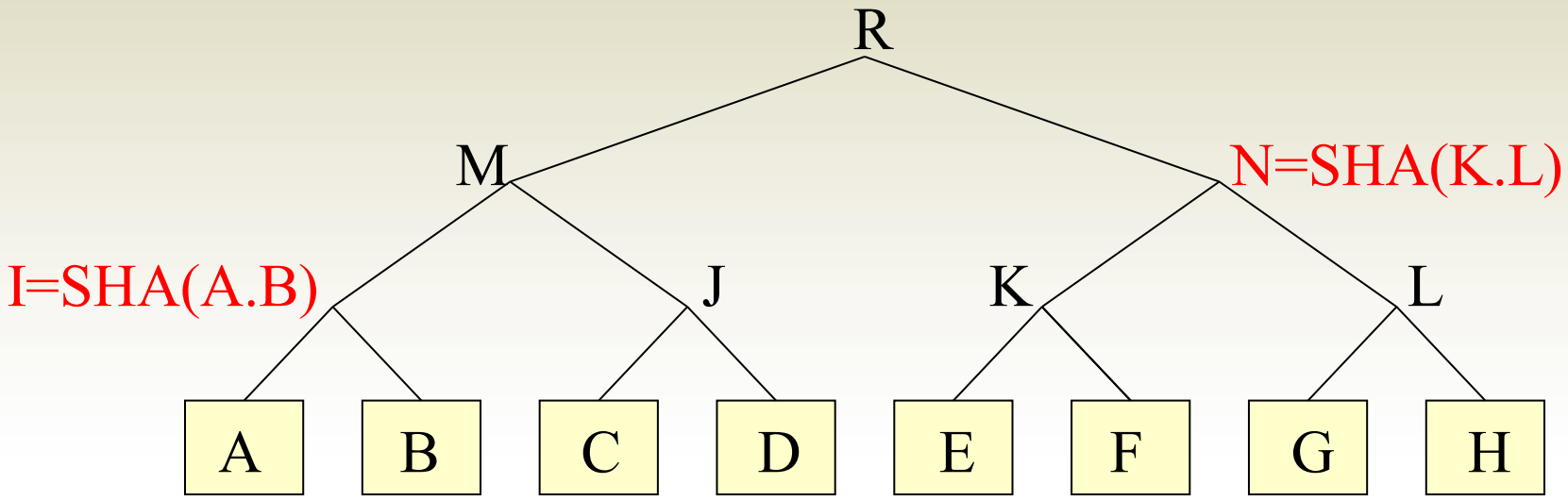


Replication Meets Epidemics

- Candidate Algorithm
  - For each (k,v) stored locally, compute SHA($k.v$)
  - Every period, pick a random leaf set neighbor
  - Ask neighbor for all its hashes
  - For each unrecognized hash, ask for key and value
- This is an epidemic algorithm
  - All N members will have all (k,v) in *log(N)* periods
  - But as is, the cost is O(C), where C is the size of the set of items stored at the original node

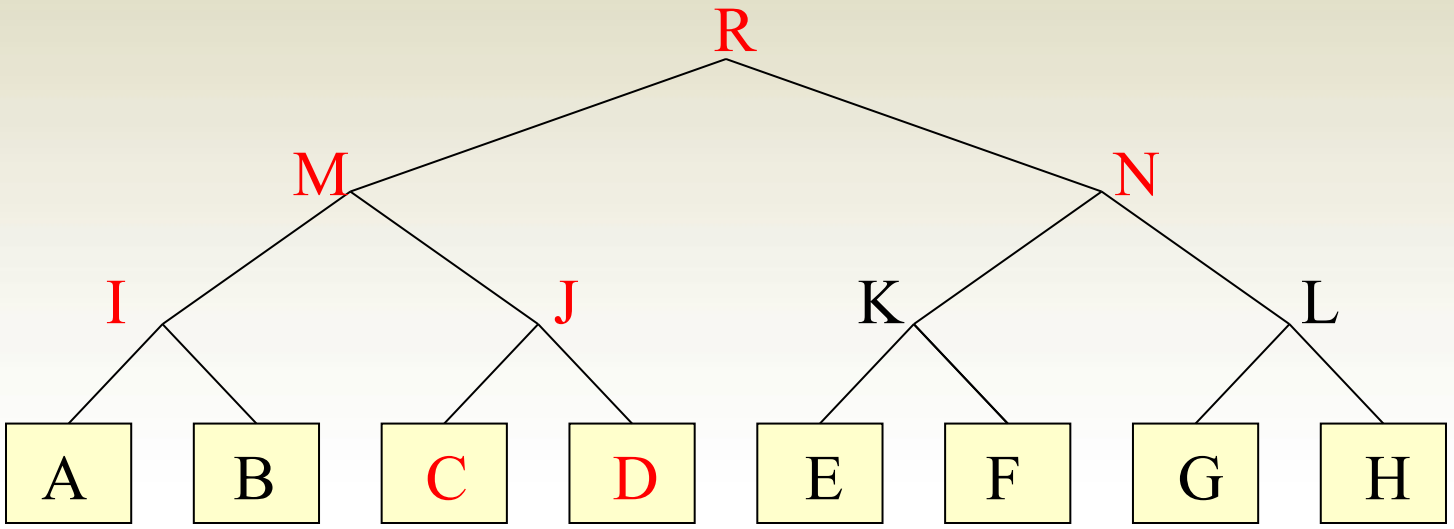
Merkle Trees

- An efficient summarization technique
  - Interior nodes are the secure hashes of their children
  - E.g., I = SHA(A.B), N = SHA(K.L), etc.



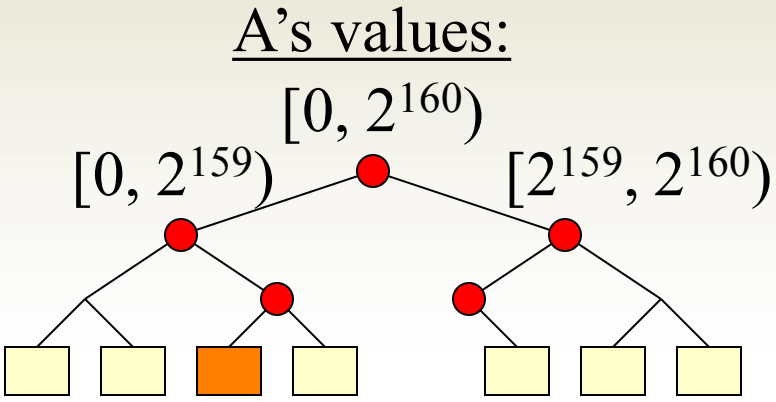
Merkle Trees

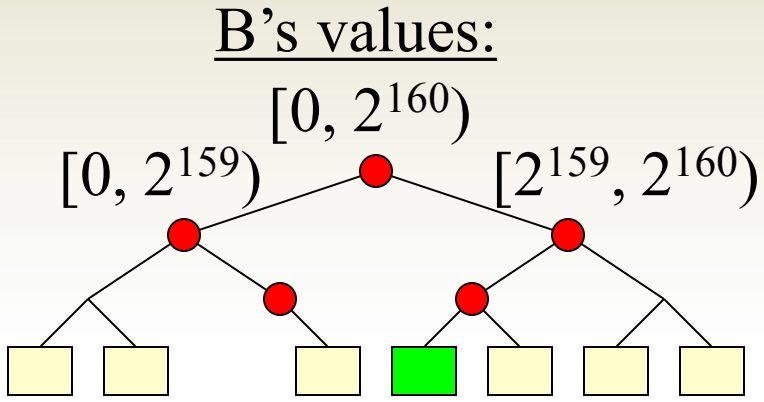
- Merkle trees are an efficient summary technique
  - If the top node is signed and distributed, this signature can later be used to verify any individual block, using only O(log *n*) nodes, where *n* = # of leaves
  - E.g., to verify block C, need only R, N, I, C, & D



Using Merkle Trees as Summaries

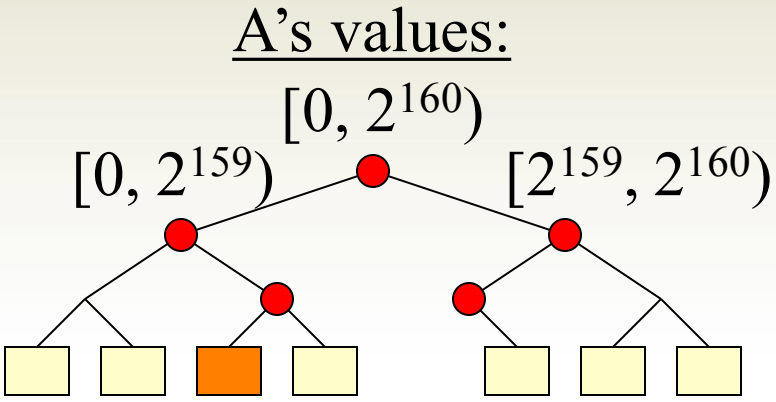
- Improvement: use Merkle tree to summarize keys
  - B gets tree root from A, if same as local root, done
  - Otherwise, recurse down tree to find difference

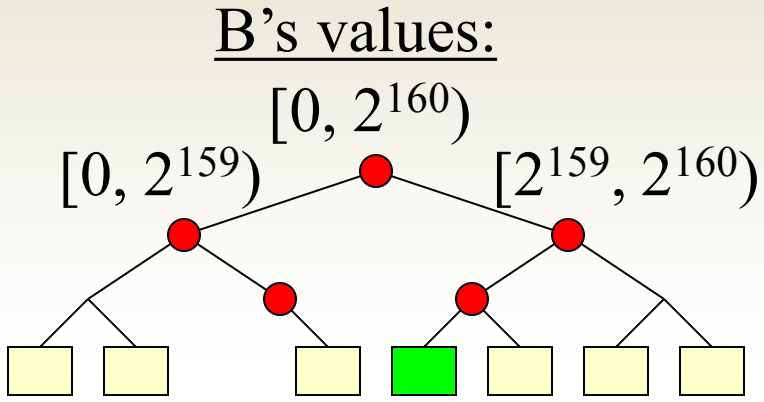




Using Merkle Trees as Summaries

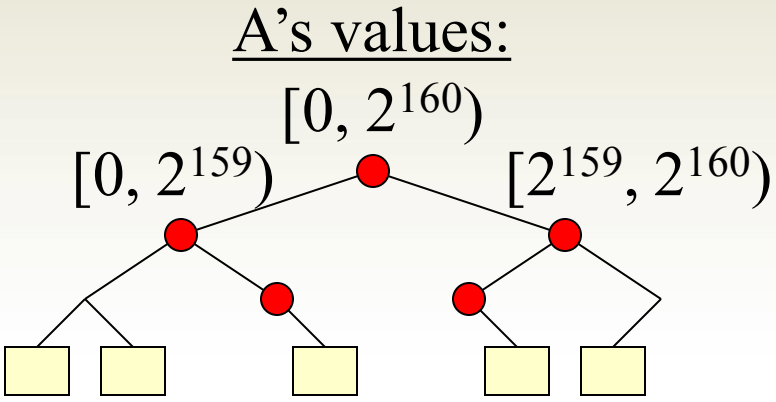
- Improvement: use Merkle tree to summarize keys
  - B gets tree root from A, if same as local root, done
  - Otherwise, recurse down tree to find difference
- New cost is O(d log C)
  - d = number of differences, C = size of disk

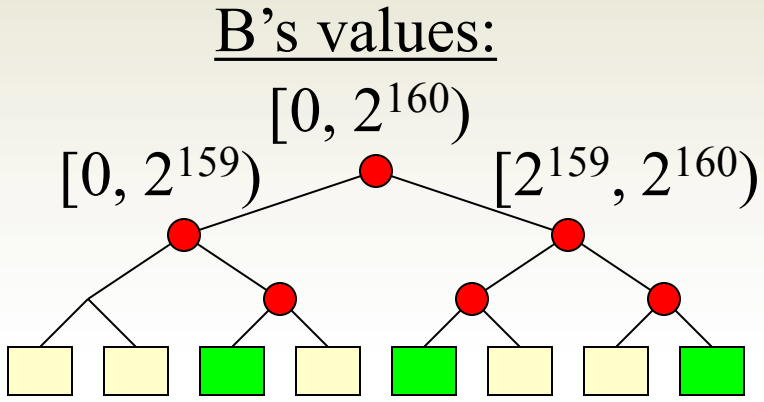




Using Merkle Trees as Summaries

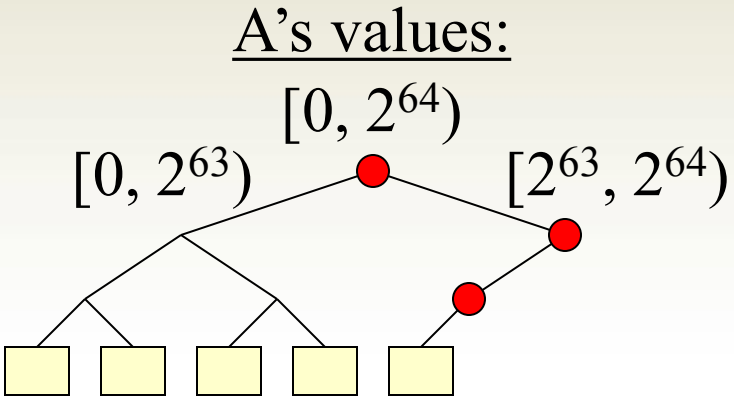
- Still too costly:
  - If A is down for an hour, then comes back, changes will be randomly scattered throughout tree

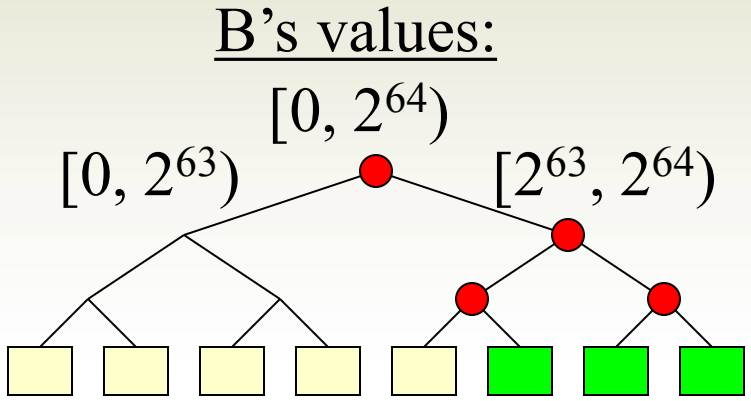




Using Merkle Trees as Summaries

- Still too costly:
  - If A is down for an hour, then comes back, changes will be randomly scattered throughout tree
- Solution: order values by time instead of hash
  - Localizes values to one side of tree

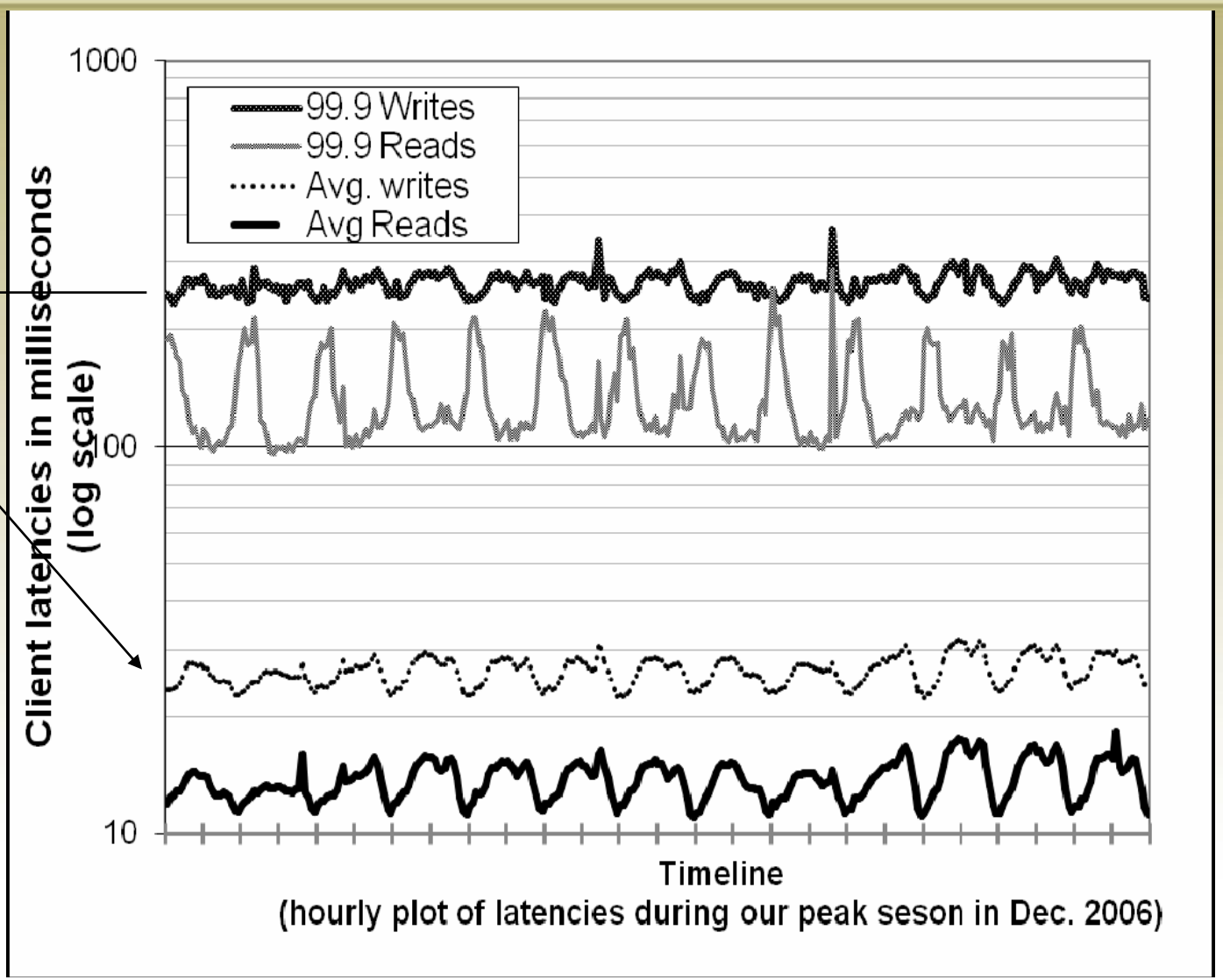




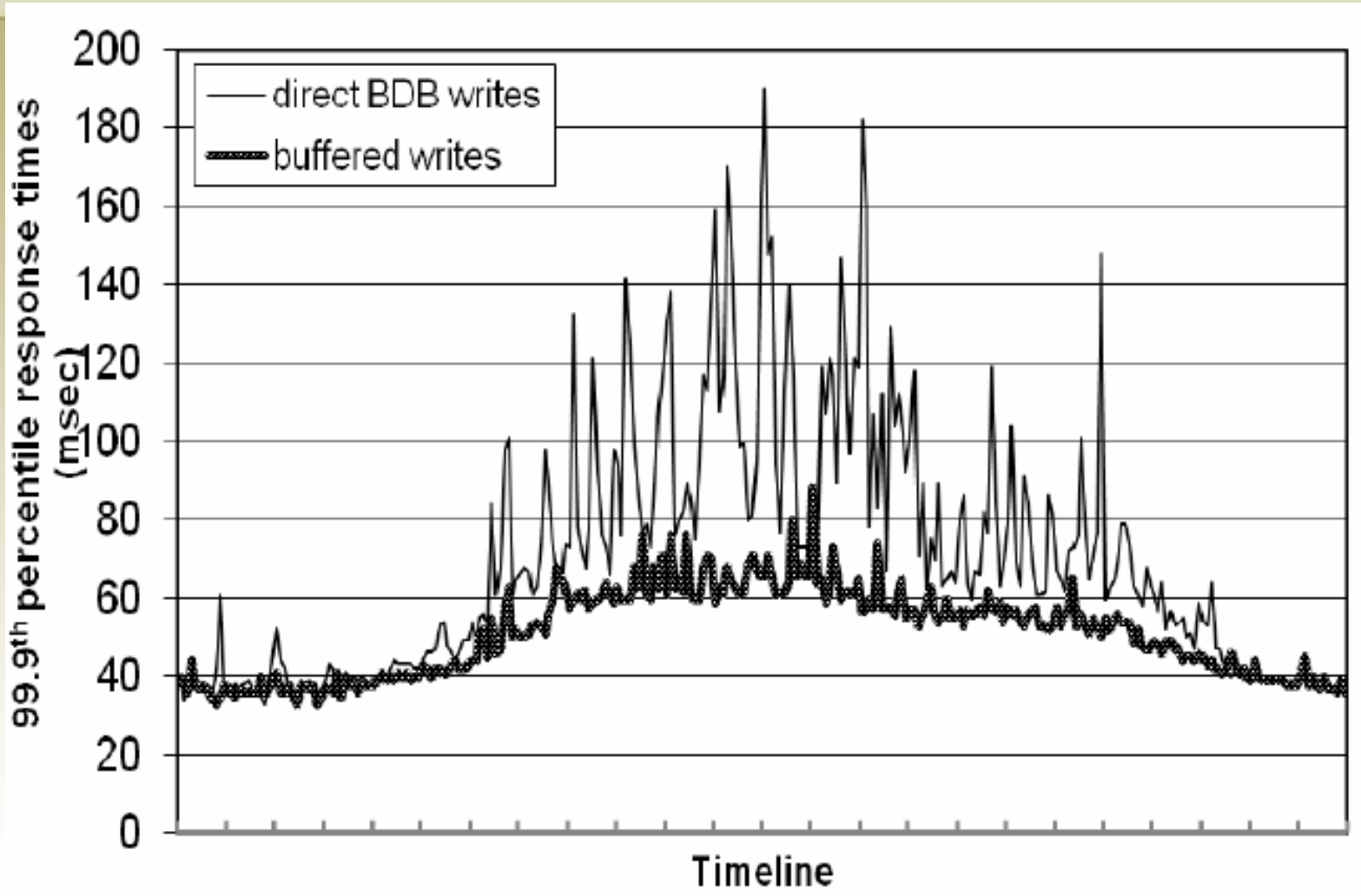
Implementation

- Java
  - non-blocking IO
- Local persistence component allows for different storage engines to be plugged in:
  - Berkeley Database (BDB) Transactional Data Store: object of tens of kilobytes
  - MySQL: larger objects

Performance evaluation



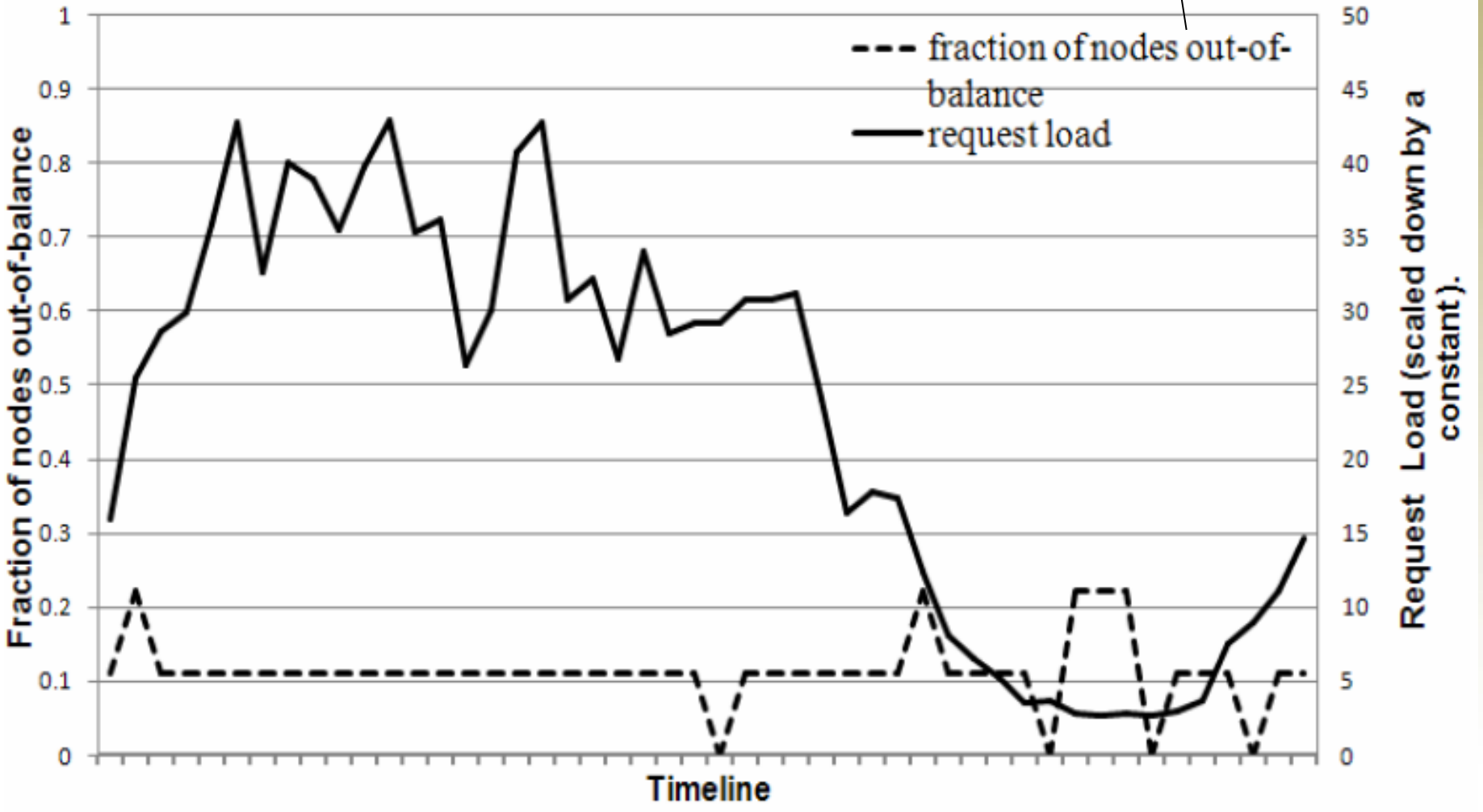
Trading between latency & durability



Comparison of performance of 99.9th %-tile latencies for buffered vs. non-buffered writes over 24 hours. The intervals between consecutive ticks in the x-axis correspond to one hour.

Load balance

out-of-balance: nodes with request load above 15% from the average system load



Divergent versions rarely created in practice

1 version → 99.94%

2 versions → 0.0057%

3 versions → 0.00047%

4 versions → 0.00007

Source: High volume of concurrent writes … robots?

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	• Eventual consistency • 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.

Consistent Hashing

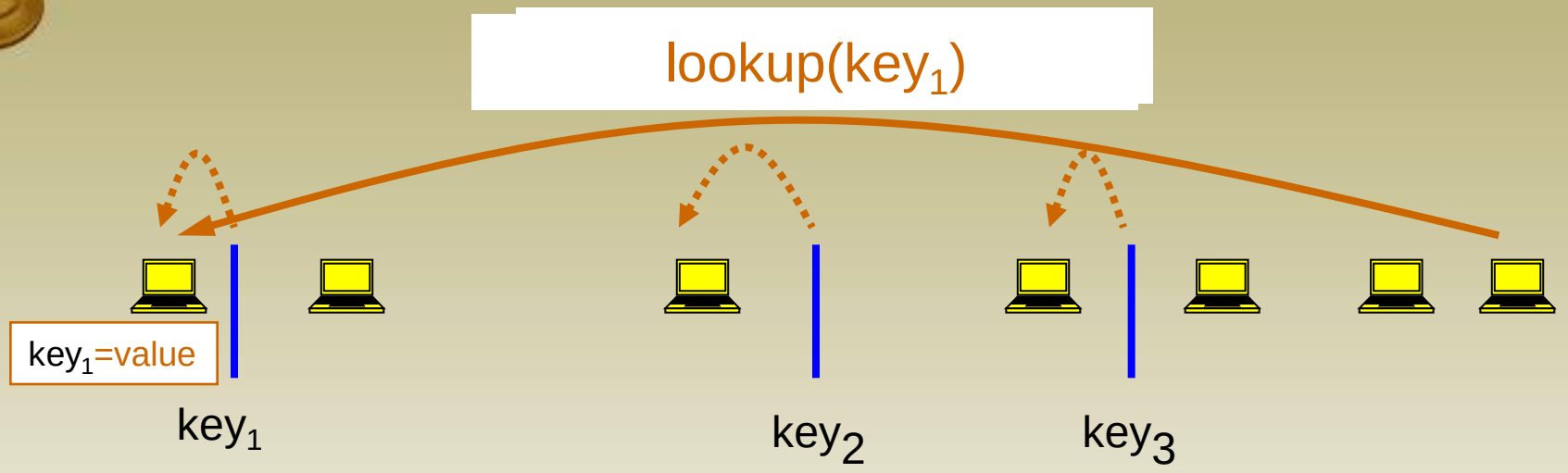
Basics of hashing

- The goal is to let any client perform a local computation that maps an object (a URL) to the particular node that contains it
- Hashing is a commonly used tool for this purpose
- Given a set of 23 caches numbered 0…22
- We hash URL u to cache **h(u)= 7u + 4 mod 23**
- A common intuition about hash functions is that they tend to distribute their inputs "randomly" among the possible locations
- Standard hashing has several drawbacks when applied to a storage system
  - Perhaps the most obvious is that machines will come up and go down over time
- If we add one, then **h(u)= 7u + 4 mod 24**, but under such a change, essentially every URL is remapped to a new cache

Basics of consistent hashing

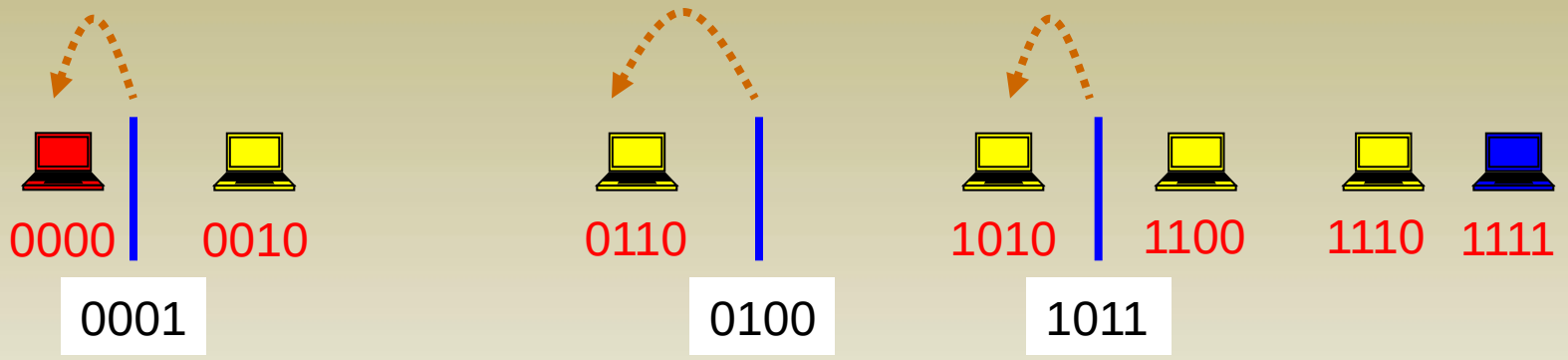
- Choose some standard base hash function that maps strings to the number range [0,…,M]
- Dividing by M, think of it as a hash function that maps to the range [0,1], which can in turn be thought of as the unit circle
- Each URL is thus mapped to a point on the unit circle
- At the same time, map every node in the system to a point on the unit circle
- Now assign each URL to the first node whose point it encounters moving clockwise from the URL's point

Consistent Hashing



- Consistent hashing partitions key-space among nodes
- Contact appropriate node to lookup/store key
  - Blue node determines red node is responsible for key_1
  - Blue node sends lookup or insert to red node

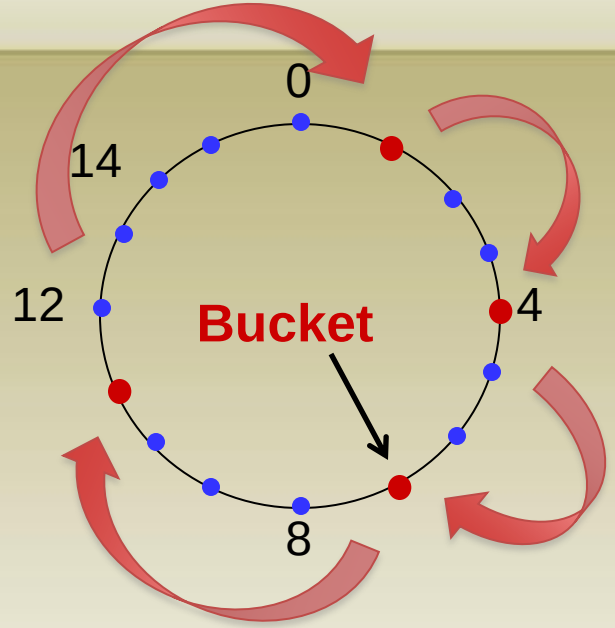
Consistent Hashing



- Partitioning key-space among nodes
  - Nodes choose random identifiers: e.g., hash(IP)
  - Keys randomly distributed in ID-space: e.g., hash(URL)
  - Keys assigned to node "nearest" in ID-space
  - Spreads ownership of keys evenly across nodes

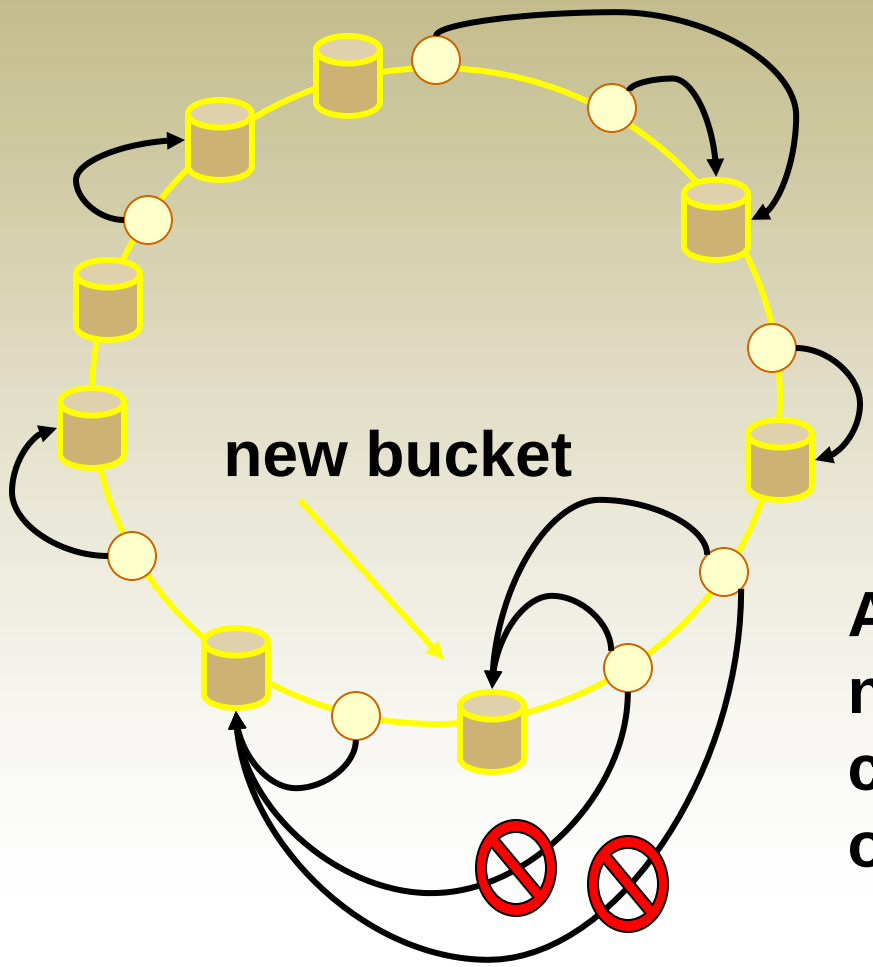
Consistent Hashing

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



- Desired features
  - Balanced: No bucket has disproportionate number of objects
  - Smoothness: Addition/removal of bucket does not cause movement among existing buckets (only immediate buckets)
  - Spread and load: Small set of buckets that lie near object

Inserting new nodes



object

bucket

Assign object to next bucket on circle in clockwise order.

Merkle Trees

Definition

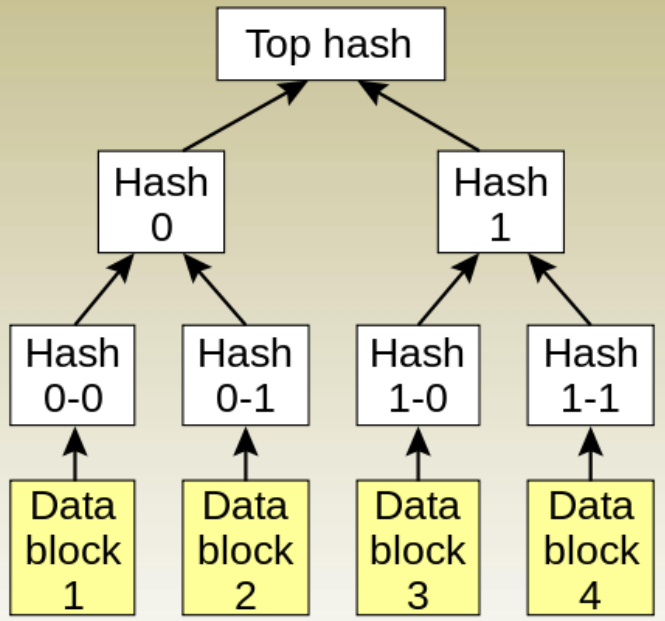
- A hash tree is a tree of hashes in which the leaves are hashes of data blocks in, for instance, a file or set of files
- Nodes further up in the tree are the hashes of their respective children
- For example, in the picture hash 0 is the result of hashing hash 0-0 and then hash 0-1. That is, hash 0 = hash(hash 0-0 || hash 0-1) where || denotes concatenation
- Usually, a cryptographic hash function such as SHA-1, Whirlpool, or Tiger is used for the hashing. If the hash tree only needs to protect against unintentional damage, much less secure checksums such as CRCs can be used

Operation

- In the top of a hash tree there is a top hash (or root hash or master hash)
- Before downloading a file on a p2p network, in most cases the top hash is acquired from a trusted source, for instance a friend or a web site that is known to have good recommendations of files to download
- When the top hash is available, the hash tree can be received from any non-trusted source, like any peer in the p2p network
- Then, the received hash tree is checked against the trusted top hash, and if the hash tree is damaged or fake, another hash tree from another source will be tried until the program finds one that matches the top hash

Example

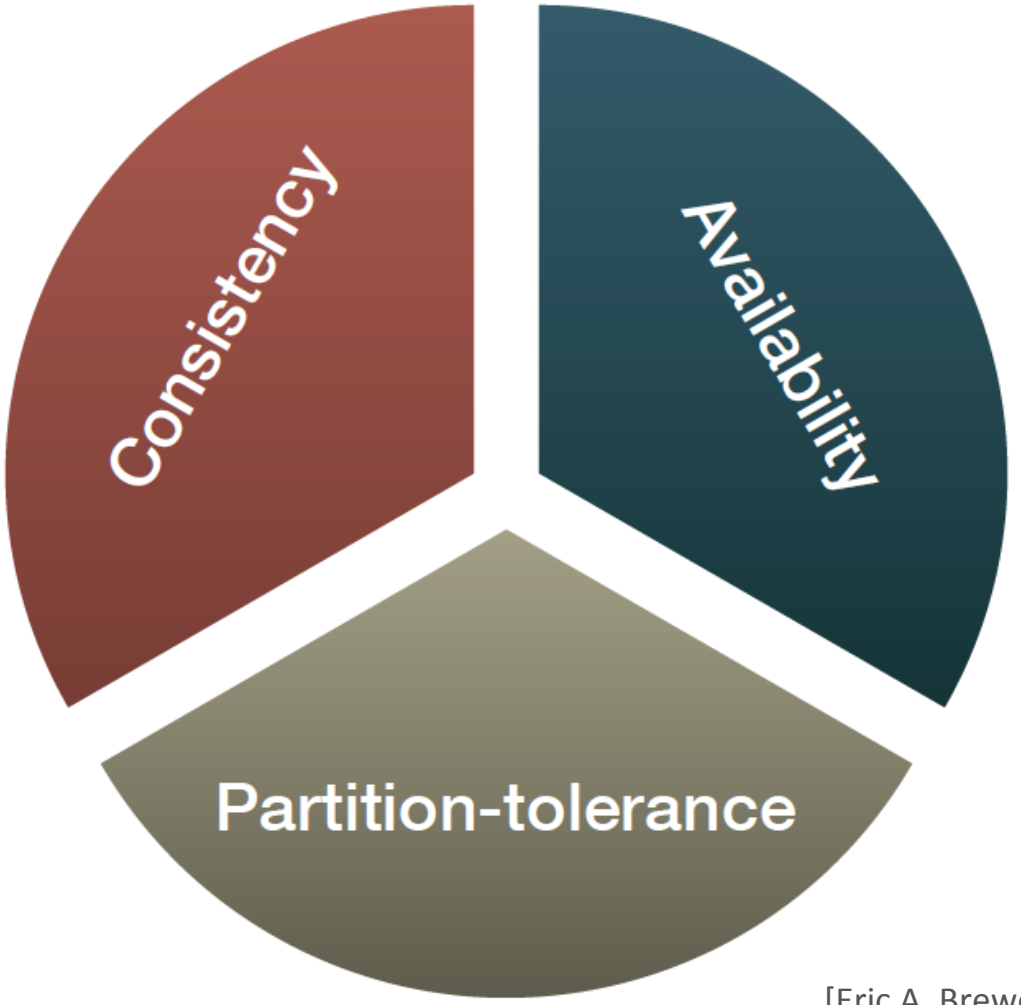
- The integrity of data block 2 can be verified immediately if the tree already contains hash 0-0 and hash 1 by hashing the data block and iteratively combining the result with hash 0-0 and then hash 1 and finally comparing the result with the top hash
- Similarly, the integrity of data block 3 can be verified if the tree already has hash 1-1 and hash 0
- This can be an advantage since it is efficient to split files up in very small data blocks so that only small blocks have to be redownloaded if they get damaged. If the hashed file is very big, such a hash tree or hash list becomes fairly big. But if it is a tree, one small branch can be downloaded quickly, the integrity of the branch can be checked, and then the downloading of data blocks can start



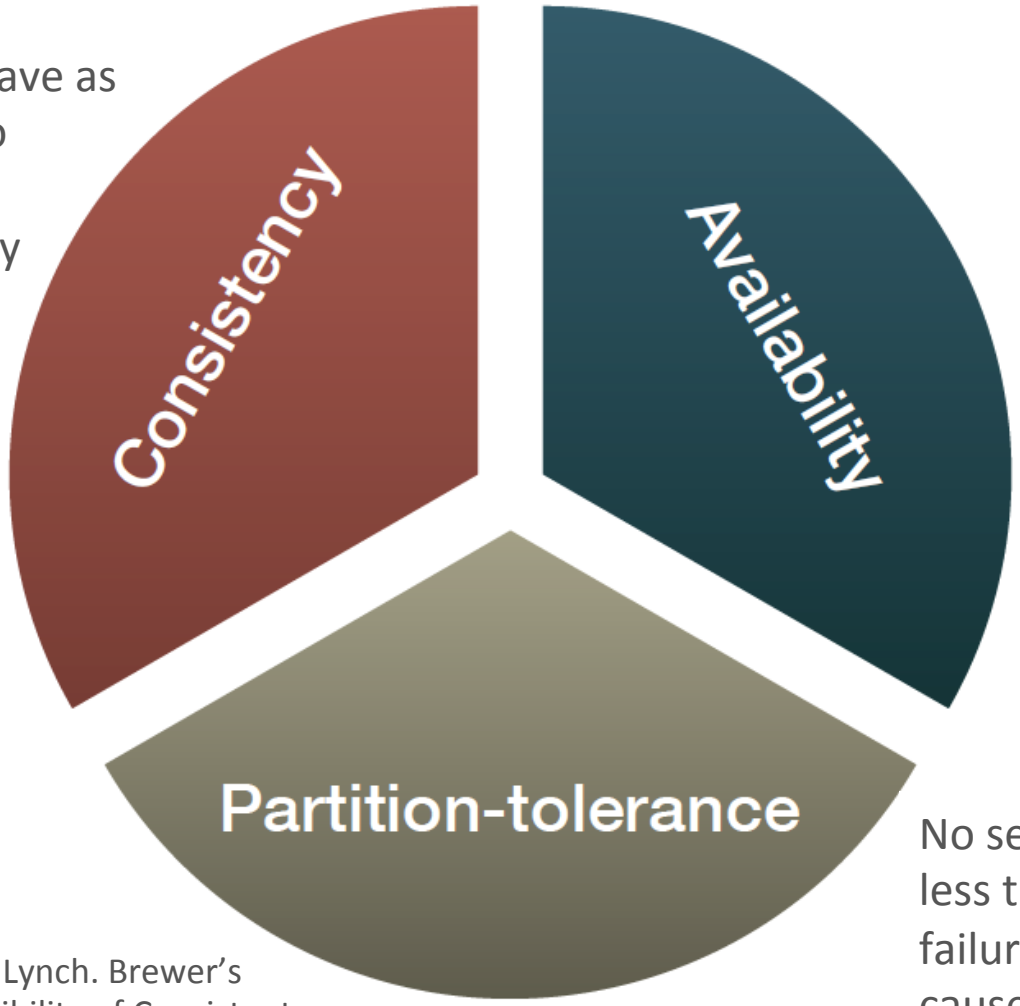
Webscale Data Management

5 CAP Theorem

© Prof. Dr. -Ing. Wolfgang Lehner



[Eric A. Brewer. Towards Robust Distributed Systems. PODC Keynote, 2000.]



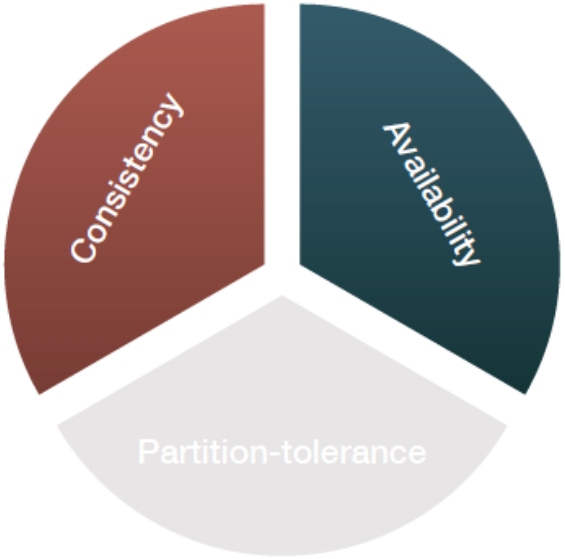
Operations behave as if there were no concurrency
→ Linearizability

Every request received by a non-failing node must result in a response

No set of failures less than total network failure is allowed to cause the system to respond incorrectly

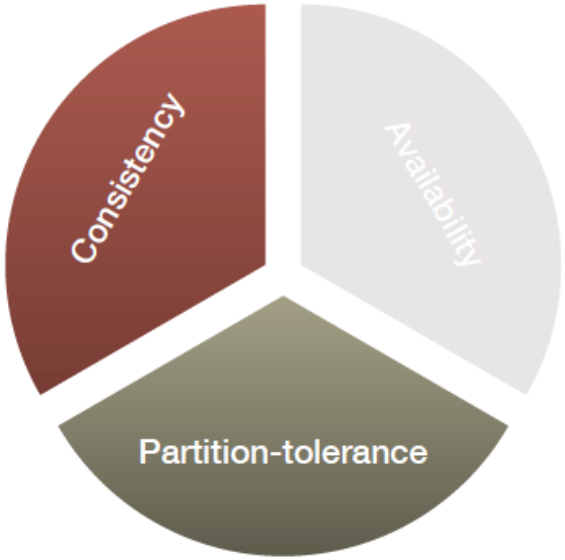
[Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. ACM SIGACT News, vol. 33, no. 2, 2002, p. 51-59.]

In shared-data systems only two of the three CAP properties can be achieved at one moment in time



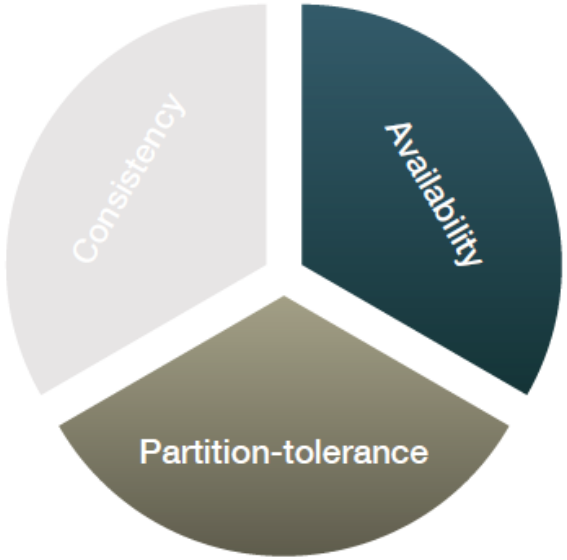
CA

Available, and consistent, unless there is a partition.



CP

Always consistent, even in a partition, but a reachable replica may deny service without agreement of the others (e.g., quorum).



AP

A reachable replica provides service even in a partition, but may be inconsistent.

Providing transactional semantics requires either

- That all nodes are in contact with each other
- Or to put everything in one atomically-failing unit

Features

- Two phase commit
- Cache validation protocols

Examples

- Single-site databases
- Cluster databases

It's better to fail the complete system than to break distribution transparency

- On a partition event, simply wait until data is consistent again

Features

- Pessimistic locking
- Minority partitions unavailable

Examples

- Distributed databases
- Distributed locking
- Quorum protocols

Without consistency guarantees, life is easy

Features

- Cache expiration / leases
- Updating with conflict resolution
- Optimistic strategies

Examples

- DNS
- Web caches
- Coda

Formalization of the notion of Consistency, Availability and Partition Tolerance

- Atomic Data Object
- Available Data Object
- Partition Tolerance

Atomic/Linearizable Consistency

- There must exist a total order on all operation such that each operation looks as if it were completed at a single instant
- This is equivalent to requiring requests on the distributed shared memory to act as if they are executing on single node, responding to operations one at the time

For a distributed system to be continuously available, every request received by a non-failing node in the system must result in a response

In some ways, this is a weak definition of availability

- It puts no bounds on how long the algorithm may run before terminating, and therefore allows unbounded computation

On the other hand, when qualified by the need for partition tolerance, this can be seen as a strong definition of availability

- Even when severe network failures occur, every request must terminate

*In order to model partition tolerance, the network is allowed to **lose arbitrary many messages** sent from one node to another*

When a network is partitioned, all messages sent from nodes in one component of the partition to another component are lost

*The **atomicity** requirement implies that every response will be atomic, even though arbitrary messages sent as part of the algorithm might not be delivered*

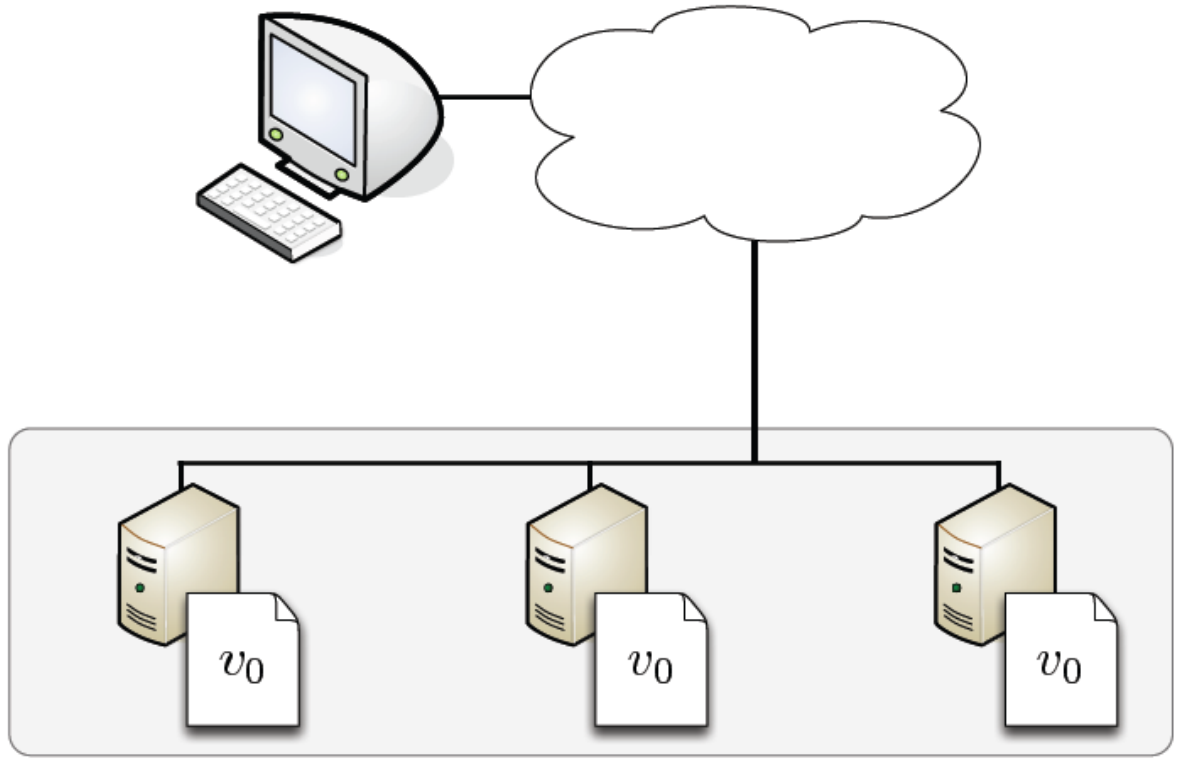
*The **availability** requirement therefore implies that every node receiving request from a client must respond, even through arbitrary messages that are sent may be lost*

Partition Tolerance

- No set of failures less than total network failure is allowed to cause the system to respond incorrectly

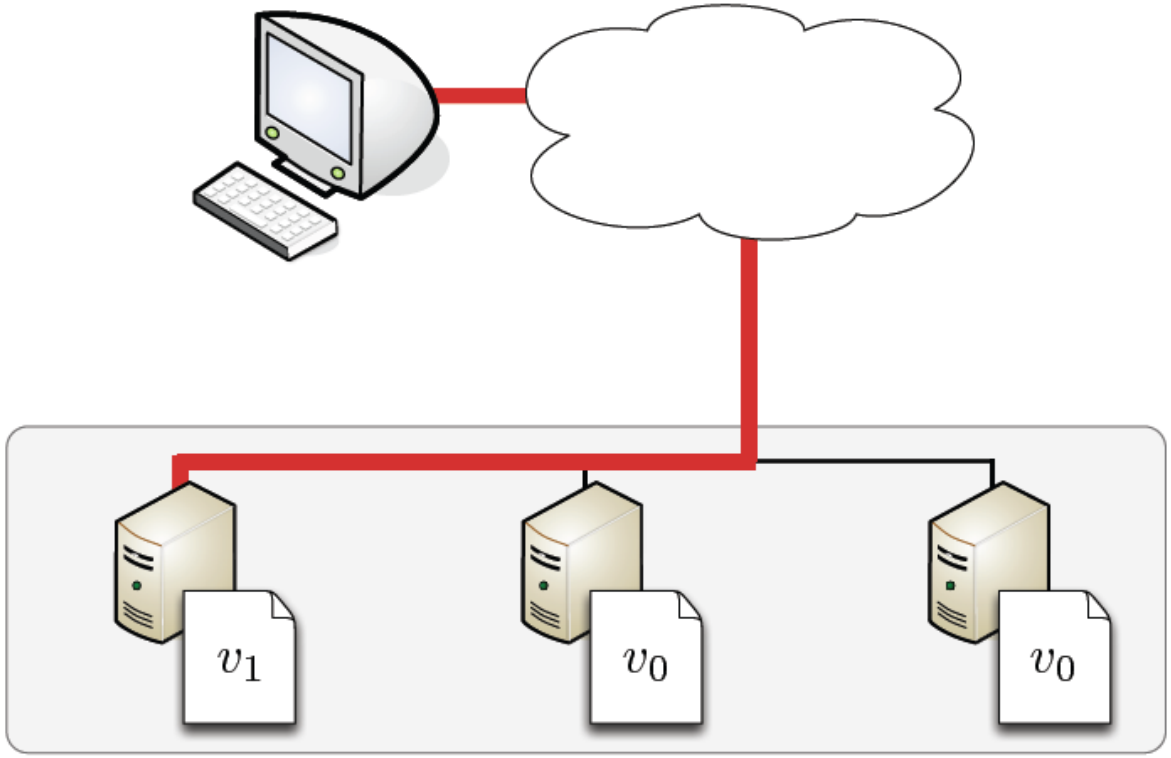
> Proof sketch of the CAP theorem

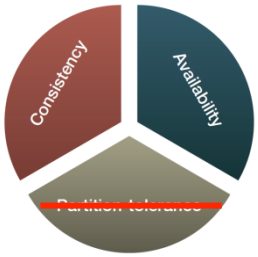
- Let v_0 be the initial value of an atomic object.
- A single *write* of a value not equal to v_0 occurs. Assume that no other client requests occur.
- We know that this write completes, by the availability requirement.
- A single *read* occurs, and no other client requests occur, ending with the termination of the *read* operation.
  The read operation returns v_1.



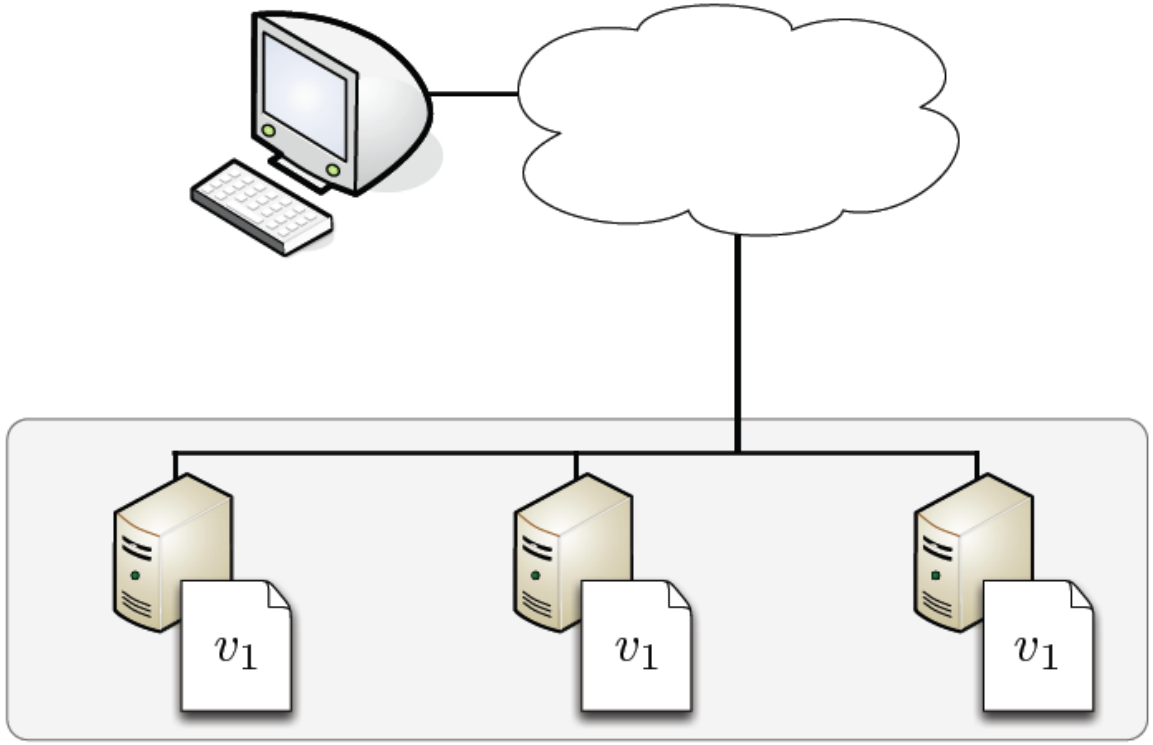
> Proof sketch of the CAP theorem (2)

- Let v_0 be the initial value of an atomic object.
- A single *write* of a value not equal to v_0 occurs. Assume that no other client requests occur.
- We know that this write completes, by the availability requirement.
- A single *read* occurs, and no other client requests occur, ending with the termination of the *read* operation.
  The read operation returns v_1.



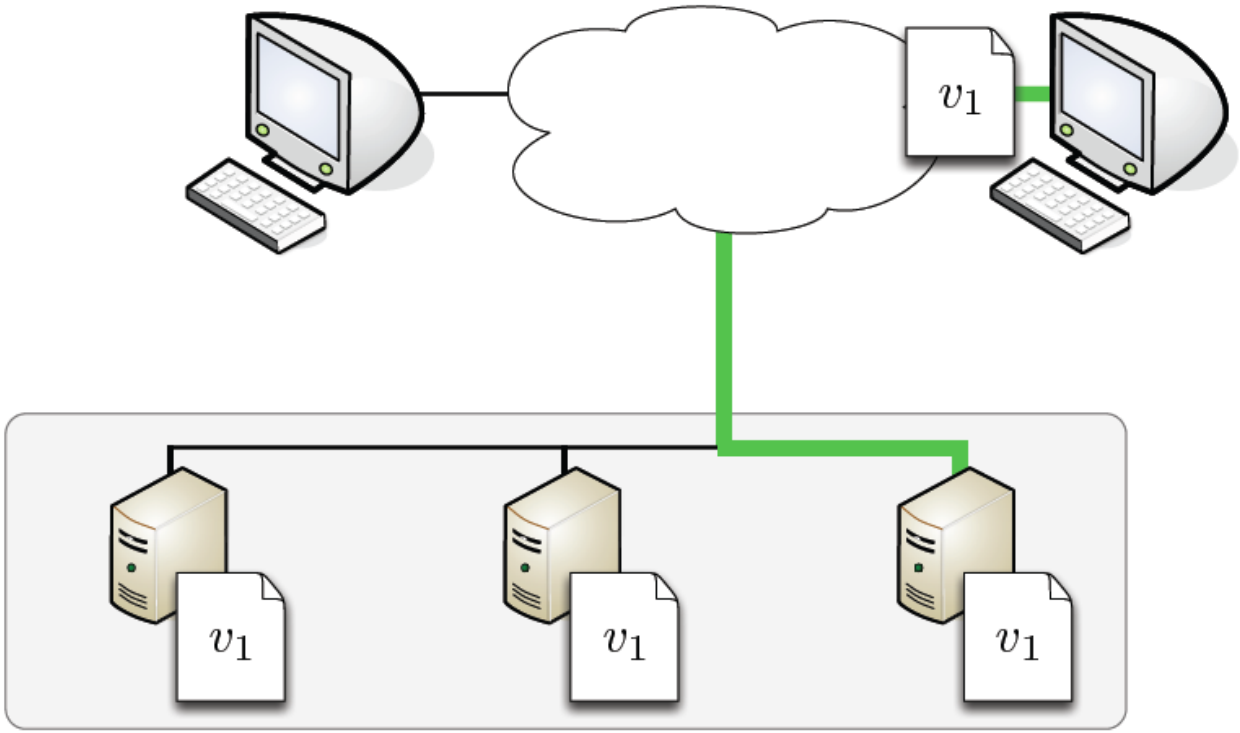


- Let v_0 be the initial value of an atomic object.
- A single *write* of a value not equal to v_0 occurs. Assume that no other client requests occur.
- We know that this write completes, by the availability requirement.
- A single *read* occurs, and no other client requests occur, ending with the termination of the *read* operation.
  The read operation returns v_1.

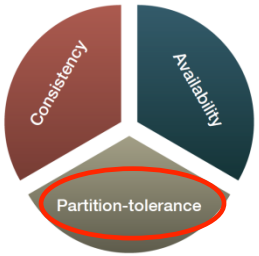


> Proof sketch of the CAP theorem (4)

- Let v_0 be the initial value of an atomic object.
- A single *write* of a value not equal to v_0 occurs. Assume that no other client requests occur.
- We know that this write completes, by the availability requirement.
- A single *read* occurs, and no other client requests occur, ending with the termination of the *read* operation.
  The read operation returns v_1.



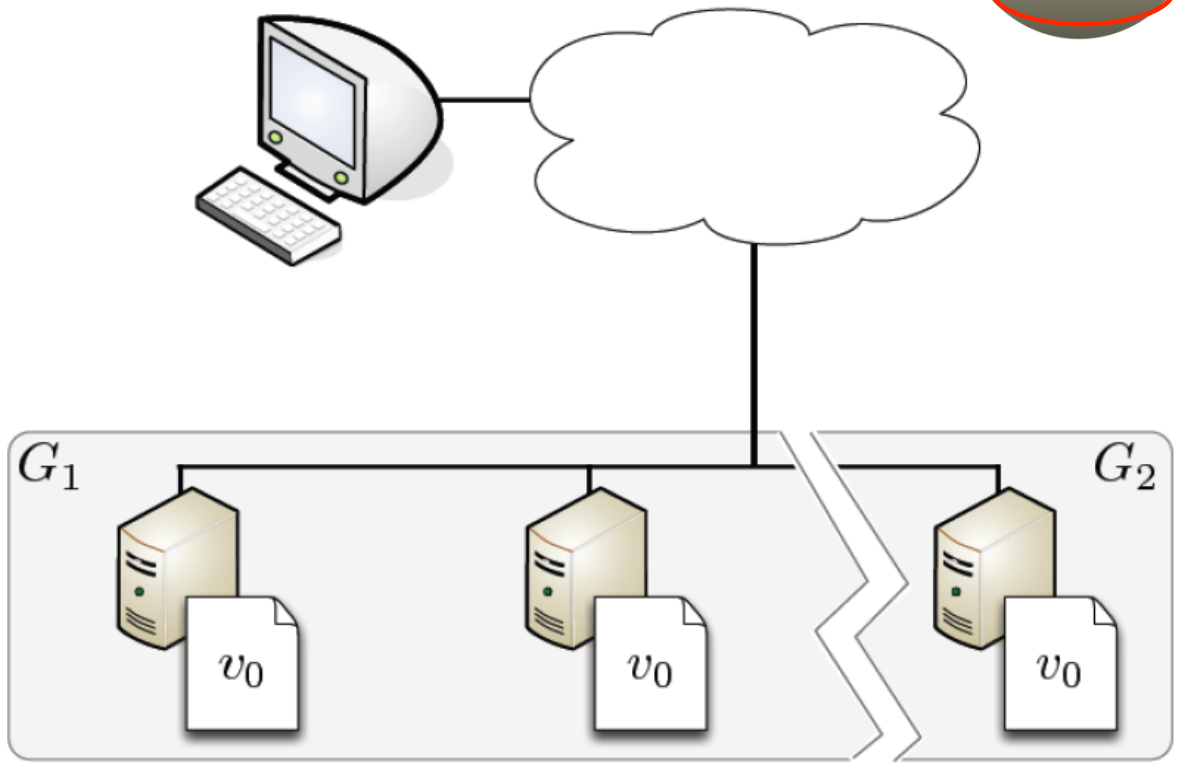
> Proof sketch of the CAP theorem (5)



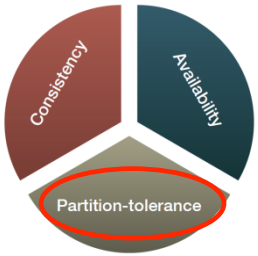
- Let v_0 be the initial value of an atomic object.

  Assume the network is divided into two disjoint sets $\{G_1, G_2\}$ and that all messages between G_1 and G_2 are lost.

- A single write of a value not equal to v_0 occurs on G_1.



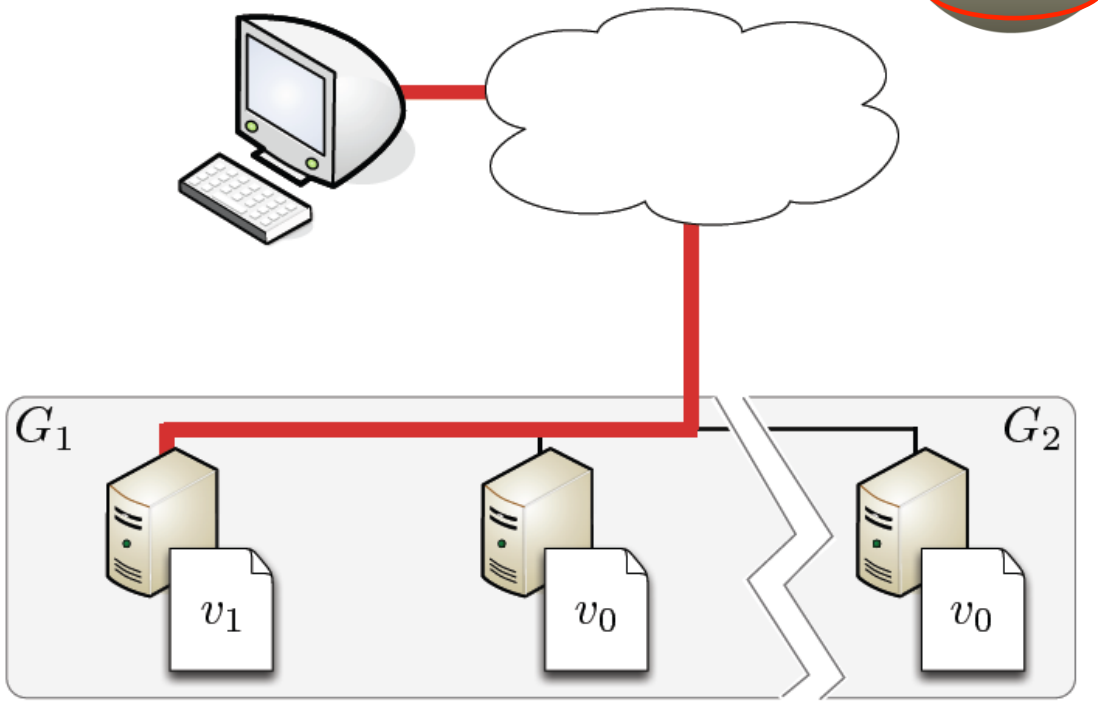
> Proof sketch of the CAP theorem (6)

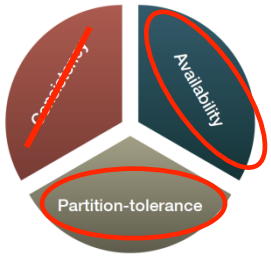


- Let v_0 be the initial value of an atomic object.

  Assume the network is divided into two disjoint sets $\{G_1,G_2\}$ and that all messages between G_1 and G_2 are lost.

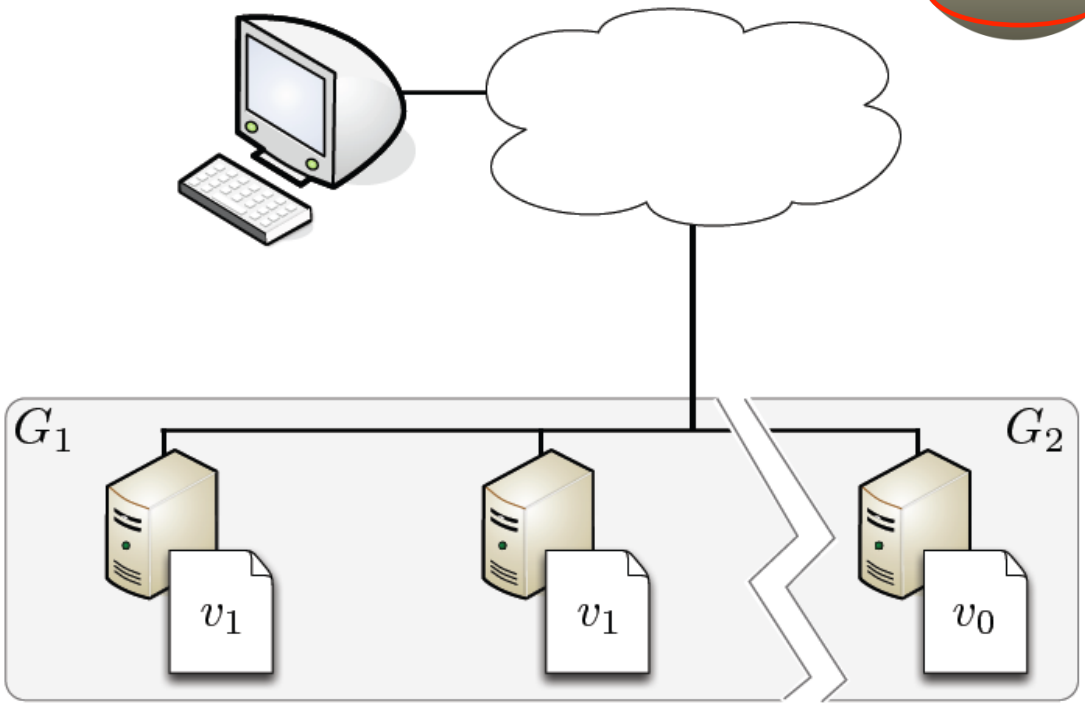
- A single write of a value not equal to v_0 occurs on G_1.



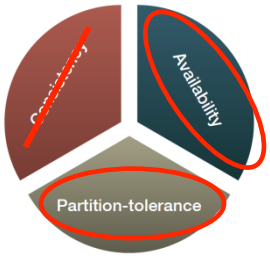


- Let v_0 be the initial value of an atomic object.
  Assume the network is divided into two disjoint sets $\{G_1,G_2\}$ and that all messages between G_1 and G_2 are lost.
- A single write of a value not equal to v_0 occurs in G_1.

A)
- By the availability requirement, this write completes.
- A single read occurs, and no other client requests occur, ending with the termination of the read operation.
  The read operation returns v_0.

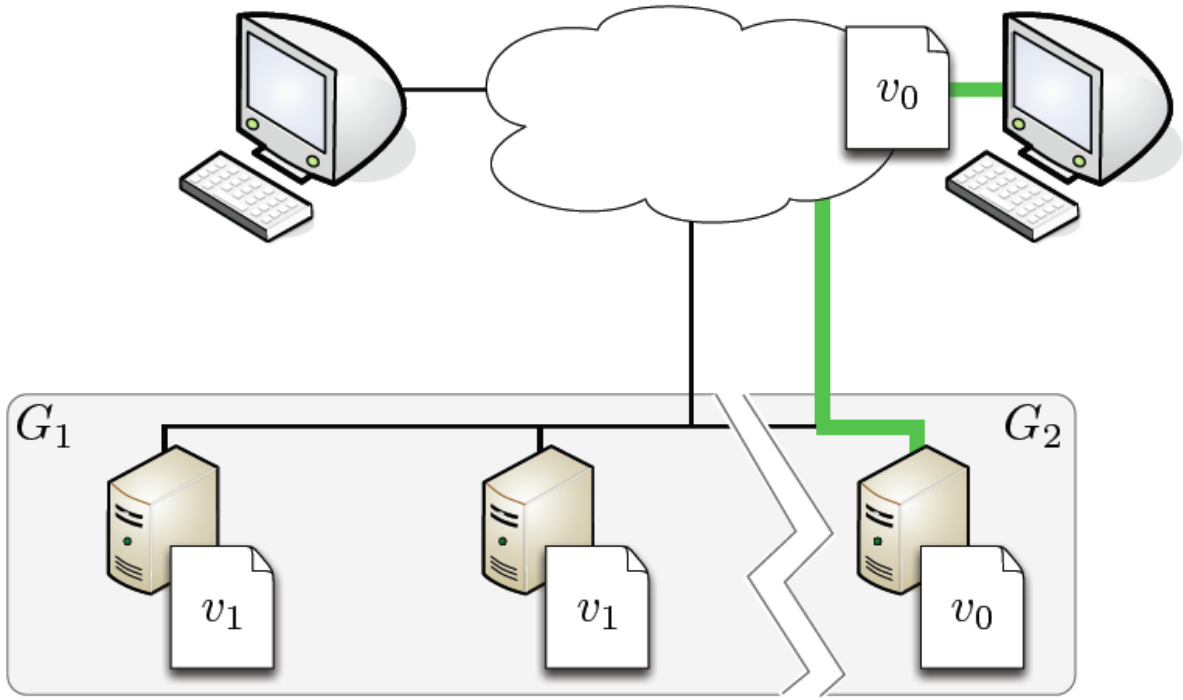


> Proof sketch of the CAP theorem (8)



- Let v_0 be the initial value of an atomic object.
  Assume the network is divided into two disjoint sets $\{G_1, G_2\}$ and that all messages between G_1 and G_2 are lost.
- A single write of a value not equal to v_0 occurs in G_1.

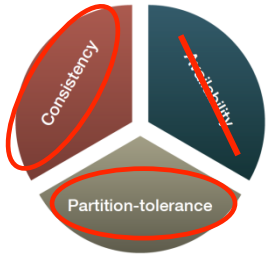
A)
- By the availability requirement, this write completes.
- A single read occurs, and no other client requests occur, ending with the termination of the read operation.
  The read operation returns v_0.

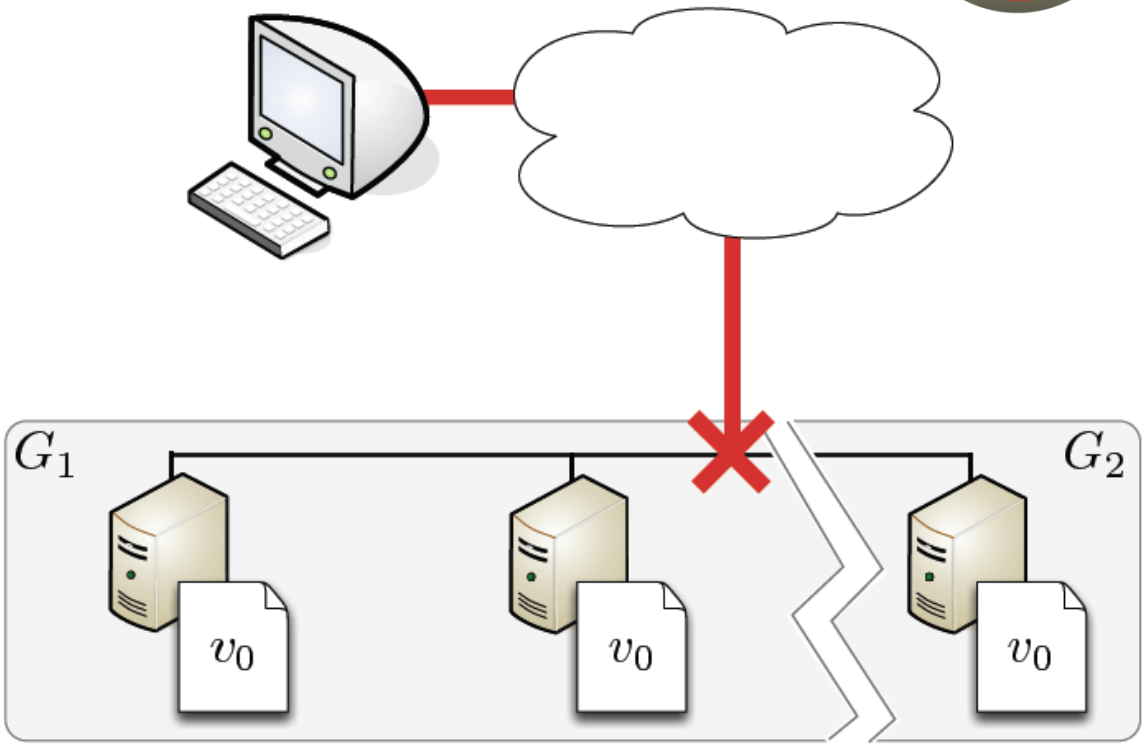


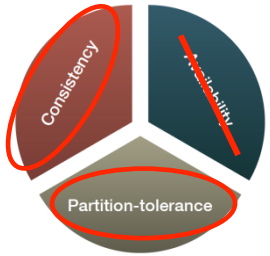
> Proof sketch of the CAP theorem (9)

- Let v_0 be the initial value of an atomic object.
  Assume the network is divided into two disjoint sets $\{G_1, G_2\}$ and that all messages between G_1 and G_2 are lost.
- A single write of a value not equal to v_0 occurs in G_1.

B)
- The write operation does not terminate. The availability requirement is violated.
- A single read occurs, and no other client requests occur, ending with the termination of the read operation.
  The read operation returns v_0.

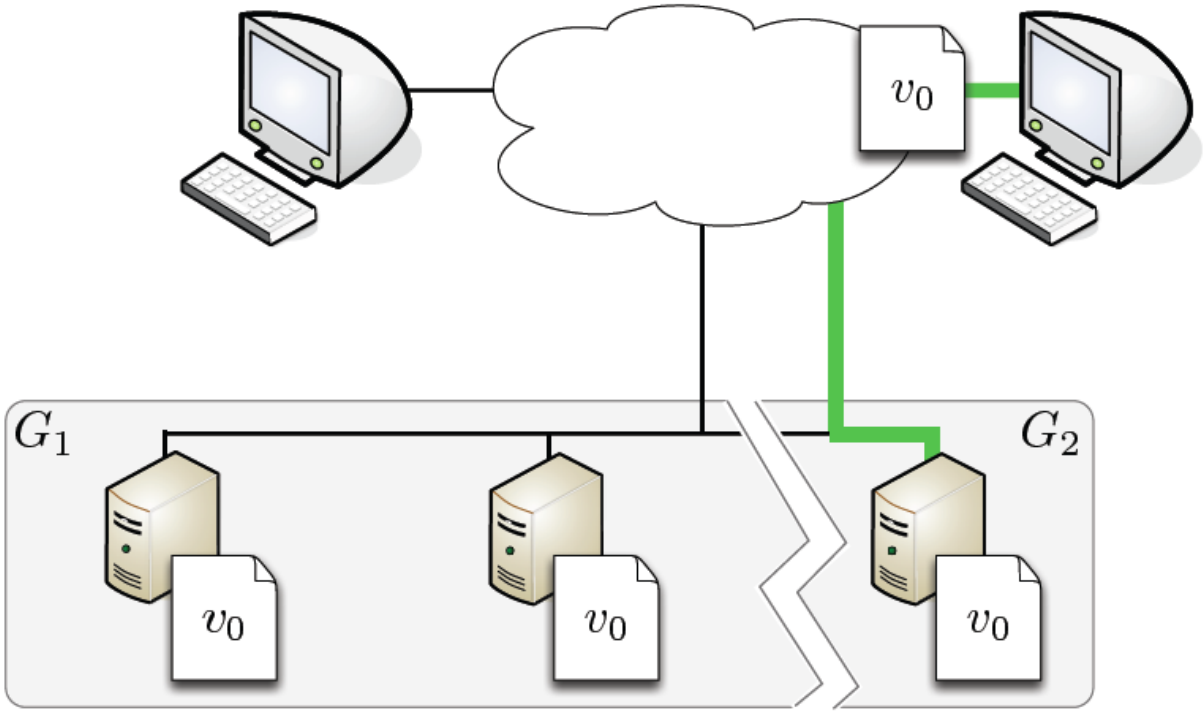






- Let v_0 be the initial value of an atomic object.
  Assume the network is divided into two disjoint sets $\{G_1, G_2\}$ and that all messages between G_1 and G_2 are lost.
- A single write of a value not equal to v_0 occurs in G_1.

B)
- The write operation does not terminate. The availability requirement is violated.
- A single read occurs, and no other client requests occur, ending with the termination of the read operation.
  The read operation returns v_0.

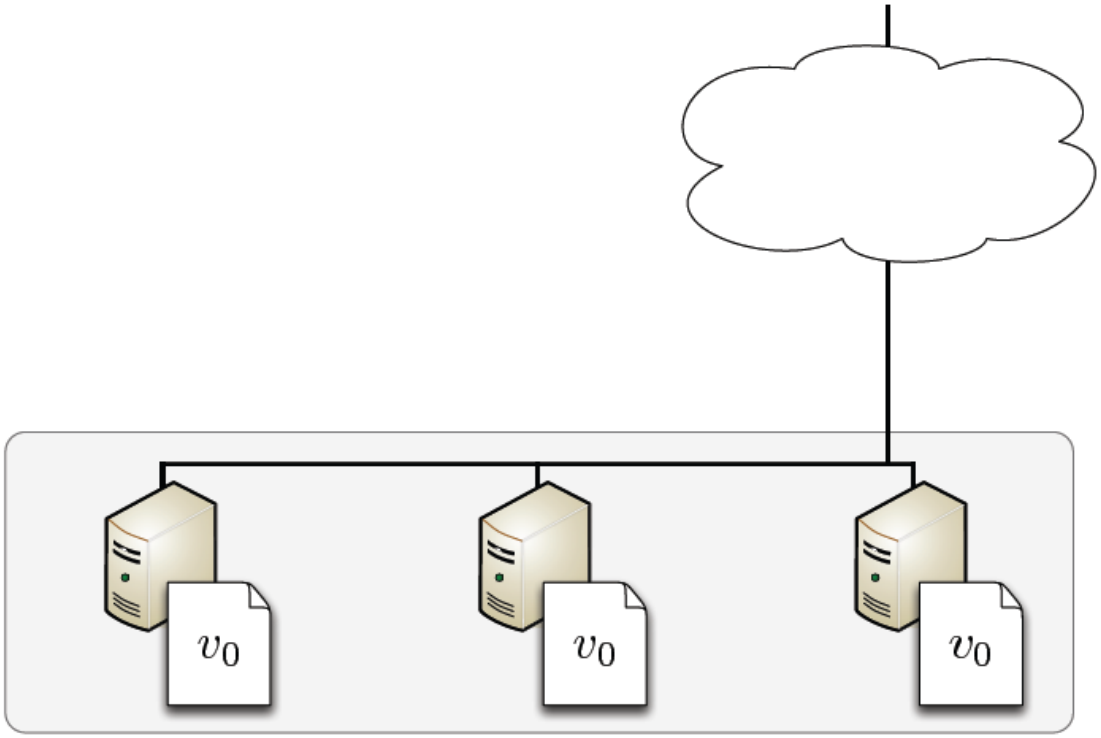


t-Connected Consistency

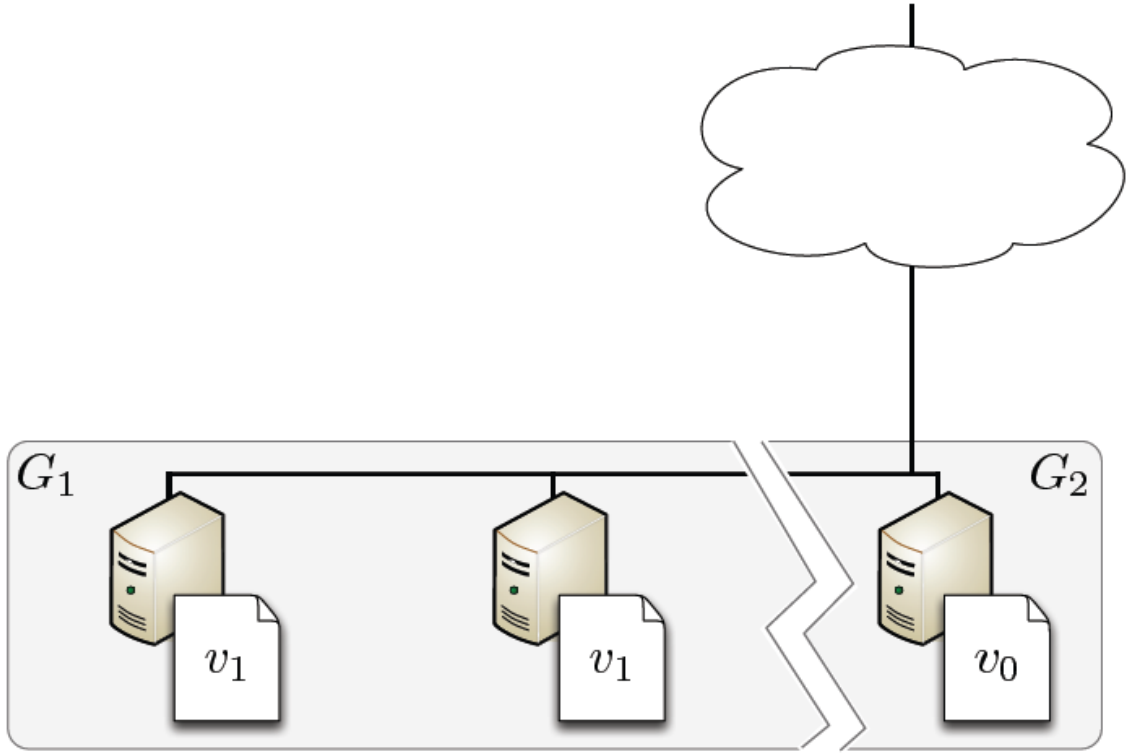
- In the presence of no partitions, the system is consistent
- In the presence of partitions, stale data can be returned
- Once a partition heals, there is a time limit on how long it takes for consistency to return

[Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. ACM SIGACT News, vol. 33, no. 2, 2002, p. 51-59.]

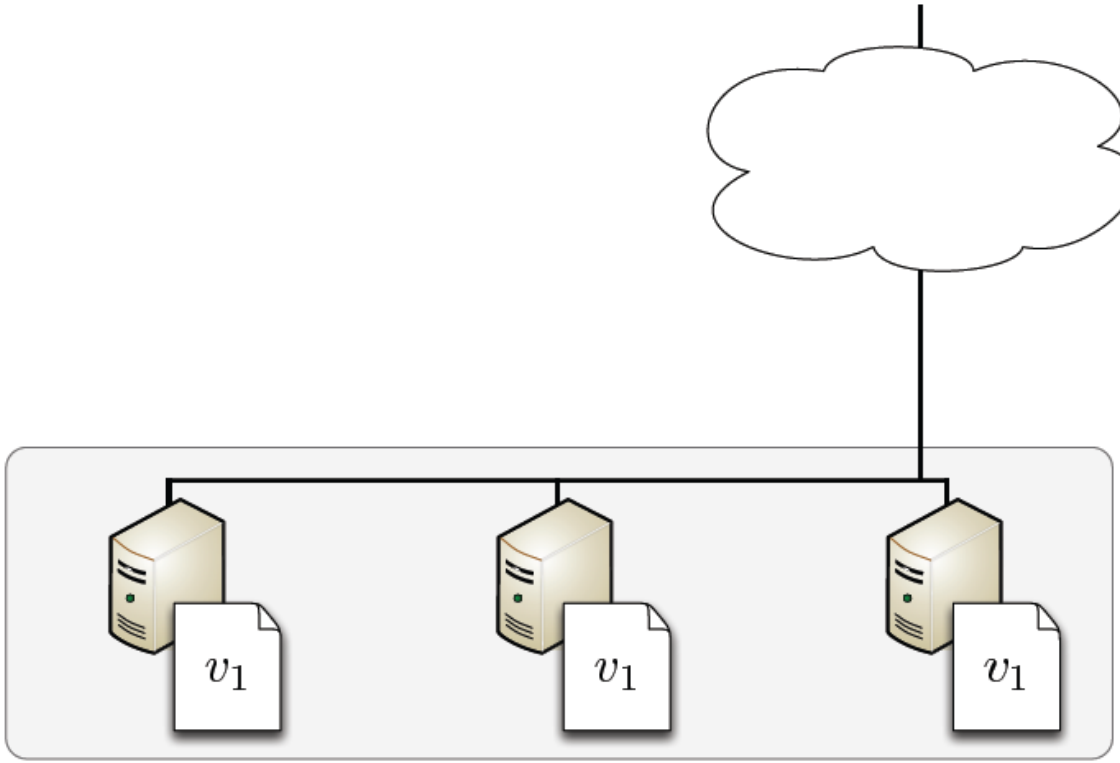
- In the presence of no partitions, the system is consistent.
- In the presence of partitions, stale data can be returned.
- Once a partition heals, there is a time limit on how long it takes for consistency to return.



- In the presence of no partitions, the system is consistent.
- In the presence of partitions, stale data can be returned.
- Once a partition heals, there is a time limit on how long it takes for consistency to return.



In the presence of no partitions, the system is consistent.
In the presence of partitions, stale data can be returned.
Once a partition heals, there is a time limit on how long it takes for consistency to return.



It is a simple solution

- Nobody understands what sacrificing „P“ means
- Sacrificing „A“ is unacceptable in the Web
- Possible to push the problem to app developer

„C“ not needed in many applications

- Airline reservation only transacts reads
- MySQL ship by default in lower isolation level

Data is noisy and inconsistent anyway

- Making it, say, 1% worse does not matter

Traditional distributed data management solutions focus on ACID semantics

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

Modern Web-scale data management focuses on BASE

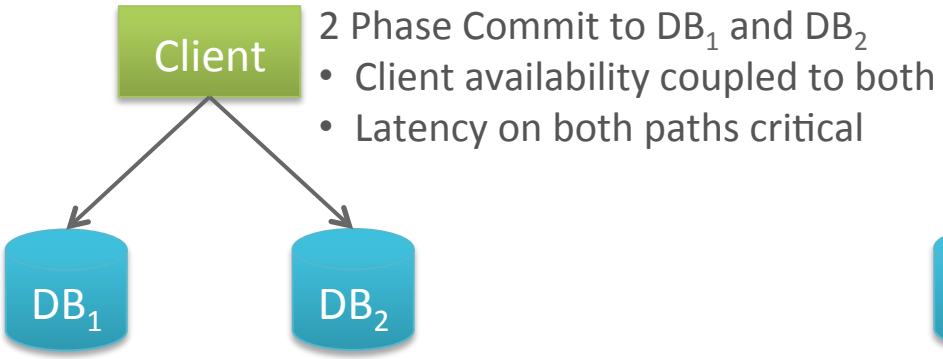
- **B**asically **A**vailable
- **S**oft-state
- **E**ventually consistent

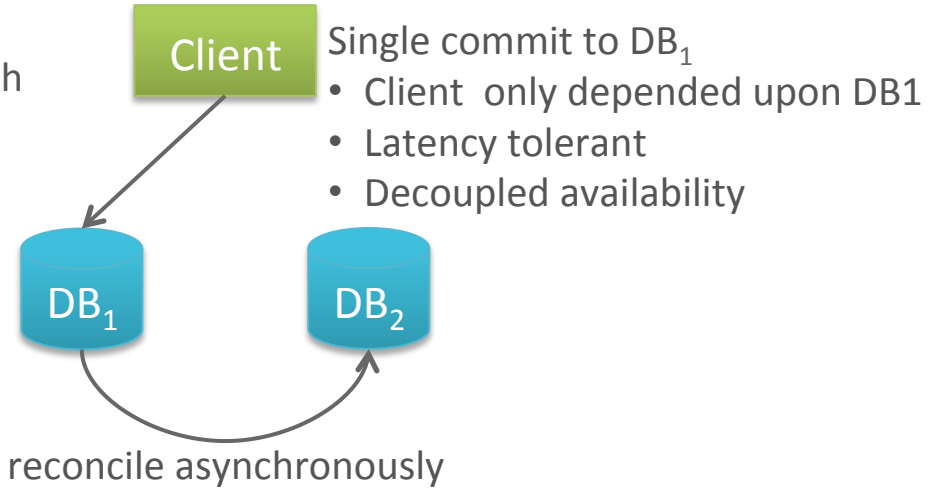
ACID

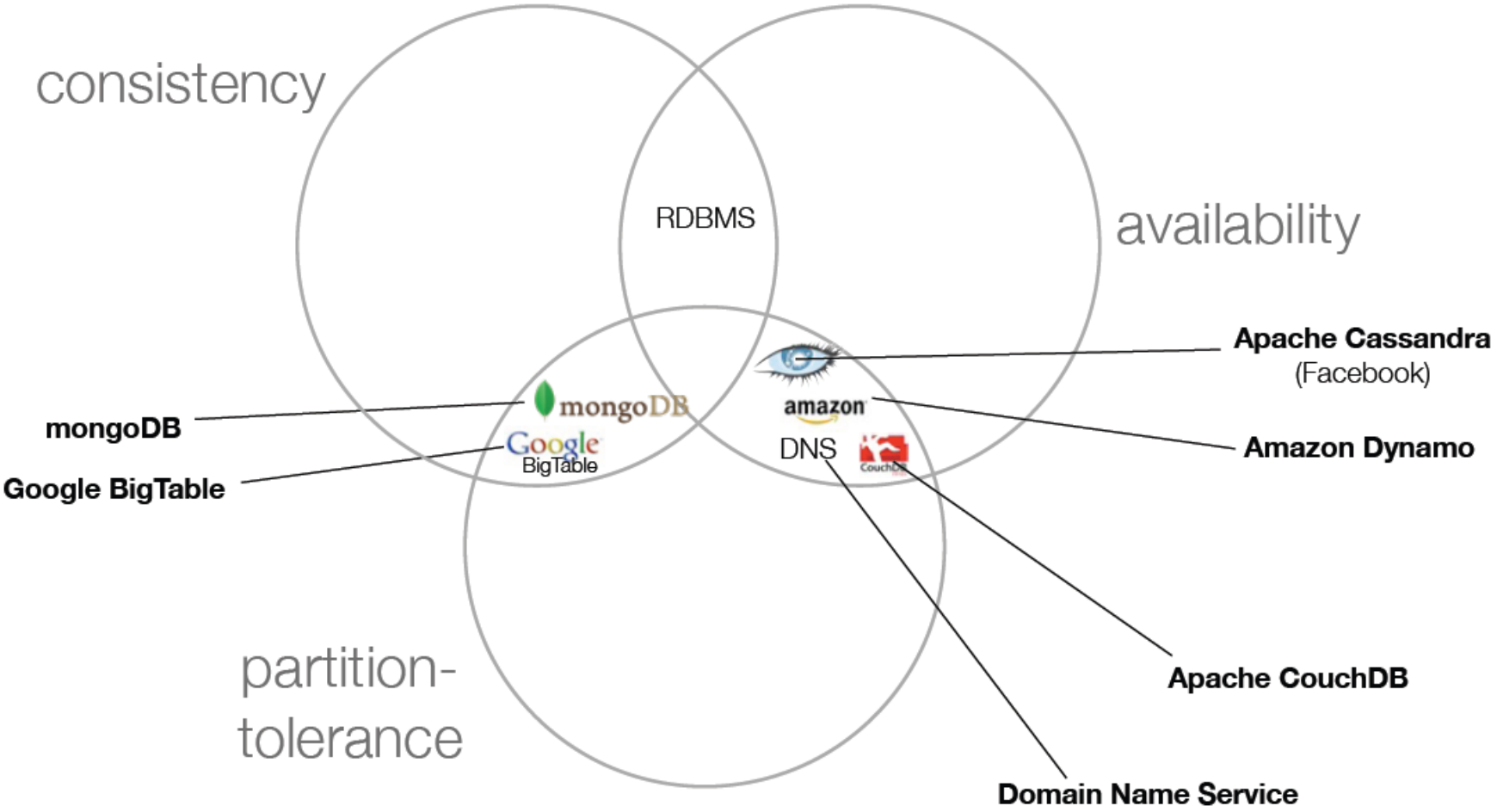
- Strong consistency
- Isolation
- Focus on “commit“
- Availability?
- Pessimistic

BASE

- Weak consistency
- Availability first
- Best effort
- Optimistic (aggressive)
- Fast and simple






consistency
RDBMS
availability
partition-
tolerance
mongoDB
Google BigTable
amazon
DNS
mongoDB
Google BigTable
Apache Cassandra
(Facebook)
Amazon Dynamo
Apache CouchDB
Domain Name Service

CAP implies that there are three types of distributed systems one can build

- CA: consistent and available, but not tolerant of partitions
- AP: available and tolerant of network partitions, but not consistent
- CP: consistent and tolerant of network partitions, but not available → strange, a clearly useless system ☺

- Availability is only sacrificed when there is a network partition
- The roles of the A and C in CAP are asymmetric

So in reality, there are only two types of systems: CP/CA and AP. I.e., if there is a partition, does the system give up availability or consistency?

Yahoo PNUTS relaxes consistency by only guaranteeing "timeline consistency"

- Replicas may not be consistent with each other but updates are guaranteed to be applied in the same order at all replicas
- If the master replica for a particular data item is unreachable, that item becomes unavailable for updates

Why would anyone want to give up both consistency and availability? CAP says you only have to give up just one

The answer is latency (L)

- PNUTS gives up consistency not for the goal of improving availability
- Instead, it is to lower latency
- Consequently, in order to reduce latency, replication must be performed asynchronously → reduces consistency

http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html

If there is a partition (P) how does the system tradeoff between

- Availability (A) and
- Consistency (C)

else when the system is running as normal in the absence of partitions, how does the system tradeoff between

- Latency (L) and
- Consistency (C)

Bounded Staleness for Partial Quorums

Analyzing eventual consistency

Quorum foundations

- Systems designers have long proposed quorum systems as a replication strategy for distributed data
- Under quorum replication, a data store writes a data item by sending it to a set of replicas, called a write quorum
- To serve reads, the data store fetches the data from a possibly different set of replicas, called a read quorum
- For reads, the data store compares the set of values returned by the replicas, and, given a total ordering of versions, can return the most recent value (or all values received, if desired)

Quorum foundations: Theory

- For each operation, the data store chooses read and write quorums from a set of sets of replicas, known as a quorum system, with one system per data item
- There are many kinds of quorum systems, but one simple configuration is to use read and write quorums of fixed sizes, which we will denote R and W, for a set of nodes of size N

- To summarize, a quorum replicated data store uses one quorum system per data item
- Across data items, quorum systems need not be identical

Strict quorum systems

- A strict quorum system is a quorum system with the property that any two quorums (sets) in the quorum system overlap (have non-empty intersection)
  - This ensures consistency
  - The minimum sized quorum defines the system's fault tolerance, or availability
- A simple example of a strict quorum system is the majority quorum system, in which each quorum is of size ceil(N/2)

Partial quorum systems

- Partial quorum systems are natural extensions of strict quorum systems: at least two quorums in a partial quorum system do not overlap
- There are two relevant variants of partial quorum systems in the literature
  - probabilistic quorum systems
  - k-quorums

Probabilistic quorum systems

- Probabilistic quorum systems provide probabilistic guarantees of quorum intersection
- By scaling the number of replicas, one can achieve an arbitrarily high probability of consistency
- Intuitively, this is a consequence of the Birthday Paradox: as the number of replicas increases, the probability of non-intersection between any two quorums decreases
- Probabilistic quorums are typically used to predict the probability of strong consistency but not (multi-version) bounded staleness

Probabilistic quorum systems

- As an example of a probabilistic quorum system, consider N replicas with randomly chosen read and write quorums of sizes R and W
- We can calculate the probability that the read quorum does not contain the last written version
- This probability is the number of quorums of size R composed of nodes that were not written to in the write quorum divided by the number of possible read quorums:

$$p_s = \frac{\binom{N-W}{R}}{\binom{N}{R}} \qquad (1)$$

Probabilistic quorum systems: Examples

- The probability of inconsistency is high except for large N
- With N = 100, R = W = 30, $p_s = 1{,}88 \times 10^{-6}$
- However, with N = 3, R = W = 1, $p_s = 0{,}6$.
- The asymptotics of these systems are excellent—but only asymptotically

k-quorum systems

- k-quorum systems provide deterministic guarantees that a partial quorum system will return values that are within k versions of the most recent write
- In a single writer scenario, sending each write to ceil(N/k) replicas with round-robin write scheduling ensures that any replica is no more than k versions out-of-date
- However, with multiple writers, we lose the global ordering properties that the single-writer was able to control, and the best-known algorithm for the pathological case results in a lower bound of (2N-1)(k-1) + N versions staleness

Quorum Foundations: Practice

- In practice, many distributed data management systems use quorums as a replication mechanism. Amazon's Dynamo is the progenitor of a class of eventually consistent key-value stores that include
  - Apache Cassandra
  - Basho Riak
  - LinkedIn's Project Voldemort
- All use the same variant of quorum-style replication

Quorum Foundations: Practice

- Dynamo-style quorum systems employ one quorum system per key, typically maintaining the mapping of keys to quorum systems using a consistent-hashing scheme or a centralized membership protocol
- Each node stores multiple keys.
- As shown in Figure, clients send read and write requests to a node in the system cluster, which forwards the request to all nodes assigned to that key as replicas

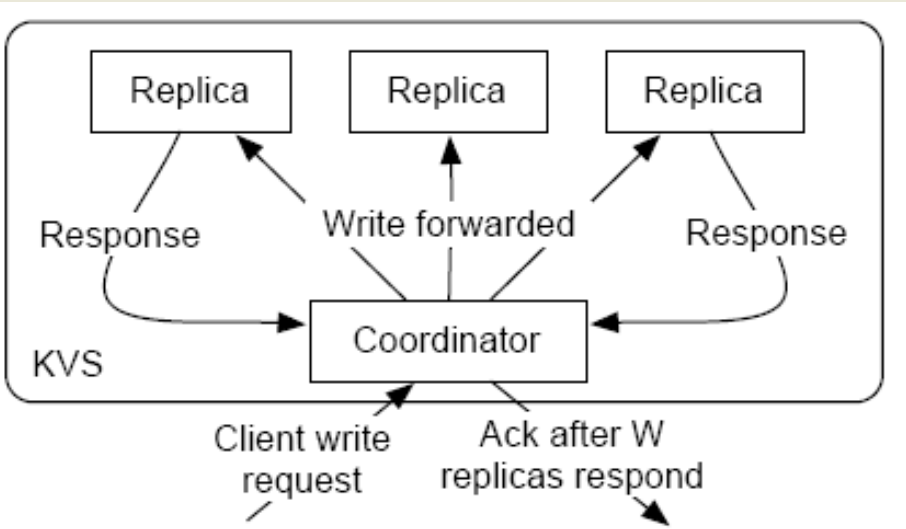


Diagram of control flow for client write to Dynamo-style quorum (N = 3,W = 2)

A coordinator node handles the client write and sends it to all N replicas. The write call returns after the coordinator receives W acknowledgments.

Quorum Foundations: Practice

- This coordinating node considers an operation complete when it has received responses from a pre-determined number of replicas (typically set per-operation)
- Without message loss, all replicas eventually receive all writes
- This means that the write and read quorums chosen for a request depend on which nodes respond to the request first
- Dynamo denotes the replication factor of a key as N, the number of replica responses required for a successful read as R, and the number of replica acknowledgments required for a successful write as W
- Under normal operation, Dynamo-style systems guarantee consistency when R+W>N. Setting W>ceil(N/2) ensures consistency in the presence of concurrent writes

Quorum configurations: Cassandra

- Cassandra defaults to N=3, R=W=1
- The Apache Cassandra 1.0 documentation claims that "a majority of users do writes at consistency level [W=1]", while the Cassandra Query Language defaults to R=W=1
- Production Cassandra users report using R=W=1 in the "general case" because it provides "maximum performance" which appears to be a commonly held belief

Quorum configurations: Riak

- Riak defaults to N=3, R=W=2
- Users suggest using R=W=1, N=2 for “low value” data (and strict quorum variants for “web,” “mission critical,” and “financial” data)

Quorum configurations: Voldemort

- Voldemort does not provide sample configurations, but Voldemort's authors (and operators) at LinkedIn often choose N=c, R=W=ceil(c/2) for odd c
- For applications requiring "very low latency and high availability," LinkedIn deploys Voldemort with N=3, R=W=1
- For other applications, LinkedIn deployments Voldemort with N=2, R=W=1, providing "some consistency," particularly when three-way replication is not required

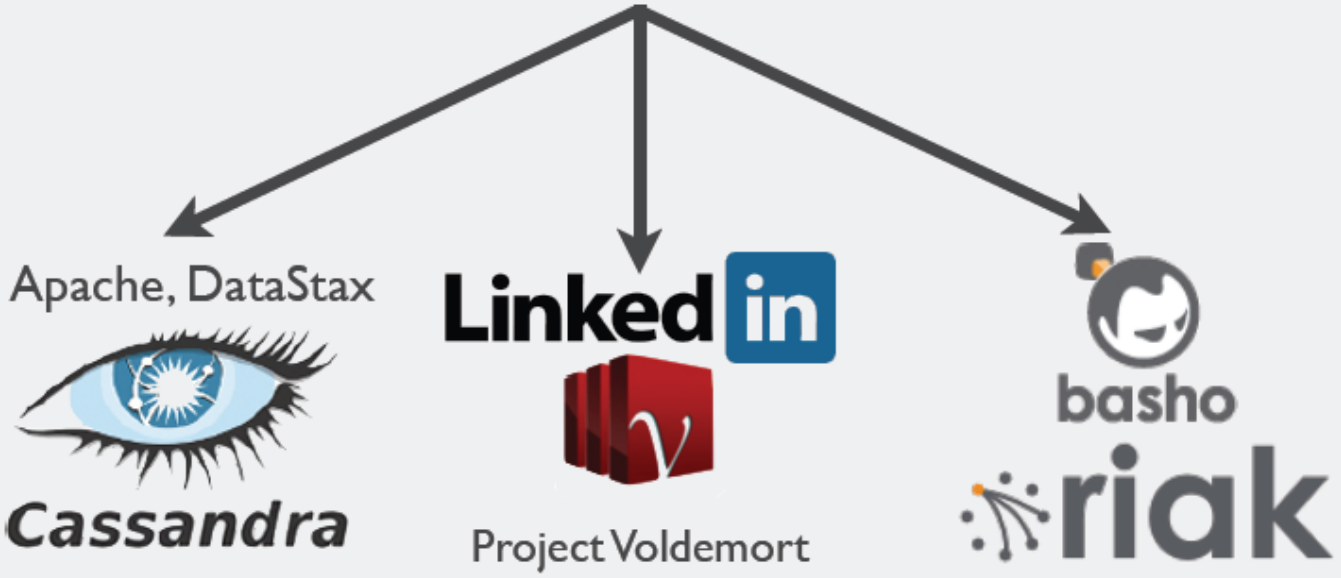
Quorum configurations: Voldemort

- Unlike Dynamo, Voldemort sends read requests to R of N replicas (not N of N)
- This decreases load per replica and network traffic at the expense of read latency and potential availability
- Provided staleness probabilities are independent across requests
- This does not affect staleness: even when sending reads to N replicas, coordinators only wait for R responses

Dynamo:

Amazon's Highly Available Key-value Store

SOSP 2007

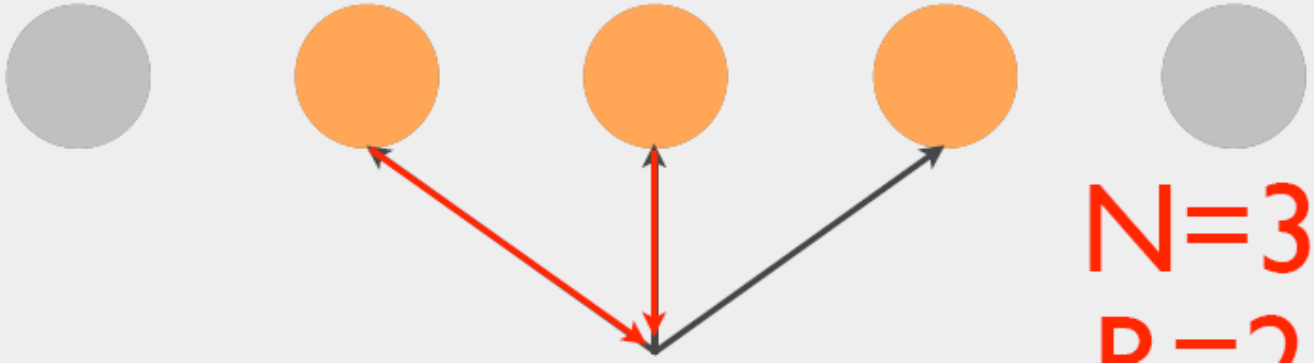


IBM
Adobe
Rackspace
Spotify
Palantir
Twitter
Cisco
Netflix
Cassandra
Reddit
Morningstar
Digg
Mozilla
Rhapsody
Shazam
Soundcloud
Gowalla
Yammer
Ask.com
Best Buy
Voldemort
Aol
Riak
Comcast
GitHub
LinkedIn
Gilt Groupe
Boeing
JoyentCloud

N replicas/key

read: wait for *R* replies

write: wait for *W* acks



N=3

R=2

if: $R+W > N$

then: "strong" consistency

else: eventual consistency

"strong" consistency ≡

$R+W > N$

reads return the last acknowledged write or an in-flight write *(per-key)*

regular register

Latency

LinkedIn disk-based model

N=3

R

	99th	99.9th
1	1x	1x
2	1.59x	2.35x
3	4.8x	6.13x

W

	99th	99.9th
1	1x	1x
2	2.01x	1.9x
3	4.96x	14.96x

⇧ consistency, ⇧ latency

wait for more replicas,
read more recent data

⇩ consistency, ⇩ latency

wait for fewer replicas,
read less recent data

eventual consistency

$R+W \leq N$

"if no new updates are made to the object, eventually all accesses will return the last updated value"

W. Vogels, *CACM* 2008

How eventual?

How long do I have to wait?

How consistent?

What happens if I don’t wait?

eventual consistency in the wild

"maximum performance"

okay for "most data"

"very low latency"

"general case"

anecdotally, EC
"good enough" for
many kinds of data

How eventual?
How consistent?

"eventual and consistent enough"

Can we do better?

can't make *promises*

can give *expectations*

Probabilistically Bounded Staleness

How eventual?

How long do I have to wait?

Probabilistically Bounded Staleness

- ***PBS k-staleness***, which probabilistically bounds the staleness of versions returned by read quorums
- ***PBS t-visibility***, which probabilistically bounds the time before a committed version appears to readers
- ***PBS <k,t>-staleness***, a combination of the two prior models

PBS k-staleness

- A quorum system obeys ***PBS k-staleness*** consistency if
  - with probability 1 - p_{sk}
  - at least one value in any read quorum has been committed within k versions of the latest committed version when the read begins
- The probability of returning a version of a key within the last k versions committed is equivalent to intersecting one of k independent write quorums
- Given the probability of a single quorum non-intersection p, the probability of non-intersection with one of the last k independent quorums is p^k
- In our example, the probability of non-intersection is Equation 1 exponentiated by k:

$$p_{sk} = \left(\frac{\binom{N-W}{R}}{\binom{N}{R}} \right)^k \qquad (2)$$

PBS k-staleness

- When N=3, R=W=1, this means that the probability of returning a version within 2 versions is 0,5, within 3 versions, 0,703, 5 versions, > 0,868, and 10 versions, > 0,98.
- When N=3, R=1, W=2 (or, equivalently, R=2, W=1), these probabilities increase: k=1 → 0,6, k=2 → 0,8, and k=5 → 0,995
- This closed form solution holds for quorums that do not change size over time. For *expanding partial quorum systems*, this solution is an upper bound on the probability of staleness

How eventual?

t-visibility: probability *p* of consistent reads after *t* seconds

(e.g., 10ms after write, 99.9% of reads consistent)

PBS t-visibility

- PBS t-visibility models the probability of inconsistency for expanding quorums. t-visibility is the probability that a read operation, starting t seconds after a write commits, will observe the latest value of a data item
- This t captures the expected length of the "window of inconsistency." Recall that we consider in-flight writes—which are more recent than the last committed version— as non-stale

- A quorum system obeys ***PBS t-visibility*** consistency if
  - with probability $1 - p_{st}$
  - any read quorum started at least t units of time after a write commits returns at least one value that is at least as recent as that write

PBS t-visibility

- Overwriting data items effectively resets t-visibility; the time between writes bounds t-visibility
- If we space two writes to a key m milliseconds apart, then the t-visibility of the first write for t > m milliseconds is undefined; after m milliseconds, there will be a newer version
- We denote the cumulative density function describing the number of replicas W_r that have received a particular version v exactly t seconds after v commits as $P_w(W_r, t)$

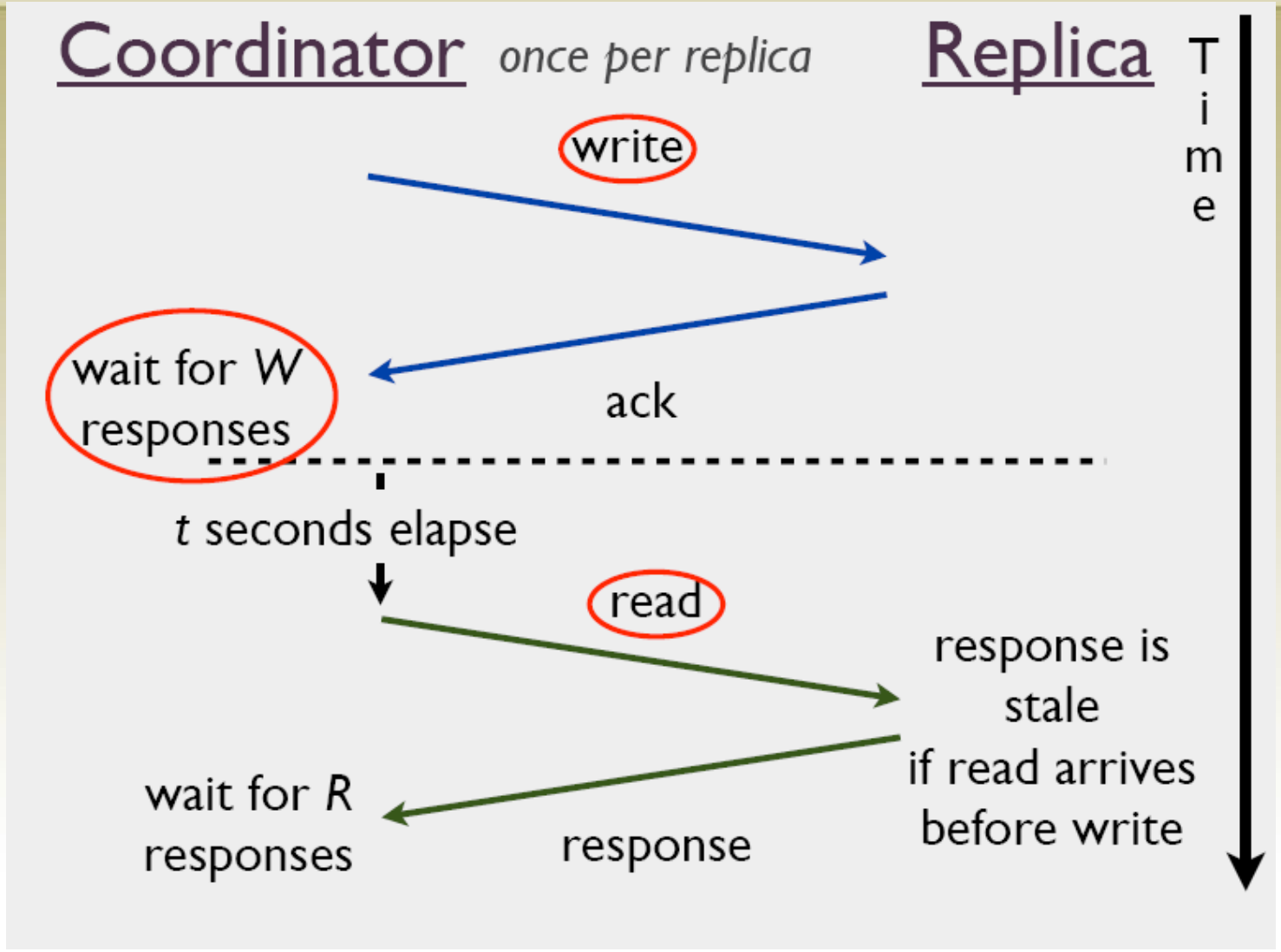
PBS t-visibility

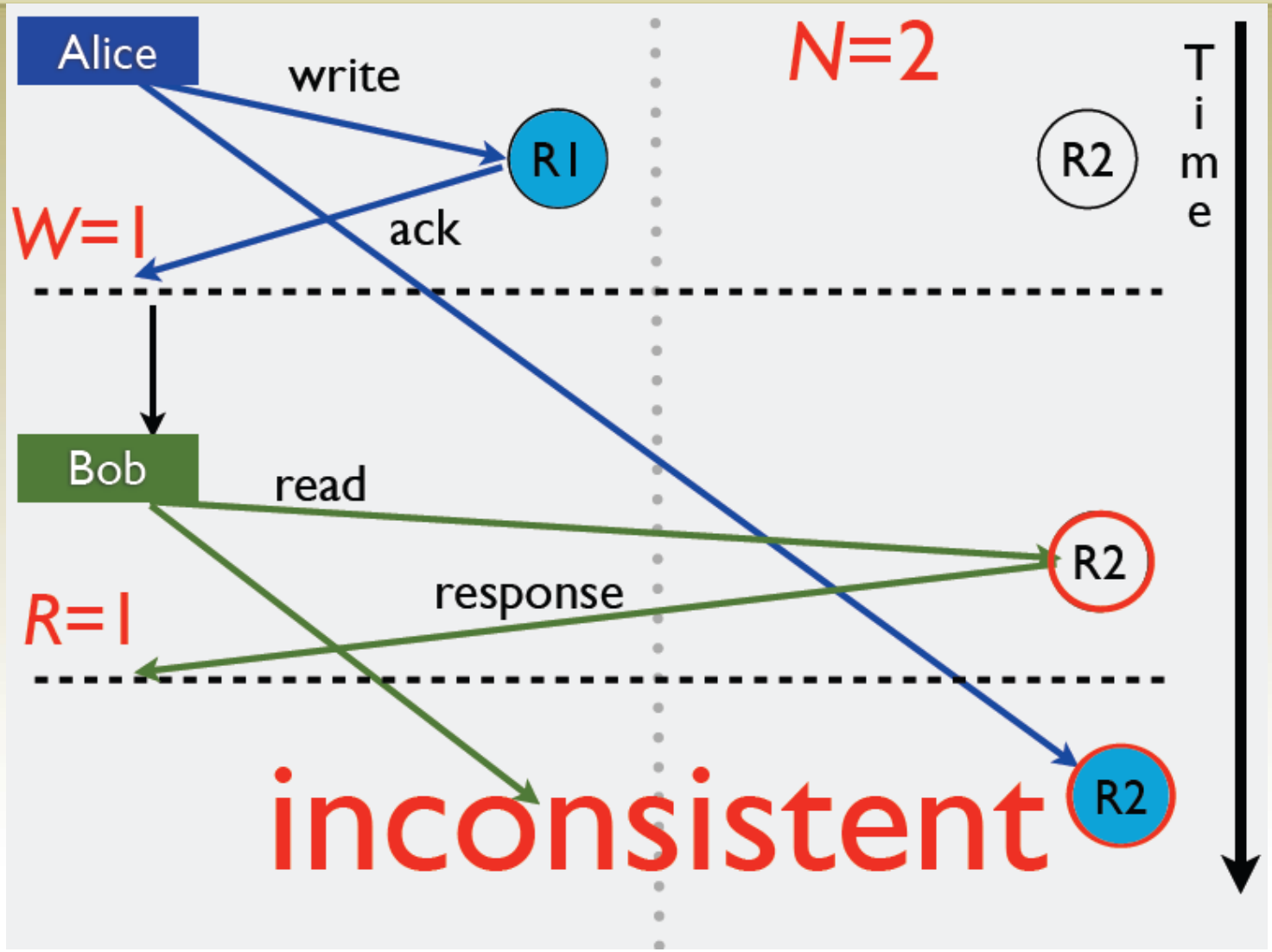
- By definition, for expanding quorums, $\forall c \in [0,W]$, $P_w(c, 0) = 1$; at commit time, W replicas will have received the value with certainty
- We can model the probability of PBS t-visibility for given t by summing the conditional probabilities of each possible W_r:

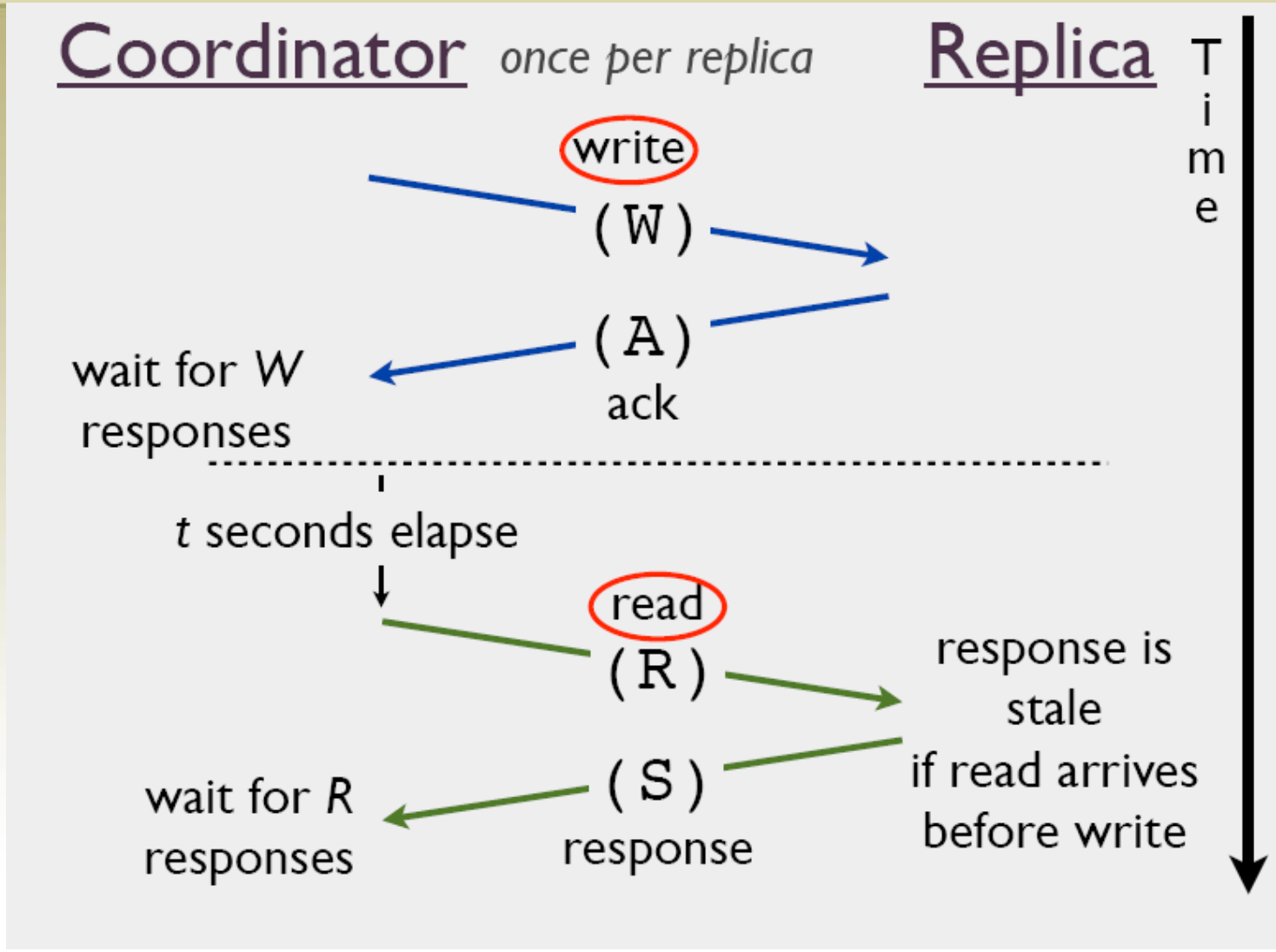
$$p_{st} = \frac{\binom{N-W}{N}}{\binom{N}{R}} + \sum_{c \in (W,N]} \frac{\binom{N-c}{N}}{\binom{N}{R}} \cdot [P_w(c+1,t) - P_w(c,t)] \quad (4)$$

PBS t-visibility

- However, the above equation assumes reads occur instantaneously and writes commit immediately after W replicas have the version (i.e., there is no delay acknowledging the write to the coordinating node)
- In the real world, coordinators wait for write acknowledgments and read requests take time to arrive at remote replicas, increasing t
- Accordingly, Equation 4 is a conservative upper bound on p_{st}


Coordinator
once per replica
Replica
Time
write
wait for W responses
ack
t seconds elapse
read
response is stale if read arrives before write
wait for R responses
response


Alice
write
R1
N=2
R2
Time
W=1
ack
Bob
read
R2
response
R=1
R2
inconsistent


Coordinator
once per replica
Replica
Time
write
(W)
(A)
ack
wait for W
responses
t seconds elapse
read
(R)
response is
stale
if read arrives
before write
(S)
response
wait for R
responses

solving *WARS*:

order statistics
dependent variables

instead:

Monte Carlo methods

to use *WARS*:

gather latency data

W	A	R	S
53.2	10.3	15.3	9.6
44.5	8.2	22.4	14.2
101.1	11.3	19.8	6.7
...	...	...	...

run simulation

Monte Carlo, sampling

Monte Carlo simulation

- Denoting the i-th sample drawn from distribution D as D[i]:
- Draw N samples from W, A, R, and S at time t,
- Compute w_t, the W-th smallest value of {W[i]+A[i],i ∈ [0;N)},
- Check whether the first R samples of R, ordered by R[i] + S[i] obey $\mathbf{w_t+R[i]+t \leq W[i]}$

WARS accuracy

real Cassandra cluster
varying latencies:

t-visibility RMSE: 0.28%
latency N-RMSE: 0.48%

How eventual?

t-visibility: consistent reads with probability *p* after *t* seconds

key: *WARS* model

need: latencies

LinkedIn

175M+ users

built and uses Voldemort

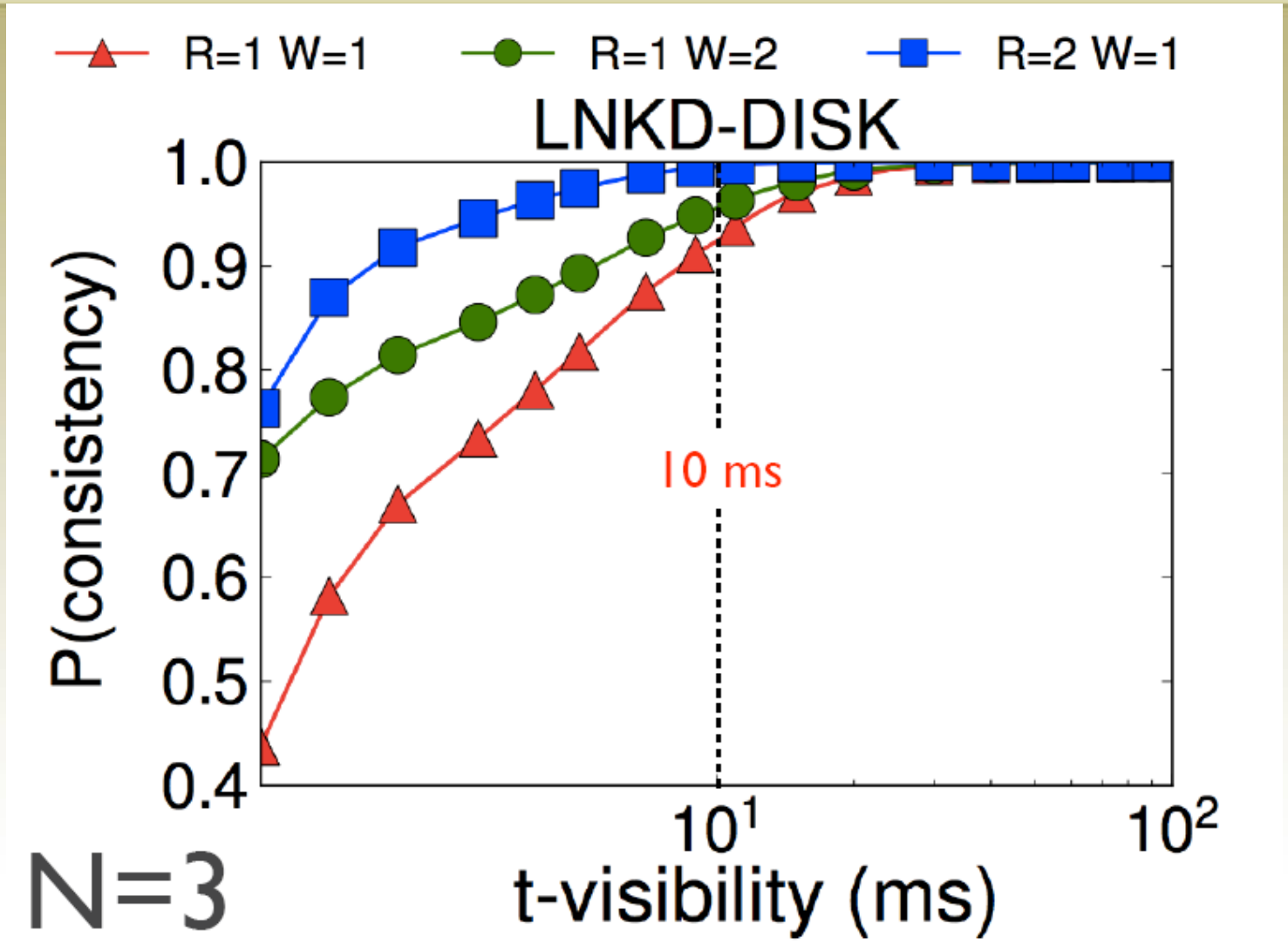
Yammer

100K+ companies

uses Riak

production latencies

fit gaussian mixtures


R=1 W=1
R=1 W=2
R=2 W=1
LNKD-DISK
1.0
0.9
0.8
0.7
0.6
0.5
0.4
P(consistency)
10 ms
10^1
10^2
t-visibility (ms)
N=3

N=3

LNKD-DISK

R=2, W=1, t =13.6 ms

99.9% consistent:

Latency: 12.53 ms

16.5% faster

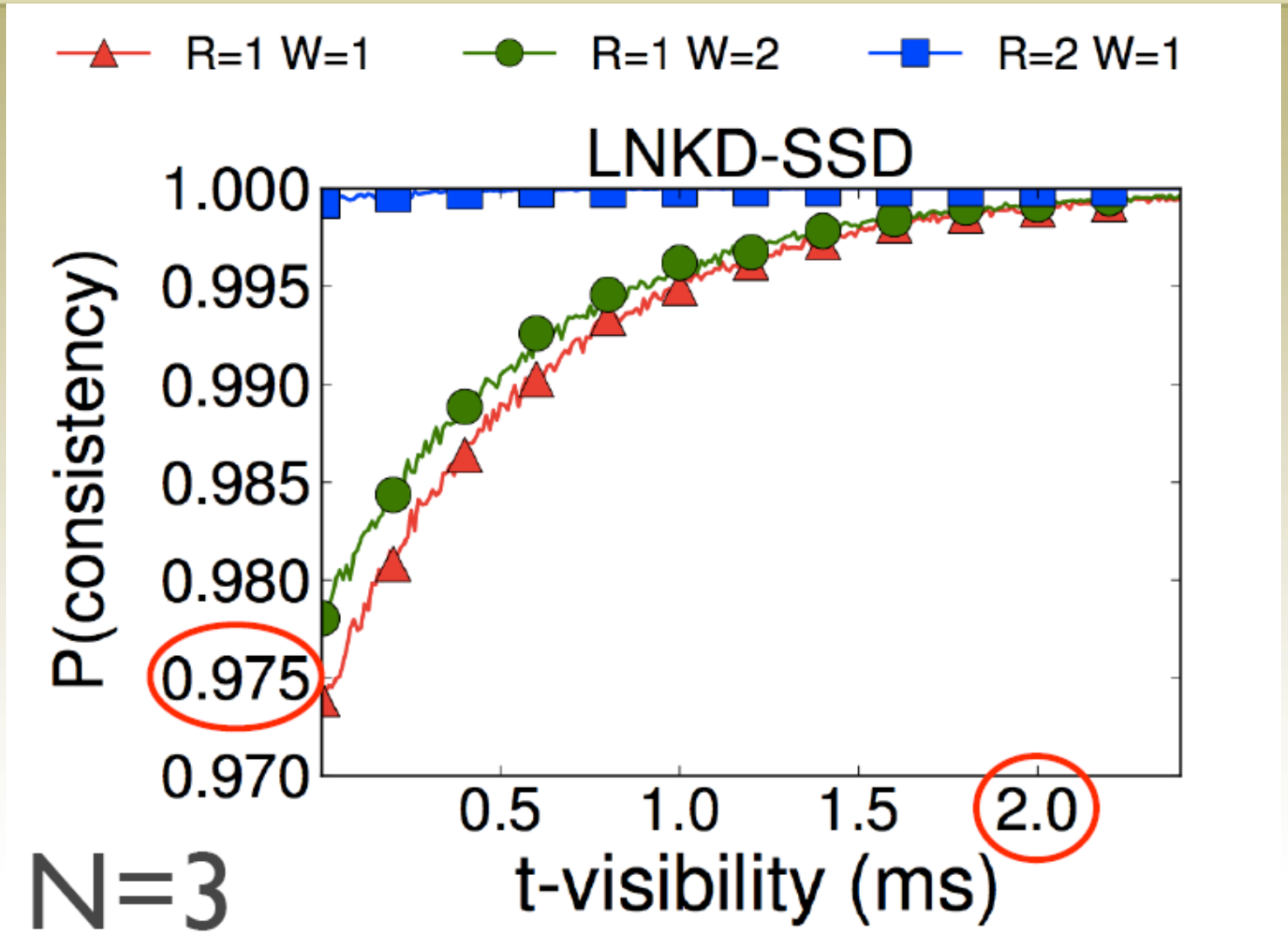
worthwhile?

R=3, W=1

100% consistent:

Latency: 15.01 ms

Latency is combined read and write latency at 99.9th percentile


R=1 W=1
R=1 W=2
R=2 W=1
LNKD-SSD
1.000
0.995
0.990
0.985
0.980
0.975
0.970
P(consistency)
0.5
1.0
1.5
2.0
t-visibility (ms)
N=3

N=3

LNKD-SSD

R=1, W=1, t = 1.85 ms

99.9% consistent:

Latency: 1.32 ms

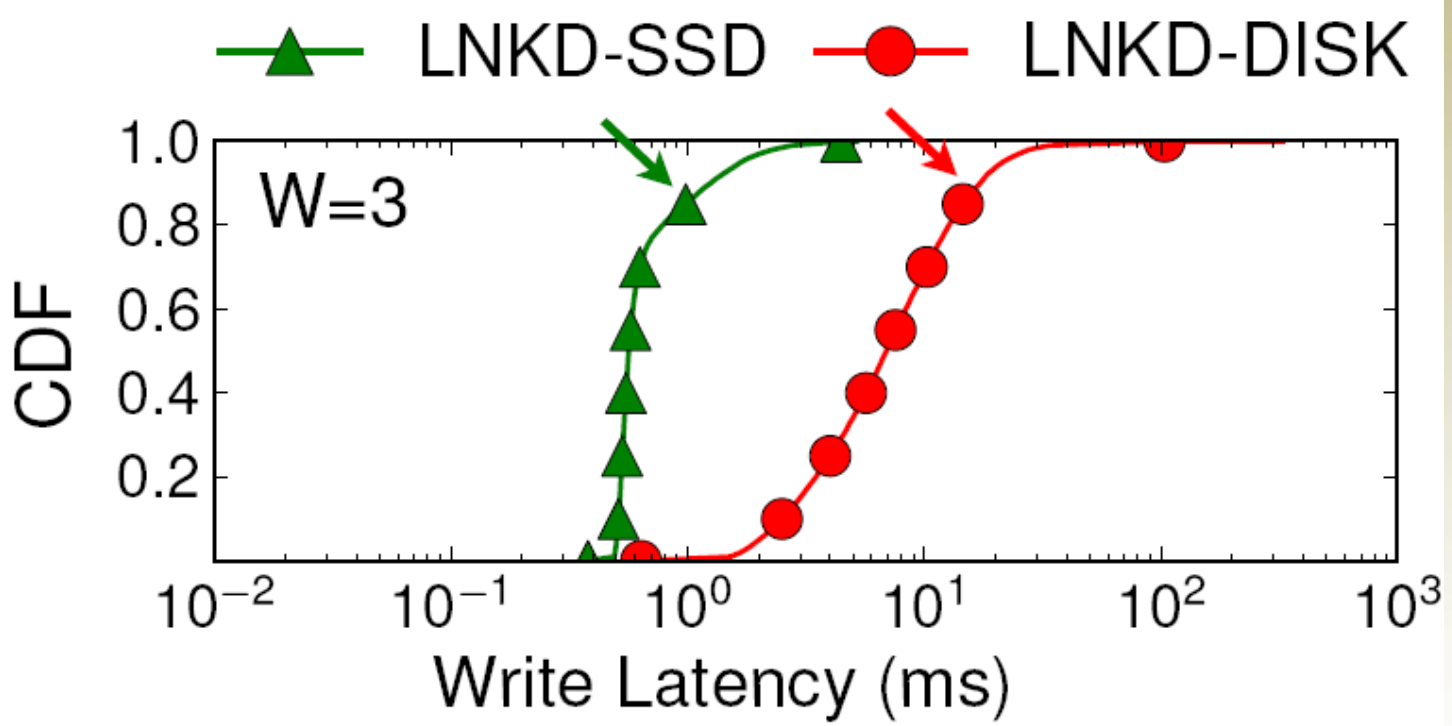
59.5% faster

R=3, W=1

100% consistent:

Latency: 4.20 ms

Latency is combined read and write latency at 99.9th percentile


LNKD-SSD
LNKD-DISK
W=3
CDF
1.0
0.8
0.6
0.4
0.2
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
Write Latency (ms)

N=3