

CSE 444: Database Internals

Lectures 26

NoSQL: Key Value Stores

References

- **Scalable SQL and NoSQL Data Stores**, Rick Cattell, SIGMOD Record, December 2010 (Vol. 39, No. 4)
- **Dynamo: Amazon's Highly Available Key-value Store**. By Giuseppe DeCandia et. al. SOSP 2007.
- Online documentation: **Amazon DynamoDB**.

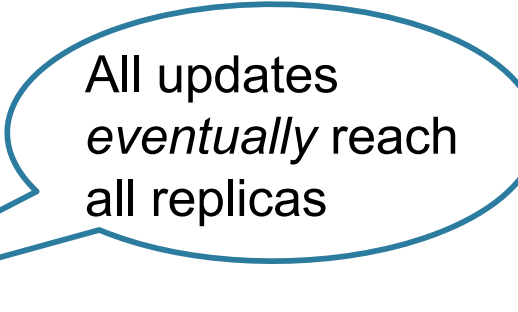
NoSQL Motivation

- Originally motivated by Web 2.0 applications
- Goal is to scale simple OLTP-style workloads to thousands or millions of users
- Users are doing both updates and reads

Why NoSQL as the Solution?

- Hard to scale *transactions*
 - Need to partition the database across multiple machines
 - If a transaction touches one machine, life is good
 - If a transaction touches multiple machines, ACID becomes extremely expensive! Need two-phase commit
- Replication
 - Replication can help to increase throughput and lower latency
 - Create multiple copies of each database partition
 - Spread queries across these replicas
 - Easy for reads but writes, once again, become expensive!

NoSQL Key Feature Decisions

- Want a data management system that is
 - Elastic and highly scalable
 - Flexible (different records have different schemas)
 - To achieve above goals, willing to give up
 - Complex queries: e.g., give up on joins
 - Multi-object transactions
 - ACID guarantees: e.g., *eventual consistency* is OK
 - Eventual consistency: If updates stop, all replicas will *converge* to the same state and all reads will return the same value
 - BASE (Basically Available, Soft state, Eventually consistent)
 - *Not all NoSQL systems give up all these properties*
- 
- All updates eventually reach all replicas

NoSQL

“Not Only SQL” or “Not Relational”.

Six key features:

1. Scale horizontally “simple operations”
2. Replicate/distribute data over many servers
3. Simple call level interface (contrast w/ SQL)
4. Weaker concurrency model than ACID
5. Efficient use of distributed indexes and RAM
6. Flexible schema

Data Models

- **Tuple** = row in a relational db
- **Key-value** = records identified with keys have values that are opaque blobs
- **Extensible record** = families of attributes have a schema, but new attributes may be added
- **Document** = nested values, extensible records (XML, JSON, protobuf, attribute-value pairs)

Different Types of NoSQL

Taxonomy based on data models:



Today

- **Key-value stores**
 - e.g., Project Voldemort, Memcached, Redis
- **Extensible Record Stores**
 - e.g., HBase, Cassandra, PNUTS
- **Document stores**
 - e.g., SimpleDB, CouchDB, MongoDB
- **Most recently: Graph databases**
- **New types of RDBMSs.. not really NoSQL**
 - Next lecture

Key-Value Store: Dynamo

- **Dynamo: Amazon's Highly Available Key-value Store.** By Giuseppe DeCandia et. al. SOSP 2007.
- Main observation:
 - “There are many services on Amazon’s platform that only need **primary-key access** to a data store.”
 - Best seller lists, shopping carts, customer preferences, session management, sales rank, product catalog

Basic Features

- **Data model:** (key,value) pairs
 - Values are binary objects (blobs)
 - No further schema
- **Operations**
 - Insert/delete/lookup by key
 - No operations across multiple data items
- **Consistency**
 - Replication with eventual consistency
 - Goal to NEVER reject any writes (bad for business)
 - Multiple versions with conflict resolution during reads

Operations

- **get(key)**
 - Locates object replicas associated with *key*
 - Returns a single *object*
 - Or a list of objects with conflicting versions
 - Also returns a *context*
 - Context holds metadata including version
 - Context is opaque to caller
- **put(key, context, object)**
 - Determines where replicas of object should be placed
 - Location depends on key value
 - Data stored persistently including context

Storage: Distributed Hash Table

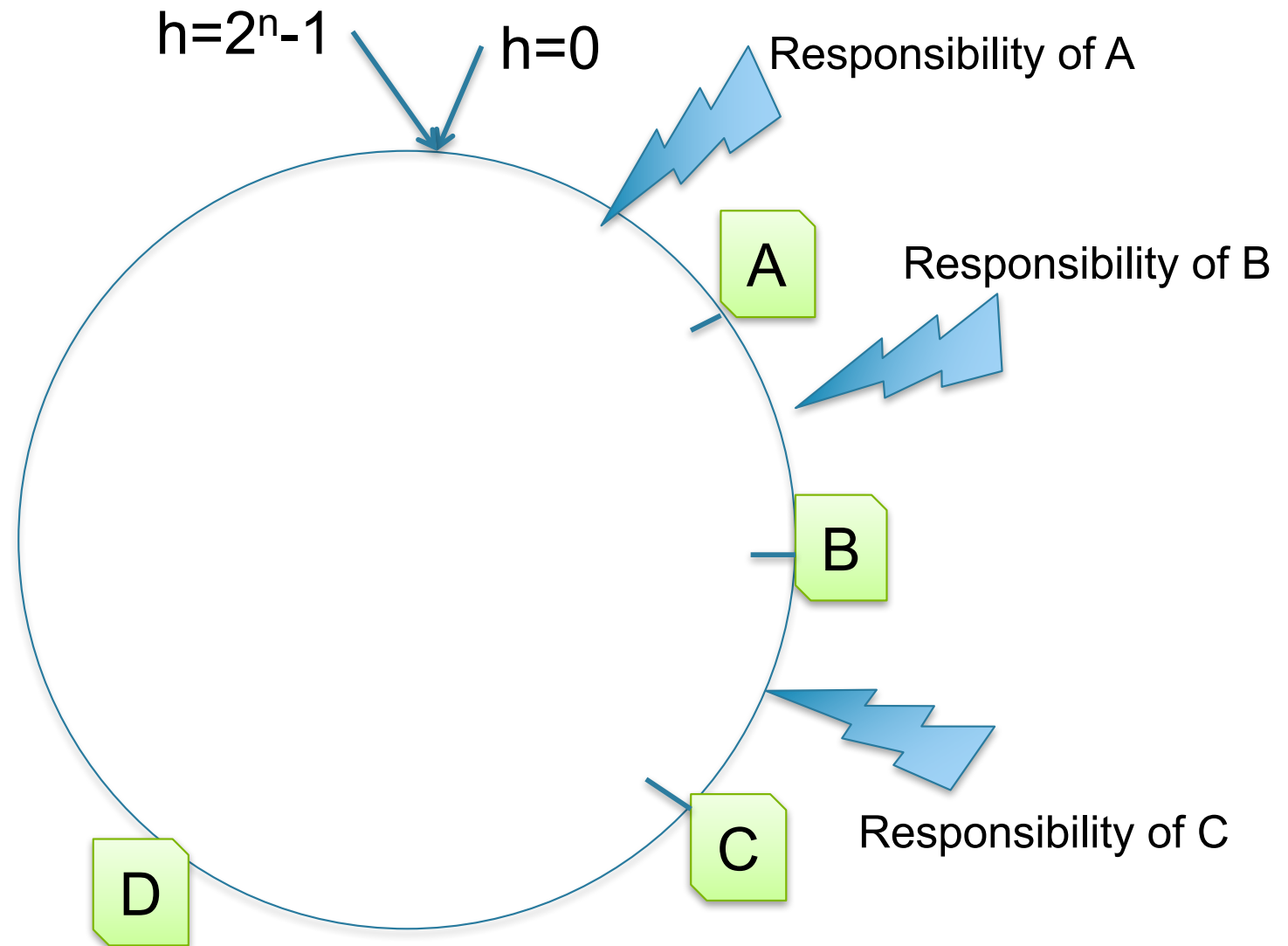
Implements a distributed storage

- Each key-value pair (k,v) is stored at some server $h(k)$
- API: `write(k,v)`; `read(k)`

Use standard hash function: service key k by server $h(k)$

- Problem 1: a client knows only one server, doesn't know how to access $h(k)$
- Problem 2. if new server joins, then $N \rightarrow N+1$, and the entire hash table needs to be reorganized
- Problem 3: we want replication, i.e. store the object at more than one server

Distributed Hash Table



Distributed Hash Table Details

- This type of hashing called “**consistent hashing**”
- Basic approach leads to load imbalance
 - Solution: Use V virtual nodes for each physical node
 - Virtual nodes provide better load balance
 - Nb of virtual nodes can vary based on capacity

Problem 1: Routing

A client doesn't know server $h(k)$, but some other server

- Naive routing algorithm:
 - Each node knows its neighbors
 - Send message to nearest neighbor
 - Hop-by-hop from there
 - Obviously this is $O(n)$, so no good
- Better algorithm: “finger table”
 - Memorize locations of other nodes in the ring
 - $a, a + 2, a + 4, a + 8, a + 16, \dots a + 2^n - 1$
 - Send message to closest node to destination
 - Hop-by-hop again: this is $\log(n)$

Problem 1: Routing

h(k) handled by server G

Read(k)

$h=2^n-1$ $h=0$

Client only "knows" server A

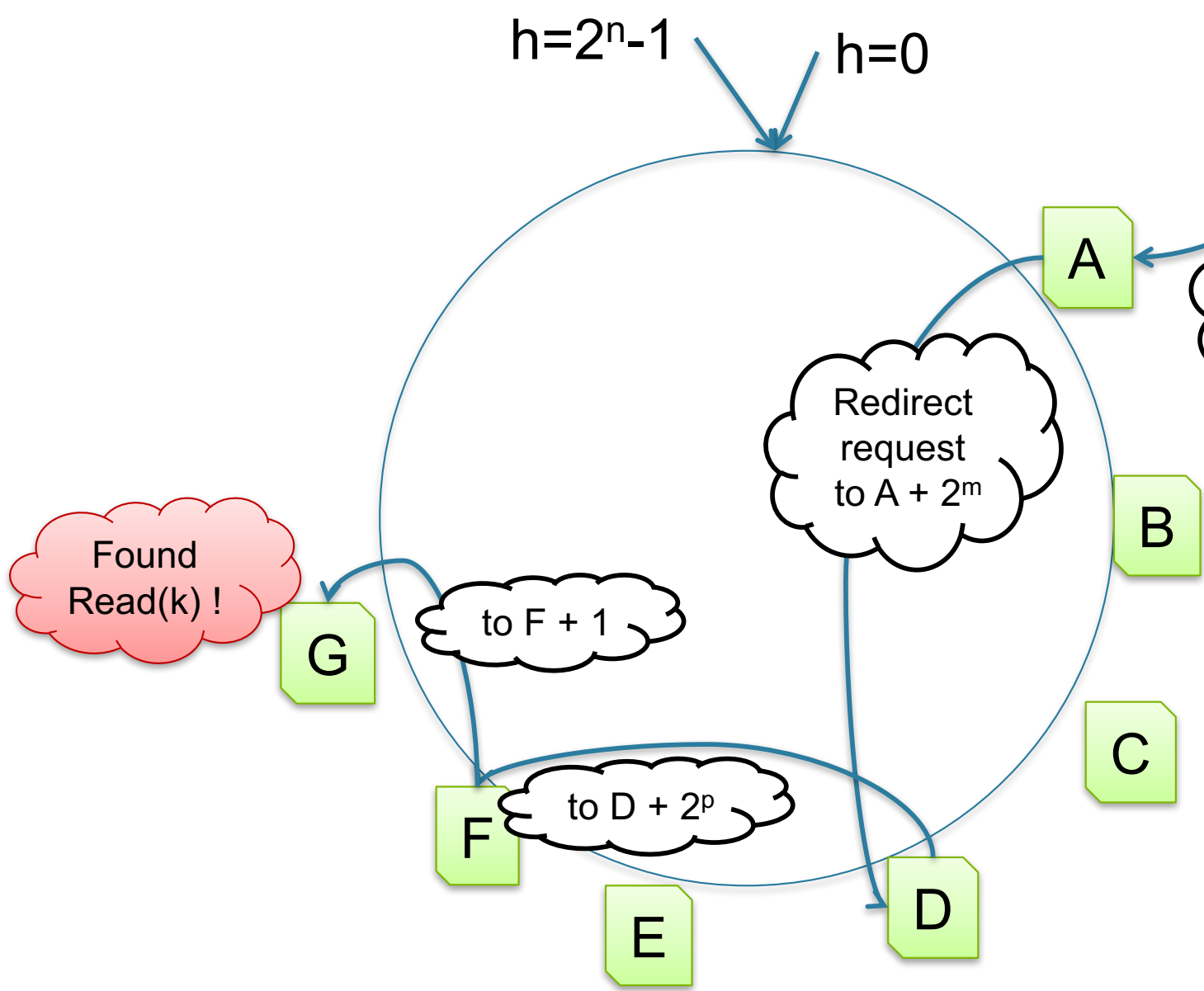
Redirect request to $A + 2^m$

to $F + 1$

to $D + 2^p$

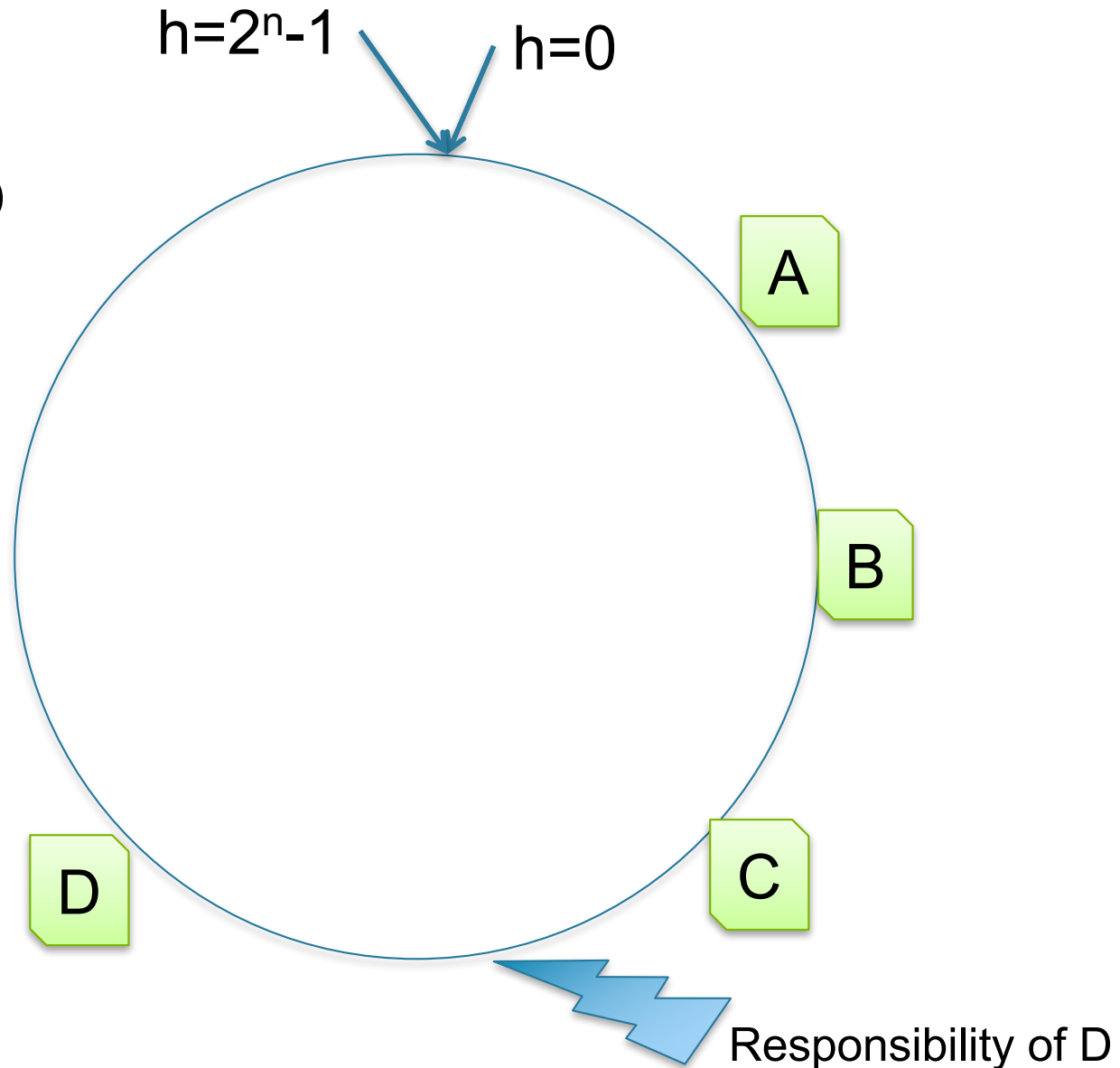
Found Read(k)!

$O(\log n)$



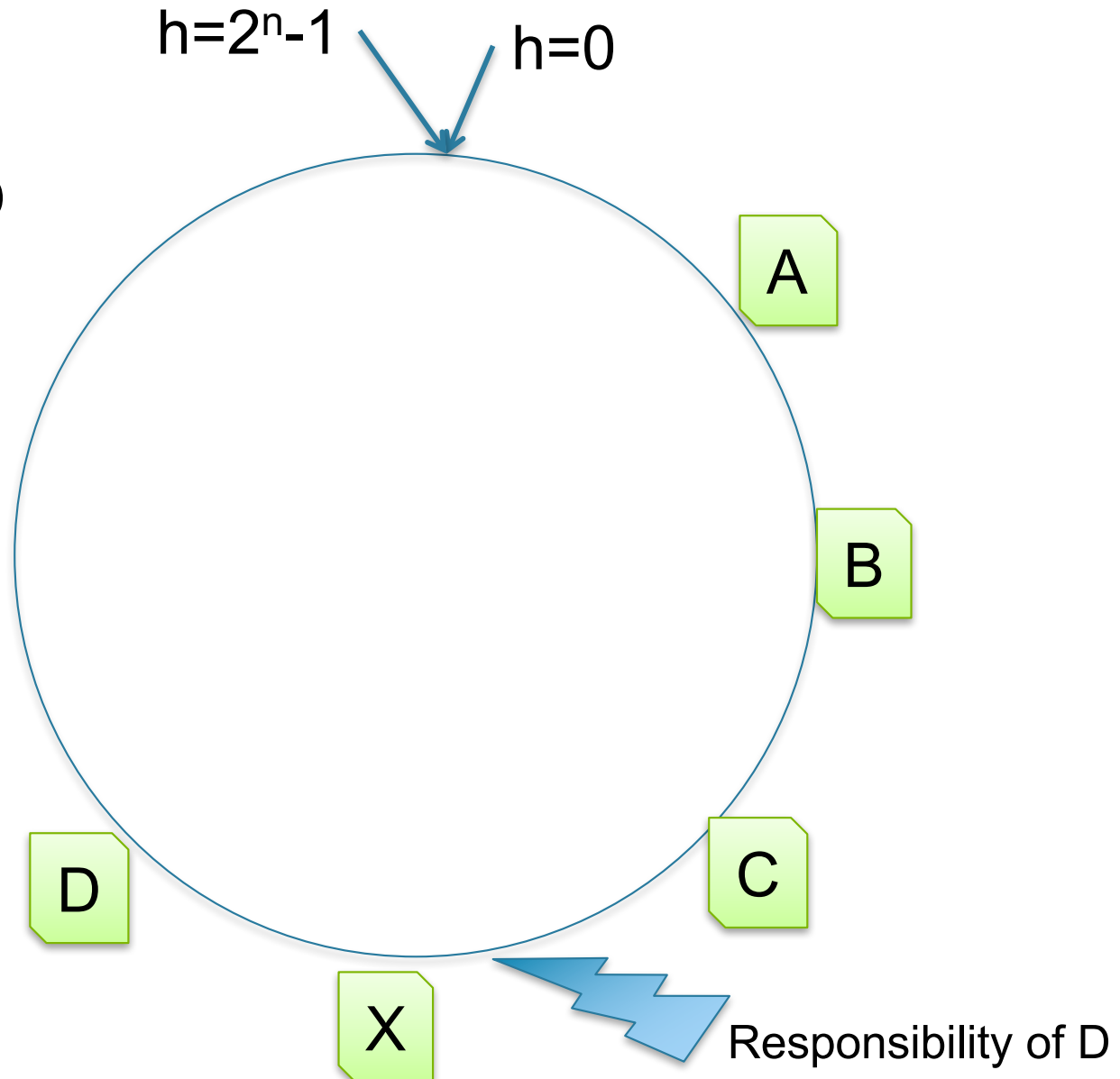
Problem 2: Joining

When X joins:
select random ID



Problem 2: Joining

When X joins:
select random ID



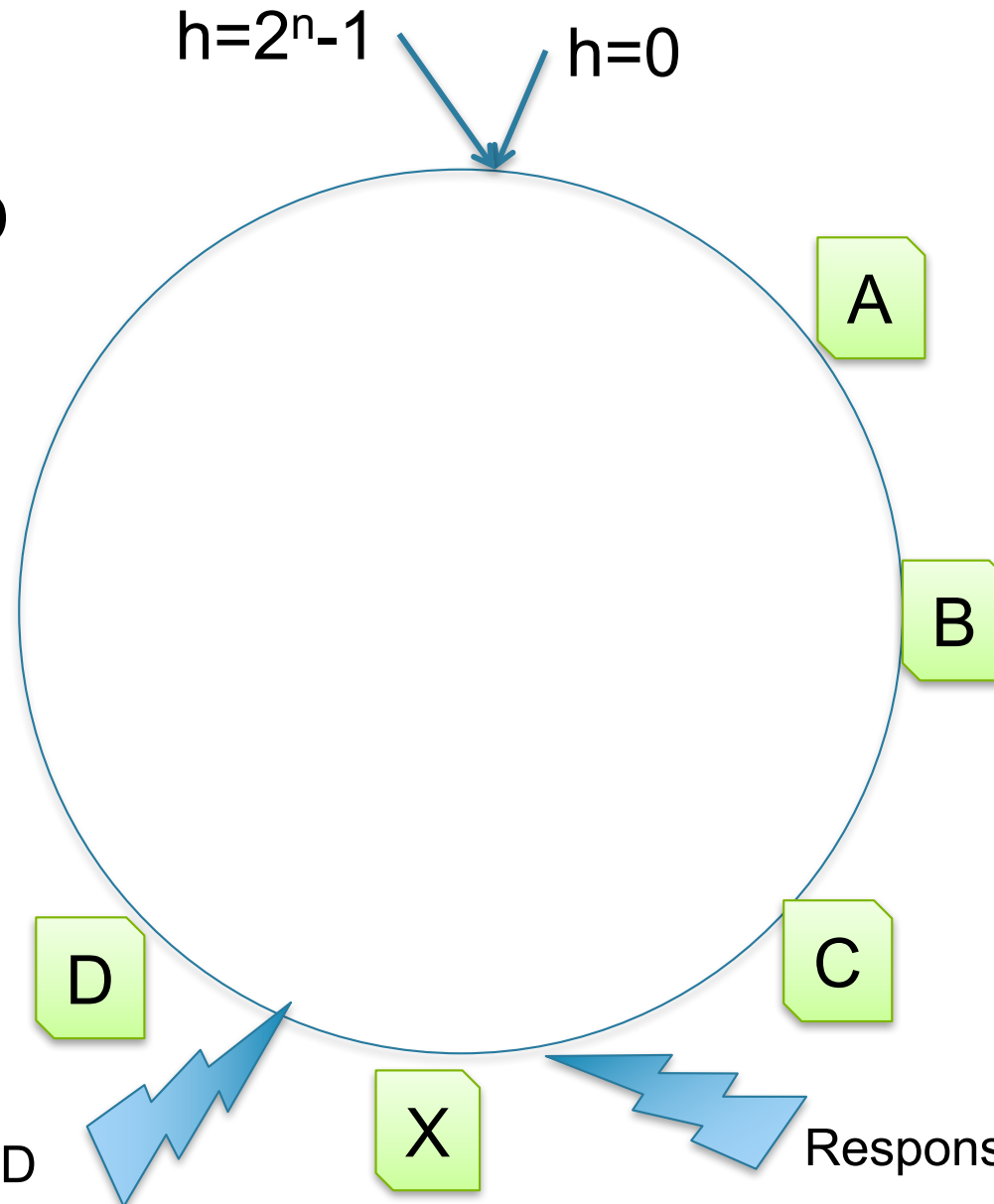
Problem 2: Joining

When X joins:
select random ID

Redistribute
the load at D

Responsibility of D

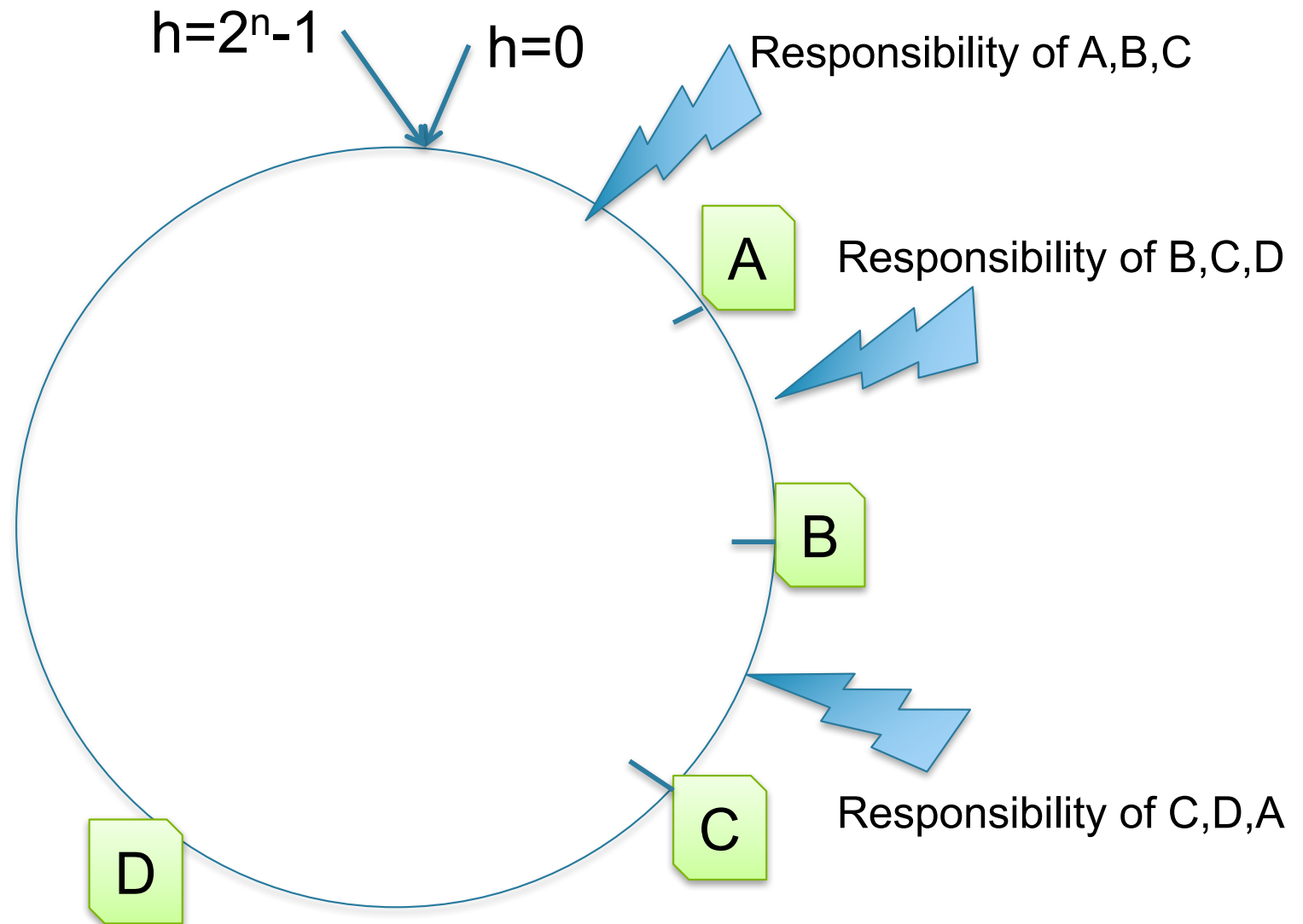
Responsibility of X



Problem 3: Replication

- Need to have some degree of replication to cope with node failures
- Let N =degree of replication
- Assign key k to $h(k), h(k)+1, \dots, h(k)+N-1$

Problem 3: Replication



Additional Dynamo Details

- Each key assigned to a *coordinator*
- Coordinator responsible for replication
 - Replication skips virtual nodes that are not distinct physical nodes
- Set of replicas for a key is its *preference list*
- One-hop routing:
 - Each node knows preference list of each key
- “Sloppy quorum” replication
 - Each update creates a new version of an object
 - Vector clocks track causality between versions

Vector Clocks

- An extension of Multiversion Concurrency Control (MVCC) to multiple servers
- Standard MVCC:
each data item X has a timestamp t :
 $X_4, X_9, X_{10}, X_{14}, \dots, X_t$
- Vector Clocks:
 X has set of [server, timestamp] pairs
 $X([s1,t1], [s2,t2], \dots)$

Vector Clocks

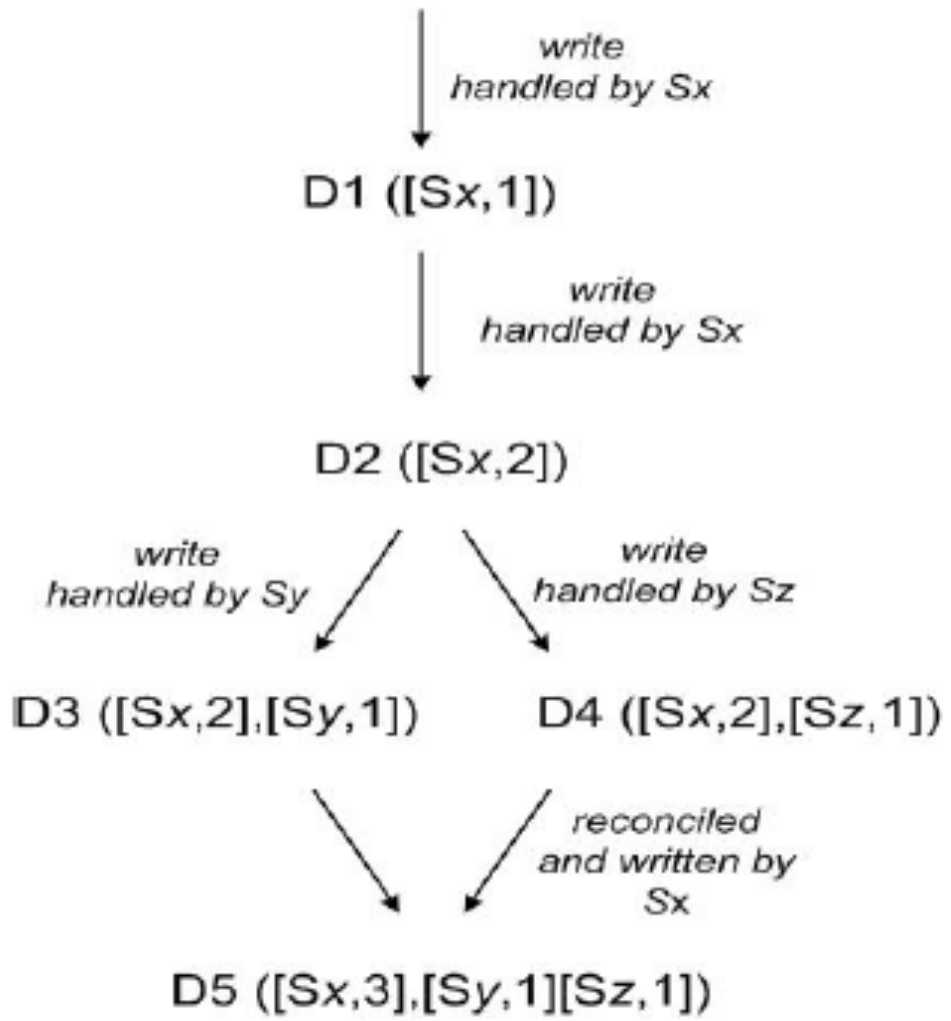


Figure 3: Version evolution of an object over time.

Vector Clocks: Example

- A client writes D1 at server SX:
D1 ([SX,1])
- Another client reads D1, writes back D2; also handled by server SX:
D2 ([SX,2]) (D1 garbage collected)
-
-
-

Vector Clocks: Example

- A client writes D1 at server SX:
D1 ([SX,1])
- Another client reads D1, writes back D2; also handled by server SX:
D2 ([SX,2]) (D1 garbage collected)
- Another client reads D2, writes back D3;
handled by server SY:
D3 ([SX,2], [SY,1])
-
-

Vector Clocks: Example

- A client writes D1 at server SX:
D1 ([SX,1])
- Another client reads D1, writes back D2; also handled by server SX:
D2 ([SX,2]) (D1 garbage collected)
- Another client reads D2, writes back D3; handled by server SY:
D3 ([SX,2], [SY,1])
- Another client reads D2, writes back D4; handled by server SZ:
D4 ([SX,2], [SZ,1])

-

Vector Clocks: Example

- A client writes D1 at server SX:
D1 ([SX,1])
- Another client reads D1, writes back D2; also handled by server SX:
D2 ([SX,2]) (D1 garbage collected)
- Another client reads D2, writes back D3; handled by server SY:
D3 ([SX,2], [SY,1])
- Another client reads D2, writes back D4; handled by server SZ:
D4 ([SX,2], [SZ,1])
- Another client reads D3 and D4: **CONFLICT !**

Vector Clocks: Meaning

- A data item $D[(S_1, v_1), (S_2, v_2), \dots]$ means a value that represents version v_1 for S_1 , version v_2 for S_2 , etc.
- If server S_i updates D , then:
 - It must increment v_i , if (S_i, v_i) exists
 - Otherwise, it must create a new entry $(S_i, 1)$

Vector Clocks: Conflicts

- A data item D is an ancestor of D' if for all $(S, v) \in D$ there exists $(S, v') \in D'$ s.t. $v \leq v'$
- Otherwise, D and D' are on parallel branches, and it means that they have a conflict that needs to be reconciled semantically

Vector Clocks: Conflict or not?

Data 1	Data 2	Conflict ?
([SX,3],[SY,6])	([SX,3],[SZ,2])	

Vector Clocks: Conflict or not?

Data 1	Data 2	Conflict ?
([SX,3],[SY,6])	([SX,3],[SZ,2])	Yes

Vector Clocks: Conflict or not?

Data 1	Data 2	Conflict ?
([SX,3],[SY,6])	([SX,3],[SZ,2])	Yes
([SX,3])	([SX,5])	

Vector Clocks: Conflict or not?

Data 1	Data 2	Conflict ?
([SX,3],[SY,6])	([SX,3],[SZ,2])	Yes
([SX,3])	([SX,5])	No

Vector Clocks: Conflict or not?

Data 1	Data 2	Conflict ?
([SX,3],[SY,6])	([SX,3],[SZ,2])	Yes
([SX,3])	([SX,5])	No
([SX,3],[SY,6])	([SX,3],[SY,6],[SZ,2])	

Vector Clocks: Conflict or not?

Data 1	Data 2	Conflict ?
([SX,3],[SY,6])	([SX,3],[SZ,2])	Yes
([SX,3])	([SX,5])	No
([SX,3],[SY,6])	([SX,3],[SY,6],[SZ,2])	No

Vector Clocks: Conflict or not?

Data 1	Data 2	Conflict ?
([SX,3],[SY,6])	([SX,3],[SZ,2])	Yes
([SX,3])	([SX,5])	No
([SX,3],[SY,6])	([SX,3],[SY,6],[SZ,2])	No
([SX,3],[SY,10])	([SX,3],[SY,6],[SZ,2])	

Vector Clocks: Conflict or not?

Data 1	Data 2	Conflict ?
([SX,3],[SY,6])	([SX,3],[SZ,2])	Yes
([SX,3])	([SX,5])	No
([SX,3],[SY,6])	([SX,3],[SY,6],[SZ,2])	No
([SX,3],[SY,10])	([SX,3],[SY,6],[SZ,2])	Yes

Vector Clocks: Conflict or not?

Data 1	Data 2	Conflict ?
([SX,3],[SY,6])	([SX,3],[SZ,2])	Yes
([SX,3])	([SX,5])	No
([SX,3],[SY,6])	([SX,3],[SY,6],[SZ,2])	No
([SX,3],[SY,10])	([SX,3],[SY,6],[SZ,2])	Yes
([SX,3],[SY,10])	([SX,3],[SY,20],[SZ,2])	

Vector Clocks: Conflict or not?

Data 1	Data 2	Conflict ?
([SX,3],[SY,6])	([SX,3],[SZ,2])	Yes
([SX,3])	([SX,5])	No
([SX,3],[SY,6])	([SX,3],[SY,6],[SZ,2])	No
([SX,3],[SY,10])	([SX,3],[SY,6],[SZ,2])	Yes
([SX,3],[SY,10])	([SX,3],[SY,20],[SZ,2])	No

(Sloppy) Quorum Read/Write

- Parameters:
 - N = number of copies (replicas) of each object
 - R = minimum number of nodes that must participate in a successful read
 - W = minimum number of nodes that must participate in a successful write
- Quorum: $R+W > N$
- Sloppy Quorum (Dynamo): allow $R+W \leq N$
 - Allow fewer than N to get better latency

Operation Execution

- Write operations
 - Initial request sent to coordinator
 - Coordinator generates vector clock & stores locally
 - Coordinator forwards new version to all N replicas
 - If at least $W-1 < N-1$ nodes respond then success!
- Read operations
 - Initial request sent to coordinator
 - Coordinator requests data from all N replicas
 - Once gets R responses, returns data
- Sloppy quorum: Involve first N *healthy* nodes

Amazon DynamoDB

Additional functionality:

- Both document and key-value store models
- Offers secondary indexes to enable queries over non-key attributes
 - So can support selection and projection queries
- Offers choice of eventual consistent vs strongly consistent read

Try Amazon DynamoDB

<http://aws.amazon.com/dynamodb/>

Amazon DynamoDB Data Model

- Tables containing Items
 - Items are described with attributes
 - One attribute must be the primary key
 - Primary key can be a single partition key attribute
 - Or a pair of (partition key k1, sort key k2)
 - Items partitioned across nodes on k1
 - Sorted within the node on k2

Amazon DynamoDB Querying

- Selection and projection queries
 - Equality predicates on primary key
 - Must create secondary indexes to query other attributes. Also equality predicates
 - Can specify attributes to return (projection)
 - Can specify path notation for document attributes

Amazon DynamoDB Consistency

- Eventually consistent read
 - “When you read data from a DynamoDB table, the response might not reflect the results of a recently completed write operation. The response might include some stale data. However, if you repeat your read request after a short time, the response should return the latest data.”
- Strongly consistent read
 - “When you request a strongly consistent read, DynamoDB returns a response with the most up-to-date data, reflecting the updates from all prior write operations that were successful. Note that a strongly consistent read might not be available in the case of a network delay or outage.”