NoSQL

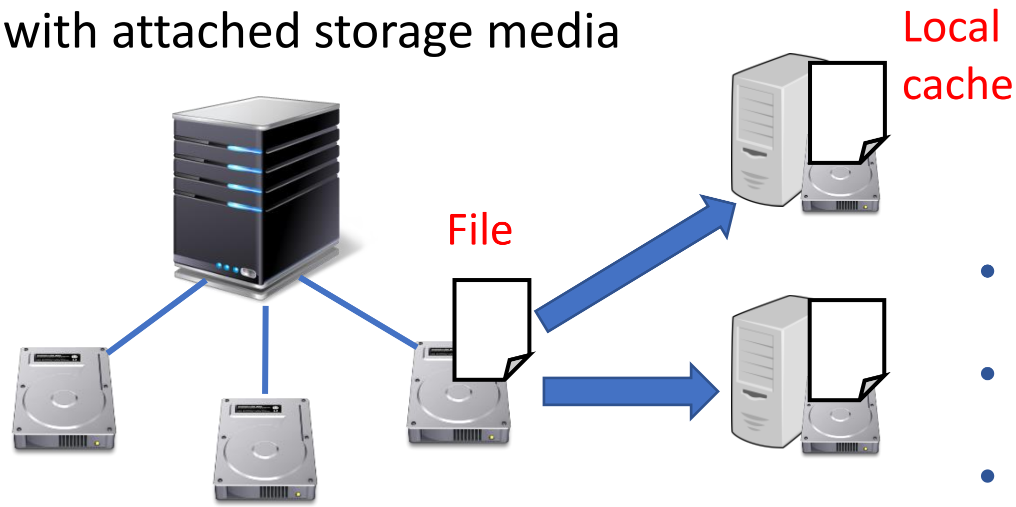
ENGR689 (Sprint)

TEXAS A&M UNIVERSITY

How to Store Data In Cloud?

- Networked File Systems

Centralized server
with attached storage media



- Hierarchical structures
- Difficult to scale for more storage
- Good for static contents
- Extremely weak consistency (open-close consistency)

RDBMS

(Relational Database Management Systems)

- RDBMS has:
  - Predefined schemas with tuples / records / rows
  - Well-defined SQL interface

```
SELECT Name, SUM(Grade) FROM
students JOIN Records
ON Students.ID = Records.ID
GROUP BY Students.ID
```

- Easy to program join/union operations
- Strong consistency for updates

Schema - Students

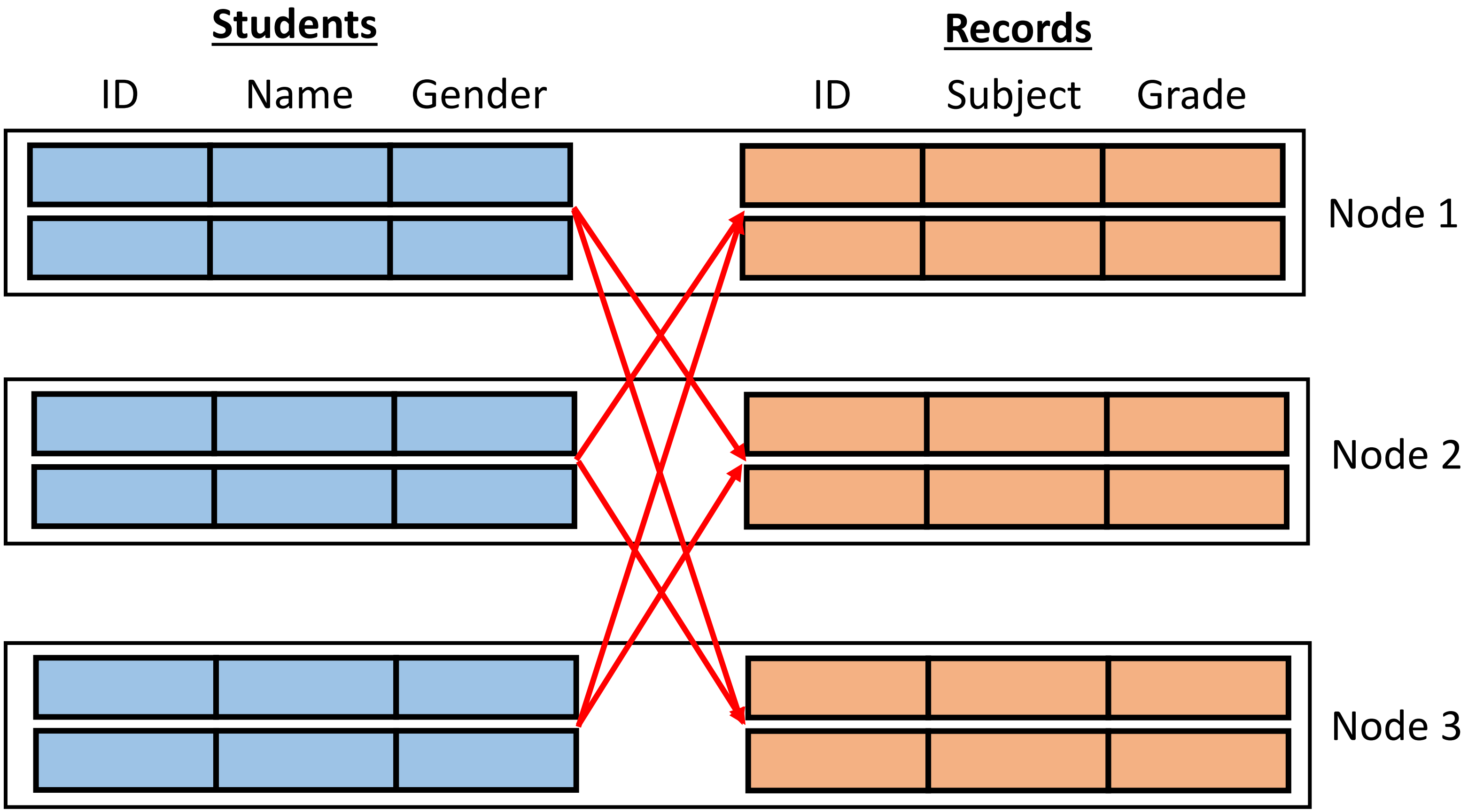
ID	Name	Gender
🟦	🟦	🟦
🟦	🟦	🟦
🟦	🟦	🟦

Schema - Records

ID	Subject	Grade
🟧	🟧	🟧
🟧	🟧	🟧
🟧	🟧	🟧

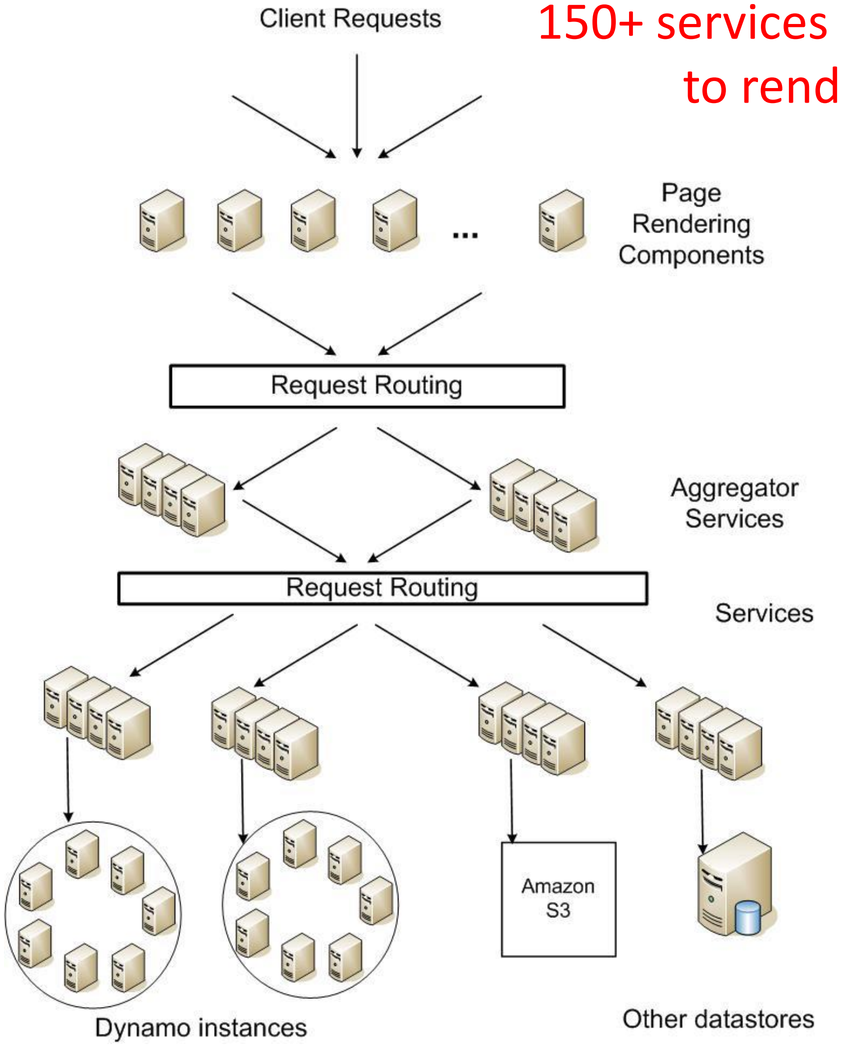
How to Scale RDBMS

- Exchanging data on partitioned DB is expensive



Data movement becomes the main bottleneck

Dynamo



150+ services inter-connected to render a page

- Highly scalable and available key-value store for Amazon
- Service Level Agreement (SLA): **99.9% of requests receive responses within 300ms**

NoSQL (Not Only SQL)

- Auto-sharding without relational operations
- Optimized for simple queries (e.g., what's the name of UID 012345?)
- Map: Key →
  - Value (Key-Value Stores, e.g., Dynamo, Memcached)
  - Document (Document Stores, e.g., MongoDB)
  - Node with connections (Graph Stores, e.g., Neo4j)
  - Columns (Wide Column Stores, e.g., HBase, Cassandra)

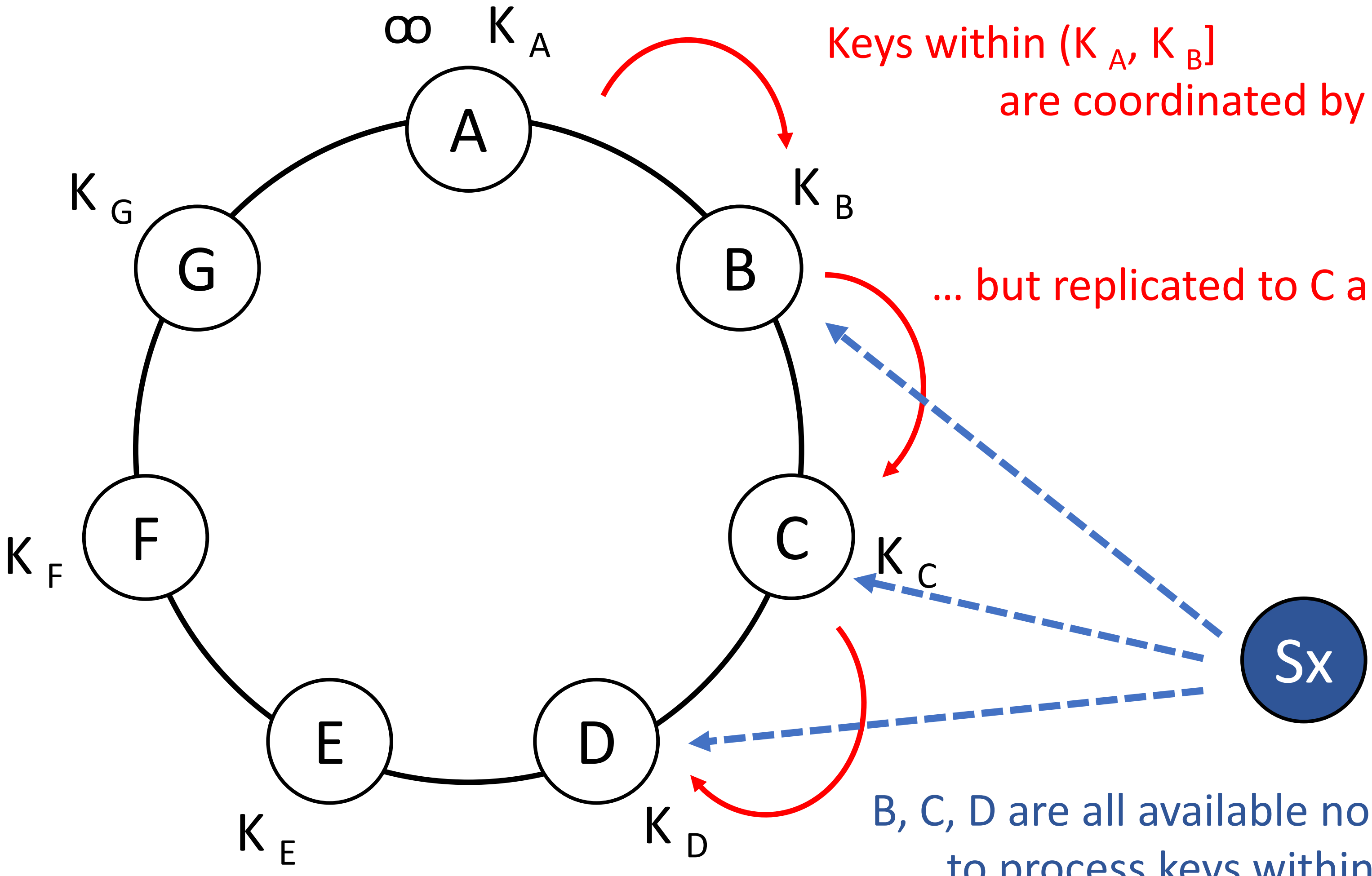
Simple Key-Value Store API

- Get(Key) → Value
- Put(Key, Value)

High Availability: Both Get and Put need to finish immediately, how?

  - Multiple replicas to ensure no blocking on node failure
  - Tolerate temporary inconsistency to prevent blocking on synchronization

Partitioning & Replication

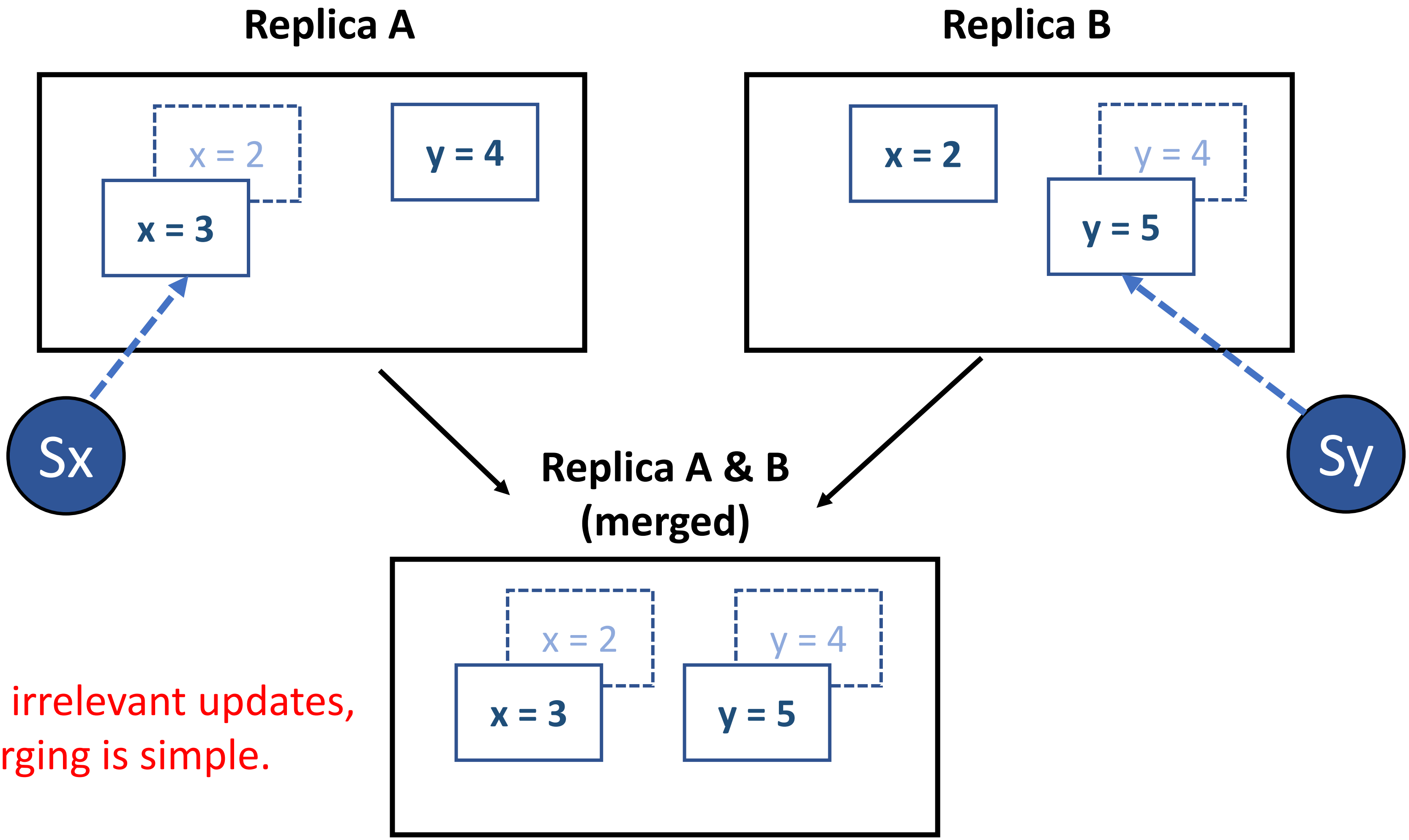


Keys within (K_A, K_B] are coordinated by B

… but replicated to C and D.

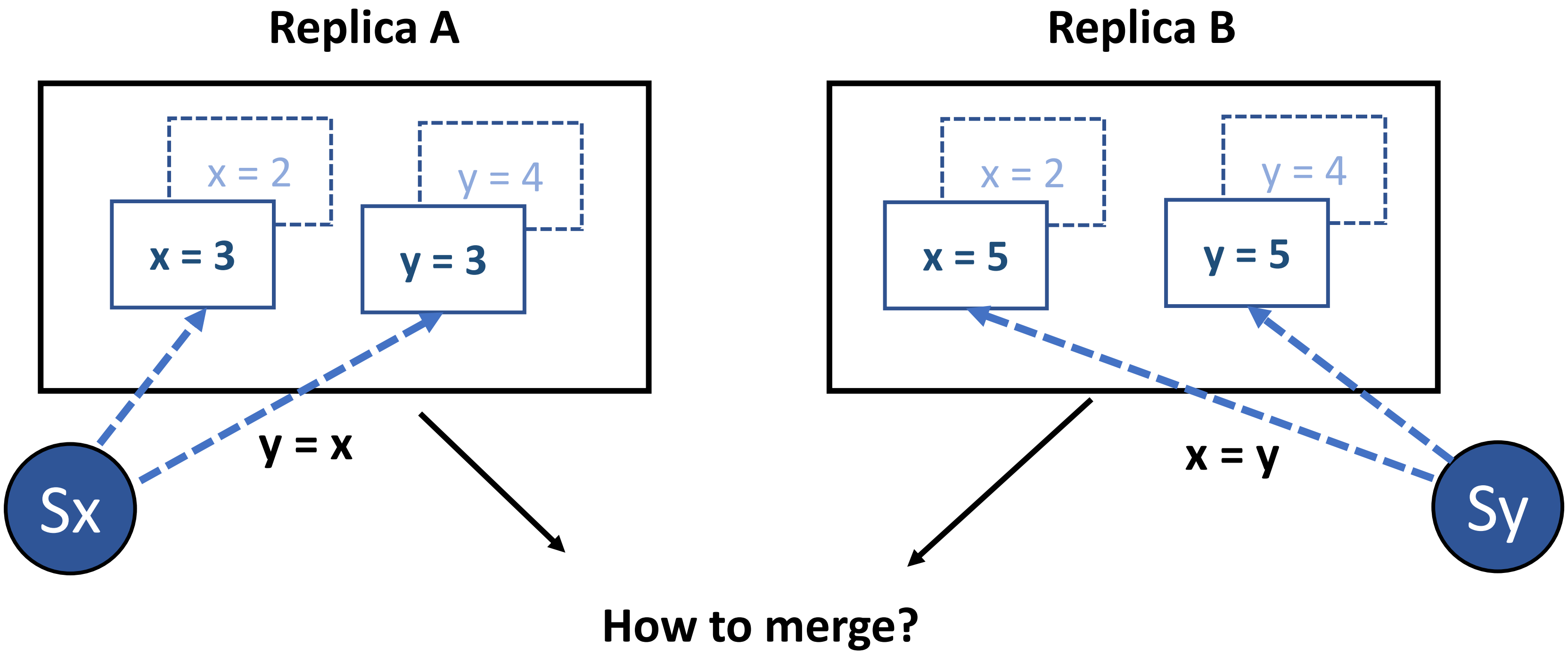
B, C, D are all available nodes to process keys within (K_A, K_B]

Eventual Consistency



For irrelevant updates, merging is simple.

Eventual Consistency

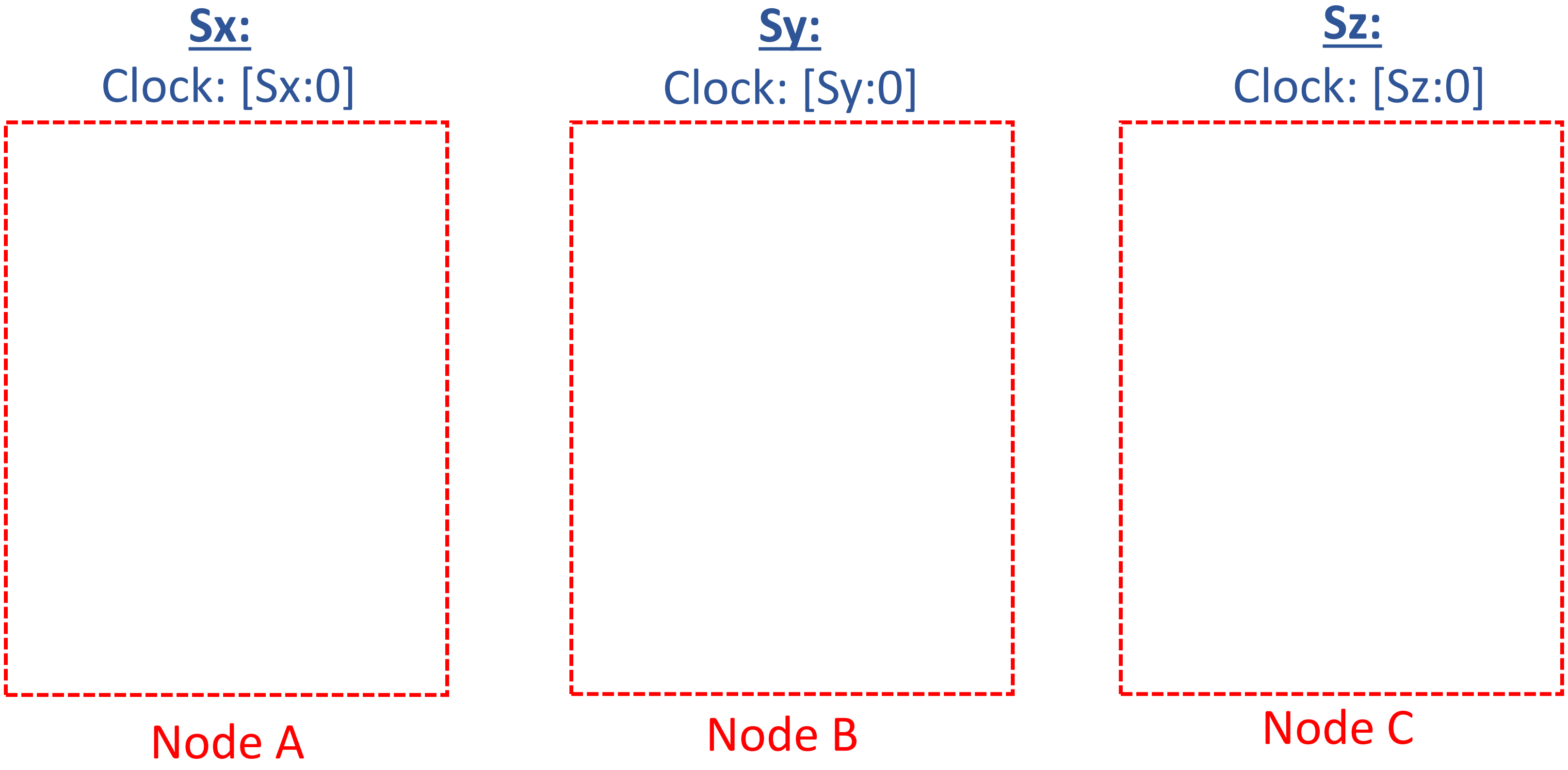


Need to keep track of the "causal" relationship:
i.e., knowing x and y are updated by the same users/servers

Versioned Data

- Keeping multiple versions until being resolved by a server

Vectored clock: a virtual clock advanced at every put() to keep track of data dependencies.



Sx:
Clock: [Sx:0]

Sy:
Clock: [Sy:0]

Sz:
Clock: [Sz:0]

Node A

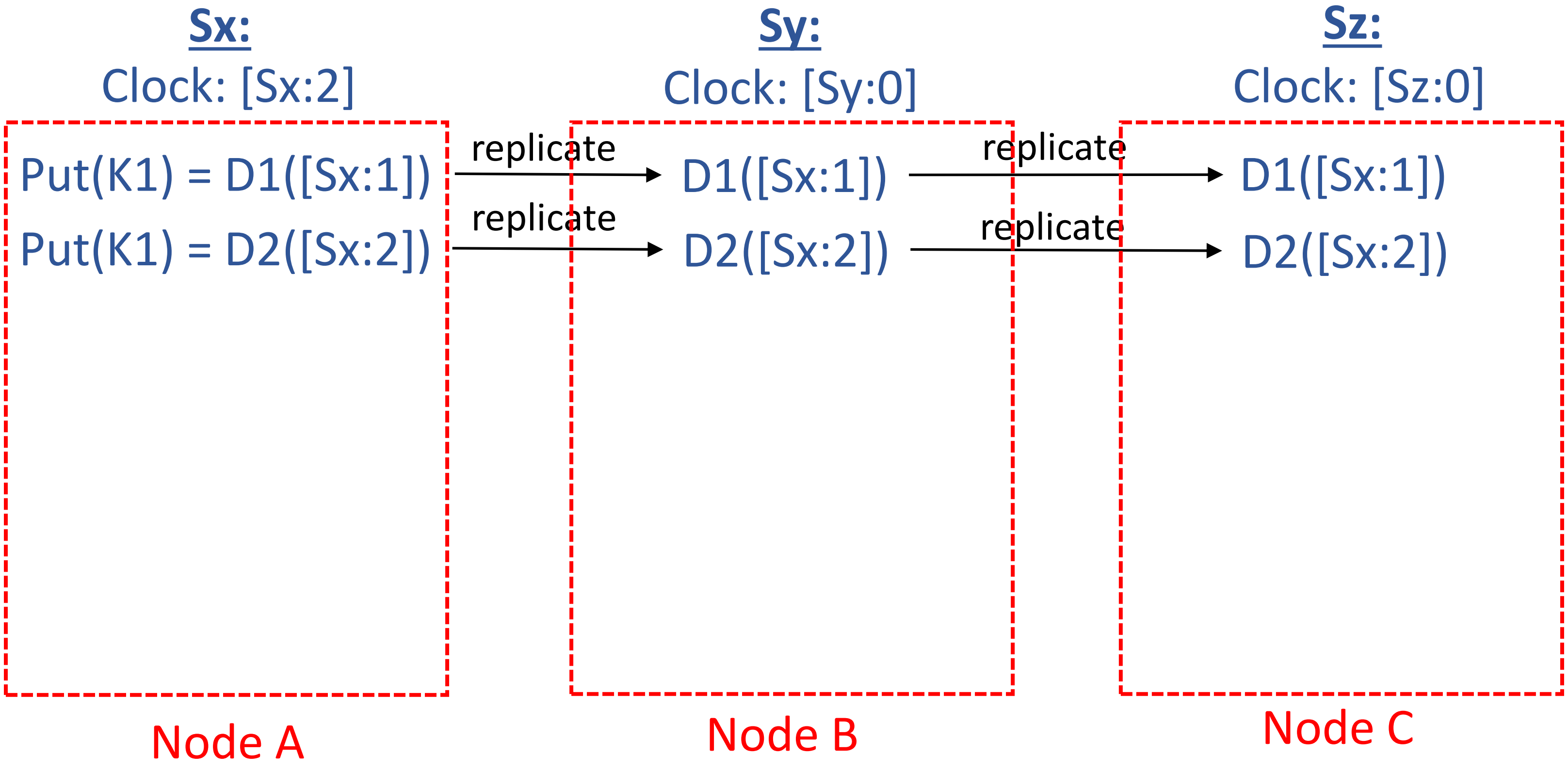
Node B

Node C

Versioned Data

- Keeping multiple versions until being resolved by a server

Vectored clock: a virtual clock advanced at every put() to keep track of data dependencies.



Sx:
Clock: [Sx:2]

Put(K1) = D1([Sx:1])
Put(K1) = D2([Sx:2])

Node A

replicate
replicate

Sy:
Clock: [Sy:0]

D1([Sx:1])
D2([Sx:2])

Node B

replicate
replicate

Sz:
Clock: [Sz:0]

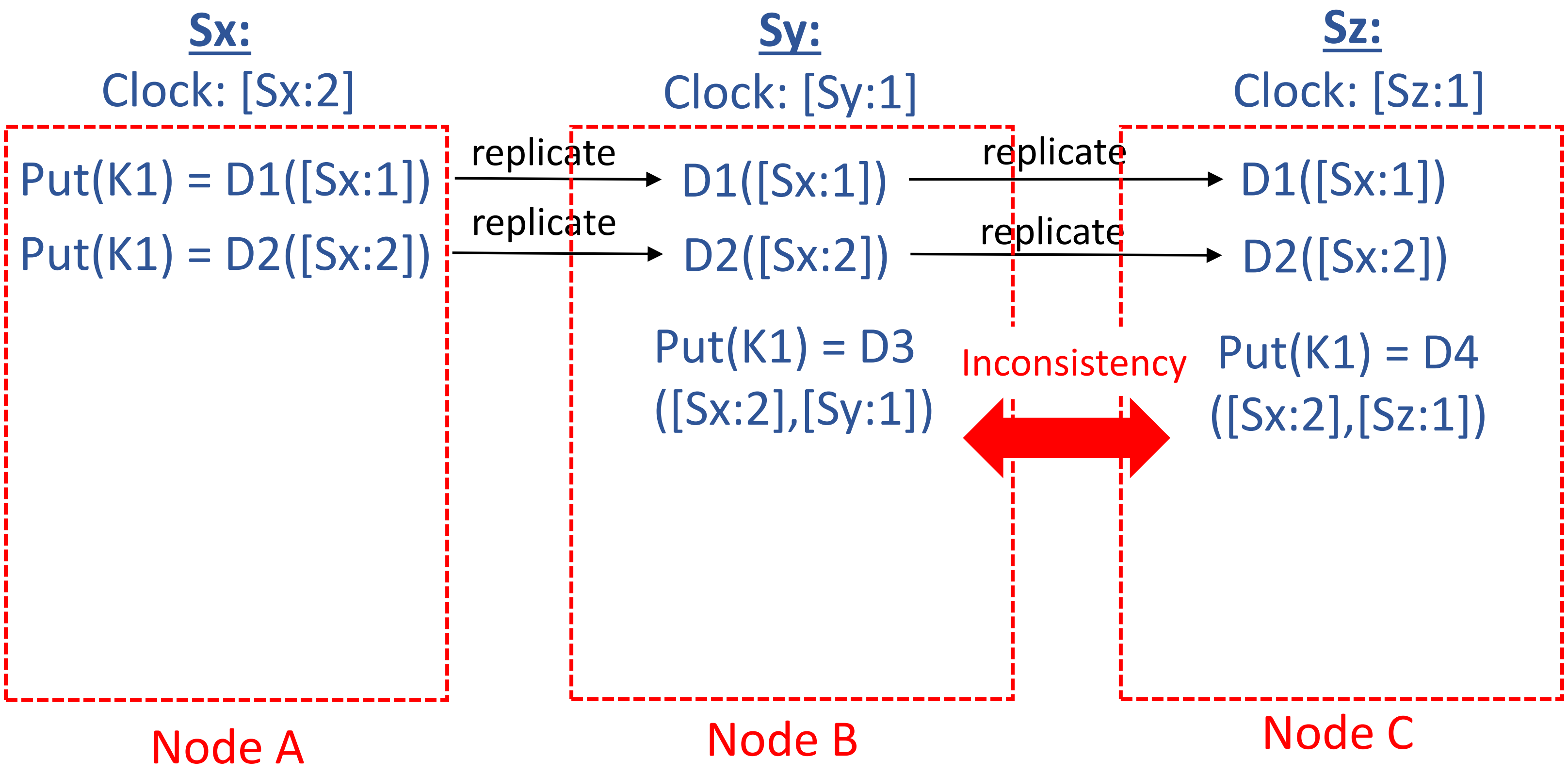
D1([Sx:1])
D2([Sx:2])

Node C

Versioned Data

- Keeping multiple versions until being resolved by a server

Vectored clock: a virtual clock advanced at every put() to keep track of data dependencies.



Sx:
Clock: [Sx:2]

Put(K1) = D1([Sx:1])
Put(K1) = D2([Sx:2])

Node A

Sy:
Clock: [Sy:1]

D1([Sx:1])
D2([Sx:2])
Put(K1) = D3
([Sx:2],[Sy:1])

Node B

Sz:
Clock: [Sz:1]

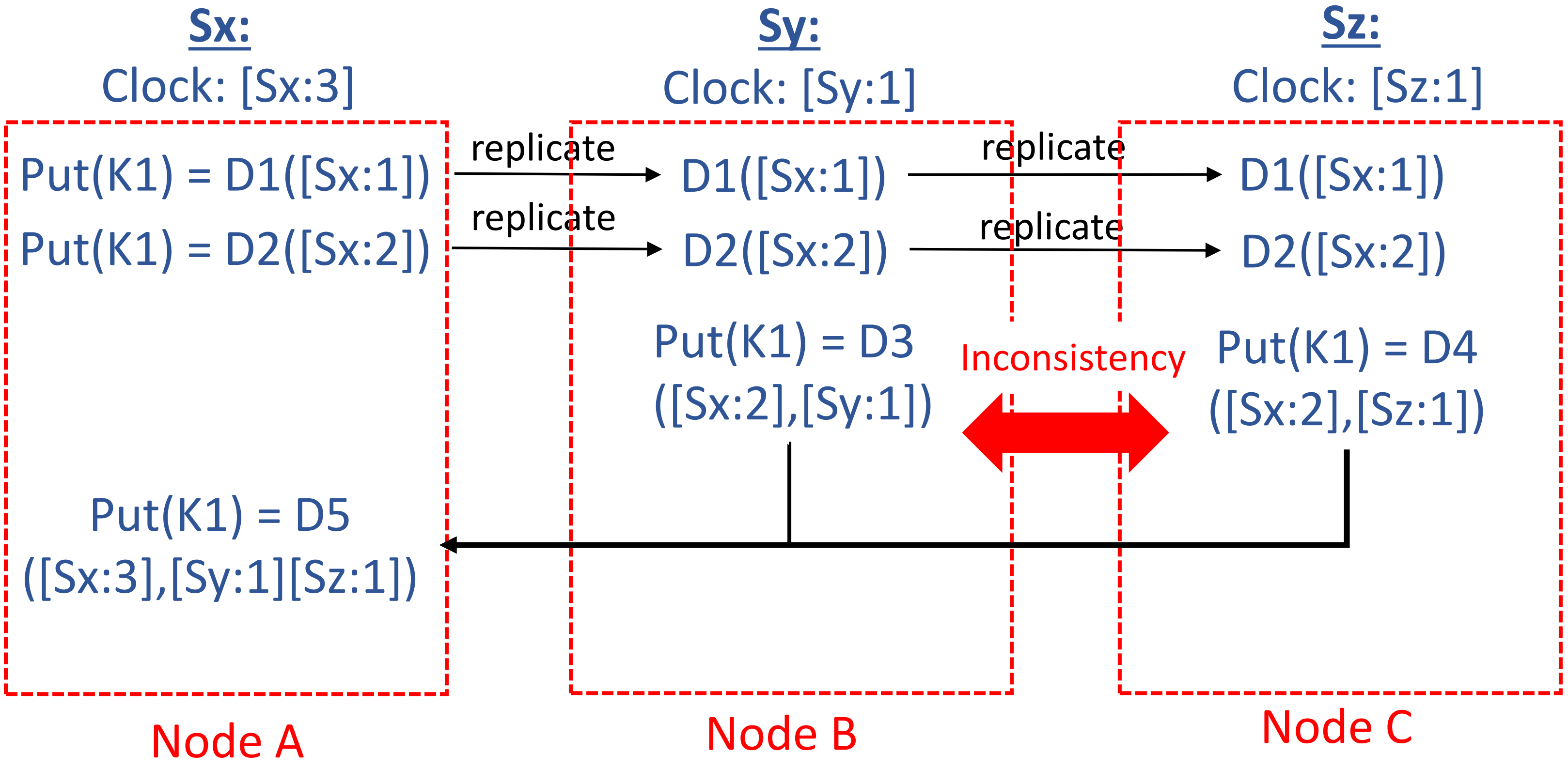
D1([Sx:1])
D2([Sx:2])
Put(K1) = D4
([Sx:2],[Sz:1])

Node C

Versioned Data

- Keeping multiple versions until being resolved by a server

Vectored clock: a virtual clock advanced at every put() to keep track of data dependencies.



Sx:
Clock: [Sx:3]

Sy:
Clock: [Sy:1]

Sz:
Clock: [Sz:1]

Put(K1) = D1([Sx:1])

Put(K1) = D2([Sx:2])

replicate

replicate

D1([Sx:1])

D2([Sx:2])

replicate

replicate

D1([Sx:1])

D2([Sx:2])

Put(K1) = D3 ([Sx:2],[Sy:1])

Inconsistency

Put(K1) = D4 ([Sx:2],[Sz:1])

Put(K1) = D5 ([Sx:3],[Sy:1][Sz:1])

Node A

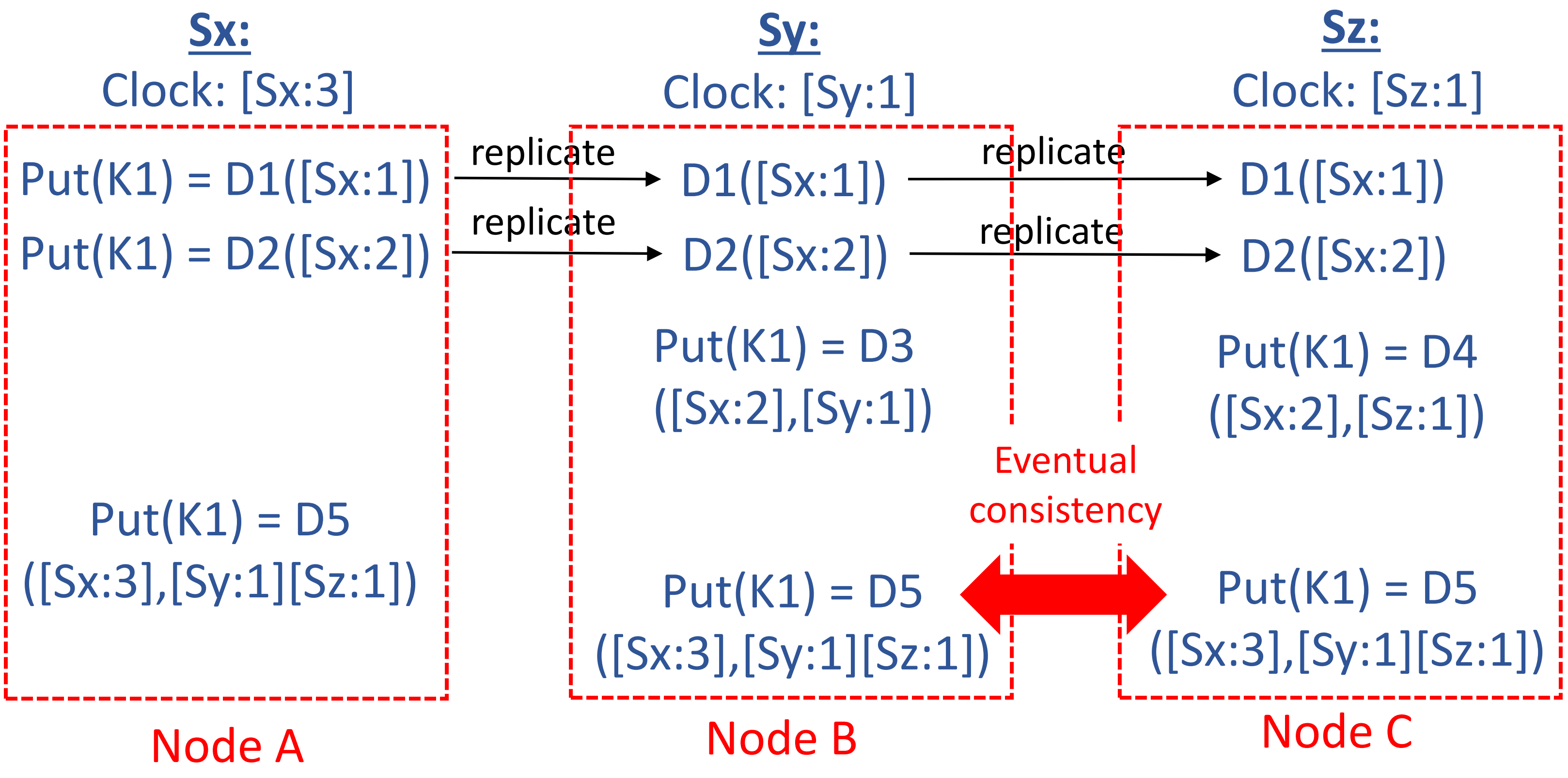
Node B

Node C

Versioned Data

- Keeping multiple versions until being resolved by a server

Vectored clock: a virtual clock advanced at every put() to keep track of data dependencies.



Sx:
Clock: [Sx:3]

Put(K1) = D1([Sx:1])
Put(K1) = D2([Sx:2])

Put(K1) = D5
([Sx:3],[Sy:1][Sz:1])

Node A

replicate
replicate

Sy:
Clock: [Sy:1]

D1([Sx:1])
D2([Sx:2])

Put(K1) = D3
([Sx:2],[Sy:1])

Put(K1) = D5
([Sx:3],[Sy:1][Sz:1])

Node B

replicate
replicate

Eventual consistency

Sz:
Clock: [Sz:1]

D1([Sx:1])
D2([Sx:2])

Put(K1) = D4
([Sx:2],[Sz:1])

Put(K1) = D5
([Sx:3],[Sy:1][Sz:1])

Node C

Dynamo API

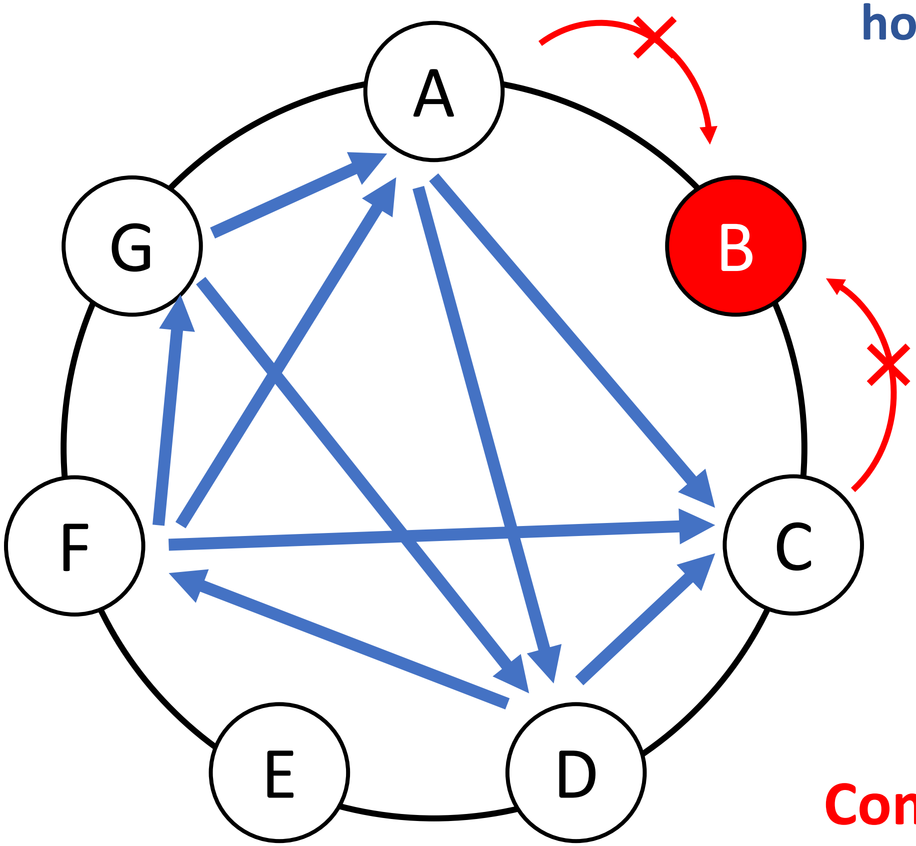
- Get(key) → Value or [(Value, Context)]

  A reader either get() a consistent value,
  or a list of versioned values with respective contexts (clocks).

- Put(Key, Context, Value)

  A writer can decide which context to put() the new value.

Failure Detection



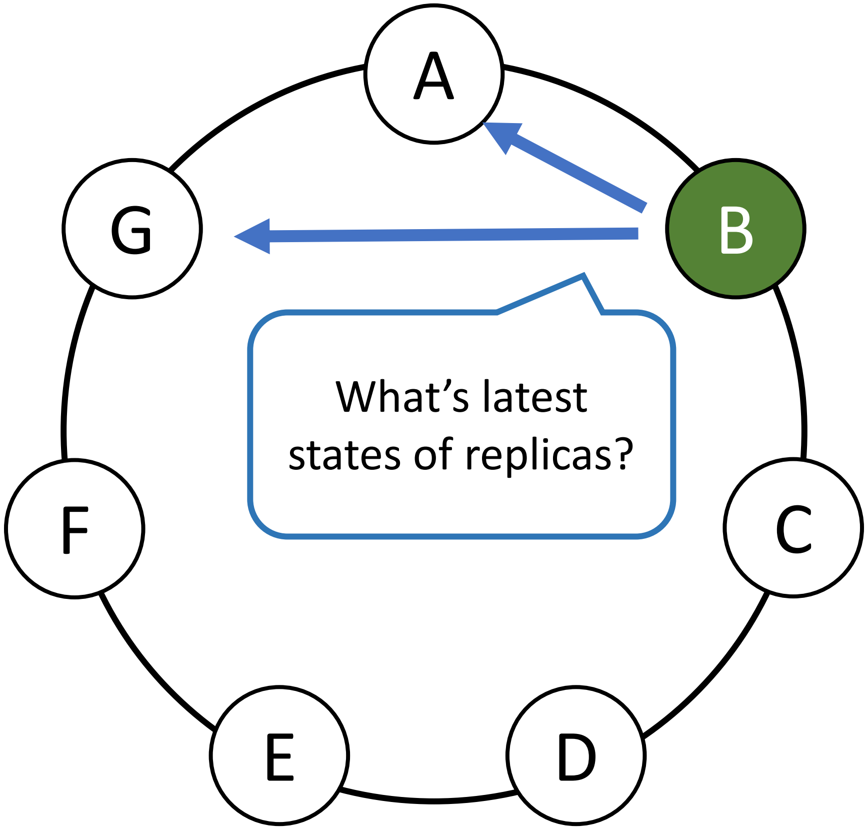
**If a node is offline,
how do other nodes know about it?**

Ans: gossip-based protocol

- Node A and C consider node B offline if receive no response from B
- Each node randomly contacts other node every second to exchange node information.

Completely decentralized failure detection protocol

Recover From Temporary Failures



- If B becomes available again, it needs to synchronize with other replicas.
- Checking the state of replicas can be expensive
  - Sending all the data
    ➔ Too expensive
  - Sending hash of the replica
    ➔ Need rehashing at updates

Summary

- NoSQL simplifies both syntaxes and requirements of distributed storage
  - Open()/Read()/Write() or SQL queries ➔ Get() and Put()
- Dynamo uses classic distributed system techniques to ensure high availability (99.9 SLA)
  - Versioned data with vectored clocks
  - Gossip-based failure detection