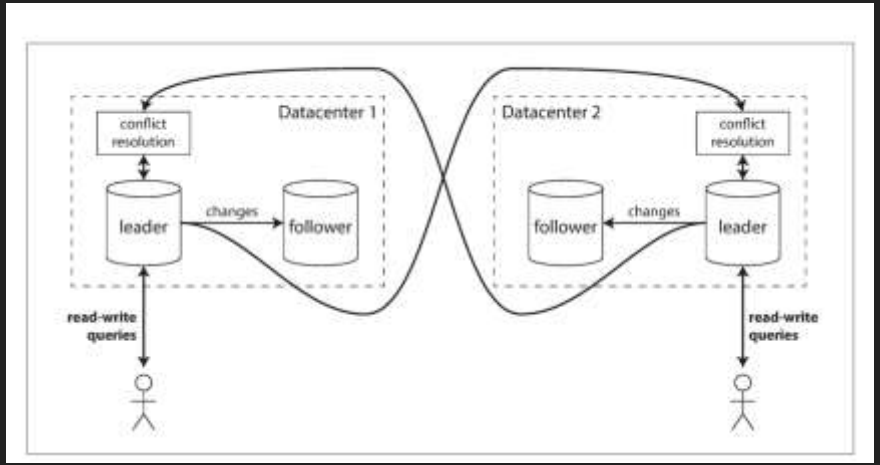
Multi Leader Replication

Types of replication

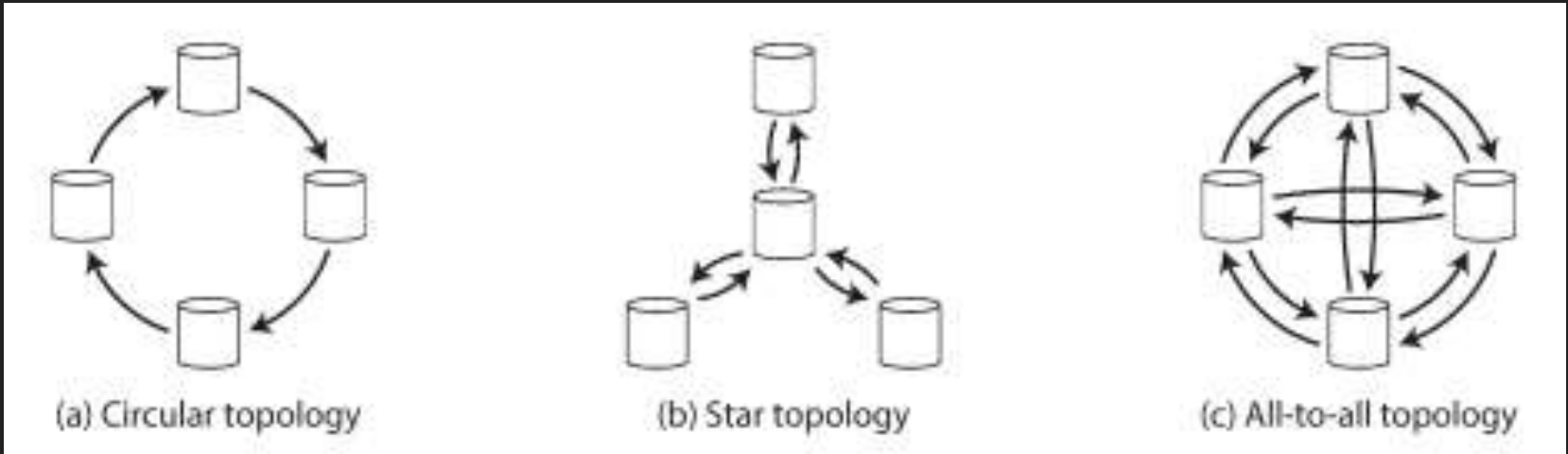
- Single leader
- Multi leader
- Leaderless

Multi Leader Replication Overview

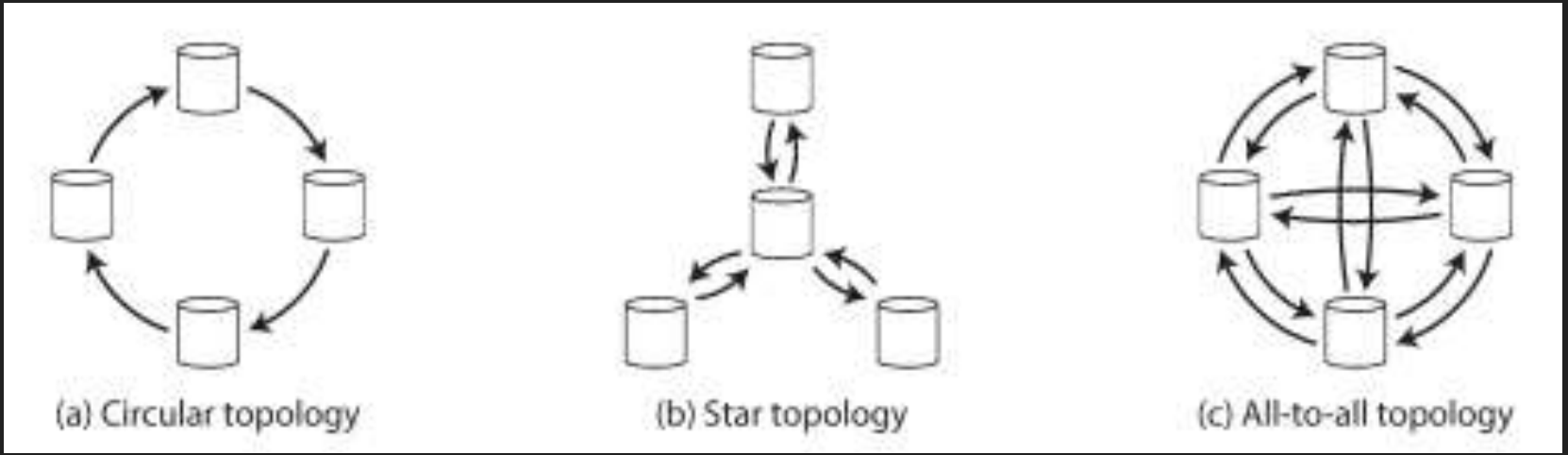
- Multiple leader database, possibly across data centers
- Leader databases send each other writes through some pre-existing topology
- Reads can come from any database



Multi Leader Replication Topologies



Multi Leader Replication Topologies



- Circle and star topologies can easily fail (any node crash for circle, central node crash for star)
- All to all topology can have writes delivered out of order due to race condition

Multi Leader Replication Tradeoffs

Pros:

- Can have a leader in each data center, makes having a global service more manageable (outages between datacenters do not break application)
- Are not limited to the write throughput of a single node

Cons:

- Having to deal with write conflicts between multiple leaders

Conflict Resolution

- Conflict Avoidance
- Last Write Wins
- On read
- On write

Conflict Avoidance

- The easiest way to deal with conflicts is to just avoid them
- Have all writes to the same item go to a given leader
- Not always possible if leader is down or you want to change your database configuration

Last Write Wins

Each write is assigned a timestamp, write with latest timestamp is kept

Pros:

- Very easy to implement

Cons:

- Do we use the client or the server timestamp?
- Writes will be lost
- Clock skew on servers means that their times are not exactly synchronized

On read

- When database detects concurrent/conflicting writes, store both writes
- Return both values to a user on the next read to manually merge the values and store the resulting merged value in the database

On write

- When database detects concurrent writes, call snippet of code to merge them somehow (often application specific)
- This is the idea between conflict free replicated data types

Detecting Concurrent Writes

Two writes are concurrent if each client did not know about the write of the other when they made the write, has nothing to do with actual time of write.

If one client knew about the write of the other, we would have a causality relationship between the two.

Hence, detecting concurrent writes is all about keeping track of what a client has seen from the database before making a write.

Version Vectors

For each key: keep track of an increasing version number for each key for each replica

Client 1

Replica A 

Replica B 

Client 2

Version Vectors

For each key: keep track of an increasing version number for each key for each replica

Client 1 [0, 0] add PS5

Replica A  [1, 0] | [PS5]

Replica B 

Client 2

Version Vectors

For each key: keep track of an increasing version number for each key for each replica

Client 1 [0, 0] add PS5

Replica A  [1, 0] | [PS5]

Replica B  [0, 1] | [xbox]

Client 2 [0, 0] add xbox

Version Vectors

For each key: keep track of an increasing version number for each key for each replica

Client 1 [0, 0] add PS5 [1, 0] add lotion

Replica A  [1, 0] | [PS5] [2, 0] | [PS5, lotion]

Replica B  [0, 1] | [xbox]

Client 2 [0, 0] add xbox

Version Vectors

For each key: keep track of an increasing version number for each key for each replica

Client 1 [0, 0] add PS5 [1, 0] add lotion

Replica A  [1, 0] | [PS5] [2, 0] | [PS5, lotion] [2, 1] | [PS5, lotion], [xbox]

Replica B  [0, 1] | [xbox] [2, 1] | [PS5, lotion], [xbox]

Client 2 [0, 0] add xbox

Version Vector Semantics

- Every time a client reads from a database, the database gives it the version vector of a key
- Every time a client writes to a database, it passes the database its most recently read version vector for a key and the database supplies it with a new one
- Two values have a happens-before relationship if one version vector is strictly greater than the other (for each element of the list, the number is greater than or equal to the corresponding element of the other list)
- All other version vectors are concurrent
    - The corresponding values can either be merged somehow or kept as “siblings” in the database
    - Merging version vectors means taking the max of both version vectors at every index

Multi Leader Replication Tradeoffs

Pros:

- Can have a leader in each data center, makes having a global service more manageable (outages between datacenters do not break application)
- Are not limited to the write throughput of a single node

Cons:

- Having to deal with write conflicts between multiple leaders