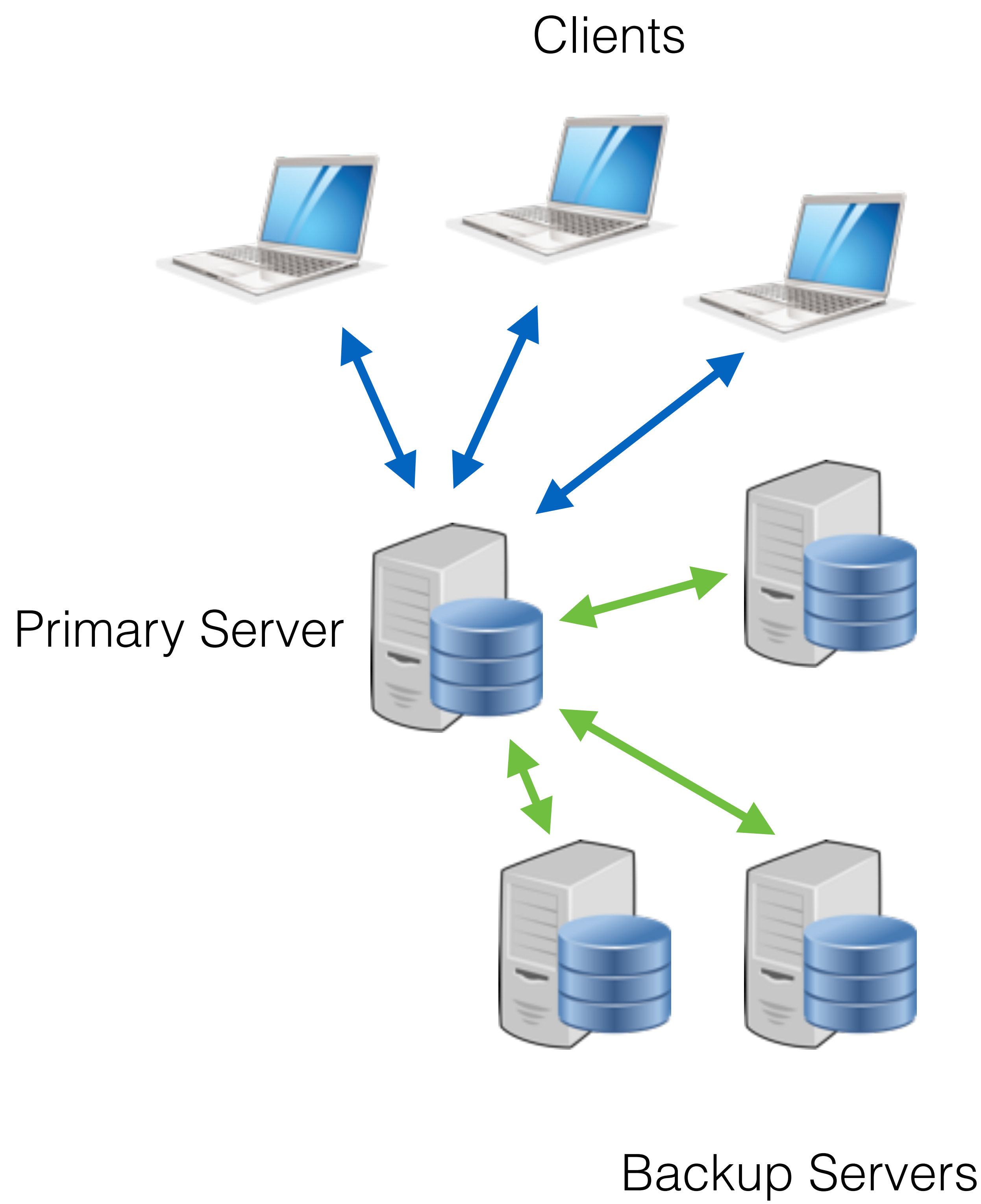
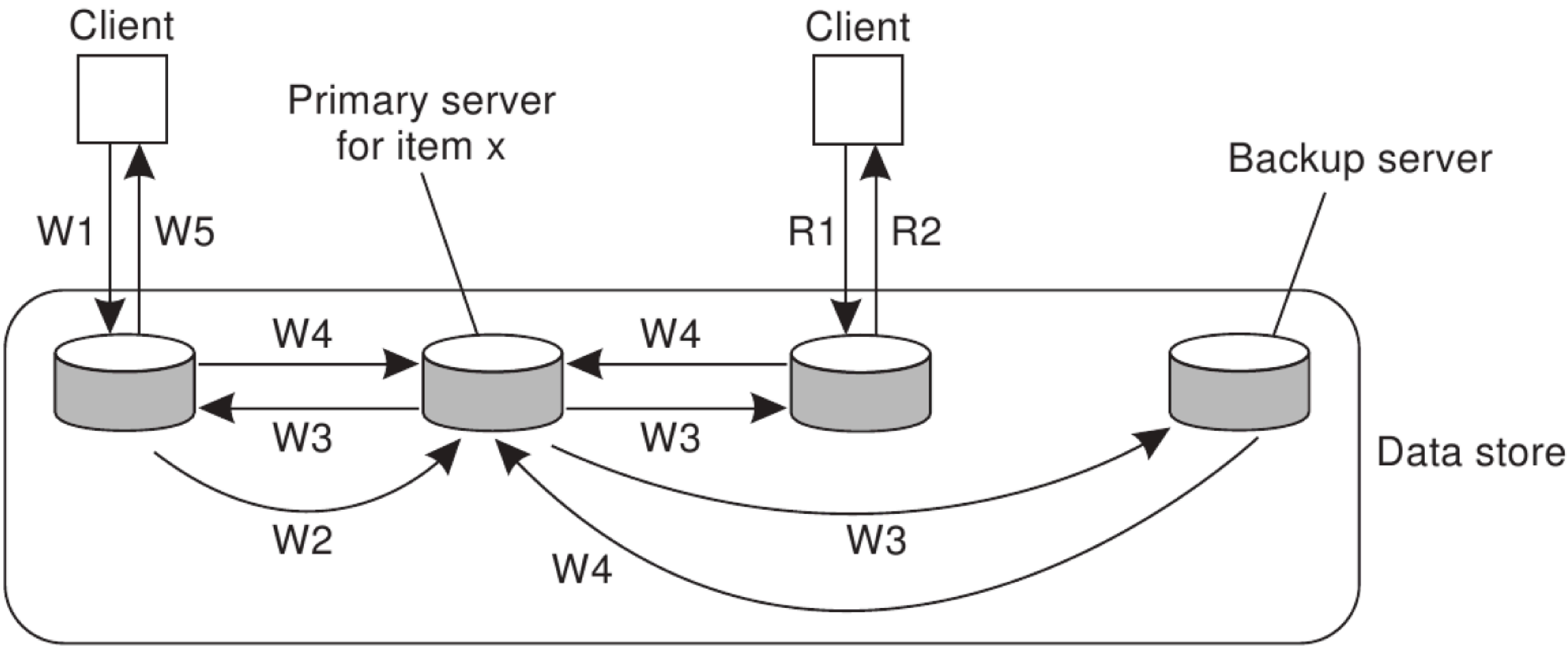
Passive Replication

- Clients communicate with primary server
- WRITES are atomically forwarded from primary server to backup servers
- READS are replied by the primary server
- Also known as Primary Copy (or Backup) Replication
- Specifications:
  - At most one replica can be the primary server at any time.
  - Each client maintains a variable L (leader) that specifies the replica to which it will send requests. Requests are queued at the primary server.
  - Backup servers ignore client requests.



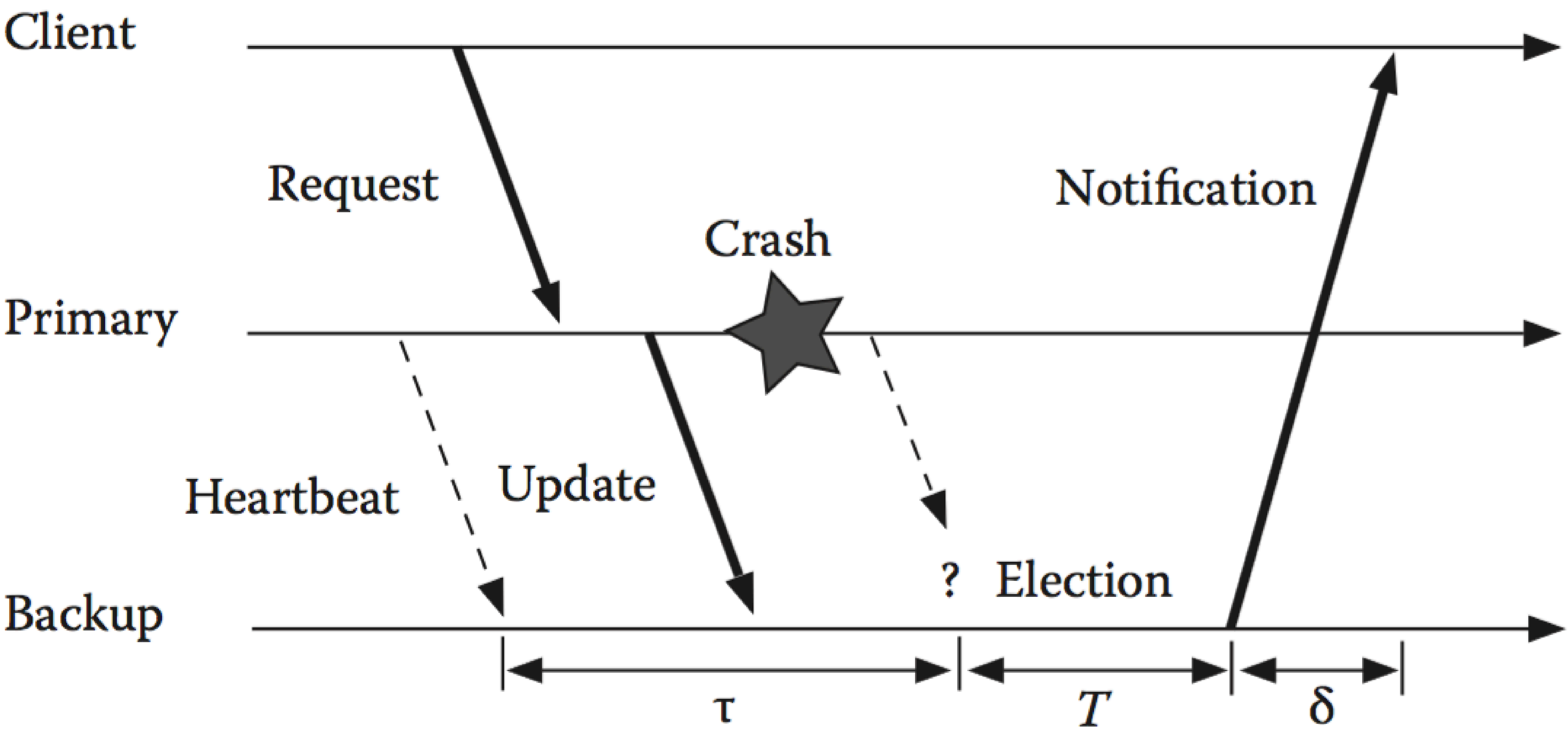
Passive Replication Protocol



W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

Passive Replication Protocol



Implementing Primary Backup

- Clients communicate with primary server
- The primary server updates the backup servers
- Backup servers detect the failure of the primary using a heartbeat mechanism
- Clients learn from the service when the primary server fails and the service “fails over” to a backup

Simple Protocol

- System model:
  - point-to-point communication
  - no communication failures → no network partitions
  - upper bound on message delivery time → synchronous communications
  - FIFO channels
  - at most one server crashes
- Two servers:
  - The primary p_1
  - The backup p_2
- Variables:
  - At *server* p_i, primary = true if p_i acts as the current primary
  - At *clients*, primary is equal to the identifier of the current primary

Simple Protocol

Protocol executed by the primary p_1

upon initialization **do**
- *primary* $\leftarrow$ **true**

upon receive $\langle \text{REQ}, r \rangle$ **from** c **do**
- *state* $\leftarrow$ update(*state*, r) % Update local state
- **send** $\langle \text{STATE}, state \rangle$ **to** p_2 % Send update to backup
- **send** $\langle \text{REP}, \text{reply}(r) \rangle$ **to** c % Reply to client

repeat every τ seconds
- **send** $\langle \text{HB} \rangle$ **to** p_2 % Heartbeat message

upon recovery after a failure **do**
- { start behaving like a backup }

Simple Protocol

Protocol executed by the backup p_2

upon initialization **do**
  $primary \leftarrow$ **false**

upon receive $\langle$STATE, $s\rangle$ **do**
  $state \leftarrow s$ % Update local state

upon not receiving a heartbeat for $\tau + \delta$ seconds **do**
  $primary \leftarrow$ **true** % Becomes new primary
  send $\langle$NEWP$\rangle$ **to** c % Inform the client of new primary
  { start behaving like a primary }

Simple Protocol

Protocol executed by client c

---

upon initialization **do**

  $primary \leftarrow p_1$ % Initial primary

upon receive $\langle$NEWP$\rangle$ **from** p_2 **do**

  $primary \leftarrow p_2$ % Backup

upon operation(r) **do**

  while not received a reply **do**

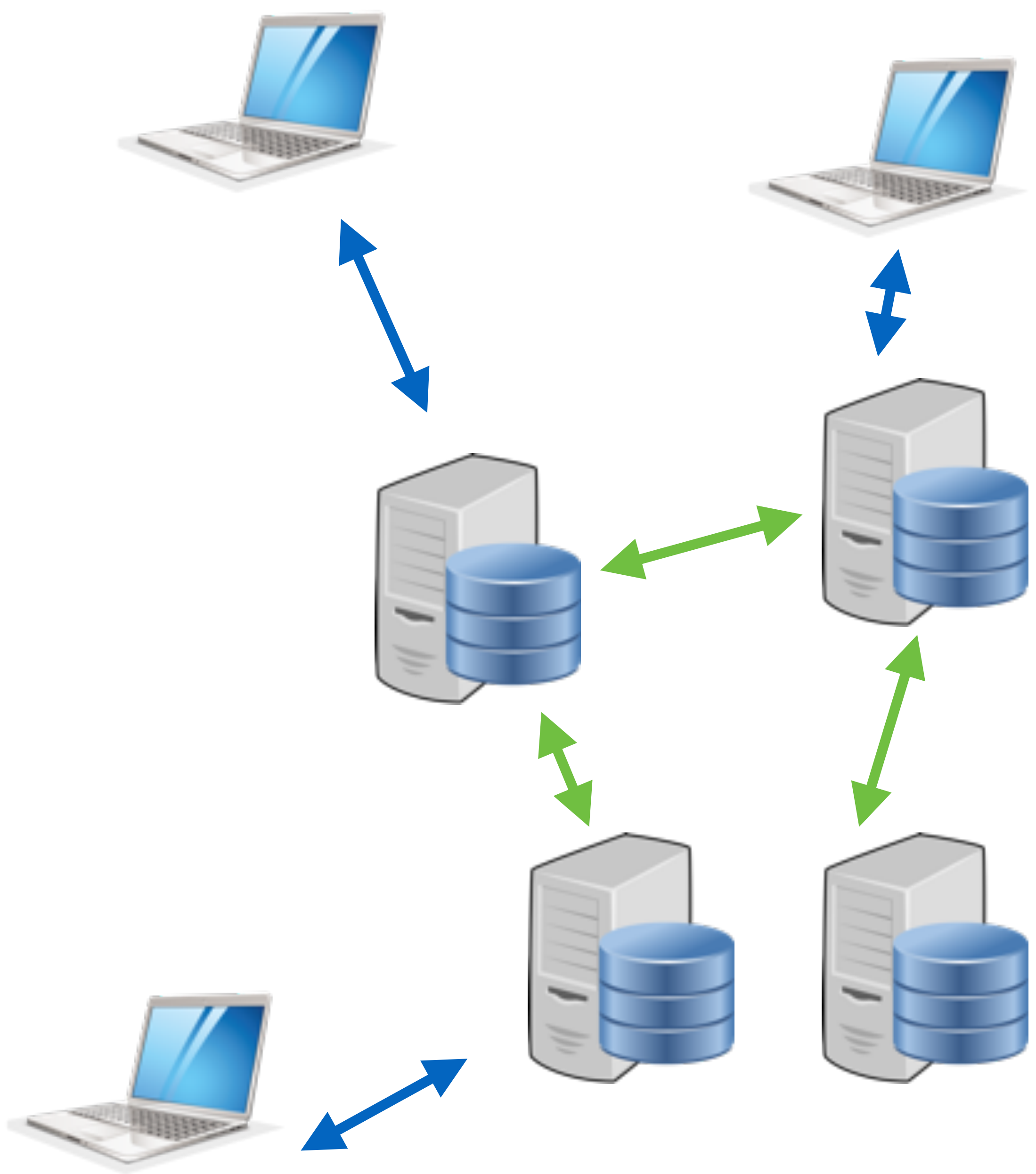
    send $\langle$REQ, $r\rangle$ **to** $primary$

    wait receive $\langle$REP, $v\rangle$ **or receive** $\langle$NEWP$\rangle$

  return v

Active Replication

- Clients communicate with several/all servers
- Every server handles any operation and sends the response
- WRITES must be applied in the same order (**total order broadcast**)
- One way to implement totally-ordered multicast is to use logical clocks
- Another solution is to use a **centralized sequencer**
    - Each write is forwarded to the sequencer
    - The sequencer assigns a unique sequence number to the WRITE and forwards the WRITE to all replicas
    - Each replica carries out the WRITES in the order of their sequence number



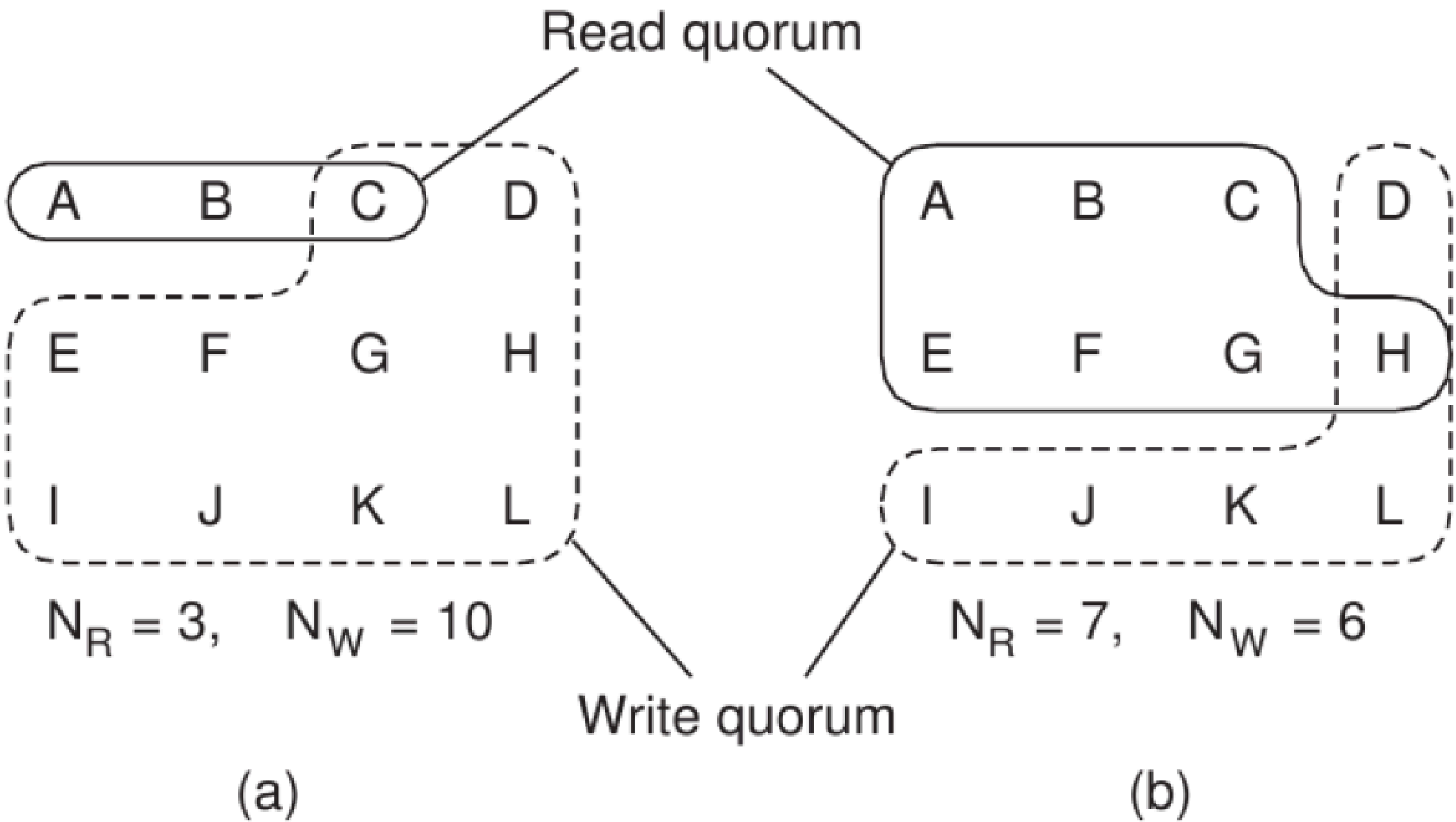
Quorum Protocols

- Proposed by Gifford in 1979
- **Quorum-based protocols** guarantee that each operation is carried out in such a way that a *majority vote* (a quorum) is established.
  - *Write quorum W*: the number of replicas that need to acknowledge the receipt of the update to complete the update
  - *Read quorum R*: the number of replicas that are contacted when a data object is accessed through a read operation

Quorum Systems

- Formally, a **quorum system** $S = \{S_1, \ldots, S_N\}$ is a collection of **quorum sets** $S_i \subseteq U$ such that two quorum sets have at least an element in common
- For replication, we consider two quorum sets, a **read quorum** R and a **write quorum** W
- **Rules**:
    1. Any read quorum must overlap with any write quorum
    2. Any two write quorums must overlap
- U is the set of replicas, i.e., $|U| = N$

Quorum Examples



Quorum Examples

- Read rule: $|R| + |W| > N \Rightarrow$ read and write quorums overlap
- Write rule: $2\,|W| > N \Rightarrow$ two write quorums overlap
- The quorum sizes determine the costs for read and write operations
- Minimum quorum sizes for are

$$\min|W| = \left\lfloor \frac{N}{2} \right\rfloor + 1 \qquad \min|R| = \left\lceil \frac{N}{2} \right\rceil$$

  - Write quorums requires majority
  - Read quorum requires at least half of the nodes
- ROWA (R,W,N) = (N = N, R = 1, W = N)
- Amazon’s Dynamo (N = 3, R = 2, W = 2)
- Linkedin's Voldemort (N = 2 or 3, R = 1, W = 1 default)
- Apache's Cassandra (N = 3, R = 1, W = 1 default)

Client-centric Consistency Models

- Each WRITE operation is assigned a unique identifier
    - Done by the server where the operation is requested
- For each client c, we keep track of:
    - Read set WS_R : contains write operations relevant to the read operations performed by c
    - Write set WS_W : contains write operations relevant to the write operations performed by c
- For each server, we keep track of:
    - Write set WS : contains the write operations executed so far

Read-Your-Write Implementation

- To perform a READ:
  - A client
    - sends READ and its WS_W to a server S.
  - The server S:
    - Checks if the $WS_W \subseteq WS$, i.e., all the WRITES seen from the client have been applied by the server
    - If not, asks the other servers the missing WRITES
    - Applies the missing WRITES locally and update its WS
    - Return the requested value to the client

Read-Your-Write Implementation

- To perform a WRITE:
  - A client
    - sends WRITE and adds it to its WS_W
  - The server S:
    - Perform the WRITE
    - adds it to its WS

Monotonic-Read Implementation

- To perform a READ:
  - A client
    - sends READ and its WS_R to a server S.
  - The server S:
    - Checks if the $WS_R \subseteq WS$, i.e., all the WRITES seen from the client have been applied by the server
    - If not, asks the other servers the missing WRITES
    - Applies the missing WRITES locally and update its WS
    - Return the requested value and WS to the client
  - The client
    - adds WS to its WS_R

Monotonic-Read Implementation

- To perform a WRITE:
  - A client
    - sends WRITE
  - The server S:
    - Perform the WRITE
    - adds it to its WS

Writes-Follow-Reads & Monotonic-Writes

- Two additional constraints on the server:
  - When a server S accepts a new WRITE W_2 at time t, it ensures that WriteOrder(W_1,W_2) is true for any WRITE W_1 already in DB(S,t).
  - Anti-entropy is performed such that if WRITE W_2 is propagated from server S_1 to server S_2 at time t then any W_1 in DB(S_1,t) such that WriteOrder(W_1,W_2) is also propagated to S_2.

References

- Ghosh, *Distributed Systems - An Algorithmic Approach* (2nd ed), chapter 16
- D. Terry et al., *Session Guarantees for Weakly Consistent Replicated Data*, https://www.cis.upenn.edu/~bcpierce/courses/dd/papers/SessionGuaranteesPDIS.ps
- D. Terry, *Replicated Data Consistency Explained Through Baseball*, http://research.microsoft.com/pubs/157411/ConsistencyAndBaseballReport.pdf