# Replication

- Part 1: Remote write and local write protocols

- Part 2: Quorum-based protocols

Implementation Issues

- Two techniques to implement consistency models
  - Primary-based protocols
    - Assume a primary replica for each data item
    - Primary responsible for coordinating all writes
  - Replicated write protocols
    - No primary is assumed for a data item
    - Writes can take place at any replica
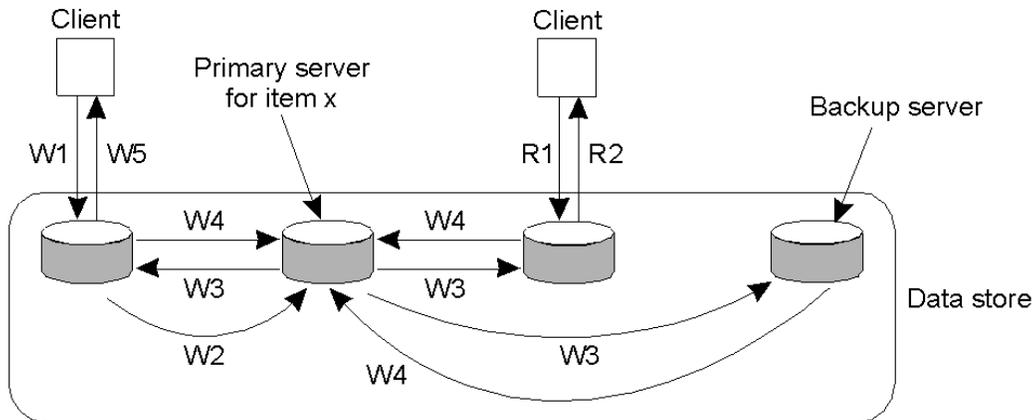
Remote-Write Protocols
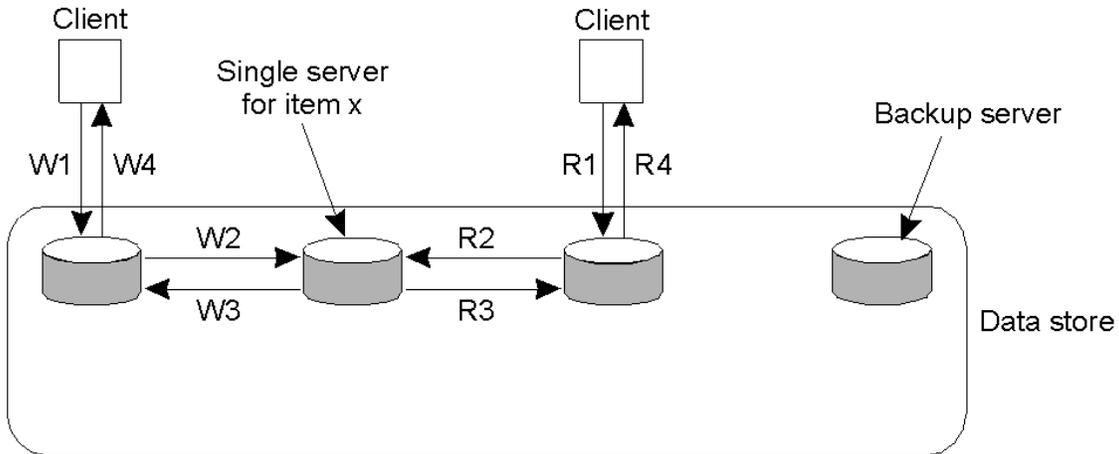
W1. Write request
W2. Forward request to server for x
W3. Acknowledge write completed
W4. Acknowledge write completed

R1. Read request
R2. Forward request to server for x
R3. Return response
R4. Return response

- **Traditionally used in client-server systems (no replication)**
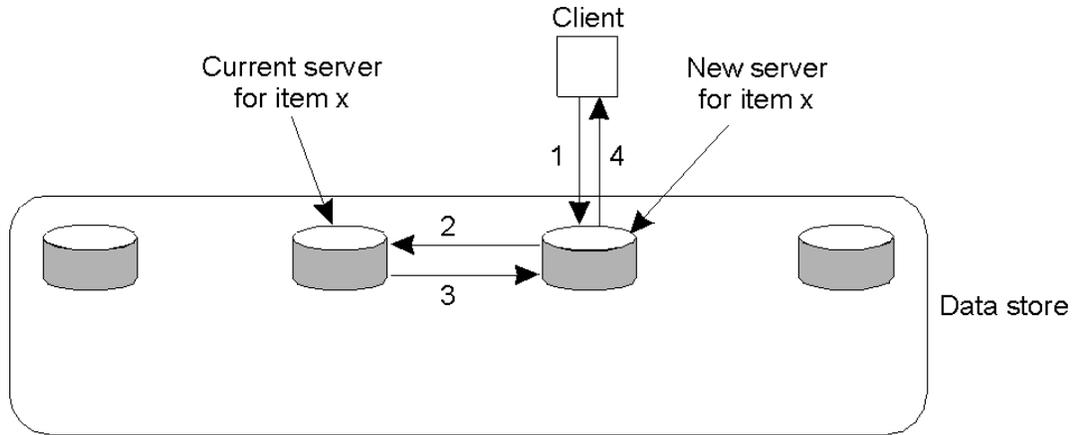
Remote-Write Protocols (2)

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

- Primary-backup protocol (1 prim, 3backup)
  - Allow local reads, sent writes to primary
  - Block on write until all replicas are notified
 - Implements sequential consistency

Local-Write Protocols (1)

1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server

- Primary-based local-write protocol in which a single copy is migrated between processes.
  - Limitation: need to track the primary for each data item

Local-Write Protocols (2)
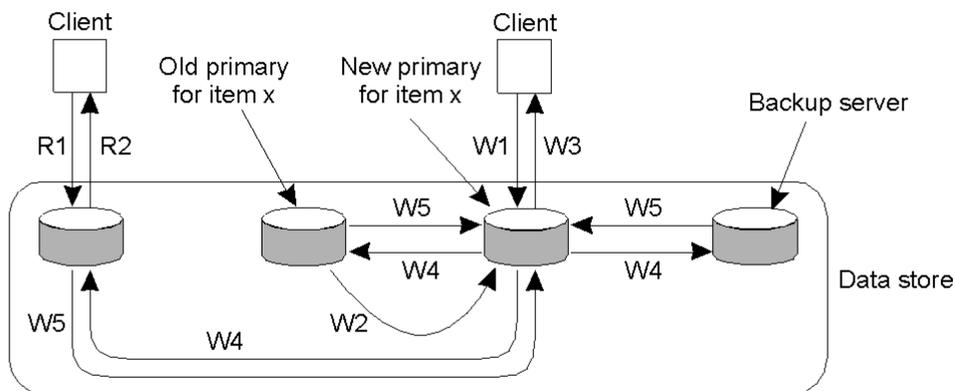
W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

- Primary-backup protocol in which the primary migrates to the process wanting to perform an update
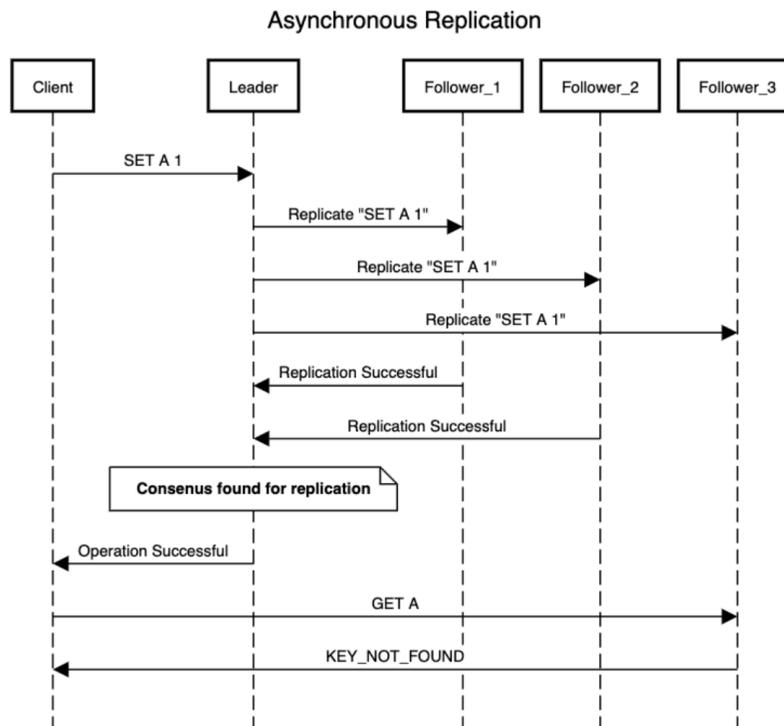
Replicated-write Protocols

- Relax the assumption of one primary
  - No primary, any replica is allowed to update
  - Consistency is more complex to achieve

- Synchronous writes to all replicas

- Asynchronous writes to all replicas

-

Synchronous Replication

Synchronous Replication

Asynchronous Replication
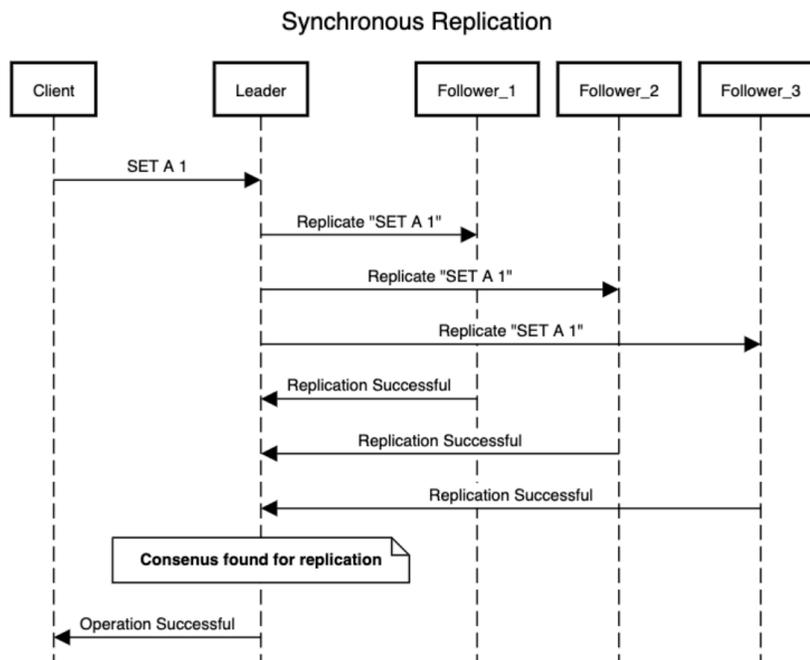
Asynchronous Replication

Replicated-write Protocols

- Relax the assumption of one primary ("leaderless")
  - No primary, any replica is allowed to update
  - Consistency is more complex to achieve
- Quorum-based protocols
  - Use voting to request/acquire permissions from replicas
  - Consider a file replicated on N servers
    - $N_R + N_W > N$ $N_W > N/2$
  - Update: contact N_W servers and get them to agree to do update (associate version number with file)
  - Read: contact N_R and obtain version number
    - If all servers agree on a version number, read

Gifford's Quorum-Based Protocol

Read quorum

A	B	C	D
E	F	G	H
I	J	K	L

$N_R = 3$, $N_W = 10$

Write quorum

(a)

A	B	C	D
E	F	G	H
I	J	K	L

$N_R = 7$, $N_W = 6$

(b)

A	B	C	D
E	F	G	H
I	J	K	L

$N_R = 1$, $N_W = 12$

(c)
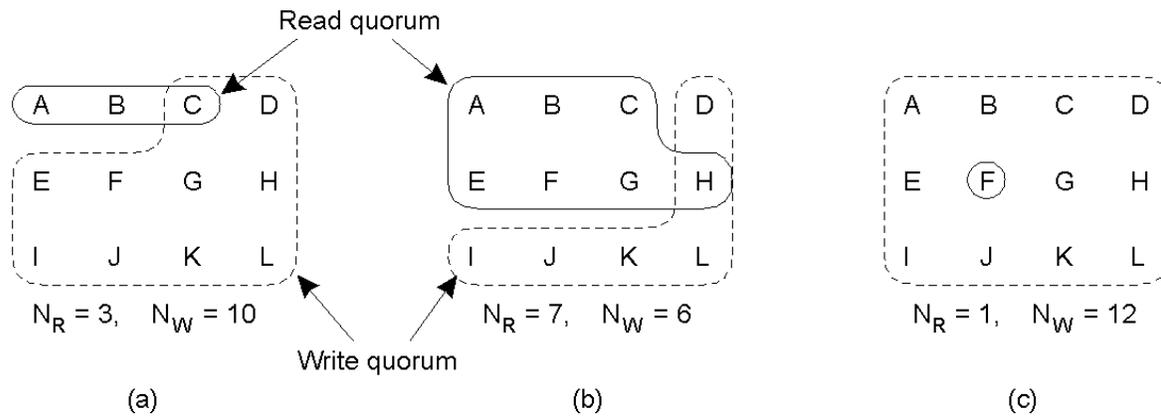
- Three examples of the voting algorithm:
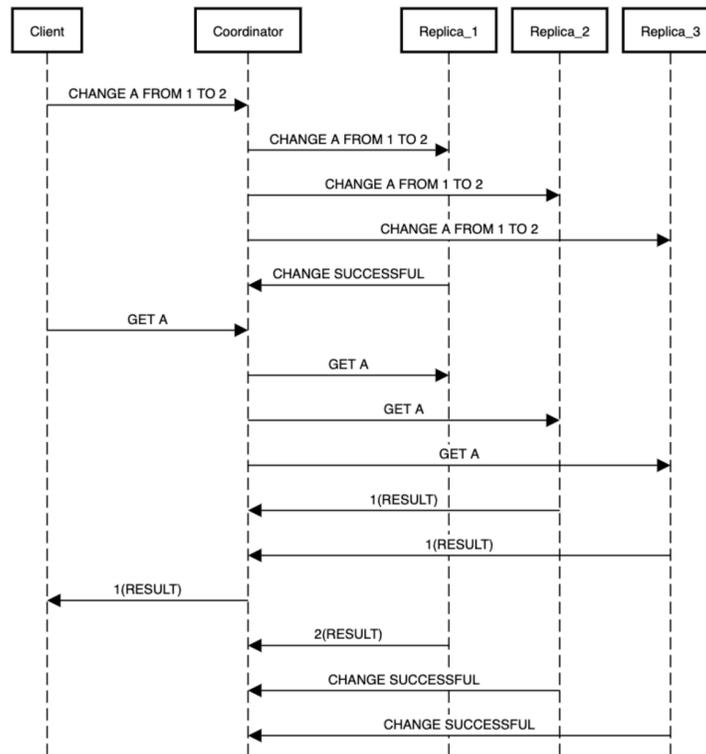a) A correct choice of read and write set
b) A choice that may lead to write-write conflicts
c) A correct choice, known as ROWA (read one, write all)

Quorums In Action

Read operation in Leaderless replication

Client	Coordinator	Node_1	Node_2	Node_3

GET A

GET A

GET A

GET A

A = 1 (Version_1)

A = 2 (Version_2)

A = 2 (Version_2)

Result(A = 2)
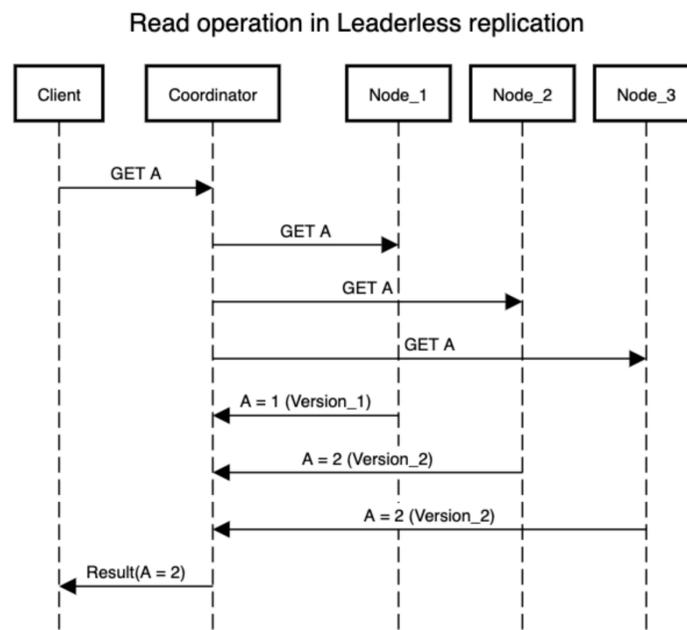
Quorums in Action

Replica Management

- Replica server placement
  - Web: geophically skewed request patterns
  - Where to place a proxy?
    - K-clusters algorithm
- Permanent replicas versus temporary
  - Mirroring: all replicas mirror the same content
  - Proxy server: on demand replication

- Server-initiated versus client-initiated

Final Thoughts

- Replication and caching improve performance in distributed systems
- Consistency of replicated data is crucial
- Many consistency semantics (models) possible
  - Need to pick appropriate model depending on the application
  - Example: web caching: weak consistency is OK since humans are tolerant to stale information (can reload browser)
  - Implementation overheads and complexity grows if stronger guarantees are desired

Fault Tolerance

- Single machine systems
  - Failures are all or nothing
    - OS crash, disk failures
- Distributed systems: multiple independent nodes
  - Partial failures are also possible (some nodes fail)
- *Question:* Can we automatically recover from partial failures?
  - Important issue since probability of failure grows with number of independent components (nodes) in the systems
  - Prob(failure) = Prob(Any one component fails)=1-P(no failure)

A Perspective

- Computing systems are not very reliable
  - OS crashes frequently (Windows), buggy software, unreliable hardware, software/hardware incompatibilities
  - Until recently: computer users were "tech savvy"
    - Could depend on users to reboot, troubleshoot problems
  - Growing popularity of Internet/World Wide Web
    - "Novice" users
    - Need to build more reliable/dependable systems
  - Example: what is your TV (or car) broke down every day?
    - Users don't want to "restart" TV or fix it (by opening it up)
- Need to make computing systems more reliable
  - Important for online banking, e-commerce, online trading, webmail…

Basic Concepts

- Need to build *dependable* systems
- Requirements for dependable systems
  - Availability: system should be available for use at any given time
    - 99.999 % availability (five 9s) => very small down times
  - Reliability: system should run continuously without failure
  - Safety: temporary failures should not result in a catastrophic
    - Example: computing systems controlling an airplane, nuclear reactor
  - Maintainability: a failed system should be easy to repair
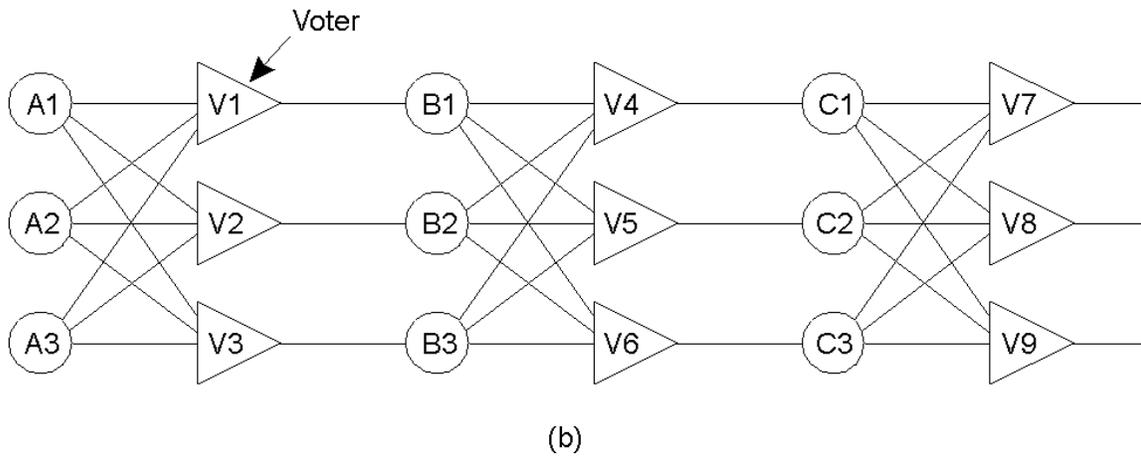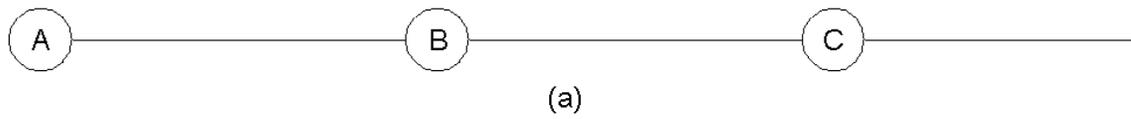
Basic Concepts (contd)

- Fault tolerance: system should provide services despite faults
    - Transient faults
    - Intermittent faults
    - Permanent faults

Failure Models

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	The server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

- Different types of failures.

Failure Masking by Redundancy

A ——————————— B ——————————— C

(a)

Voter

A1 —— V1 ——— B1 —— V4 ——— C1 —— V7

A2 —— V2 ——— B2 —— V5 ——— C2 —— V8

A3 —— V3 ——— B3 —— V6 ——— C3 —— V9

(b)

- Triple modular redundancy.

CS677: Distributed and Operating Systems   Lecture 17, page 21