

UNIT 7

Consistency and Replication

Distributed Systems (CAC352) — BCA III/VI

⚠ Please note

*These notes were prepared with AI assistance and **may contain errors or omissions**. Please cross-check definitions, algorithms and numerical examples against your prescribed textbook (e.g., Tanenbaum & van Steen, *Distributed Systems*), your class notes and your instructor before relying on them for exams.*

7.1 Introduction

Replication means keeping copies of data (or services) on multiple machines. There are two main reasons: **reliability** (if one replica fails, another can be used; corrupted copies can be detected by comparing) and **performance** (load is shared and copies can be placed close to clients — scaling in size and geography). The price is **consistency**: when one copy is updated, the other copies become stale and must be updated too, which costs bandwidth and coordination. In the worst case each update needs global synchronisation, which limits scalability. The solution is to **relax consistency** — accept that copies may differ for a while — provided applications can tolerate it.

Consistency models

A **consistency model** is a contract between the data store and its processes: if the processes obey certain rules, the store promises to behave in a defined way. The stronger the model, the easier to program but the more expensive it is.

Continuous consistency (conits)

Yu and Vahdat defined a **conit** (consistency unit) and three axes of allowed deviation between replicas: **numerical deviation** (how many/what value of updates are unseen), **staleness deviation** (how old the copy may be) and **ordering deviation** (how much the order of tentative updates may differ). Applications pick bounds on each axis, e.g., a stock ticker may be stale by at most 5 seconds.

7.2 Data-Centric Consistency Models

These models describe the view of the whole data store as seen by all processes. Notation: $W_i(x)a$ = process i writes value a to item x ; $R_i(x)b$ = process i reads b from x .

Model	Guarantee	Remarks
Strict	Every read returns the value of the most recent write in absolute global time	Impossible in a distributed system (no global time)
Linearizability	Operations appear in some sequential order consistent with real-time (timestamp) order	Expensive; used for critical data
Sequential	Result is as if all operations were executed in one sequential order, and each process's operations appear in program order	Popular; all processes see the same interleaving; no real-time constraint
Causal	Writes that are causally related must be seen by all processes in the same order; concurrent writes may be seen in different orders	Weaker; cheaper
FIFO (PRAM)	Writes by one process are seen by everyone in the order issued; writes of different processes may be seen in different orders	Easy to implement
Weak / Release / Entry	Consistency is enforced only at synchronisation points (synchronisation variables, acquire/release)	Fine-grained control; used in distributed shared memory

Example 7.1 — Sequential consistency

P1: W(x)a. P2: W(x)b. Then two observers: **Case A** — P3 reads R(x)b, R(x)a and P4 reads R(x)b, R(x)a. **Case B** — P3 reads R(x)b, R(x)a and P4 reads R(x)a, R(x)b. Which cases are sequentially consistent? Which are causally consistent?

Solution

Case A: both observers see b first and then a, matching the single global order W(x)b then W(x)a. **Sequentially consistent** (and causal).

Case B: P3 sees the order b, a but P4 sees a, b — no single global order satisfies both, so it is **not sequentially consistent**.

The two writes are made by different processes with no causal link (concurrent), so a causal store **may** show different orders → Case B is **causally consistent**.

Weak consistency and synchronisation

With **weak consistency** only special **synchronisation variables** are consistent: (1) all earlier writes are completed before a synchronisation, and (2) no access is allowed until all previous synchronisations have completed. **Release consistency** splits this into *acquire* (before entering a critical region) and *release* (on leaving); changes are propagated at release. **Entry consistency** associates each shared item with its own lock, so only the data guarded by the lock is made consistent.

7.3 Client-Centric Consistency Models

Data-centric models assume many concurrent processes updating the store. In many systems (Web, DNS, mobile users' mailboxes) most operations are reads, and updates are rare and by one process. Then only **eventual consistency** is needed: if no new updates occur, all replicas eventually converge. The problem arises when one client connects to *different replicas* at different times. **Client-centric models** give guarantees from the point of view of a single client:

Model	Guarantee	Example
Monotonic reads	If a process reads x, later reads of x return the same or a more recent value	E-mail: once a message has been read, it does not vanish from the inbox at another replica
Monotonic writes	A write by a process is completed before any later write by the same process	Software library update: a patch is applied only after the previous version is installed
Read-your-writes	A write by a process is always seen by its subsequent reads	Changing a Web page password, then logging in on another replica
Writes-follow-reads	A write following a read on x is applied to the same or a newer version of x	Reply to a post is only stored after the post itself

*Note: Implementation idea: every write gets a globally unique id; each client keeps a **read set** and a **write set** (ids of relevant writes). Before serving a request, a replica checks that it has applied all writes in the client's sets (often using version vectors), otherwise it first fetches them or forwards the request.*

7.4 Replica Management

Replica placement

- **Permanent replicas** — initial set, created by the owner; e.g., mirror sites and clusters at fixed locations.
- **Server-initiated replicas** — created dynamically by the server near heavy demand (push caches, CDN edge nodes).
- **Client-initiated replicas** — client caches; created by the client to reduce access time.

Update propagation — what to propagate

- **Invalidation protocols** — send a message saying data is out of date; cheap in bandwidth, good when there are many writes and few reads.
- **State transfer** — send the new value (or a delta); good when read/write ratio is high.

- **Operation transfer (active replication)** — send the operation; small messages but each replica must run it deterministically.

Push versus pull

Issue	Push-based (server-initiated)	Pull-based (client-initiated)
State at server	Keeps track of clients/caches	None
Messages	Update (and later fetch if invalidation)	Poll and update
Response time	Immediate	Fetch-time delay on a miss
Best when	Read-to-update ratio is high	Read-to-update ratio is low; many clients

Leases combine both: the server promises to push updates for a limited time; after the lease expires, the client must pull or renew. Lease durations can adapt to age of item, client demand and server load. Updates can be sent by **unicast** to each replica or by **multicast** (cheaper in LANs).

7.5 Consistency Protocols

A **consistency protocol** implements a consistency model.

Primary-based protocols

Each data item has a **primary** replica responsible for coordinating writes.

- **Remote-write (primary-backup)** — all writes are forwarded to a fixed primary, which updates the backups and then acknowledges. Provides sequential consistency (all writes ordered by the primary) but writes are slow (blocking).
- **Local-write** — the primary copy migrates to the process that wants to write; the process updates locally and then propagates (useful for mobile / disconnected operation).

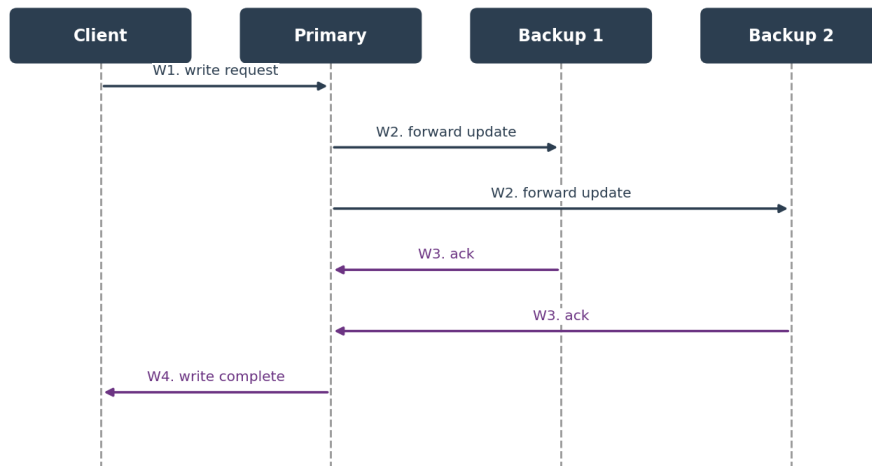


Figure 7.1 — Remote-write (primary-backup) protocol.

Replicated-write protocols

- **Active replication** — every replica executes each operation. Operations must be applied in the same order at all replicas, achieved with a **sequencer** or totally-ordered multicast (Lamport timestamps). Problem: replicated invocations of other services.
- **Quorum-based protocols** (Gifford) — a client must acquire a **read quorum NR** or a **write quorum NW** of the N replicas, subject to $NR + NW > N$ (prevents read-write conflicts) and $NW > N/2$ (prevents write-write conflicts). Each replica has a version number; a read returns the highest version among the quorum.

Example 7.2 — Quorum validity

$N = 12$ replicas. Are (a) $NR = 3, NW = 10$, (b) $NR = 7, NW = 6$, (c) $NR = 1, NW = 12$ valid quorum choices?

Solution

- (a) $3 + 10 = 13 > 12$ ✓ and $10 > 6$ ✓ → **valid**; reads are cheap, writes expensive.
- (b) $7 + 6 = 13 > 12$ ✓ but 6 is **not** > 6 ✗ → **invalid**: two clients could write to disjoint sets of 6 replicas at the same time (write-write conflict).
- (c) $1 + 12 = 13 > 12$ ✓ and $12 > 6$ ✓ → **valid**: this is **ROWA** (read-one, write-all).

Cache-coherence protocols

Used in multiprocessors and distributed shared memory. **Write-through** propagates every write immediately; **write-back** propagates when the cache line is evicted; **write-invalidate** invalidates other copies before updating; **write-update** sends the new value to all caches. Coherence detection can be static (compile time) or dynamic (run time).

7.6 Caching and Replication in the Web

The Web is a huge read-mostly system in which caching and replication are essential to scale.

- **Client-side caching** — the browser stores fetched pages/objects; **proxy (forward) caches** shared by an organisation or ISP improve hit ratios; **reverse proxies** cache close to the origin server.
- **Cache consistency in HTTP** — the server sets *Cache-Control: max-age, Expires, Last-Modified* and *ETag* headers. A cache validates a stale entry with a **conditional GET** (*If-Modified-Since / If-None-Match*); the server returns **304 Not Modified** (no body) if unchanged.
- **Cooperative caching** — caches ask sibling caches (e.g., Internet Cache Protocol) before fetching from the origin.
- **Server replication** — mirror sites and **Content Delivery Networks** (Akamai, Cloudflare) replicate content on edge servers; DNS redirection or anycast sends clients to a nearby edge.
- **Dynamic content** — pages generated from databases are harder to cache; techniques: cache fragments, edge-side includes, cache query results, replicate the database at the edge.

Example 7.3 — Cache hit ratio

A proxy cache has hit ratio 0.6. A hit costs 10 ms and a miss costs 200 ms (fetch from origin). What is the average access time?

Solution

$T = 0.6 \times 10 + 0.4 \times 200 = 6 + 80 = \mathbf{86 \text{ ms}}$ (compared with 200 ms without a cache).

Key Takeaways

- ✓ Replication improves reliability and performance but creates the need for consistency; models trade cost for guarantees.
- ✓ Data-centric: strict > linearizable > sequential > causal > FIFO; weak/release/entry use synchronisation points.
- ✓ Client-centric (eventual consistency): monotonic reads, monotonic writes, read-your-writes, writes-follow-reads.
- ✓ Replica management covers placement (permanent, server-initiated, client-initiated), what to propagate (invalidate/state/operation) and push vs pull vs leases.
- ✓ Protocols: primary-backup, local-write, active replication, quorums with $NR + NW > N$ and $NW > N/2$.
- ✓ Web scaling relies on proxy caches, conditional GET, CDNs and cooperative caching.

— *End of Unit 7* —

bcalink.com