Lab 1.3.1 — Replication and Consistency in a Distributed KV Store

Session 1.3 — Distributed Systems Fundamentals
Hands-on lab
Tools: Scala 3 (`scala-cli`), `nodelib.scala` (provided)

---

1. Objective

You will build two replication strategies on top of a simulated distributed key-value store and observe how each behaves under network partitions. By the end you will have experienced the CAP trade-off firsthand: the same cluster, the same data, but radically different behavior depending on the replication model you choose.

2. The Framework

A library file `nodelib.scala` provides the infrastructure. You write your node behavior in `lab.scala`.

2.1. What the library gives you

- **`Cluster`** — the simulated network. Creates nodes, routes messages with random latency (50–500ms), and can simulate network partitions.
- **`KVServer`** — a file-backed key-value store that handles `GET`, `PUT`, `DEL`. Each node gets its own isolated store directory.
- **`KVClient`** — sends requests to a node through the cluster network. Returns `Future[RPCResponse]`.
- **`NodeCommand`** — the message type your node behavior receives: `Stop` or `Request(rpc)`.
- **`ServerCommand`** — what you send to the `KVServer` child actor: the RPC plus a `replyTo` for the response.

2.2. What you write

Your node is a Pekko `Behavior[NodeCommand]` that receives client requests and decides what to do with them. The default node simply forwards every request to its local `KVServer`:

```
object MyNode:
  def init(cluster: Cluster, id: String, storeRoot: Path): Behavior[NodeCommand] =
    Behaviors.setup: ctx =>
      val serverRef = ctx.spawn(KVServer.init(id, storeRoot), "server")
      Behaviors.receiveMessage:
        case NodeCommand.Stop => Behaviors.stopped
        case NodeCommand.Request(rpc) =>
          serverRef ! ServerCommand(rpc, rpc.replyTo)
          Behaviors.same
```

To implement replication you will modify this behavior to intercept requests before (or after) forwarding them to the local store, and use `cluster.send(from, to, msg)` to communicate with peer nodes.

2.3. Key APIs

Call	Effect
`cluster.send(from, to, msg)`	Send a `NodeCommand` from node `from` to node `to` through the network. Returns `false` if partitioned.
`cluster.spawn(id)`	Spawn a node with the default behavior.
`cluster.spawn(id, factory)`	Spawn a node with a custom `NodeFactory`.
`cluster.isolate(id)`	Partition a node from all others and the client.
`cluster.rejoin(id)`	Remove all partition rules involving this node.
`cluster.partition(a, b)`	Partition two specific nodes (symmetric).
`cluster.heal(a, b)`	Heal a specific partition.
`cluster.client(id)`	Get a `KVClient` for an existing node.

3. Part 1 — Async Replication (AP system)

3.1. 1.1 — Fan-out writes

Modify your node behavior so that on every `PUT`, the node:
1. Writes to its own local `KVServer` (and replies to the client immediately)
2. Forwards the same `PUT` to all peer nodes via `cluster.send`

Your node needs to know the list of peer node IDs. Pass them as a parameter to your factory.

Hint: `cluster.send(myId, peerId, NodeCommand.Request(rpc))` sends the same RPC to a peer. The peer's node will process it against its own local store. You do not need to wait for the peers to acknowledge — this is *asynchronous* replication.

3.2. 1.2 — Read from any node

Once writes fan out, every node eventually has the same data. Test this:
1. `PUT` a key on `alice`
2. Wait briefly for replication (e.g. `Thread.sleep(1000)`)
3. `GET` the same key from `bob` and `carol`

Verify that all three nodes return the same value.

3.3. 1.3 — Observe stale reads

Now remove the sleep. Fire a `PUT` on `alice` and immediately `GET` from `bob`. Do you always get the latest value? Why not?

Discussion: this is *eventual consistency*. The write is acknowledged before replicas have caught up. Reads from different nodes may return different values during the replication window.

3.4. 1.4 — Partition behavior

1. Write `color = red` on `alice`
2. Partition `alice` from `bob`: `cluster.partition("alice", "bob")`
3. Write `color = blue` on `alice`
4. Read `color` from `bob` — what do you get?
5. Heal the partition, wait, read again

Discussion: during the partition, `bob` serves stale data — but it *does* serve data. The system remains **available** (every reachable node answers) but sacrifices **consistency** (nodes disagree). This is an **AP** system.

3.5. 1.5 — Write conflicts

1. Partition `alice` from `bob`
2. Write `color = red` on `alice`
3. Write `color = blue` on `bob`
4. Heal the partition, wait for replication
5. Read `color` from both — what happened?

Discussion: both writes succeed (availability), but after healing, the nodes may disagree or one overwrites the other depending on message ordering. There is no conflict resolution — last-writer-wins by accident. Real AP systems (Dynamo, Cassandra) use vector clocks or CRDTs to handle this.

4. Part 2 — Leader / Follower Replication (CP system)

In this part you will implement a different replication model. One node is the **leader** — all writes go through it. The other nodes are **followers** — they receive replicated writes from the leader and serve reads.

There is no leader election. You designate the leader at startup. This is not a course on Raft — the goal is to observe CP behavior, not to implement consensus.

4.1. 2.1 — Leader behavior

Write a `LeaderNode` factory. On `PUT` / `DEL`:
1. Write to the local `KVServer`
2. Forward the write to every follower via `cluster.send`
3. Reply to the client only **after** all followers have been sent the update

On `GET`: serve from the local store directly.

Hint: the leader does not need to wait for follower acknowledgments in this simplified version. The key difference from Part 1 is that **only the leader accepts writes**.

4.2. 2.2 — Follower behavior

Write a `FollowerNode` factory. On `PUT` / `DEL`:
1. If the request comes from the leader (forwarded via `cluster.send`), apply it to the local store
2. If the request comes from a client, **reject it** — followers do not accept writes

Hint: you can distinguish leader-forwarded requests from client requests by adding a wrapper `NodeCommand` case, or by convention (e.g. followers only accept requests through `cluster.send`, not through `cluster.ask`).

On `GET`: serve from the local store. Followers can serve reads.

4.3. 2.3 — Test the happy path

1. Spawn `alice` as the leader, `bob` and `carol` as followers
2. Write data through `alice`
3. Read from `bob` and `carol` — verify they have the data

4.4. 2.4 — Partition the leader

1. Write some data through `alice`
2. Isolate `alice`: `cluster.isolate("alice")`
3. Try to write through `alice` — what happens?
4. Try to read from `bob` — does it work?
5. Try to write through `bob` — what happens?

Discussion: when the leader is partitioned, **writes are unavailable** — no node can accept them. Reads from followers still work (they have stale but consistent data). The system chose **consistency over availability**: rather than risk conflicting writes, it refuses to write at all. This is a **CP** system.

4.5. 2.5 — Heal and recover

1. Rejoin `alice`
2. Write new data — verify it replicates to followers

3. The system resumes normal operation with no conflicts

Discussion: because only one node ever writes, there are no conflicts to resolve after a partition heals. Compare this to Part 1 where both sides could write independently.

5. Part 3 — Comparison and Discussion

5.1. 3.1 — Discussion questions

1. *When would you choose an AP system?* Think of use cases where availability matters more than strict consistency (shopping carts, social media likes, DNS).
2. *When would you choose a CP system?* Think of use cases where consistency is critical (bank transfers, inventory counts, configuration management).
3. *What is missing from our CP system?* We hardcoded the leader. What happens if the leader crashes permanently? This is the problem that consensus protocols (Raft, Paxos, ZAB) solve — but that is a topic for another day.
4. *Can you have both?* Some systems (e.g. CockroachDB, Spanner) use consensus for writes but allow stale reads from followers. Where does that sit on the CAP spectrum?

6. Key Takeaways

1. **Replication is not free.** Every replication strategy trades something — latency, availability, or consistency.
2. **AP systems stay available during partitions** but may serve stale or conflicting data. Conflict resolution is the hard problem.
3. **CP systems stay consistent during partitions** but become unavailable for writes. Choosing a leader avoids conflicts but creates a single point of failure.
4. **CAP is not a toggle.** Real systems mix strategies: strong consistency for writes, eventual consistency for reads, tunable quorum levels per operation.
5. **Network partitions are inevitable.** Your code must handle them — the question is how.