0) The problem ZooKeeper solves (the “why”)

Imagine you have a **distributed system** = many servers working together. In that world: servers crash, restart, join, leave, network breaks. The scary part is: **they must still agree on shared decisions**, like:

- “Who is the leader?”
- “Which servers are alive?”
- “What configuration should everyone use?”
- “Who gets to do this job first?”

Without a coordinator, the system can get into fights:

- Two servers both think they are leader → conflict / corruption.

ZooKeeper is that coordinator. Your slide literally calls it a “small, super-reliable manager / class monitor.”
It keeps common settings, tracks who’s alive, picks a leader, coordinates work, and sends alerts when something changes.

---

1) ZooKeeper in one line

ZooKeeper runs a **small cluster (ensemble)** of servers that all keep a **tiny shared “tree” of data** (called znodes). Clients read from any server; updates (writes) are coordinated through a leader, using a protocol called **ZAB** for consistent ordering.

---

2) Ensemble / cluster (what it is + why 3–5 servers)

An **ensemble** is just a **group of ZooKeeper servers working together** as one service.

Why not one server?

- If one server dies, your whole system would lose coordination.
- With an ensemble, if one fails, others continue.

Big exam keyword: ZooKeeper requires a **majority (quorum)** to operate.
Example:

- 3 servers → need 2 alive

- 5 servers → need 3 alive

Quorum/majority = what exactly?

If N servers exist, majority = **floor(N/2) + 1**.

- N=3 → 2
- N=5 → 3
- N=7 → 4

This majority rule prevents “split brain” (two different groups both acting like the real system).

---

3) The three roles: Leader, Followers, Observers

In ZooKeeper, every server has a role: **Leader / Follower / Observer**.

A) Leader (the boss for writes)

Only **one leader at a time**.
Leader:

- Processes **all write requests** (create/delete/setData/setACL/multi)
- Assigns transaction IDs (**zxid**)
- Proposes updates to followers and runs **ZAB**

Your class note matches this: **write replication is coordinated by leader**.

B) Followers (voting members)

Followers:

- Handle **read** requests directly
- Forward writes to leader
- **Vote** in leader election + write agreement (ACKs)

Leader + followers form the **quorum (majority group)**.
Slide even gives: in a 5-node ensemble, if Leader + 2 followers alive → ZooKeeper operates.

C) Observers (non-voting read scalers)

Observers:

- Handle reads like followers

- **Do not vote**
- **Do not count toward quorum**
- Purpose: scale reads without slowing writes

Why does adding normal servers slow writes?
Because writes need majority ACK. If you increase voting servers:

- Majority becomes bigger → leader waits for more ACKs → writes slower.
  Observers avoid that.

Your note: “Observers are non voting” — correct.

---

4) The ZooKeeper “data” is a TREE (znode hierarchy)

ZooKeeper keeps a small **tree structure** like folders.
Each item is a **znode**.

What is a znode?

A znode is like a tiny file:

- stores small data (usually a few KB; up to ~1MB)
- has metadata (version, ACL, timestamps)
- can have children (like directories)

So the tree looks like:

- `/config`
- `/leaders`
- `/members`
  etc.

Your class note: “Each tree ke andar sub tree is also a node” — yes: children are znodes too.

---

5) Special znode types (this is VERY testable)

1) Persistent znode

- Stays until explicitly deleted
- Doesn’t disappear if client disconnects/session expires

Use: configuration storage.

2) Ephemeral znode

- Exists only while client session is alive
- Auto-deletes when session ends
- Used for membership / “who is alive”
- **Cannot have children**

Use: service registration, leader election.

3) Sequential znode

- ZooKeeper appends a monotonically increasing number
- Used for ordering, locks, elections

---

6) Watches (the “I don’t want to keep checking” feature)

A **watch** is a lightweight notification: client says “tell me if this changes.”

Key facts:

- Watches avoid constant polling
- Watch is a **one-time trigger** (fires once)
- Can trigger on:
    - data change
    - deletion
    - creation
    - children list change

Your note explains it perfectly: “I can’t keep track of everything, so I put a watch.”

---

7) How Leader Election works (simple story)

ZooKeeper election uses **ephemeral-sequential** znodes:

Steps:

1. Every server creates a node like:
   `/election/server-00005`

2. The one with the **smallest number becomes leader**
3. If leader dies → its ephemeral node disappears → next smallest becomes leader

This is why ephemeral and sequential are so important.

---

8) Group Membership (who is alive?)

Each server creates an **ephemeral znode** under `/members`.
If a server crashes:

- its ephemeral node disappears automatically
- everyone immediately sees updated membership

This is “service discovery / membership”.

---

9) The write pipeline: why writes are “complicated”

ZooKeeper keeps all data **in memory** for fast reads.
But writes must be consistent across machines:

Write flow:

1. Client sends write (create/setData/delete)
2. Goes to **leader**
3. Leader broadcasts to followers
4. Majority ACK
5. Leader commits
6. Everyone applies in same order

This ensures a **single system image**: it behaves like one datastore.

---

10) ZAB (ZooKeeper Atomic Broadcast) — the heart

ZAB is the consensus/replication protocol.

Why “Atomic”?

Each update is applied **completely or not at all** across all nodes.

Why “Broadcast”?

Leader sends updates to all followers to ensure **strong consistency**.

ZAB steps (exam perfect):

1. Leader election
2. Proposal: leader assigns zxid and proposes
3. Followers persist + ACK
4. Commit: after quorum ACK, leader commits and tells followers to commit
5. Crash recovery: new leader continues from last committed transaction

---

11) Guarantees (very exam-friendly)

ZooKeeper guarantees:

- **Total order** of updates (global ordering)
- **Atomicity** (multi ops succeed or fail entirely)
- **Single system image**
- **Reliability** (once quorum ACK, it persists)
- **Session semantics** (ephemerals tied to session timeout)

---

12) Use-cases (you must be able to write these with examples)

Common use cases listed:

- Leader election
- Membership/service discovery
- Distributed locks
- Configuration store
- Barriers/queues primitives

---

13) ZooKeeper API (don’t memorize code, but know meanings)

Core calls:

- `create(path, data, acl, flags)` (flags = EPHEMERAL, SEQUENTIAL)
- `delete(path, version)`
- `getData / setData`
- `exists(path, watch)`
- `getChildren(path, watch)`
- `multi(ops)` atomic transaction
- `sync(path)` ensure read is up-to-date with leader

---

14) The BIG example in slides: HOD with ZooKeeper (WATCHES + EPHEMERAL/SEQUENTIAL)

This part is very "scenario based" — your teacher loves it.

Problem (HOD):

- Client starts JobTracker (JT) + TaskTrackers (TTs)
- JT IP/port not known before startup
- TTs need to find JT
- Client needs to find JT

Solution with ZooKeeper:

1. Client creates znode: `/hod/jt-` with **Ephemeral|Sequence** → `/hod/jt-1`
2. Client spawns JT and TT and passes the znode path
3. JT writes contact info into znode using `setData`
4. Client/TT use `getData(..., watch=true)` to get notified when JT writes data
5. JT and TT also watch znode existence; if node disappears, they exit (shutdown/failure)

This example teaches: **watches + ephemeral nodes + coordination without polling**.

---

15) What ZooKeeper is NOT (super testable)

ZooKeeper is:

- Not a database
- Not for big data
- Not for large files

- Not for high-write workloads
- ONLY for tiny metadata + coordination

---

16) Performance intuition (why reads are great, writes get slower)

Slides show: as %writes increases, performance drops, and adding more servers can reduce throughput at high write rates.
Reason: writes need quorum ACK → more voting servers → slower writes (this is why observers exist).

Q1. What is ZooKeeper and why is it required in distributed systems?

Answer:
ZooKeeper is a **distributed coordination service** used to manage and synchronize distributed applications.

It is required because:

1. Distributed systems need **coordination**, not just storage.
2. Components need **leader election, configuration management, and synchronization**.
3. It provides **consistency, ordering, and fault tolerance** which are hard to achieve manually.

Example:
In Hadoop, NameNode coordination and job tracking rely on ZooKeeper to avoid split-brain scenarios.

Q2. Explain ZooKeeper's tree structure.

Answer:
ZooKeeper organizes data in a **hierarchical tree structure**, similar to a file system.

Key points:

1. Each node in the tree is called a **znode**.

2. Every znode can have **child znodes (subtrees)**.
3. Each subtree itself behaves like a node.

Important class note:
If a tree has 5 znodes, **each subtree is also a znode**.

Q3. What is a znode? What can it contain?

Answer:
A **znode** is the basic data unit in ZooKeeper.

A znode can contain:

1. **Small data** (configuration, metadata).
2. **Metadata** (timestamps, version number).
3. **Children znodes** (subdirectories).

ZooKeeper is **not for large data storage**.

Q4. Explain the three types of znodes.

Answer:
ZooKeeper supports three znode types:

1. **Persistent znode**
    - Exists until explicitly deleted.
2. **Ephemeral znode**
    - Exists only while the client session is active.
3. **Sequential znode**
    - ZooKeeper appends a unique sequence number automatically.

Example:
Leader election uses **ephemeral + sequential** znodes.

Q5. Why are ephemeral znodes important?

Answer:
Ephemeral znodes are important because:

1. They automatically disappear if a client crashes.
2. They help detect **node failures**.
3. They ensure system state stays **consistent without manual cleanup**.

Example:
If a worker node dies, its ephemeral znode disappears, informing the system instantly.

Q6. What is ZooKeeper's cluster architecture?

Answer:
ZooKeeper works in a **replicated cluster**.

Components:

1. **Leader** – handles all write requests.
2. **Followers** – replicate data and serve read requests.
3. **Observers** – non-voting members.

This ensures **high availability and fault tolerance**.

Q7. Who handles read and write requests in ZooKeeper?

Answer:

1. **Write requests** always go through the **leader**.
2. The leader replicates writes to followers.
3. **Read requests** can be served by followers.

Important note:
Rights (writes) are replicated by the **leader itself** to followers.

Q8. What are observers in ZooKeeper?

Answer:
Observers are special ZooKeeper nodes that:

1. **Do not participate in voting**.
2. **Do not affect quorum size**.

3. Only replicate data and serve read requests.

They improve **read scalability** without slowing leader elections.

Q9. Explain quorum and majority in ZooKeeper.

Answer:
Quorum means **more than half of the nodes must agree**.

Key points:

1. Required for write consistency.
2. Prevents split-brain.
3. Ensures system correctness.

Numerical example:
In a 5-node cluster, quorum = **3 nodes**.

Q10. Why is majority important in leader election?

Answer:
Majority ensures:

1. Only **one leader** is elected.
2. Network partitions do not create multiple leaders.
3. System remains **consistent and safe**.

Without majority, distributed systems break.

Q11. How does ZooKeeper perform leader election?

Answer:
Leader election works by:

1. Nodes creating **ephemeral sequential znodes**.
2. Node with **lowest sequence number** becomes leader.
3. Others watch the node just before them.

If leader fails, election automatically restarts.

Q12. Why does ZooKeeper guarantee ordering?

Answer:
ZooKeeper guarantees strict ordering because:

1. All writes pass through the leader.
2. Writes are assigned **monotonically increasing transaction IDs**.
3. Updates are applied in the same order on all nodes.

Important:
Ordering **never breaks**.

Q13. What is ZAB (ZooKeeper Atomic Broadcast)?

Answer:
ZAB is ZooKeeper's **consensus protocol**.

It ensures:

1. Atomic broadcast of updates.
2. Same order of updates on all nodes.
3. Consistency during leader changes.

It is **not Paxos**, but conceptually similar.

Q14. Explain the phases of ZAB.

Answer:
ZAB works in two phases:

1. **Leader election phase** – elect a leader.
2. **Broadcast phase** – leader proposes updates, followers acknowledge, then commit.

Only after majority ACK does a write commit.

Q15. What are watches in ZooKeeper?

Answer:
Watches are **one-time notifications** set by clients.

They:

1. Notify clients when data changes.
2. Avoid continuous polling.
3. Improve performance and responsiveness.

Q16. Why are watches needed?

Answer:
Watches are needed because:

1. Clients cannot track everything continuously.
2. Polling wastes resources.
3. Watches enable **event-driven communication**.

Class example:
“I cannot keep track of everything, so I put a watch.”

Q17. Explain watch behavior using Hadoop/YARN example.

Answer:
In Hadoop:

1. NameNode and YARN place watches on znodes.
2. When a Map task finishes, a znode entry appears.
3. The watch triggers immediately.

Result: **No waiting, instant notification**.

Q18. Are watches persistent?

Answer:
No, watches are:

1. **One-time triggers**.
2. Removed after firing.
3. Must be re-registered if needed again.

This avoids notification storms.

Q19. What happens when a watched znode is created?

Answer:

1. Client receives a notification.
2. Client wakes up instantly.
3. Client performs the next operation.

This is **event-driven coordination**.

Q20. What is a lock in distributed systems?

Answer:
A lock ensures:

1. Mutual exclusion.
2. Only one process accesses a resource at a time.
3. Consistency of shared resources.

Distributed locks are harder than local locks.

Q21. How does ZooKeeper implement distributed locks?

Answer:
ZooKeeper uses:

1. Ephemeral sequential znodes.
2. Ordering by sequence number.
3. Watches to notify next client.

This avoids deadlocks and starvation.

Q22. Which Hadoop ecosystem applications need locks?

Answer:
Locks are needed in:

1. HDFS NameNode operations.
2. Resource scheduling (YARN).
3. Leader-based services like HBase Master.

Locks prevent **conflicting writes**.

Q23. Why is ZooKeeper not used as a database?

Answer:
ZooKeeper is not a database because:

1. It stores **very small data only**.
2. Optimized for coordination, not queries.
3. Frequent writes would degrade performance.

Q24. Explain fault tolerance in ZooKeeper.

Answer:
ZooKeeper achieves fault tolerance via:

1. Replicated cluster.
2. Leader election.
3. Majority quorum.

System continues working as long as **quorum exists**.

Q25. What happens if the leader crashes?

Answer:

1. Followers detect failure.
2. Leader election starts.
3. New leader is chosen using ZAB.

System remains consistent.

Q26. Can ZooKeeper handle network partitions?

Answer:
Yes, because:

1. Only majority partition remains active.
2. Minority partition stops accepting writes.
3. Prevents split-brain.

Q27. Compare ZooKeeper leader and follower roles.

Answer:

Leader	Follower
Handles writes	Replicates data
Assigns order	ACKs proposals
Controls consistency	Serves reads

Q28. Why are sequential znodes useful in elections?

Answer:
Sequential znodes:

1. Provide total ordering.
2. Avoid conflicts.
3. Simplify leader election logic.

Lowest sequence number always wins.

Q29. Why does ZooKeeper use in-memory data?

Answer:
ZooKeeper keeps data in memory to:

1. Achieve low latency.
2. Serve reads quickly.
3. Support real-time coordination.

Disk is used only for durability.

Q30. Explain ZooKeeper's consistency guarantee.

Answer:
ZooKeeper guarantees:

1. **Linearizable writes**.
2. **FIFO client ordering**.
3. **Global ordering of updates**.

This is stronger than eventual consistency.

Q31. How does ZooKeeper ensure no stale reads after leader change?

Answer:
Because:

1. Leader must sync followers before serving writes.
2. New leader replays committed transactions.
3. Uncommitted updates are discarded.

Q32. Why does ZooKeeper scale poorly for writes?

Answer:
Because:

1. Every write goes through leader.
2. Requires majority ACK.
3. Increases coordination overhead.

ZooKeeper is **read-heavy optimized**.

Q33. What is meant by “balanced tree” in ZooKeeper?

Answer:
Balanced tree means:

1. Predictable traversal.
2. Efficient lookup.
3. No deep skewed paths.

Improves performance and consistency.

Q34. Why does ZooKeeper avoid polling?

Answer:
Polling:

1. Wastes CPU cycles.
2. Causes unnecessary delays.
3. Scales poorly.

Watches provide **push-based notifications**.

Q35. Scenario: Two leaders appear — is this possible?

Answer:
No, because:

1. Leader requires majority.
2. Only one partition can have majority.

3. Minority partition cannot elect leader.

ZooKeeper prevents split-brain.

Q36. Why is ZooKeeper critical in Big Data architectures?

Answer:
ZooKeeper is critical because:

1. Big data systems are distributed.
2. Coordination failures break systems.
3. ZooKeeper provides **reliable synchronization backbone**.

Used by Hadoop, HBase, Kafka, YARN.