Apache ZooKeeper: Distributed Coordination Service

Comprehensive Lecture Notes

Contents

1 Introduction to Distributed Coordination

1.1 The Coordination Problem

In distributed systems, multiple servers (nodes) work together to perform tasks. However, unlike a single monolithic application, distributed systems face unique challenges:

- **Partial Failures:** Servers may crash, network cables may be cut, or messages may be dropped.
- **Race Conditions:** Multiple nodes may try to modify the same resource simultaneously.
- **Information Divergence:** Nodes must agree on shared configuration data (e.g., "Who is the master database?").

Without a coordinator, distributed processes may fight for resources, operate on stale data, or enter split-brain scenarios where two nodes believe they are the leader.

1.2 What is ZooKeeper?

Apache ZooKeeper is a high-performance, distributed coordination service. It exposes common services—such as naming, configuration management, synchronization, and group services—in a simple interface so that developers do not have to write them from scratch.

Analogy: Think of ZooKeeper as the "Class Monitor" of a distributed system. It:

1. Retains common settings (Configuration).
2. Tracks attendance (Group Membership/Liveness).
3. Decides who is in charge (Leader Election).
4. Sends alerts when the state changes (Watches).

2 System Architecture

ZooKeeper is replicated to ensure high availability. It does not run as a single server but as a cluster, referred to as an **Ensemble**.

2.1 The Ensemble and Quorum

An ensemble typically consists of an odd number of servers (3, 5, or 7).

- **Quorum:** To operate successfully, ZooKeeper requires a majority of servers to be active and communicating.
- **Formula:** If an ensemble has N servers, the quorum size is $\lfloor N/2 \rfloor + 1$.
- **Fault Tolerance:** An ensemble of N servers can tolerate F failures, where $N = 2F + 1$.
  - Example: A 3-server ensemble can tolerate 1 failure.
  - Example: A 5-server ensemble can tolerate 2 failures.

2.2 Server Roles

Nodes in a ZooKeeper ensemble assume specific roles to manage consistency and scaling.

2.2.1 1. Leader

- **Uniqueness:** There is only one leader at a time.
- **Responsibilities:**
- Handles all **write** requests (create, delete, setData).
- Converts writes into transactions and assigns them a unique identifier (ZXID).
- Coordinates replication to followers using the ZAB protocol.

2.2.2 2. Followers

- **Read Handling:** Can serve read requests directly from their local memory.
- **Write Handling:** They forward all write requests to the Leader.
- **Consensus:** They participate in the voting process (Quorum) to accept writes and elect a new leader if the current one fails.

2.2.3 3. Observers

- **Function:** Like Followers, they maintain a copy of the data and serve reads.
- **Restriction:** They **do not vote** in the consensus process or leader elections.
- **Use Case:** They increase read scalability without degrading write performance. Adding more voting Followers increases the time required to reach a quorum (more acknowledgments needed); Observers allow the cluster to serve more read clients without slowing down the write path.

3 Data Model: Znodes

ZooKeeper provides a hierarchical namespace, similar to a standard file system. Each node in this tree is called a **Znode**.

3.1 Znode Characteristics

- **Hybrid Storage:** Znodes act like both files (they store data) and directories (they can have children nodes).
- **Data Size:** Designed for metadata, not big data. Znodes usually store small amounts of data (< 1 MB), such as configuration strings or status flags.
- **In-Memory:** The entire data tree is loaded into the RAM of the ZooKeeper servers for low-latency reads.

3.2 Types of Znodes

When creating a Znode, clients specify flags that determine its behavior:

1. **Persistent:**
    - The node remains in the system until explicitly deleted.
    - *Use Case:* Storing static configuration or database connection strings.

2. **Ephemeral:**
    - The node exists only as long as the client session that created it is active.
    - If the client crashes or disconnects, ZooKeeper automatically deletes this node.
    - *Restriction:* Ephemeral nodes cannot have children.
    - *Use Case:* Service discovery (detecting if a server is online).
3. **Sequential:**
    - ZooKeeper automatically appends a monotonically increasing counter to the path name (e.g., `task-0001`, `task-0002`).
    - *Use Case:* Implementing queues or determining priority in leader election.

4 Core Mechanisms

4.1 Watches (Notifications)

Polling a distributed system (asking "Did this change?" repeatedly) is inefficient. ZooKeeper uses a **Watch** mechanism.

- **Definition:** A one-time trigger associated with a Znode.
- **Trigger Events:** Data change, Node deletion, Node creation, Child list modification.
- **Workflow:**
    1. Client reads a node and sets a watch flag.
    2. ZooKeeper returns the data.
    3. If the data changes later, ZooKeeper sends an asynchronous notification to the client.
- **Note:** Watches are one-time. Once triggered, the client must re-register the watch to receive future updates.

4.2 Guarantees

- **Sequential Consistency:** Updates from a client are applied in the order they were sent.
- **Atomicity:** Updates either succeed or fail entirely. There are no partial results.
- **Single System Image:** A client will see the same view of the service regardless of which server it connects to.
- **Reliability:** Once an update is applied, it persists until overwritten.

5 The ZAB Protocol

The consistency of ZooKeeper relies on the **ZooKeeper Atomic Broadcast (ZAB)** protocol. It is a consensus protocol optimized for high-throughput processing of primary-backup systems.

5.1 How ZAB Works

1. **Leader Election:** Upon startup or leader failure, the ensemble elects a new leader.
2. **Proposal Phase:** When a leader receives a write request, it creates a transaction proposal with a unique, monotonically increasing ID called the **ZXID**.
3. **Broadcast:** The leader sends the proposal to all followers.
4. **Acknowledgment:** Followers log the transaction to a durable Write-Ahead Log (WAL) and send an ACK to the leader.
5. **Commit:** Once the leader receives ACKs from a quorum (majority), it issues a COMMIT message. The update is applied to the in-memory database and the client receives a success response.

Analogy: Ideally like a postal system. The leader writes a letter (transaction), sends copies to followers, waits for delivery confirmation, and then officially posts it (commit). Observers receive the letter but do not confirm receipt.

6 Common Use Cases

ZooKeeper provides the primitives (Znodes, Watches, Sequencing) to build higher-level distributed coordination patterns.

6.1 1. Leader Election

In a cluster of workers, one must be the master.

- **Implementation:** All contenders create an **Ephemeral-Sequential** Znode under a path like `/election`.
- **Logic:** ZooKeeper assigns sequence numbers (e.g., `node-1`, `node-2`). The node with the **lowest** sequence number is the leader.
- **Failover:** If `node-1` dies (session ends), the node is removed. `node-2` sees it has the new lowest number and promotes itself.

6.2 2. Group Membership / Service Discovery

Determining which servers are currently alive in a cluster.

- **Implementation:** Every server, upon startup, creates an **Ephemeral** node under `/members`.
- **Discovery:** Clients watch the `/members` path. If a server crashes, its ephemeral node vanishes, triggering the watch and notifying clients immediately.

6.3 3. Distributed Locking

Ensuring mutually exclusive access to a shared resource.

- **Implementation:** Clients create **Ephemeral-Sequential** nodes under `/locks`.
- **Logic:** The client with the lowest sequence number holds the lock.
- **Wait-Free:** Clients with higher numbers place a watch on the node immediately *preceding* them. If the lock holder crashes or finishes, the next client in line is notified. This avoids the "Herd Effect" (waking up all clients simultaneously).

6.4 4. Configuration Management

- **Implementation:** Store configuration (e.g., database URL) in a **Persistent** Znode.
- **Update:** All application nodes place a watch on this Znode. When an administrator updates the Znode, all applications are notified and reload the configuration dynamically without a restart.

7 Case Study: Hadoop on Demand (HOD)

7.1 The Bootstrap Problem

In dynamic Hadoop clusters, clients need to submit jobs to a JobTracker, and TaskTrackers need to connect to that JobTracker. However, the IP address of the JobTracker isn't known until it is allocated by the resource manager (Torque).

7.2 ZooKeeper Solution

1. The client creates a sequential Znode (e.g., `/hod/jt-1`) passing the path as a parameter to the system.
2. The JobTracker starts up and writes its contact information (IP/Port) into that Znode.
3. The TaskTrackers and the Client **watch** that Znode.
4. Once data appears, TaskTrackers connect to the JobTracker, and the Client begins submitting jobs.
5. If the Znode disappears, the system knows the JobTracker failed.

8 Performance and Limitations

8.1 Read vs. Write Performance

- **Read Dominant:** ZooKeeper is optimized for workloads where reads are much more frequent than writes (e.g., 10:1 ratio). Reads are fast because they are served from local memory by any server.
- **Write Penalties:** Writes are slower because they require network round-trips to achieve consensus among the quorum.

8.2 Scalability Limits

- Adding more **Followers** improves Read performance but degrades Write performance (leader waits for more ACKs).
- Adding **Observers** improves Read performance *without* degrading Write performance.

8.3 What ZooKeeper is NOT

- It is **not** a general-purpose database (data size per node is limited).
- It is **not** for Big Data storage (use HDFS).
- It is **not** for high-frequency writes (use Redis or Kafka).