Distributed Coordination Service Apache ZooKeeper

Prof. Dr. Wolfgang Blochinger

Zookeeper

- Elasticity of applications is a main concept of cloud computing.
  - Depending on the actual workload more or less resources are employed (usually in a scale out fashion).
- Scale out paradigm leads to (highly) distributed systems!
- Zookeeper was designed to facilitate building large scale and highly dynamic distributed systems.
  - Initially developed at Yahoo!
  - Now Apache top level project
- Some examples of systems that employ Zookeeper:
  - Facebook Messages, Apache Solr, Apache HBase, Apache Kafka, HDFS, …

Distributed Systems (1)

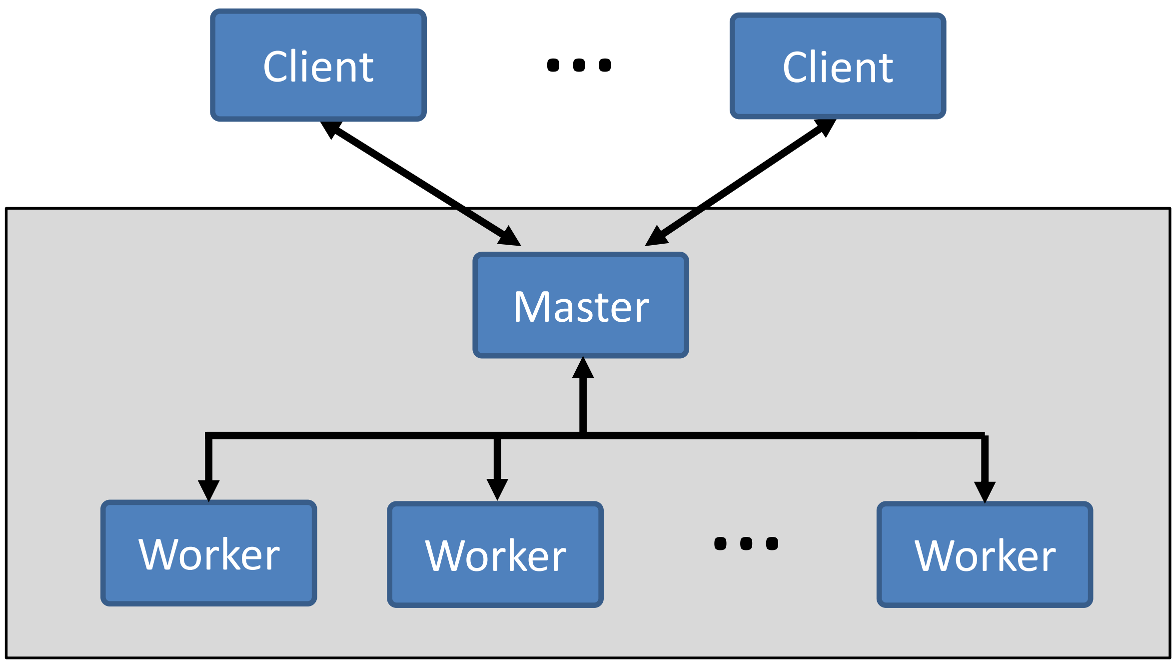
- **Distributed system:** A system comprised of software components (processes) running independently and concurrently across multiple physical machines that are connected by a network.
  - “Shared nothing architecture”
- Processes in a distributed system can communicate by
  - direct message exchange over the network or
  - read and write operations to a shared storage location.
- ZooKeeper employs the shared storage model.
  - Note: Implementing shared storage requires message exchange between the processes and the storage.

Distributed Systems (2)

- The “view” of a process on the whole system (i.e. the state of the other processes) solely results from incoming messages.
- In a distributed system messages can get arbitrarily delayed.
  - Typical reasons: Network failure or congestion (limited processing capacities at sender, intermediate router, or receiver).
- Message delays can lead to difficult issues, for example:
  - **Event Ordering:**
    Process *P* may send a message before another process *Q* sends a message, but *Q*’s message might be received first.
  - **Failure Detection:**
    It is impossible for a process P to reliably determine if an other process Q has crashed (based on message exchange with process Q).

Example: Master-Worker Architecture

- Clients generate tasks which are executed by the workers.
  – Number of workers is adjusted according to current workload.
- Master process is responsible for assigning tasks to workers and thus must keep track of the workers and tasks available.



Coordination Requirements of a Master-Worker Architecture

- **Master election:** When a master crashes a new one must be elected among the workers.
  - New master must be able to recover the state of the old one.
  - Two active masters (split-brain scenario) must be avoided.
- **Worker presence detection:** The master must be able to detect when workers crash or disconnect.
  - Exactly-Once Semantics: Ensure that a task is executed exactly once.
  - At-Most-Once Semantics: Ensure that a task is not executed more than once.
- **Group membership management:** The master must be able to figure out which workers are available to execute tasks.
- **Metadata management:** The master and the workers must be able to store assignments and execution statuses in a reliable manner.

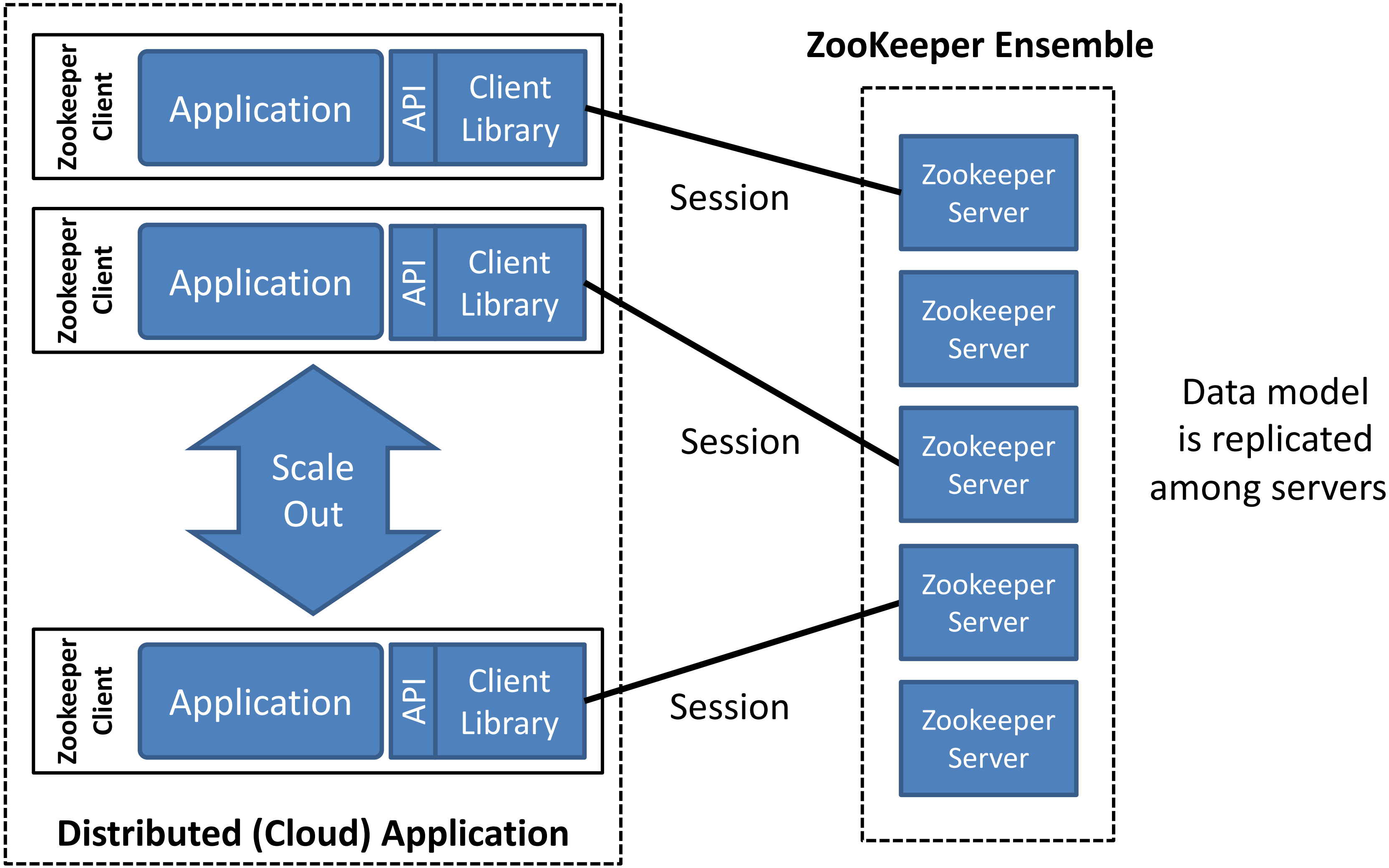
Main characteristics of Zookeeper (1)

- Zookeeper does not expose specific coordination primitives, instead it provides an API based on a generic, **tree based data model**.
  - Applications can implement their own coordination primitives *in a simple way*.
  - Alternative approach: Build for each coordination primitive a specific service, e.g.
    - Chubby (Google): Lock service
    - Centrifuge (Microsoft): Lease service
- A large number of distributed coordination primitives/protocols have been implemented (cf. “**Zookeeper recipes**”):
  - distributed queues, distributed locks, leader election, …

Main characteristics of Zookeeper (2)

- Zookeeper enables separation of application data and control/coordination data.
- Zookeeper based applications consist of a set of clients that connect to Zookeeper servers and invoke operations on them through the Zookeeper client API.
  - Zookeeper servers store the data model which can be manipulated by client requests.
- Zookeeper is designed to be highly reliable, thus avoiding to introduce a single point of failure.
  - The data model is logically centralized but physically replicated among a set of Zookeeper servers.

ZooKeeper Architecture

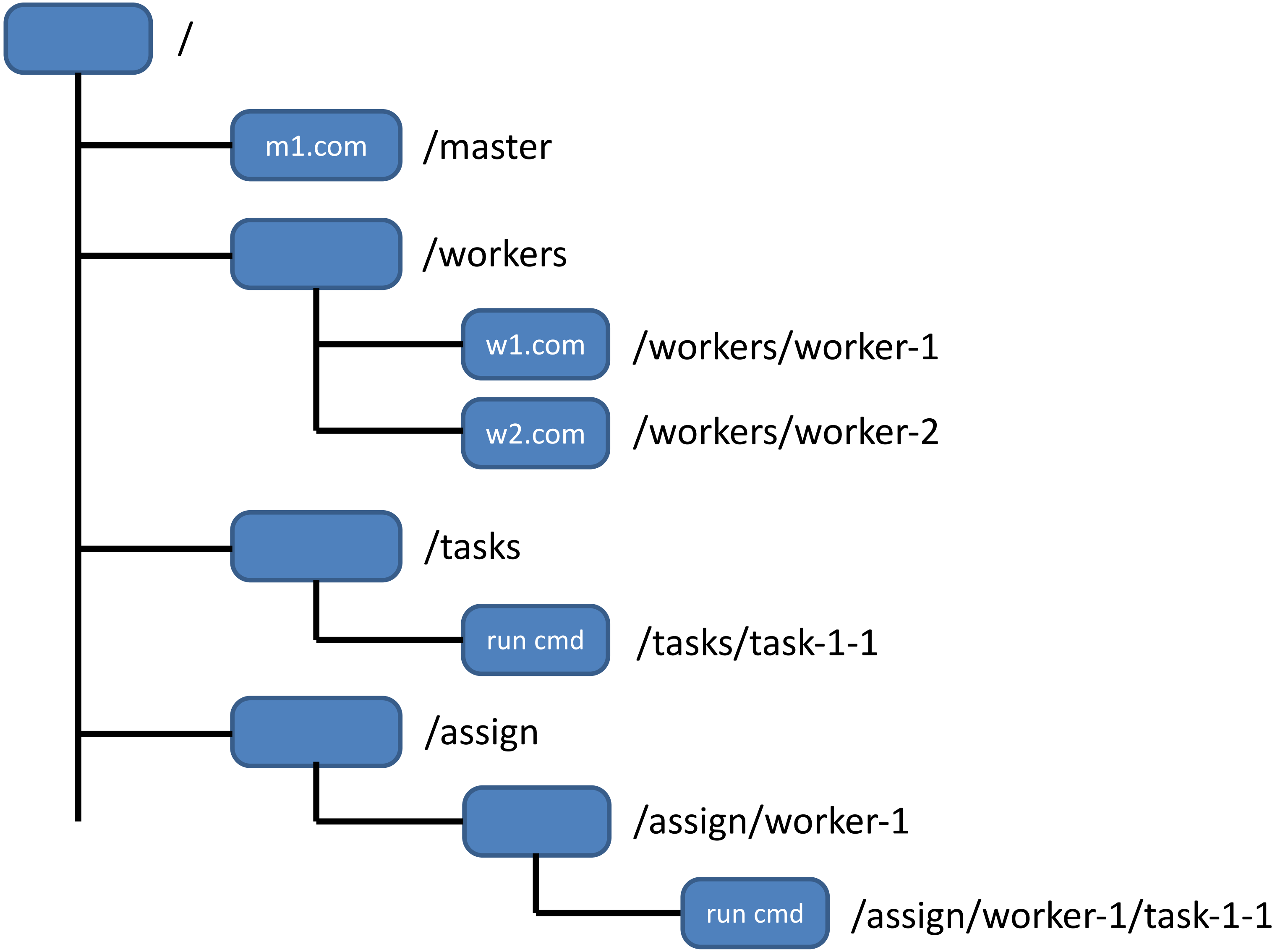


Note: For testing purposes one can use Zookeeper in standalone mode (using one server)

Zookeeper Data Model

- Zookeeper data model represents a **tree of znodes**.
  - A znode can have zero or more child znodes.
  - All znodes (inner znodes as well as leaf znodes) can (but need not) store data.
  - Data is stored as a byte array (format is application specific)
    - Limit on data size per znode: 1 MByte
  - Data access is protected by an ACL (access control list).
- Znodes are always referenced by absolute paths leading to a hierarchical namespace, e.g.
  /assign/worker-1/task-1-1

Zookeeper Data Model (Example)



Basic Zookeeper Operations (1)

- **`create (znode, data, flags)`**
  Creates a znode containing the given data
- **`delete (znode, version)`**
  Deletes a znode
- **`exists (znode, watch)`**
  Checks whether a znode exists
- **`setData (znode, data, version)`**
  Stores data at a znode
- **`getData (znode, watch)`**
  Returns the data stored at a znode
- **`getChildren (znode, watch)`**
  Returns a list of the children of a znode

(Detailed discussion on following slides)

Basic Zookeeper Operations (2)

- Data access is atomic:
  A client always receives the whole data (*getData*) or replaces the whole data (*setData*) stored at a znode. Otherwise the operations fail.
  - No append, insert or partial read operations.

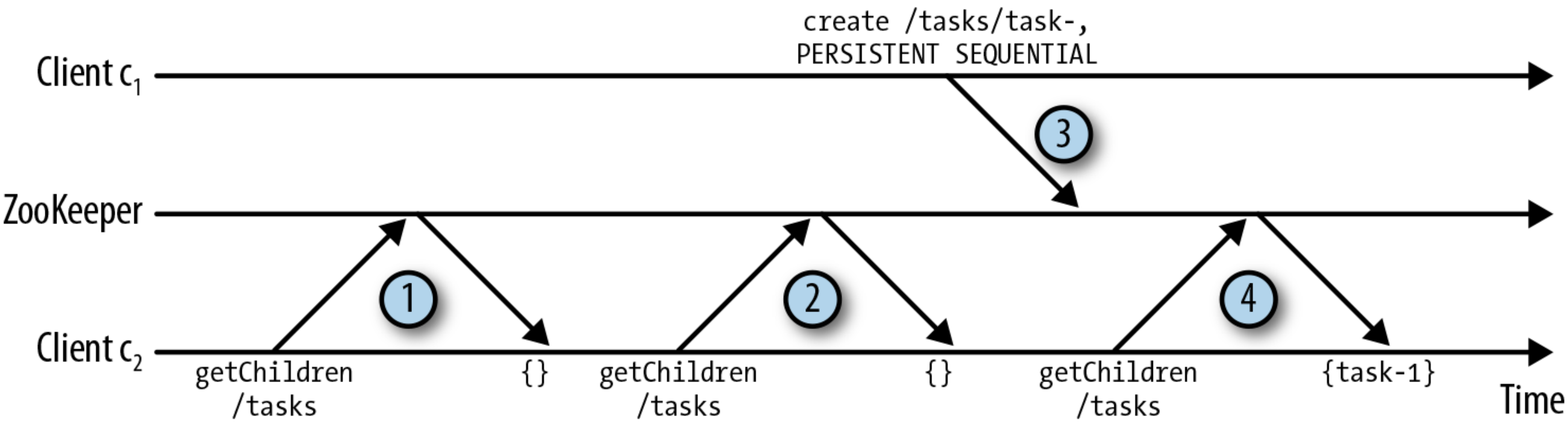
Modes of Znodes (1)

- A znode can be **persistent** or **ephemeral**
  - A **persistent znode** must be explicitly deleted by an arbitrary client (using the delete operation).
    - Master-Worker example: Assignments of tasks to workers must be available even when the master crashes, thus the mode of the “assign znodes” is persistent.
  - An **ephemeral znode** is automatically deleted when the creating client’s session ends (close or expire/crash).
    - Ephemeral znodes may not have child nodes.
    - Ephemeral znodes can also be explicitly deleted
    - Master-Worker example: The mode of “worker znodes” is ephemeral indicating the presence of a specific worker.

Modes of Znodes (2)

- A **sequential** znode is automatically given a unique, monotonically increasing sequence number.
  - Sequential znodes allow to create znodes with unique names and also to see the creation order of znodes.
  - Master-Worker example:
    - One creates a sequential znode with the path /assign/worker-1/*task-*
    - ZooKeeper automatically assigns a sequence number n
    - and the final path of the znode becomes /assign/worker-1/*task-<n>*.
- The mode of a znode is fixed at creation time using the following flags (cf. create operation)
  - PERSISTENT
  - EPHEMERAL
  - PERSISTENT_SEQUENTIAL
  - EPHEMERAL_SEQUENTIAL

Polling for State Changes

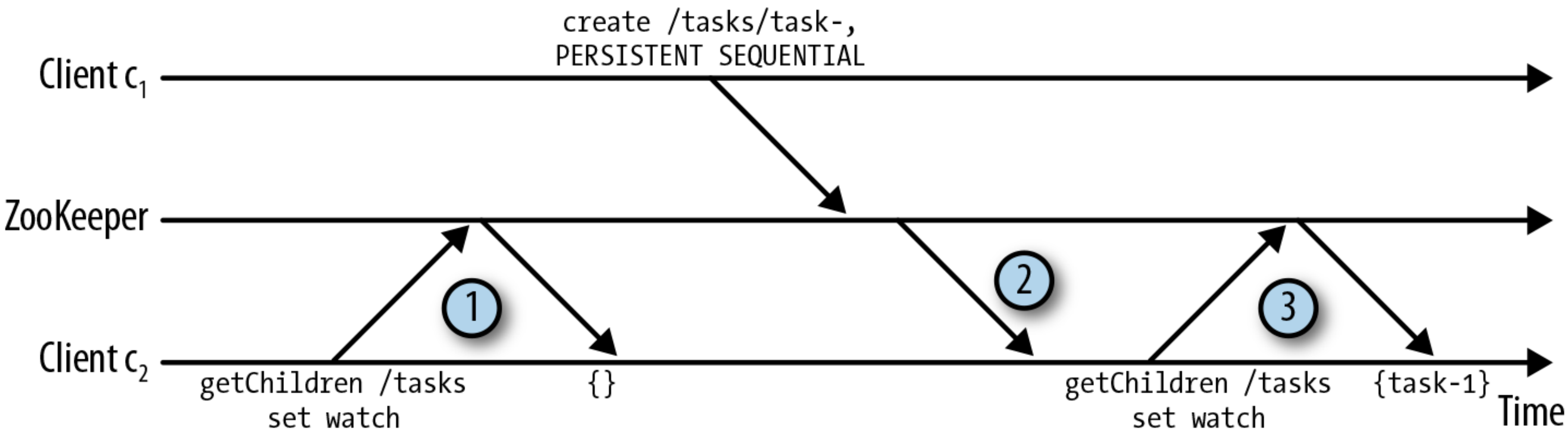


1. Client c_2 reads the list of tasks, initially empty.
2. Client c_2 reads znode again to determine whether there are new tasks.
3. Client c_1 creates a new task.
4. Client c_2 reads again and observes the change.

Watches and Notifications (1)

- Clients can register with ZooKeeper to receive notifications of changes to a znode by setting a watch.
  - Avoids polling for state changes by clients, reducing server load.
  - Watches can be set for changes to the data of a znode, changes to the children of a znode, or a znode being created or deleted.
- Watches are set by operations that read the state of a znode: *getData, getChildren, exists*
  - Reading state and setting the watch is atomic, thus clients cannot miss state changes.
- A notification is triggered when an operation takes place that changes the state of a watched znode: *setData, create, delete*

Watches and Notifications (2)



1. Client c_2 reads the list of tasks, initially empty. It sets a watch for changes.
2. When there is a change, the client is notified.
3. Client c_2 reads the children of /tasks and observes the new task.

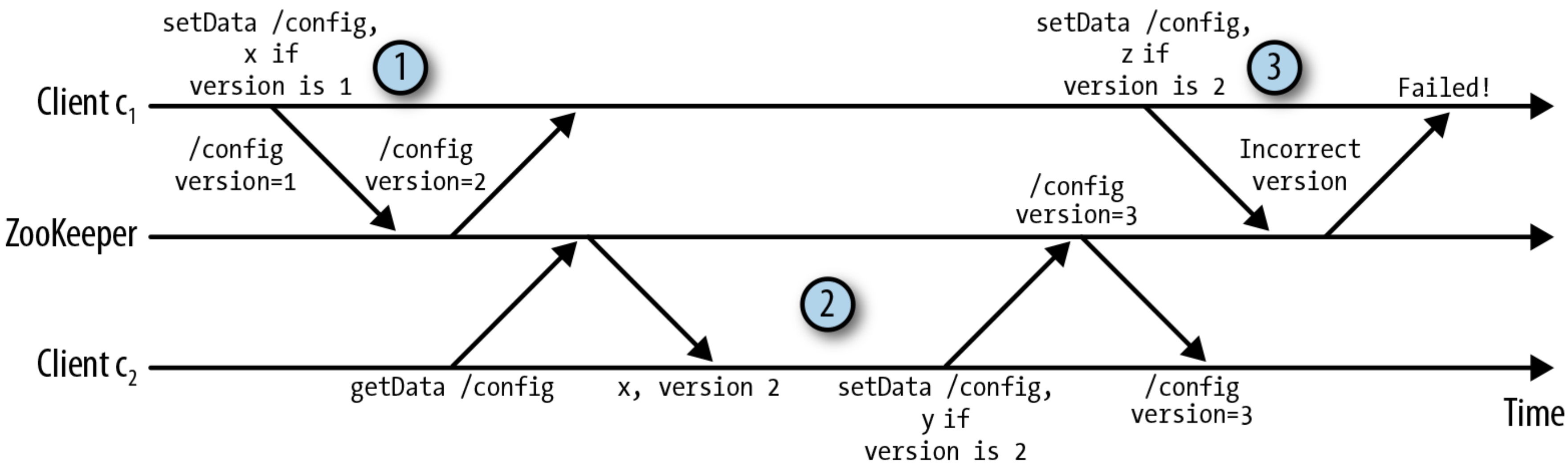
Order of Notifications

- A watch is a **one-shot operation**: it triggers exactly one notification event at the client
  - To further receive notifications, the client must set a new watch upon receiving each notification.
- Notifications preserve the order of updates the client observes.
  - Although changes to the state of ZooKeeper may end up propagating more slowly to any given client, it is guaranteed that clients observe changes to the state according to a global order (see below).

Versions of Znodes (1)

- Each znode has a version number associated with it which is incremented every time its data changes.
- *setData* and *delete* operations must provide a version number as an input parameter and succeed only if the provided version matches the current version on the server.
  - Set version = -1 to omit the conditional check.
- Employing version numbers can prevent inconsistent data updates, see next slide.

Versions of Znodes (2)



1. Client c_1 writes the first version of /config.
2. Client c_2 reads /config and writes the second version.
3. Client c_1 tries to write a change to /config, but the request fails because the version does not match.

Guarantees

- **Sequential consistency:** Updates from a particular client are applied in the order that they are sent.
  - If a client updates the value of znode /z to v_1 and later to v_2 no other client will see /z with a value v_1 after it has seen it with value v_2 (provided no other updates to z take place)
  - Variation of eventual consistency
- **Atomicity:** Updates either succeed or fail
  - If an update fails, no client will ever see it.
- **Durability:** Once an update succeeds it will persist
  - Updates will survive server failures.

Example: Group Membership

- A group keeps track of the presence of a set of clients.
  - Assume every client has a unique id <client-id>

- **createGroup**() {
  create("/group-name", "", PERSISTENT)
  }

- **joinGroup**() {
  create("/group-name/"+client-id, "address", EPHEMERAL)
  }

- **getGroupMembers**() {
  getChildren("/group-name/", false)
  }

Example: Distributed Lock

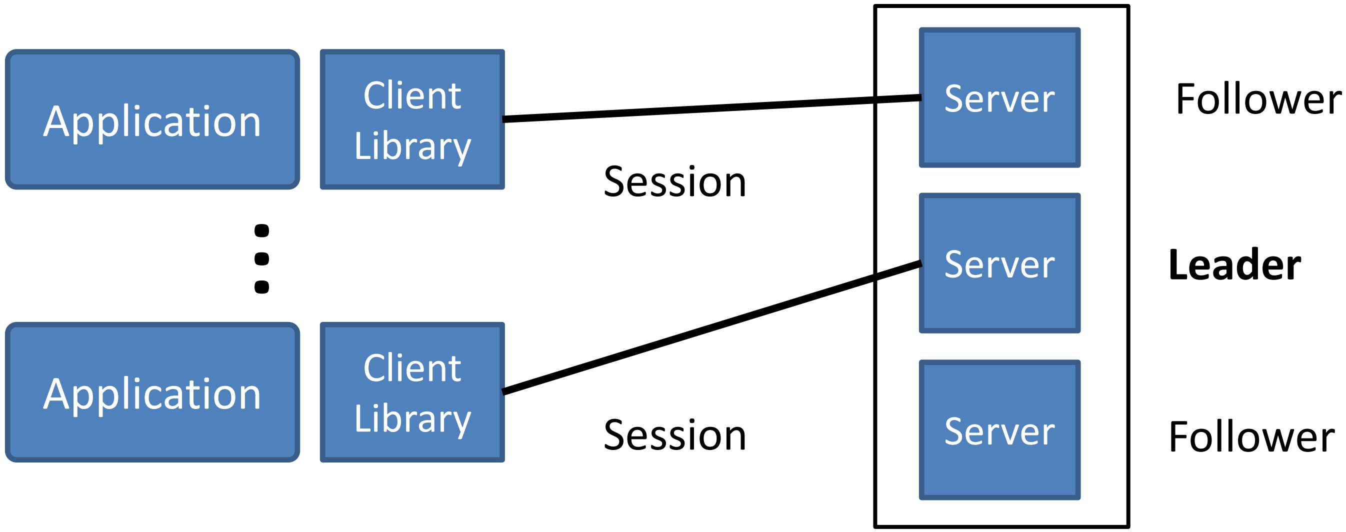
- **aquireLock**(lock-name) {

```
aquireLock(lock-name) {
    while (true) {
        create(lock-name, "", EPHEMERAL)
        if create is successfull
            return
        else {
            getData(lock-name, TRUE)
            wait for watch
        }
    }
}
```

- **releaseLock**(lock-name) {

```
releaseLock(lock-name) {
    delete (lock-name)
}
```

ZooKeeper Architecture (Quorum Mode)



- Read operations are processed locally by the server to which the client is connected.
- All write operations are forwarded to the **leader** which performs a totally ordered broadcast operation to replicate the data within the ensemble.
  - ZAB (Zookeeper Atomic Broadcast)

Quorum based Write-Operations

- ZooKeeper replicates data across all servers in the ensemble.
  - Problem: If a client had to wait for every server to store its data before continuing, the delays might be unacceptable.
- ZooKeeper employs a **majority quorum for write operations**:
  - When the majority of servers has finished storing a client's data the client is informed that the operation succeeded.
    - Other servers will eventually catch up and finish storing the data.
  - With 2f + 1 servers f failures of servers can be tolerated.
    - For an ensemble of 5 servers the quorum is 3. Thus, a loss of 2 servers can be tolerated.

Leader Operation (Simplified)

- **Phase 1: Leader election**
  - The servers in an ensemble select a leader employing a round based protocol.
    - Protocol guarantees that no follower has data with a higher transaction ID than the leader.
  - Phase ends when a majority of followers have synchronized their state with the leader.
- **Phase 2: Zookeeper Atomic Broadcast (ZAB)**
  - All write requests are forwarded to the leader, which broadcasts the update to the followers.
  - When a majority of followers has persisted the data the leader commits the update and informs the client.