Zookeeper

Zookeeper

- Goals:
  - Allow developer of a cloud computing application to concentrate on the application logic
    - Instead of on coordination and failure handling
  - Uses a simple API modeled on a file system API

Zookeeper

- Hadoop's distributed coordination server
- Design Goals
  - Simplicity
    - Distributed processes coordinate through a shared hierarchical namespace — znodes
  - Reliability
    - Uses replication

Zookeeper Mission

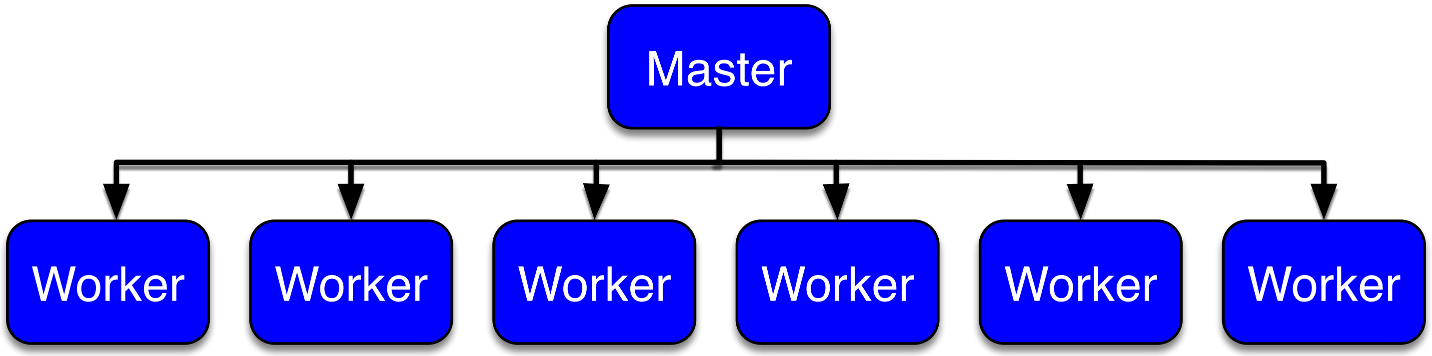
- Strong consistency, ordering, and durability guarantees
- The ability to implement typical synchronization primitives
- A simpler way to deal with many aspects of concurrency that often lead to incorrect behavior in real distributed systems

Zookeeper Mission

- Distributed systems are difficult because of
  - Message delays
  - Processor delays
  - Clock drifts

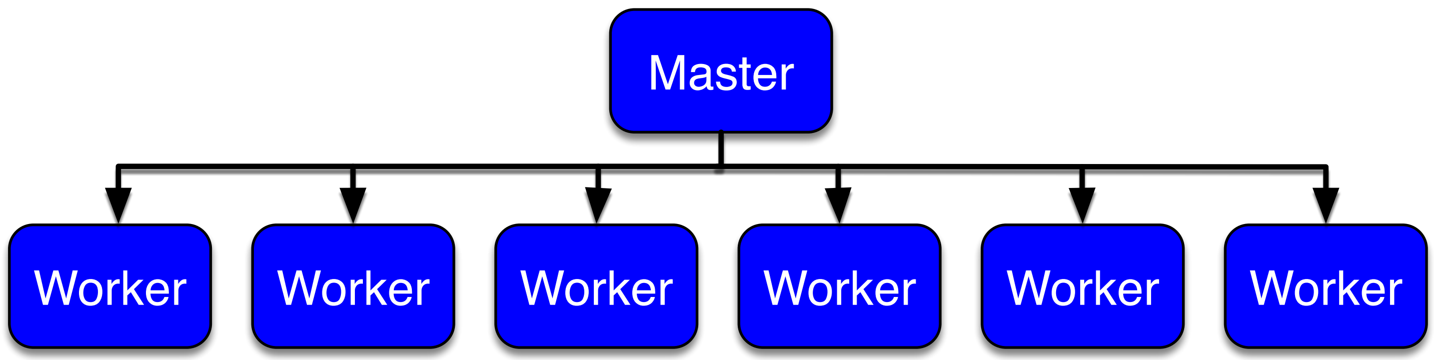
Zookeeper Example

- A simple master-worker architecture



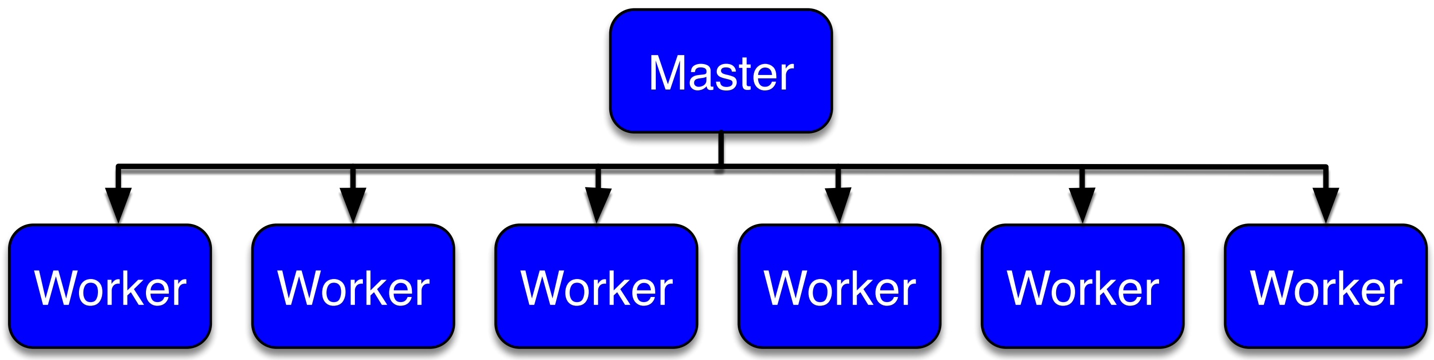
- Three fundamental problems:
  - Master crashes
  - Worker crashes
  - Master-worker communication fails

Zookeeper Example



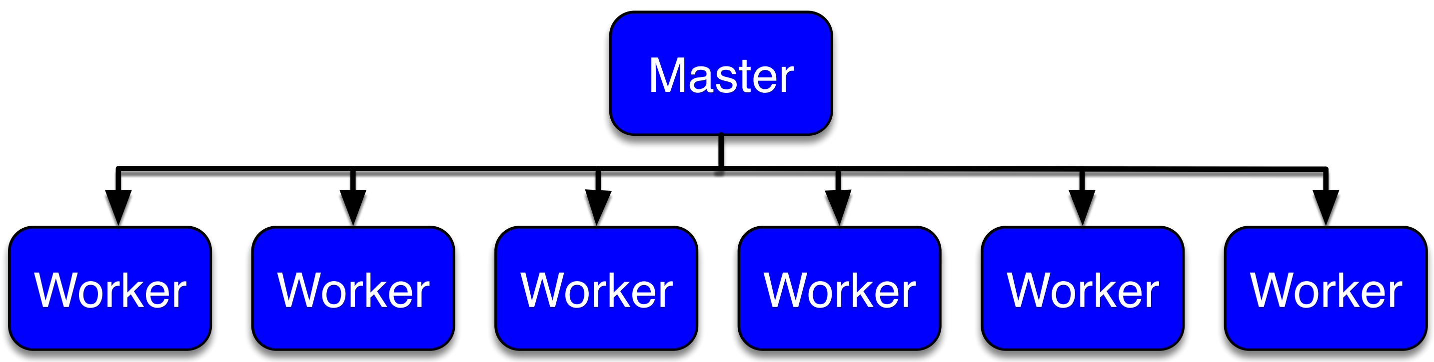
- Master failure:
  - Need a back-up master
  - But all need to agree on a take-over
  - Need to restore state of the failed master

Zookeeper Example



- Worker failure:
  - Master needs to detect worker failure
  - Master needs to replace the worker
  - Replacement worker might need to clean up
    - Work could have side effects, such as changing database tables

Zookeeper Example



- Communication failure:
    - Two workers can now be assigned the same task after reassignment
        - Problem: Need exactly-once semantics, but can only get at-least-once or at-most-once

Zookeeper

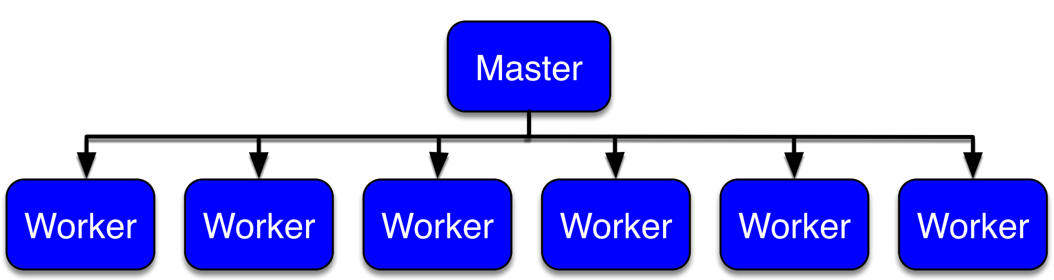
- Other solutions:
  - Amazon simple queue service
    - Provides just queuing
  - Protocols for leader election
  - Protocols for common configurations
  - Chubby for locking with strong synchronization guarantees

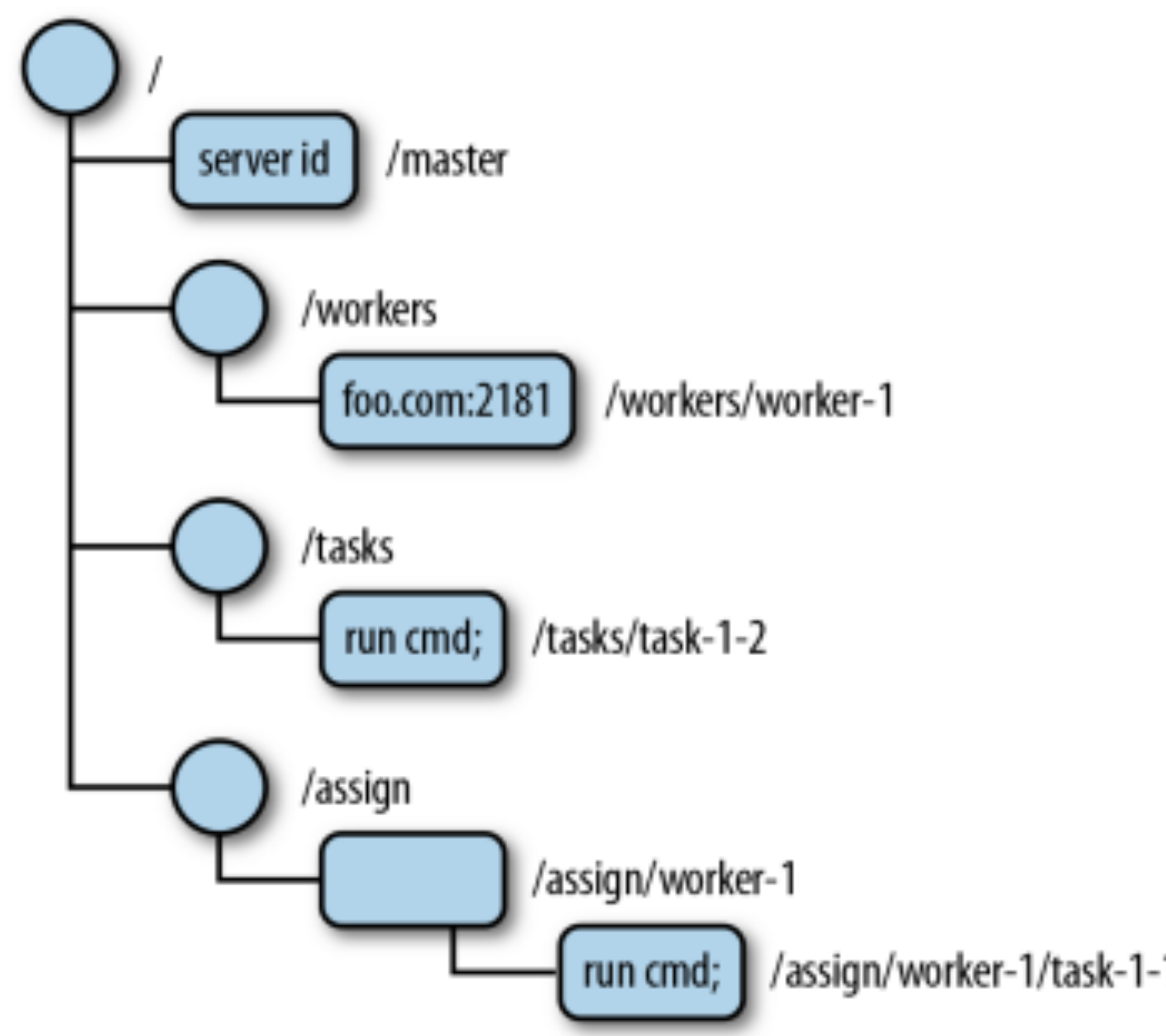
Zookeeper Example

- Thus:
  - Master election
  - Crash detection
  - Group membership
  - Metadata management
- But: no ideal solution possible

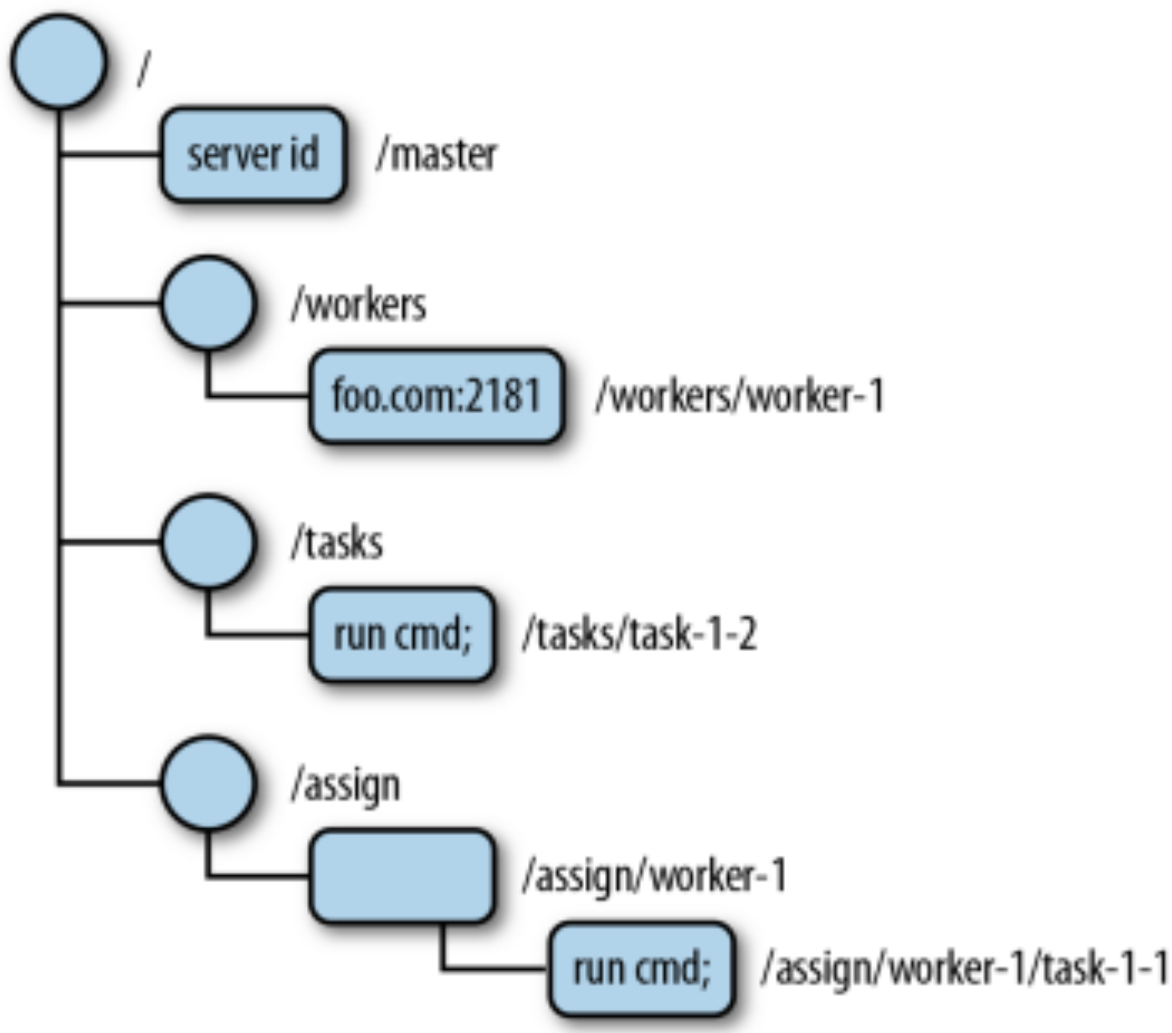
Running Zookeeper Basics

- Zookeeper does not provide primitives
- Uses recipes
- Recipes manipulate small data structures
  - z-nodes



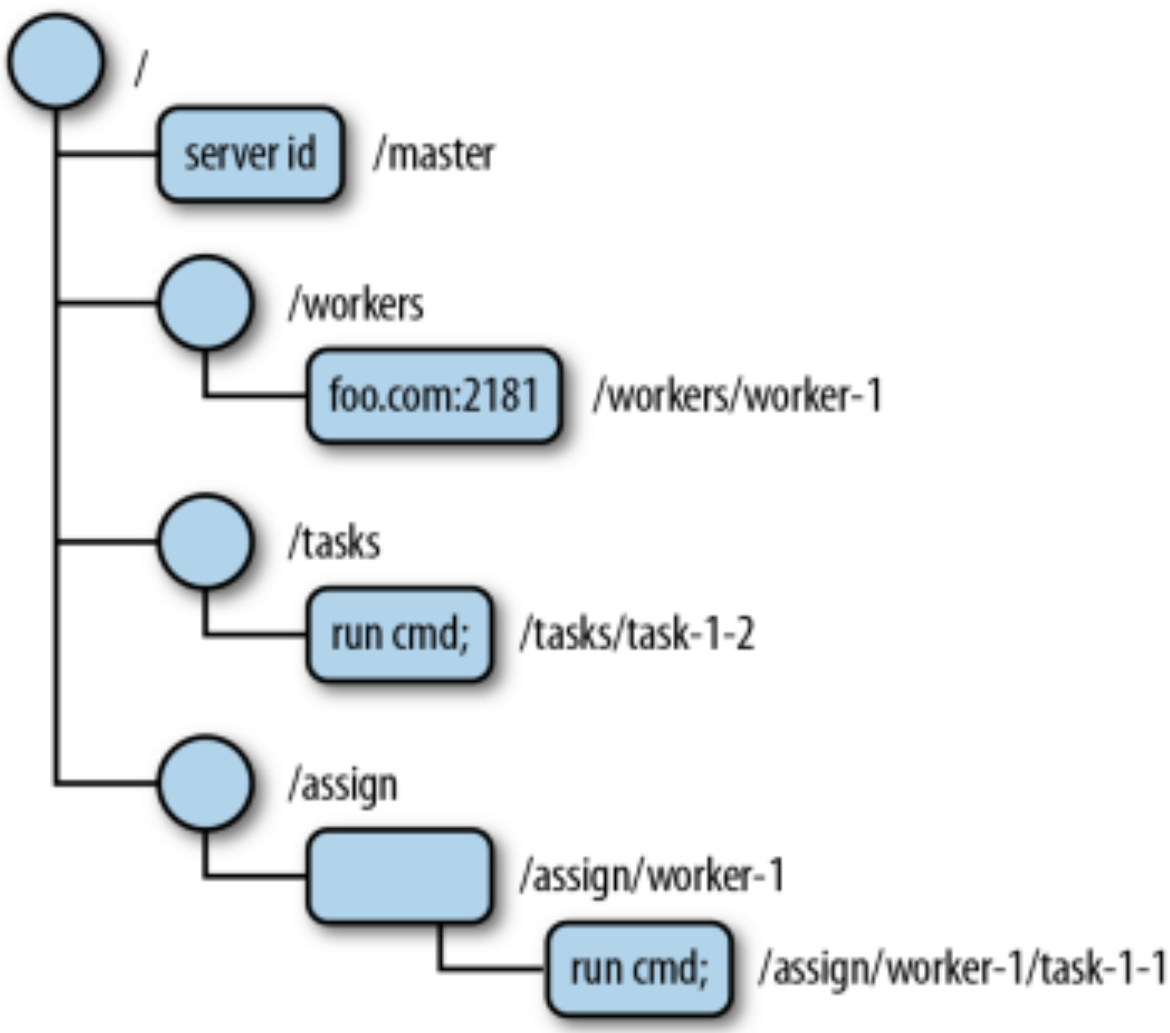


Zookeeper Basics



- There is no data in /master:
  - No master is currently assigned
- There is one node in /worker
  - One worker is assigned
- There is one task, which is assigned to the sole worker

Zookeeper Basics



- Clients will add znodes to the /tasks node
- When there is a master, the master can assign tasks to a worker by adding to the /assign node

Running Zookeeper Basics

- Zookeeper API
  - `create /path data`
    - Creates a znode named with /path and containing data
  - `delete /path`
    - Deletes the znode /path
  - `exists /path`
    - Checks whether /path exists

Zookeeper Basics

- `setData /path data`
  - Sets the data of znode /path to data
- `getData /path`
  - Returns the data in /path
- `getChildren /path`
  - Returns the list of children under /path

Zookeeper Basics

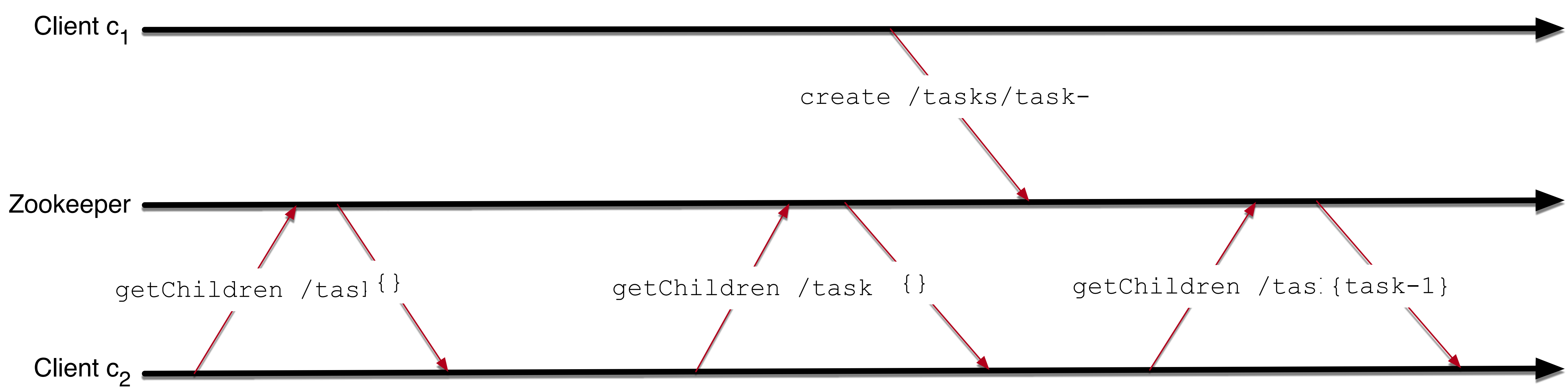
- Persistent / ephemeral Nodes
  - A persistent node /`path` can only be deleted with an explicit call /`delete /path`
  - Ephemeral nodes vanish
    1. If the process that created it has crashed or closed its zookeeper connection
    2. It has been deleted explicitly

Zookeeper Basics

- Sequential znodes:
  - A sequential znode is assigned a unique, monotonically increasing integer.
  - Sequential znode sequence numbers are attached to the path
- Example:
  - Client creates a sequential znode with the path /tasks/task-
  - First node is /tasks/task-1

Zookeeper Basics

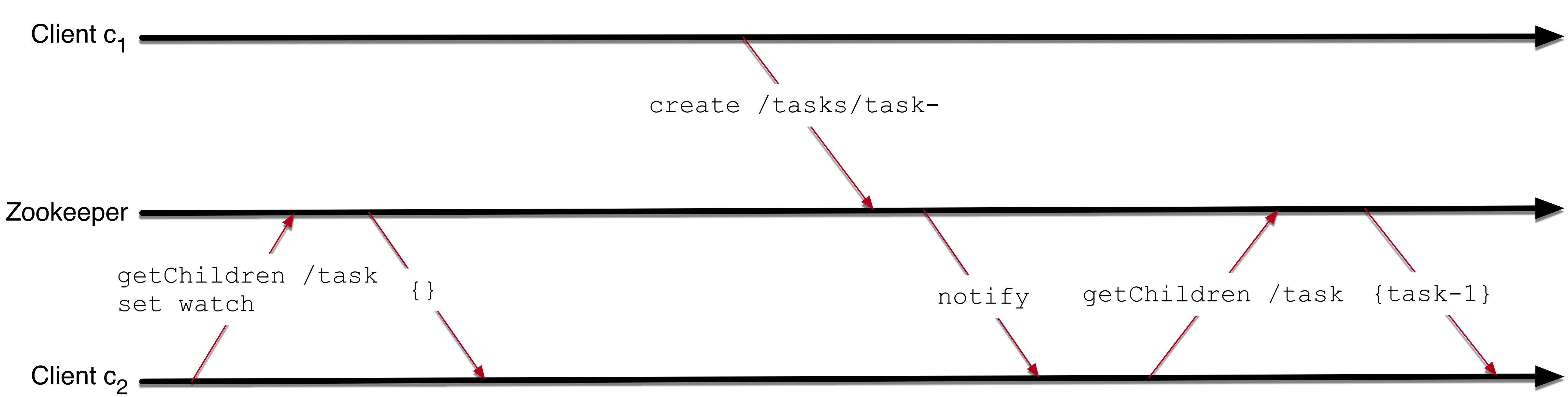
- Watches
  - Client based polling loads the communication layer



Client 2 polls a Zookeeper node until a task becomes available

Zookeeper Basics

- Watches
  - Zookeeper allows notifications



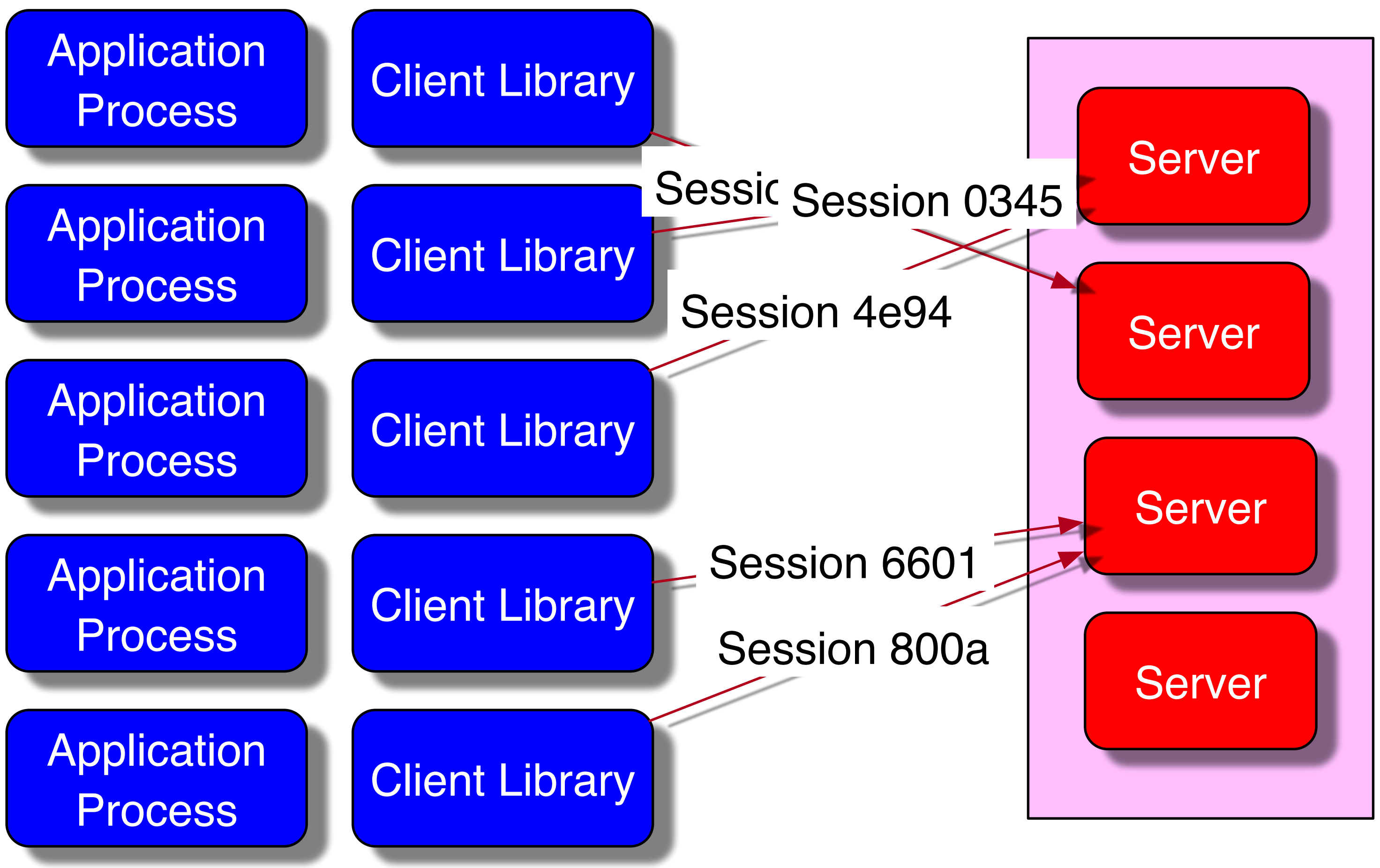
Zookeeper Basics

- Zookeeper notifications can be missed
  - Clients need to check before setting watches
  - Example:
    - Client 1 sets a watch
    - Client 2 adds a node
    - Client 1 receives the notification
    - Client 3 adds a node
    - Client 1 sets a watch
  - At this point, Client 1 does not receive a notification

Zookeeper Basics

- Versions
  - All znodes have a version number
    - `setData` and `delete` can be made conditional on the version number

Zookeeper Basics



Zookeeper Architecture:
Applications make calls to Zookeeper
servers via the Client Library

Zookeeper Basics

- Client API
  - create(path, data, flags)
  - delete(path, version)
  - exists(path, watch)
  - getData(path, watch)
  - setData(path, data, version)
  - getChildren(path, watch)
  - sync(path)
    - waits for all pending updates to propagate to servers

Zookeeper Basics

- Client API
  - Synchronous API for single ZooKeeper operations
  - Asynchronous API if there are outstanding operations and other tasks are executed in parallel
    - Client then has to guarantee that callbacks are invoked in order

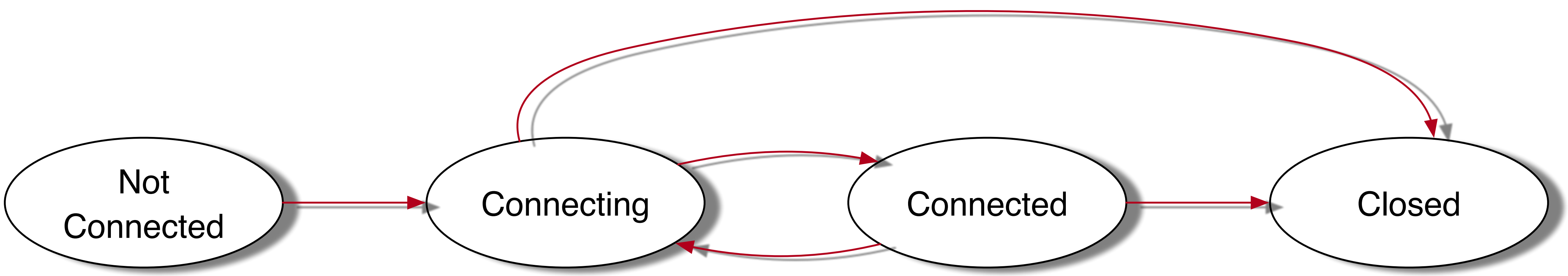
Zookeeper Basics

- Zookeeper servers run in either
  - *standalone mode*
    - Single server, no failure tolerance
  - *quorum mode*
    - Data tree is replicated across all servers
    - Quorum is the number of servers needed to acknowledge
      - Zookeeper allows assigning weights to nodes
      - A quorum needs to have combined weight

$$> \sum_{s \text{ Server}} w(s)/2$$

Zookeeper Basics

- Zookeeper clients establish sessions



  - If a client looses its connection or there is a timeout, it moves into the “Connecting” state again.

- Only the zookeeper servers can close a session

Zookeeper Basics

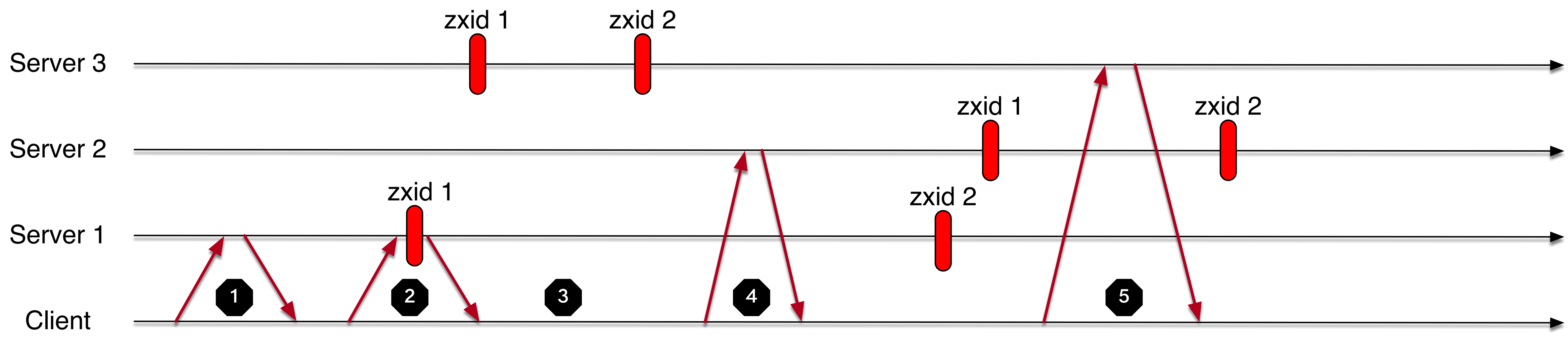
- Session time t
  - Zookeeper servers after t time closes section closed
  - Client at $t/3$ sends a heartbeat message to the server
  - Client at $2t/3$ accesses a different server

  - *Accessing a different server needs care*

Zookeeper Basics

- Accessing a different server:
  - Client cannot connect to a server that has not seen an update that the client has seen
  - Zookeeper orders all updates to servers
    - Done using transaction identifiers

Zookeeper Basics



1: Client connects to Server 1

2: Client creates a znode. Transaction zxid 1 is assigned by the server. The transaction reaches Server 3.

3: Client gets disconnected from Server 1.

4: Client tries to connect to Server 2. However, client has seen zxid 1, but Server 2 has a lower number. The connection fails.

5: Client tries to connect to Server 2. Server 2 has zxid 2, so the connection succeeds.

Locks

- Implementing simple locks:
  - Some processes want to get a lock
    - Process p creates a znode `/lock`
    - If it succeeds, then p has the lock
    - The lock is ephemeral: If p dies, the lock will be released

Locks

- Implementing locks
  - Any other process cannot create the same znode
  - They can set a watch in order to get notified if the znode vanishes

A Master Worker Example

- Master
    - watches for new workers and tasks
    - assigns tasks to workers
- Workers
    - register themselves as available
    - watch for new tasks assigned to them
- Client
    - creates new tasks and wait for responses from the system

A Master Worker Example

- There can only be one master
  - Therefore, the master process acquires a lock with an ephemeral node `/master`
  - The client is free to create a back-up master that sets a watch for `/master`.
  - Whenever the backup master detects the vanishing of the `/master` node, it can acquire it and become the new master

A Master Worker Example

- The client or a bootstrap procedure now creates
  - `/workers`
  - `/tasks`
  - `/assign`
- These nodes are all permanent
- The master also creates watches

A Master Worker Example

- A worker creates an ephemeral node under `/worker`
  - With its contact information
    - As the node is created, the master is notified
- The worker creates a node
    - `/assign/worker1.example.com`
  - and sets a watch (by using ls)
    - `ls /assign/worker1.example.com true`

A Master Worker Example

- The client adds tasks to the system
  - This is done by creating a znode in the `/task` directory
    - With version number
      - `create -s /tasks/task- “command"`
  - The client needs to set a watch for the creation of a status node
    - Created by the worker once the task is done

A Master Worker Example

- When a task is created:
  - The master is notified
    - The master looks at the available workers
    - The master creates an assignment znode to assign the task
      - `create /assign/worker1.example.com/task-0000000000 “"`
    - The worker receives a notification as the worker watches assignments

A Master Worker Example

- Once the worker has finished the task:
  - Worker creates a status znode under `/tasks`
  - The client is notified
  - The client can access the results

Zookeeper API

- Zookeeper uses primarily a Java interface
    - E.g. a zookeeper handle is created via

```
ZooKeeper(
     String connectString,
     int sessionTimeout,
     Watcher watcher)
```

- The watcher needs to be implemented

Zookeeper API

- Watcher interface:

```
public interface Watcher {
    void process(WatchedEvent event);
}
```

```java
import java.io.IOException;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.ZooKeeper;

public class Master implements Watcher {
    ZooKeeper zk;
    String hostPort;

    Master(String hostPort) {
        this.hostPort = hostPort;
    }

    void startZK() throws IOException {
        zk = new ZooKeeper(hostPort, 15000, this);
    }

    public void process(WatchedEvent e) {
        System.out.println(e);
    }

    public static void main(String args[])
        throws Exception {
        Master m = new Master(args[0]);

        m.startZK();

        // wait for a bit
        Thread.sleep(60000);
    }
}
```

Zookeeper API

- State Changes:
  - Event: execution of an update at a znode
  - Notification: executed by a watch and sent to the watcher
  - Example:
    - The client executes an exists operation on /z with the watch flag set and waits for the notification.
    - The notification comes in the form of a callback to the application client.

Zookeeper API

- Between a notification and setting another watch, events can be missed
- Usually not a problem:
  - Events change the state of the watched znode
  - znodes have versions
- All read commands `getData`, `getChildren`, and `exists` can set watches

Zookeeper API

- Multiops:
  - Not in the original Zookeeper
  - Allows to bundle operations that are executed atomically
  - Example Master-Worker: Can bundle task assignment and task deletion from the todo-list.
  - Example: Checking version numbers

Zookeeper API

- Caching:
  - Zookeeper decided against transparent caches
  - Applications need watches to maintain cache coherency

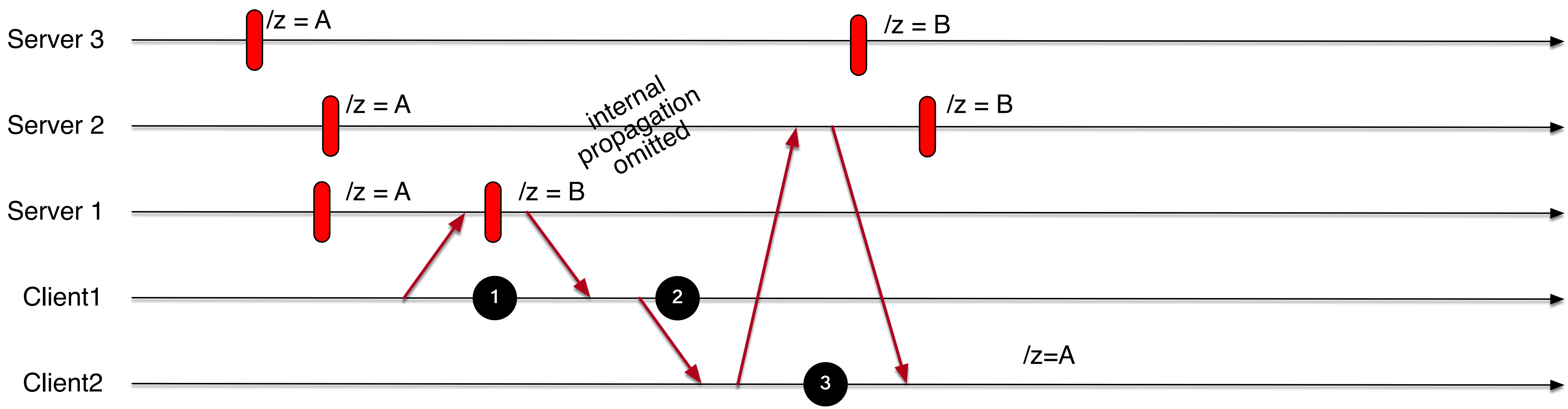
Zookeeper API

- Ordering:
  - Zookeeper servers agree on order of state changes
  - Apply them in the same order
  - But not necessarily at the same time

  - Clients can observe this if they can use hidden channels

Zookeeper API

- Hidden channel example



1. Client 1 updates /z
2. Client 1 sends a message to Client 2
3. Client 2 reads /z and receives an outdated value

Zookeeper API

- Ordering is true for notifications:
  - Example:
    - Update u to z/a
    - Update u' to z/b
    - If a client has set a watch for z/a and reads z/b:
      - Client is guaranteed to see the notification for z/a before the read result to z/b
- This allows clients to implement safety checks

Zookeeper API

- Example:
  - Configuration data in a number of znodes:
    - /config/m1, /config/m2, /config/m3, /config/m4
  - Master needs to update these znodes simultaneously
    - Master creates a znode /config/invalid
    - Master updates the other znodes
    - Master removes the /config/invalid znode
  - Clients can watch for /config/invalid and are guaranteed to only read znodes in /config that are consistent.

Zookeeper API

- Herd effect
  - Watches can be dangerous
  - Spike in load if a much watched znode changes state

Zookeeper API

- Example:
  - A large number of known clients want to get a lock
  - Clients create sequential znodes /lock/lock-
  - Client gets sequence number by
    - `getChildren(/lock)`
  - If a client has the smallest sequence number, it has the lock
  - Otherwise, the client watches for the next-smallest sequence number

Zookeeper API

- The client that created `/lock/lock-001` has the lock.
- The client that created `/lock/lock-002` watches

  `/lock/lock-001`.
- The client that created `/lock/lock-003` watches

  `/lock/lock-002`.

Zookeeper API Failures

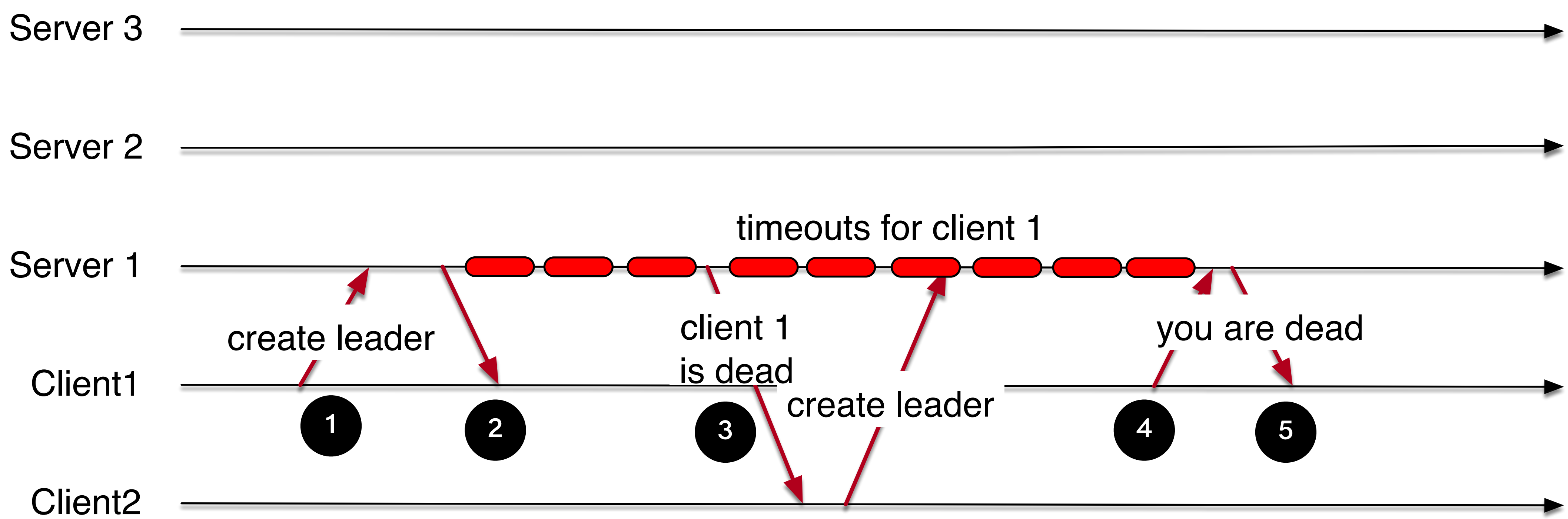
- Recoverable versus unrecoverable failures
  - Recoverable failures are transient
  - Example: A disconnected client tries to reconnect:
    - Once the session is reestablished:
      - ZooKeeper will generate a SyncConnected event and start processing requests
      - Zookeeper reregisters all watches
      - Zookeeper generates watch events that were missed

Zookeeper API

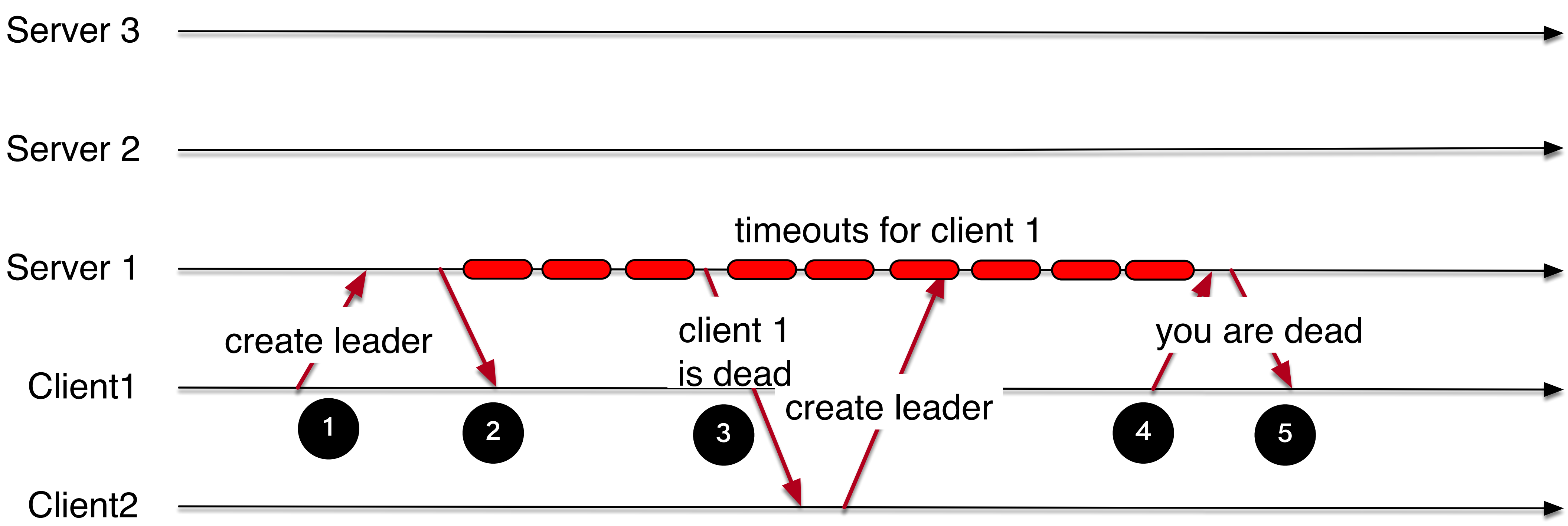
1. Client 1 creates an event
2. Server 2 has a network problem
3. Client 1 reconnects to server 3
4. Client 1 reissues the event

Zookeeper API

- Reconnections need to be handled well in order to not generate spikes after network failures
- Example: A client is a leader



Zookeeper API



1: Client 1 is the leader
2: Client 1 is disconnected
3: After the timeout, Zookeeper selects another leader
4: Client 2 accepts leadership
5: Client 1 reconnects and foolishly creates events
6: Client 1 finds out that it is declared dead

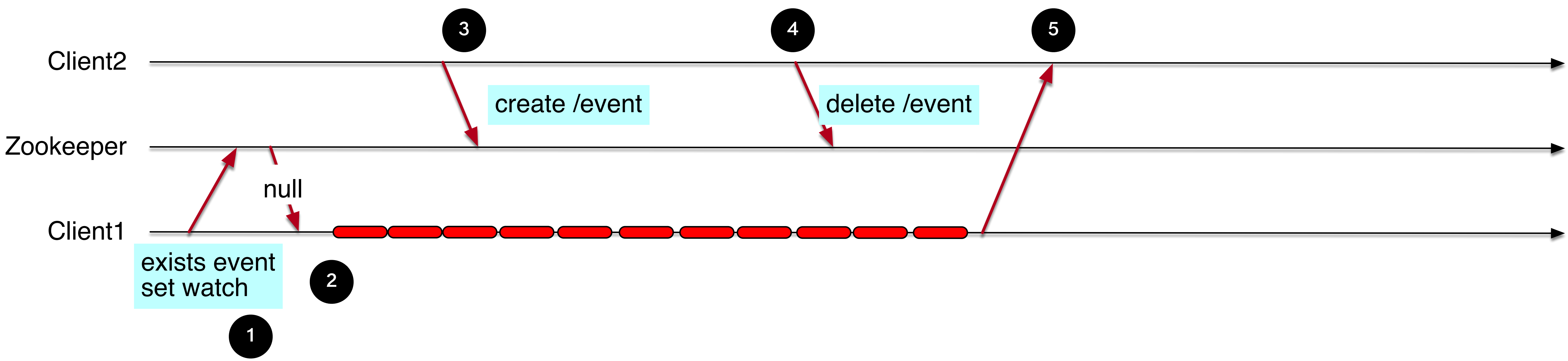
Zookeeper API

- Moral:
  - Clients need to take the “Disconnected” message seriously

Zookeeper API

- Upon reconnection:
    - Client library takes care of outstanding watches and the last zxid seen
    - Servers will go through the list of watches, check timestamps and regenerate missed notifications
    - **CAVEAT**: We can miss an exists event

Zookeeper API



1. Client 1 checks for /event and sets a watch
2. Zookeeper says that /event does not exist, afterwards disconnection.
3. Client 2 creates /event
4. Client 2 deletes /event
5. Client 1 reconnects and resets watches

Client 1 has missed out on the event

Zookeeper API

- Irrecoverable failures:
  - A session expires
  - The authentication information is no longer valid
  - Zookeeper will then loose all state information

Zookeeper API

- Zookeeper cannot protect external devices fully
- Real life example:
    - We use Zookeeper to create a leader
    - Because of Java memory crunch, Java garbage collection runs, so leader still thinks that session is valid
    - However, Zookeeper has selected another leader
    - Old leader continues to behave like the leader and sends off queued requests
    - Only then does the old leader discovers that Zookeeper has appointed another leader

Zookeeper API

- Fencing:
  - Ensures exclusive access
  - Fencing with a token
    - Leader selection with Zookeeper returns a STAT structure with a sequential czxid
    - This is the fencing token
    - When a new leader is selected, the czxid has increased
    - If the new leader interacts with a resource, it will use the new token
    - The resources will not accept commands from the old leader afterwards

Zookeeper API

- Caveats:
    - When a znode is deleted and recreated, its version number is reset
    -

Zookeeper API

- Ordering in the presence of failures:
    - If there is a connection loss event, Zookeeper cancels pending operations
        - This allows reordering of operations

1. Application submits a request to execute Op1.
2. Client library detects a connection loss and cancels pending request to execute Op1.
3. Client reconnects before the session expires.
4. Application submits a request to execute operation Op2.
5. Op2 is executed successfully.
6. Op1 returns with CONNECTIONLOSS.
7. Application resubmits Op1.

Requests, Transactions, and Identifiers

- ZooKeeper servers process read requests (exists, getData, and getChildren) locally.
- Client requests that change the state of ZooKeeper (create, delete, and setData) are forwarded to the leader.
  - Leader produces a state update, a *transaction*
  - Transactions are *idempotent*
  - ZooKeeper transactions get an ID *(zxid)*
  - Transactions are strictly ordered
    - Originally by using a single thread at the leader

Leader Selection

- Leaders are responsible for ordering operations that change the state of Zookeeper
    - `create, setData, delete`
- Leaders are unique because they need support by a quorum

Leader Selection

- Each server starts in the *Looking* state
  - If there is already a leader, the server moves to the *Follower* state
  - Otherwise, there is a leader election
  - The winning server enters the *Leading* state, otherwise the *Follower* state

Leader Selection

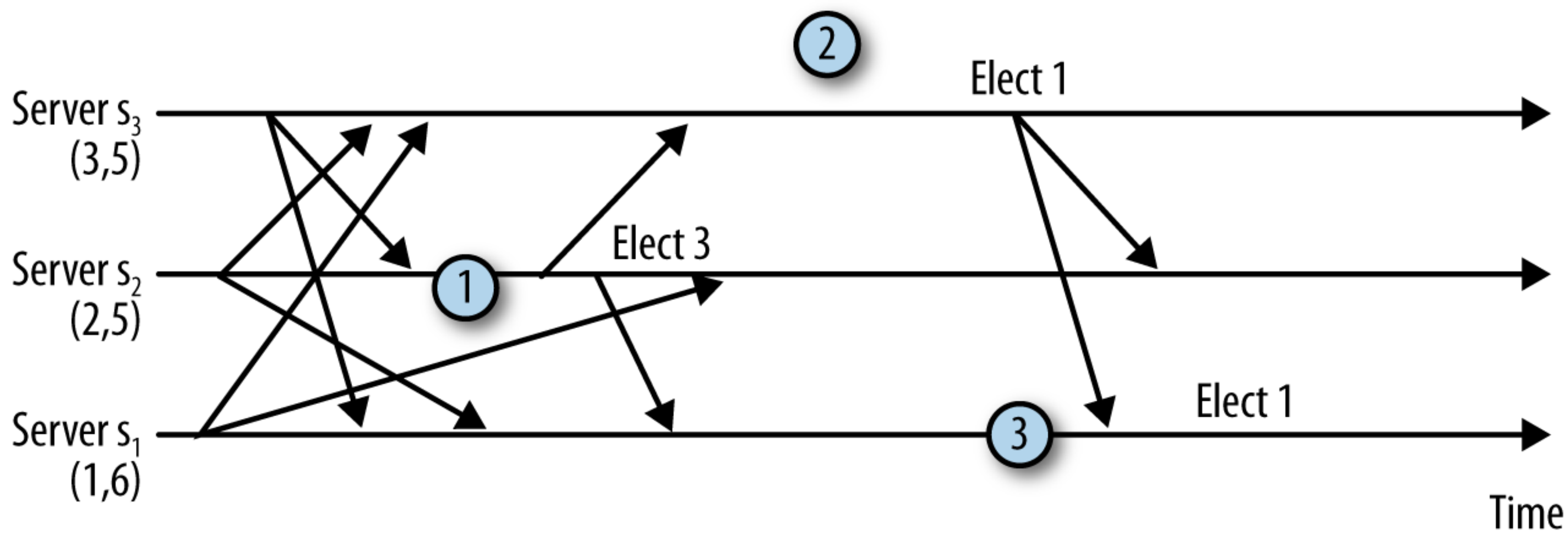
- A server in *Looking* state sends leader notifications to all servers
- Each servers sends a vote consisting of its server identity (SID) and the most recent transactions it has executed zxid
- If a server receives a leader notification (voteSID, voteZXID) and has itself (mySID, myZXID)
  - If `(myZXID > voteZXID) or (myZXID = voteZXID and mySID > voteSID)` then keep the current vot
  - Otherwise, switch to `(voteSID voteZxid)`

Leader Selection

- Once a server receives the same vote from a majority of servers, the leader has been selected
- As soon as possible, bring followers up to the state of the leader

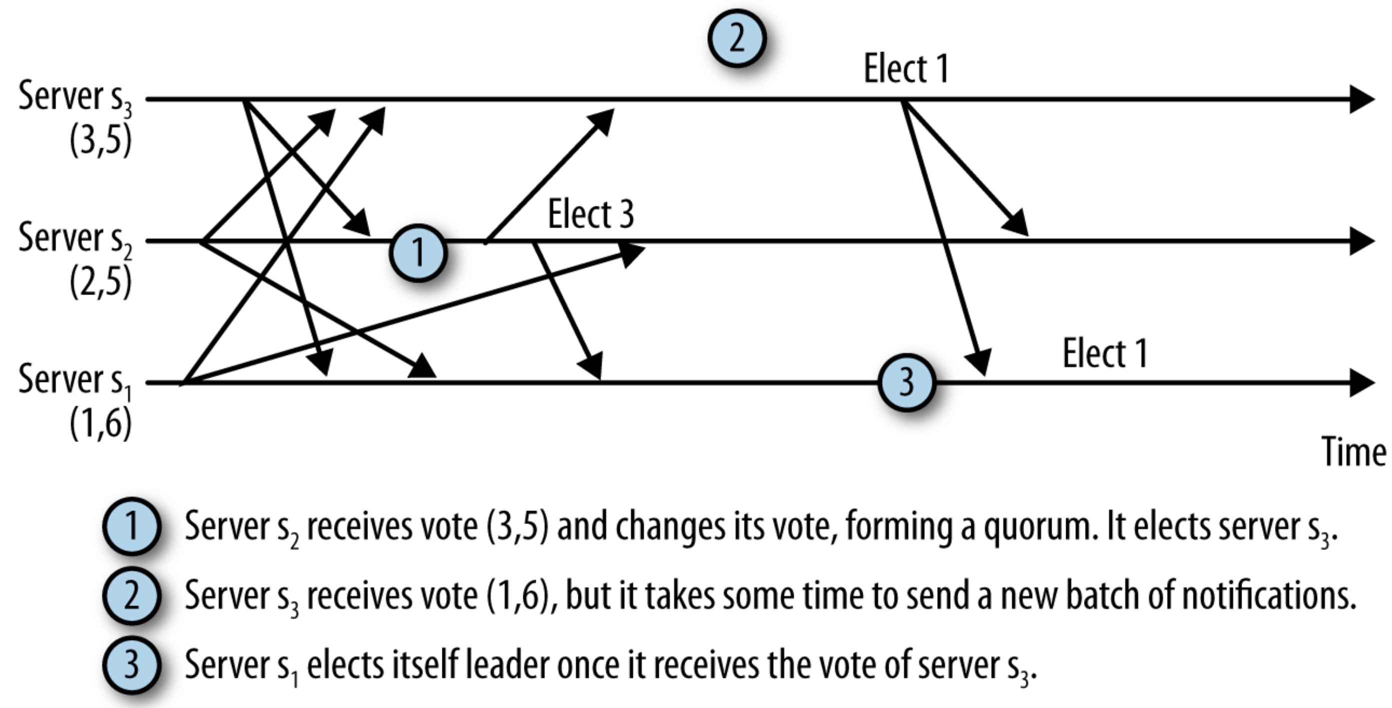
Leader Selection

- Leader election is not guaranteed to be unanimous:



1. Server s_2 receives vote (3,5) and changes its vote, forming a quorum. It elects server s_3.
2. Server s_3 receives vote (1,6), but it takes some time to send a new batch of notifications.
3. Server s_1 elects itself leader once it receives the vote of server s_3.

Leader Selection



Having s2 elect a different leader does not cause the service to behave incorrectly, because s3 will not respond to s2 as leader. Eventually s2 will time out trying to get a response from its elected leader, s3, and try again. Trying again, however, means that during this time s2 will not be available to process client requests, which is undesirable.

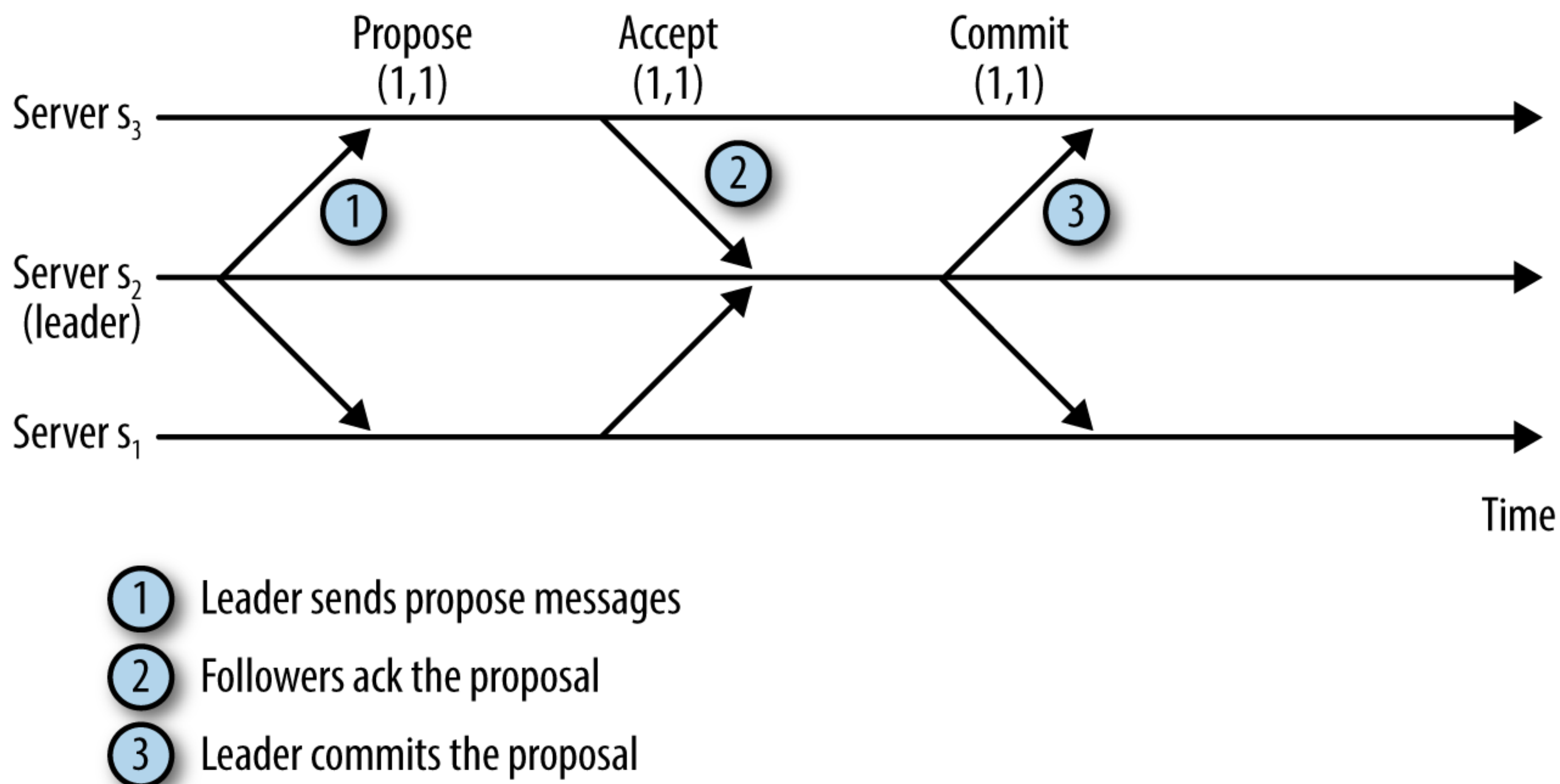
Leader Selection

- Falsely electing a leader can prolong recovery time.
- Leader election might need some time
  - FastLeaderElection uses 200 msec
    - Compromise between maximum network delay in a data center and short enough to not influence recovery time visibly

ZAB: Zookeeper Atomic Broadcast

- Upon receiving a write request:
  - Follower forwards to leader
  - Leader executes the request speculatively
  - Leader broadcasts the result of the execution as a state update (transaction)
  - Uses 2-phase commit

ZAB


Propose
(1,1)
Accept
(1,1)
Commit
(1,1)
Server s_3
Server s_2
(leader)
Server s_1
1
2
3
Time
1 Leader sends propose messages
2 Followers ack the proposal
3 Leader commits the proposal

ZAB

- If a server commits T before T′, then any server that commits T and T′ must also commit T before T′.
- If a server commits T and T′ and commits T first, then any server that commits T′ must commit T first.

ZAB

- Transactions can still end up on some servers and not on others
  - because servers can fail while trying to write a transaction to storage.
- ZooKeeper brings all servers up to date whenever a new quorum is created and a new leader chosen.

ZAB

- ZAB transaction number consists of an epoch and a sequence number
- Epoch number is incremented whenever there is a leader change

ZAB

- Split Brain:
  - Having two servers that believe they are leaders
- Split Brains are difficult to avoid, but ZAB promises
  - An elected leader has committed all transactions that will ever be committed from previous epochs before it starts broadcasting new transactions.
  - At no point in time will two servers have a quorum of supporters.