Big Data
- Zookeeper

Dr. Qing “Matt” Zhang
ITU

Outline

- Overview of Zookeeper
- Use cases and examples
- Essential internals
- Related works and references

Introduction

What is Zookeeper?

- A scalable, distributed, open-source coordination service for distributed applications
- Provides a simple set of primitives to implement higher level services for synchronization, configuration maintenance, consensus, leader election, groups and naming in a distributed system.

Why Use Zookeeper?

- If you love to have sleepless nights debugging distributed synchronization problems – please ignore the rest of the presentation
- Difficulty of implementing distributed services
  - Complex distributed algorithms are notoriously difficult to implement correctly
  - Prone to race conditions and dead locks. And distributed deadlocks are the worst!
  - Different implementations lead to management complexity when the applications are deployed

Why Use Zookeeper?

- Other programming models using distributed locks or State Machine Replication are difficult to use correctly
- Zookeeper solves these problems for us by providing a simple and already familiar programming model
- Zookeeper provides reusable code libraries for common use cases – very easy to use

Who uses Zookeeper?

- Deepdyve - Does search for research and provide access to high quality content using advanced search technologies. ZK is used to manage server state, control index deployment and a myriad other tasks
- Katta - Katta serves distributed Lucene indexes in a grid environment. ZK is used for node, master and index management in the grid
- 101tec – Does consulting in the area of enterprise distributed systems. Uses ZK to manage a system build out of hadoop, katta, oracle batch jobs and a web component

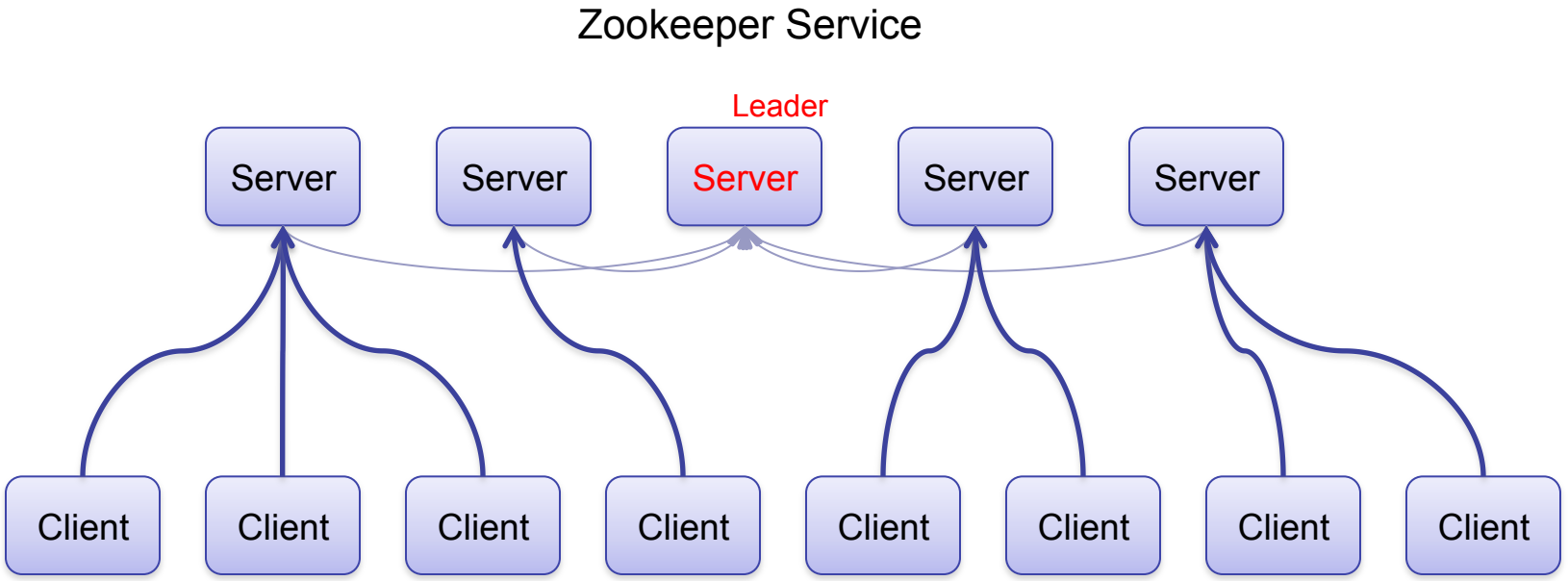
Who uses Zookeeper?

- Hbase – HBase is an open-source distributed column-oriented database on hadoop. Uses ZK for master election, server lease management, bootstrapping, and coordination between servers.
- Rackspace – Email & Apps team uses ZK to co-ordinate sharding, handling responsibility changes, and distributed locking

Design Goals

- Reliability
- Availability
- Concurrency
- Performance
- Simplicity

How does it look like?



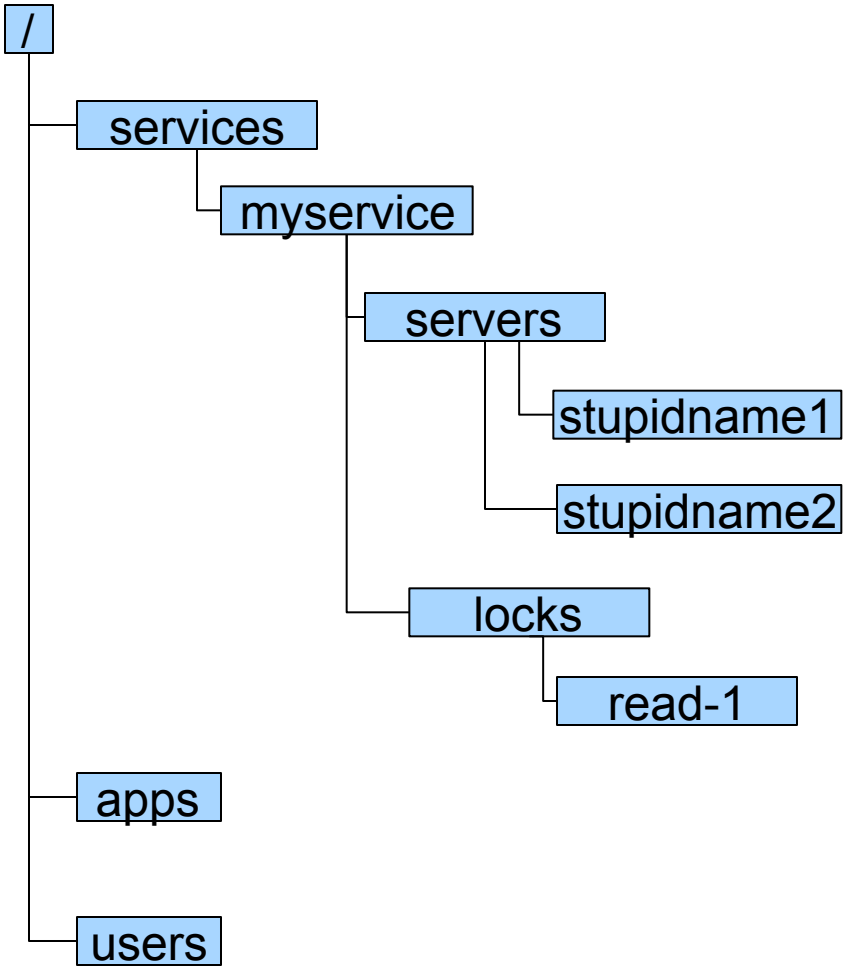
- Data is stored in-memory in all the servers
- A leader is elected at start-up
- Followers service clients, all updates go through leader
- Update responses are sent when a majority of servers have persisted the change

Basic Concepts

- Allows distributed processes to coordinate with each other through a shared hierarchal namespace which is organized similarly to a standard file system
- The namespace consists of data registers – znodes
- Provides a very simple API and programming model
- The API is similar to that of a file system, but different
- Data is kept in memory – high throughput and low latency
- Provides strictly ordered updates and accesses
- Provides certain guarantees for the operations, based on which higher level concepts can be built
- Supports additional features such as change notification (watches), ephemeral nodes and conditional updates

Data Model

- Hierarchical data model, much like a standard distributed file system
- Nodes are known as znodes, and identified by a path
- znode can have data associated with it, and children. Data is in KBs
- znodes are versioned
- Data is read/written in its entirety
- znodes can be ephemeral nodes – exists as long as the session that created it is active
- Watches can be set on znodes
- Auto generation of file names



Zookeeper API

- String create (path, data, acl, flags)
- void delete (path, expectedVersion)
- Stat setData (path, data, expectedVersion)
- byte[] getData (path, watch)
- Stat exists (path, watch)
- String[] getChildren (path, watch)
- void sync (path)

Zookeeper Session

- ZK client establishes connection to ZK service, using a language binding. (Java, C, Perl, Python, REST)
- List of servers provided – retry the connection until it is (re)established
- When a client gets a handle to the ZK service, ZK creates a ZK session, represented as a 64-bit number
- If reconnected to a different server within the session timeout, session remains the same
- Session is kept alive by periodic PING requests from the client library

Ephemeral Nodes, Watches

- Ephemeral nodes
  - Present as long as the session that created it is active
  - Cannot have child nodes
- Watches
  - Clients can set watches on znodes.
    - Changes to that znode trigger the watch and then clear the watch.
    - When a watch triggers, ZooKeeper sends the client a notification.
    - One time trigger. Have to be reset by the client if interested in future notifications

Ephemeral Nodes, Watches

- Watches
  - Not a full fledged notification system.
    - Client should verify the state after receiving the watch event
  - Ordering guarantee: a client will never see a change for which it has set a watch until it first sees the watch event
  - Default watcher – notified of state changes in the client (connection loss, session expiry, …)

Guarantees

- Since its goal is to be a basis for the construction of more complicated services such as synchronization, it provides a set of guarantees
- Sequential Consistency - Updates from a client will be applied in the order that they were sent
- Atomicity - Updates either succeed or fail. No partial results
- Single System Image - A single client will see the same view of the service regardless of the server that it connects to
- Reliability - Once an update has been applied, it will persist from that time forward until a client overwrites the update
- Timeliness - The clients view of the system is guaranteed to be up-to-date within a certain time bound

Use cases

- Use cases inside Yahoo!
  - Leader Election
  - Group Membership
  - Work Queues
  - Configuration Management
  - Cluster Management
  - Load Balancing
  - Sharding

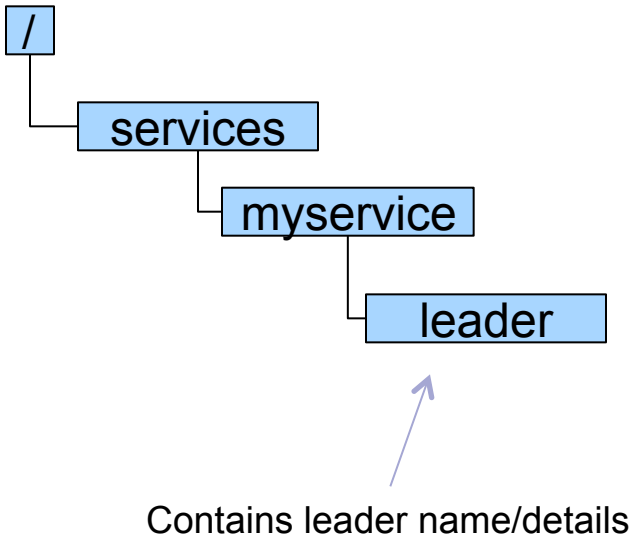
Use cases

- Use cases in HBase
  - Leader Election
  - Configuration Management – store bootstrap information
  - Group membership – discover tablet servers and finalize tablet server death
  - Store schema information and ACLs

Example – Leader Election

Leader election algorithm – when exactly one of N service providers have to be available:

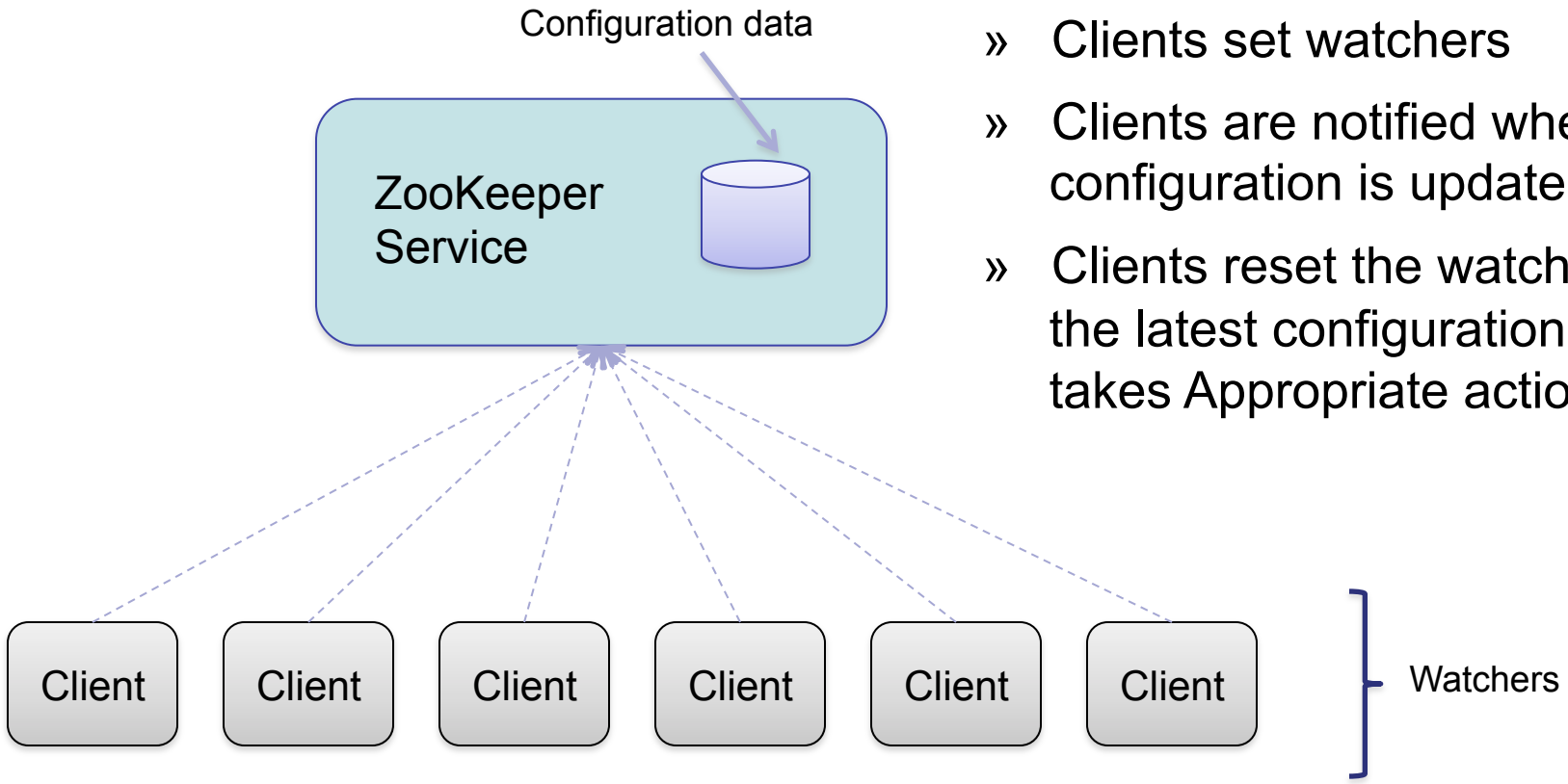
1. `getData ("/services/myservice/leader", true)`
2. `If successful, follow the leader described in the data and exit`
3. `create ("/services/myservice/leader", hostname, EPHEMERAL)`
4. `If successful, lead and exit`
5. `Go to step 1`



Note: If you want to have M processes of a set of N processes to be active, the algorithm can be modified to do so

Example – Configuration management

- Configuration data stored in znodes
- Clients set watchers
- Clients are notified when the configuration is updated
- Clients reset the watch, reads the latest configuration and takes Appropriate action



Essential Internals

- Leader + Followers, 2f+1 nodes can tolerate failure of f nodes
- Consistency model – completely ordered history of updates. All updates go through the leader
- Replication. No SPOF.
- All replicas can accept requests.
- If the leader fails, a new one is elected
- It's a system designed for few writes and many reads

Essential Internals

- Consistency using consensus – well known ways are Paxos algorithm, State Machine Replication, etc.
  - These are notoriously difficult. SMR is very difficult if your application doesn't fit that model.
- ZooKeeper uses ZooKeeper Atomic Broadcast protocol (ZAB)
  - ZAB – very similar to multi-Paxos, but the differences are real
- The implementation builds upon the FIFO property of TCP stream

References

- Algorithms
  - Paxos, multi-Paxos algorithms
  - State Machine Replication model
  - Atomic Broadcast
- Related projects
  - Chubby lock service from Google

Summary

Summary

- Zookeeper is a scalable, distributed, open-source coordination service for distributed applications
- We covered the design goals, basic concepts, and use cases with Zookeeper
- We also discussed the essential internals of Zookeeper

END