# Introduction to Apache ZooKeeper™

http://zookeeper.apache.org/

# What is a Distributed System?

> A distributed system consists of multiple computers that communicate through a computer network and interact with each other to achieve a common goal.

**- Wikipedia**

# Fallacies of Distributed Computing

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

Reference: *http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing*

# Coordination in a distributed system

- *Coordination*: An act that multiple nodes must perform together.
- Examples:
  - Group membership
  - Locking
  - Publisher/Subscriber
  - Leader Election
  - Synchronization
- Getting node coordination correct is very hard!

# Introducing ZooKeeper

> ZooKeeper allows distributed processes to coordinate with each other through a shared hierarchical name space of data registers.

***- ZooKeeper Wiki***

ZooKeeper is much more than a distributed lock server!

# What is ZooKeeper?

- An open source, high-performance coordination service for distributed applications.
- Exposes common services in simple interface:
  - naming
  - configuration management
  - locks & synchronization
  - group services

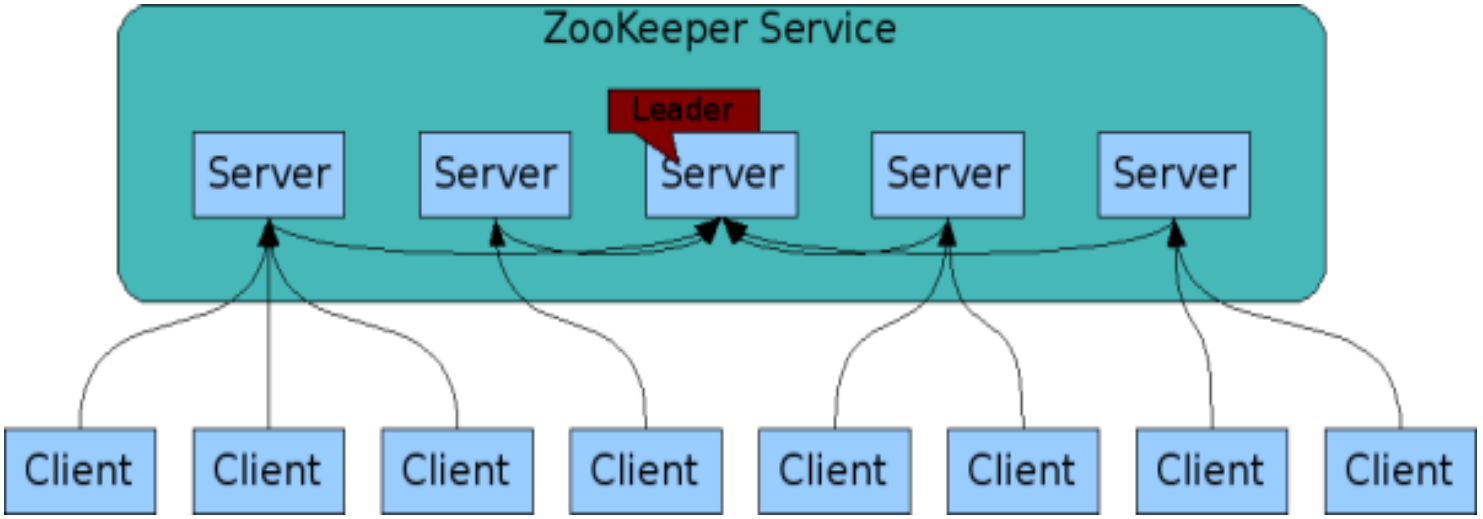
    *… developers don't have to write them from scratch*
- Build your own on it for specific needs.

# Design Goals

- Reliability
- Availability
- Concurrency
- Performance
- Simplicity

# 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




- ZooKeeper Service is replicated over a set of machines
- All machines store a copy of the data (in memory)
- A leader is elected on service startup
- Clients only connect to a single ZooKeeper server & maintains a TCP connection.
- Client can read from any Zookeeper server, writes go through the leader & needs majority consensus.

Image: https://cwiki.apache.org/confluence/display/ZOOKEEPER/ProjectDescription

# ZooKeeper Use Cases

- Configuration Management
  - Cluster member nodes bootstrapping configuration from a centralized source in unattended way
  - Easier, simpler deployment/provisioning
- Distributed Cluster Management
  - Node join / leave
  - Node statuses in real time
- Naming service – e.g. DNS
- Distributed synchronization - locks, barriers, queues
- Leader election in a distributed system.
- Centralized and highly reliable (simple) data registry

# The ZooKeeper Data Model

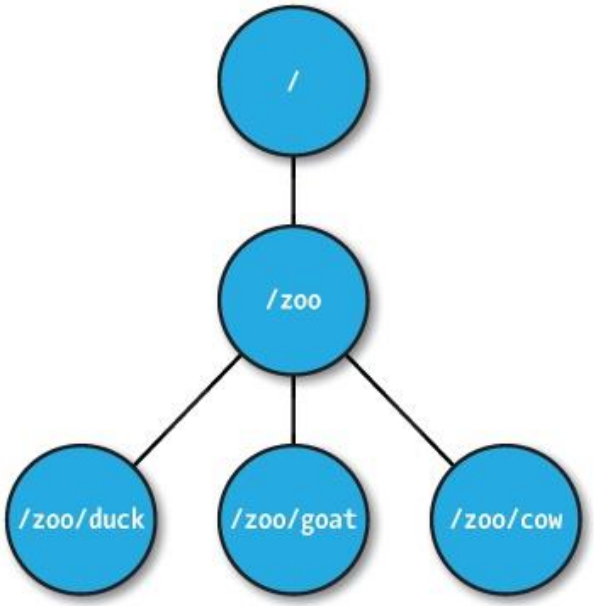
- ZooKeeper has a hierarchal name space.
- Each node in the namespace is called as a *ZNode.*
- Every ZNode has data (given as byte[]) and can optionally have children.

```
parent : "foo"
        |-- child1 : "bar"
        |-- child2 : "spam"
        `-- child3 : "eggs"
                    `-- grandchild1 : "42"
```

- ZNode paths:
  - canonical, absolute, slash-separated
  - no relative references.
  - names can have Unicode characters

# ZNodes

- Maintain a stat structure with version numbers for data changes, ACL changes and timestamps.
- Version numbers increases with changes
- Data is read and written in its entirety



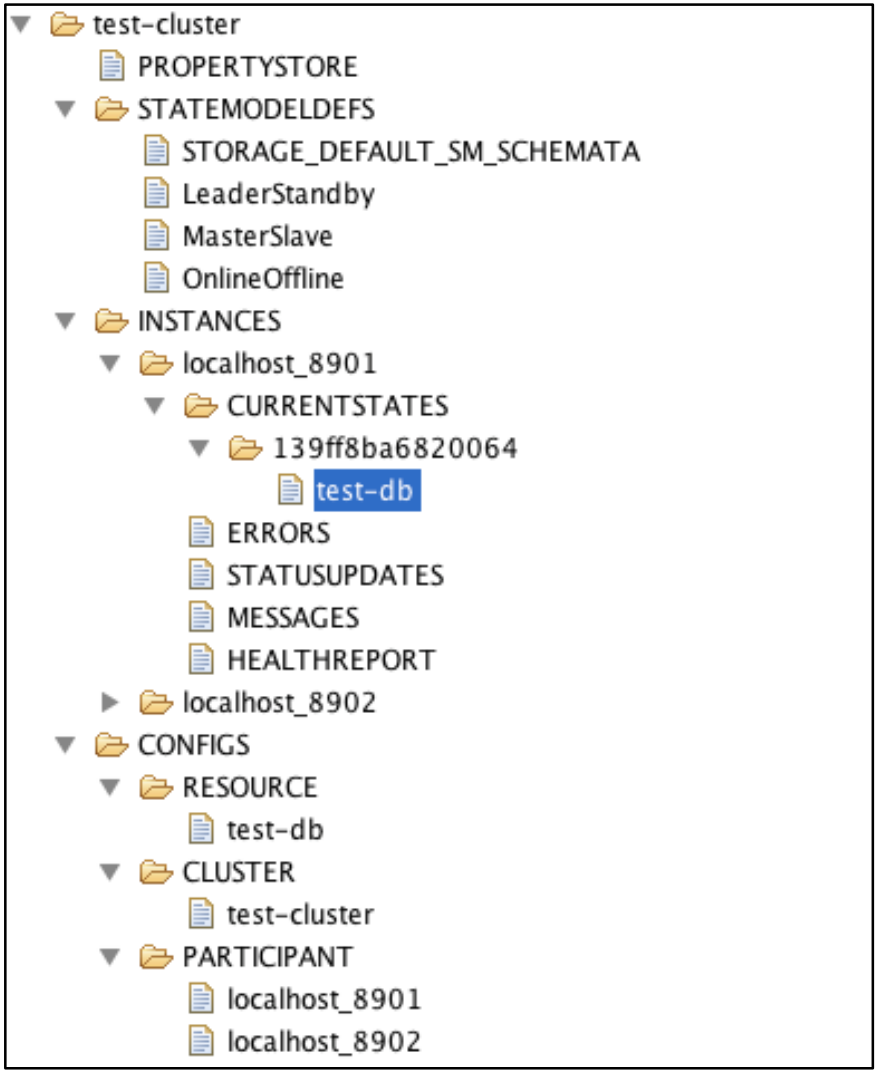




Image: http://helix.incubator.apache.org/Architecture.html

# ZNode Types

- Persistent Nodes
  - exists till explicitly deleted
- Ephemeral Nodes
  - exists as long as the session is active
  - can’t have children
- Sequence Nodes (Unique Naming)
  - append a monotonically increasing counter to the end of path
  - applies to both persistent & ephemeral nodes

| Operation | Type |
|---|---|
| create | Write |
| delete | Write |
| exists | Read |
| getChildren | Read |
| getData | Read |
| setData | Write |
| getACL | Read |
| setACL | Write |
| sync | Read |

***ZNodes are the main entity that a programmer access.***

# 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 Shell

```
[zk: localhost:2181(CONNECTED) 0] help
ZooKeeper -server host:port cmd args
        connect host:port
        get path [watch]
        ls path [watch]
        set path data [version]
        rmr path
        delquota [-n|-b] path
        quit
        printwatches on|off
        create [-s] [-e] path data acl
        stat path [watch]
        close
        ls2 path [watch]
        history
        listquota path
        setAcl path acl
        getAcl path
        sync path
        redo cmdno
        addauth scheme auth
        delete path [version]
        setquota -n|-b val path
```

```
[zk: localhost:2181(CONNECTED) 1] ls /
[hbase, zookeeper]

    [zk: localhost:2181(CONNECTED) 2] ls2 /zookeeper
    [quota]
    cZxid = 0x0
    ctime = Tue Jan 01 05:30:00 IST 2013
    mZxid = 0x0
    mtime = Tue Jan 01 05:30:00 IST 2013
    pZxid = 0x0
    cversion = -1
    dataVersion = 0
    aclVersion = 0
    ephemeralOwner = 0x0
    dataLength = 0
    numChildren = 1

    [zk: localhost:2181(CONNECTED) 3] create /test-znode HelloWorld
  Created /test-znode
   [zk: localhost:2181(CONNECTED) 4] ls /
  [test-znode, hbase, zookeeper]
[zk: localhost:2181(CONNECTED) 5] get /test-znode
        HelloWorld
```

# ZNode Watches

- Clients can set watches on znodes:
  - NodeChildrenChanged
  - NodeCreated
  - NodeDataChanged
  - NodeDeleted
- Changes to a znode trigger the watch and ZooKeeper sends the client a notification.
- Watches are one time triggers.
- Watches are always ordered.
- Client sees watched event before new znode data.
- Client should handle cases of latency between getting the event and sending a new request to get a watch.

# 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
  - Tell me when something changes. E.g. Configuration data
  - One time trigger. Have to be reset by the client if interested in future notifications
  - Not a full fledged notification system. Its like clients asking for recommendations. 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, …)

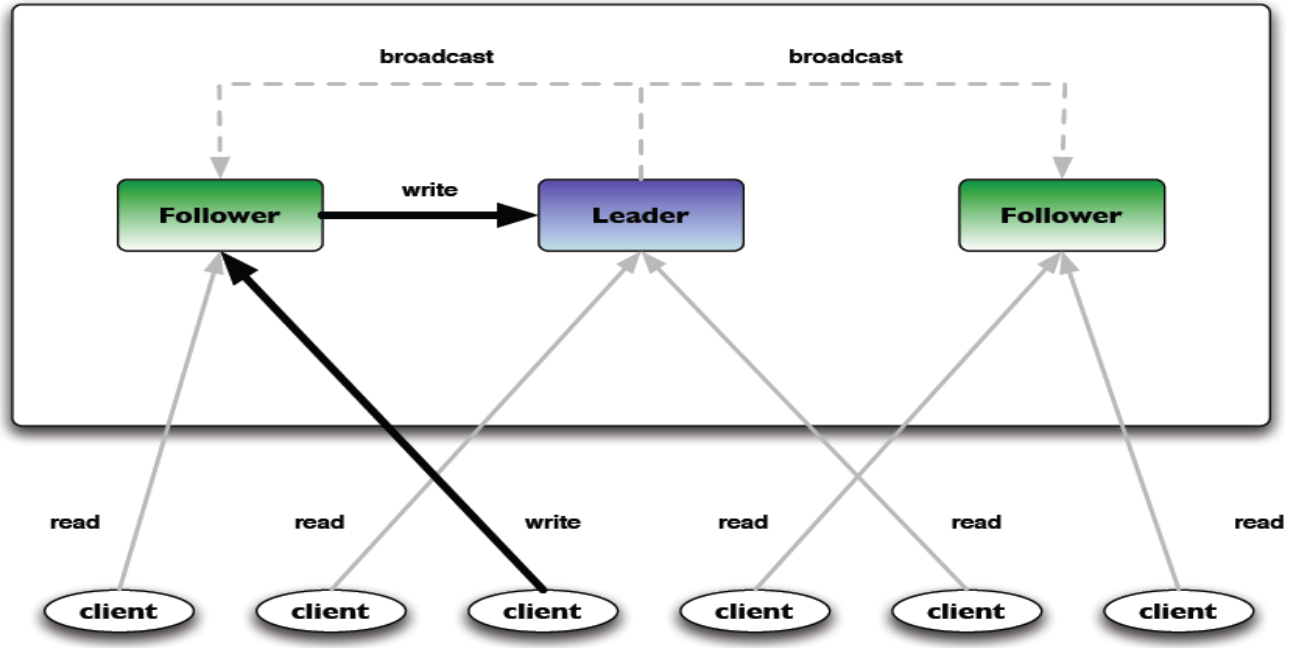
# API Synchronicity

- API methods are sync as well as async
- Sync:

```
exists(“/test-cluster/CONFIGS", null);
```

- Async:

```
exists("/test-cluster/CONFIGS", null, new StatCallback() {
    @Override
    public processResult(int rc, String path, Object ctx, Stat stat)
    {
            //process result when called back later
    }
    }, null);
```




- Read requests are processed locally at the ZooKeeper server to which the client is currently connected
- Write requests are forwarded to the leader and go through majority consensus before a response is generated.

Image: http://www.slideshare.net/scottleber/apache-zookeeper

# Consistency Guarantees

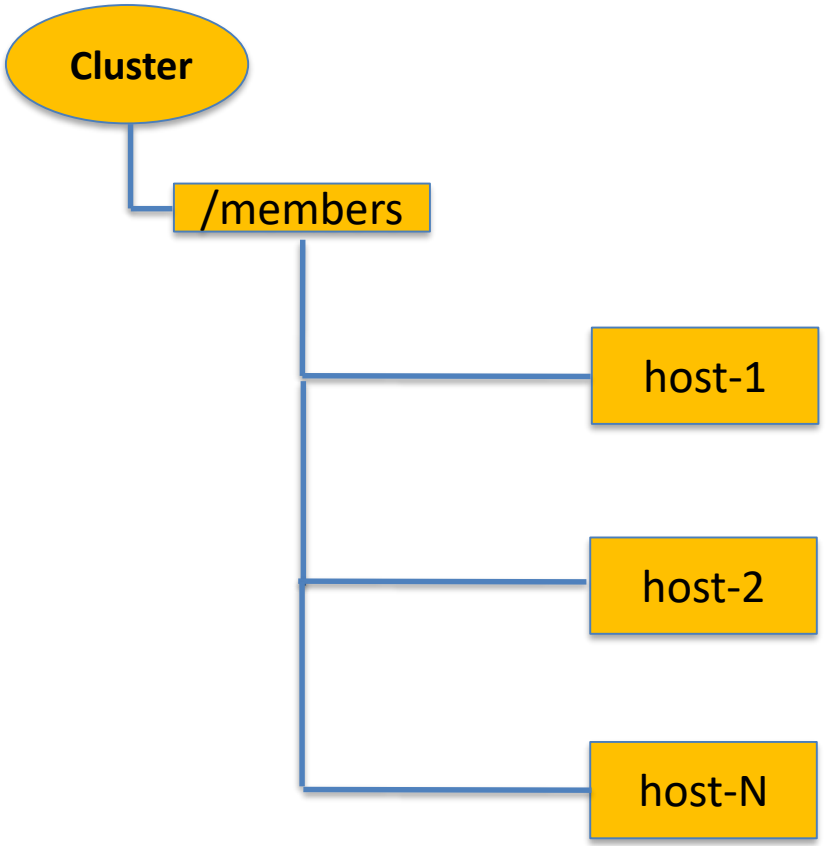
- **Sequential Consistency**: Updates are applied in order
- **Atomicity**: Updates either succeed or fail
- **Single System Image**: A client sees the same view of the service regardless of the ZK server it connects to.
- **Reliability**: Updates persists once applied, till overwritten by some clients.
- **Timeliness**: The clients' view of the system is guaranteed to be up-to-date within a certain time bound. (Eventual Consistency)

# Recipe #1: Cluster Management

Each Client Host i, i:=1 .. N

1. Watch on /members
2. Create /members/host-${i} as ephemeral nodes
3. Node Join/Leave generates alert
4. Keep updating /members/host-${i} periodically for node status changes

   (load, memory, CPU etc.)



# Recipe #2: Leader Election

1. A znode, say “/svc/election-path"
2. All participants of the election process create an ephemeral-sequential node on the same election path.
3. ***The node with the smallest sequence number is the leader.***
4. Each “follower” node listens to the node with the next lower seq. number
5. Upon leader removal go to election-path and find a new leader, or become the leader if it has the lowest sequence number.
6. Upon session expiration check the election state and go to election if needed

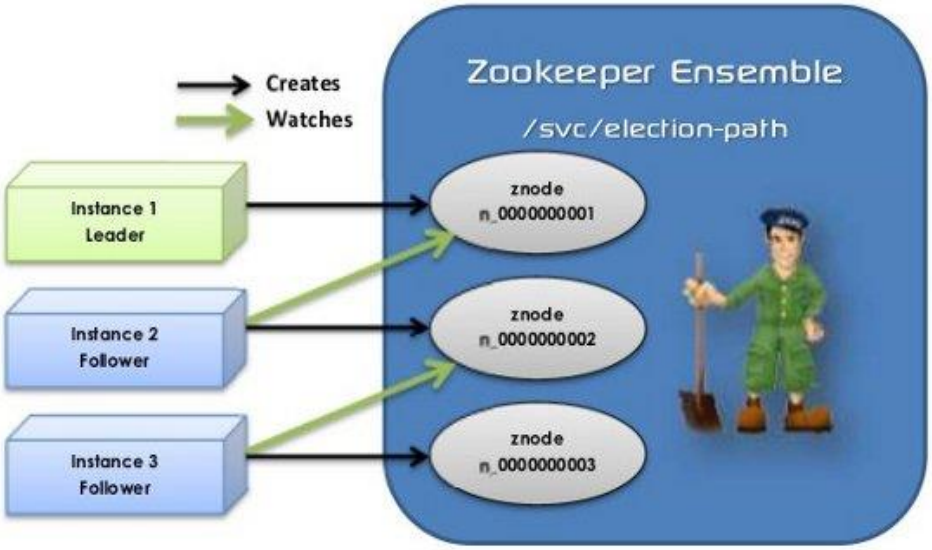



Image: http://techblog.outbrain.com/2011/07/leader-election-with-zookeeper/

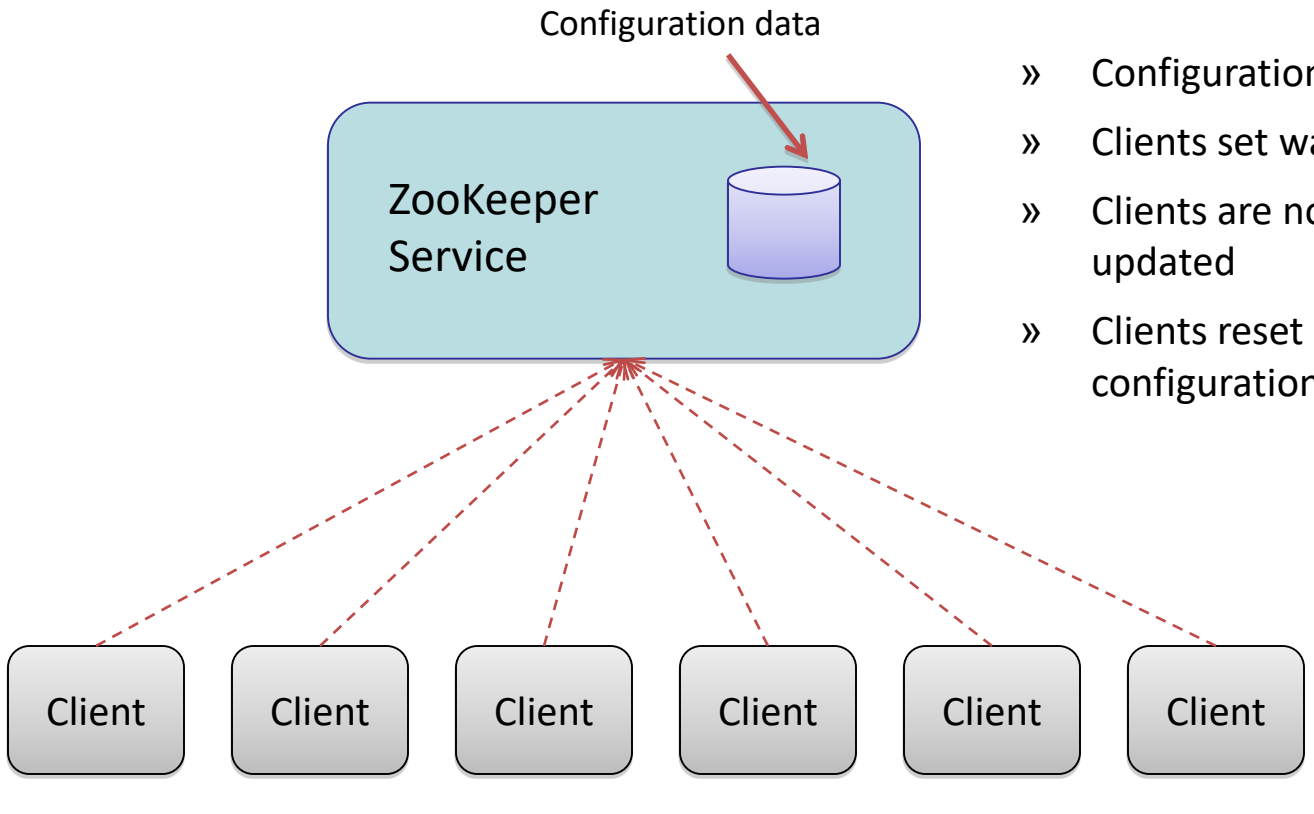
# Recipe #3: Distributed Exclusive Lock

Assuming there are N clients trying to acquire a lock

- Clients creates an ephemeral, sequential znode under the path /Cluster/_locknode_
- Clients requests a list of children for the lock znode (i.e. _locknode_)
- ***The client with the least ID according to natural ordering will hold the lock.***
- Other clients sets watches on the znode with id immediately preceding its own id
- Periodically checks for the lock in case of notification.
- The client wishing to release a lock deletes the node, which triggering the next client in line to acquire the lock.

```
ZK
|---Cluster
   +---config
   +---memberships
   +---_locknode_
      +---host1-3278451
      +---host2-3278452
      +---host3-3278453
      +--- …
      \---hostN-3278XXX
```

# Recipe #4: 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

# Recipe #5: Barrier

**Barrier -** a primitive that enables a group of processes to synchronize the beginning and the end of a computation

```
General idea: Barrier – parent node, participants = children

enter() {
   create new node for process p
   wait until child list size == barrier size
}

leave() {
   delete the node for process p
   wait until child list size == 0
}
```

# Recipe #6: Producer Consumer Queues

**Producer Consumer Queue -** a distributed data structure that a group of processes use to generate and consume items. Producer processes create new elements and add them to the queue. Consumer processes remove elements from the list, and process them.

```
General idea: Queue – parent node, items = children (using SEQUENCE flag)

produce (int i) {
   create new node with SEQUENCE flag and value I
}

int consume() {
   return the value of the child with the smallest counter
}
```

# Language Bindings

- ZooKeeper ships client libraries in:
  - Java
  - C
  - Perl
  - Python
- Community contributed client bindings available for Scala, C#, Node.js, Ruby, Erlang, Go, Haskell

  https://cwiki.apache.org/ZOOKEEPER/zkclientbindings.html

# A few points to remember

- Watches are one time triggers
  - Continuous watching on znodes requires reset of watches after every events / triggers
- Too many watches on a single znode creates the "*herd effect*" - causing bursts of traffic and limiting scalability
- If a znode changes multiple times between getting the event and setting the watch again, carefully handle it!
- Keep session time-outs long enough to handle long garbage-collection pauses in applications.
- Set Java max heap size correctly to *avoid swapping.*
- Dedicated disk for ZooKeeper transaction log

# 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
- 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

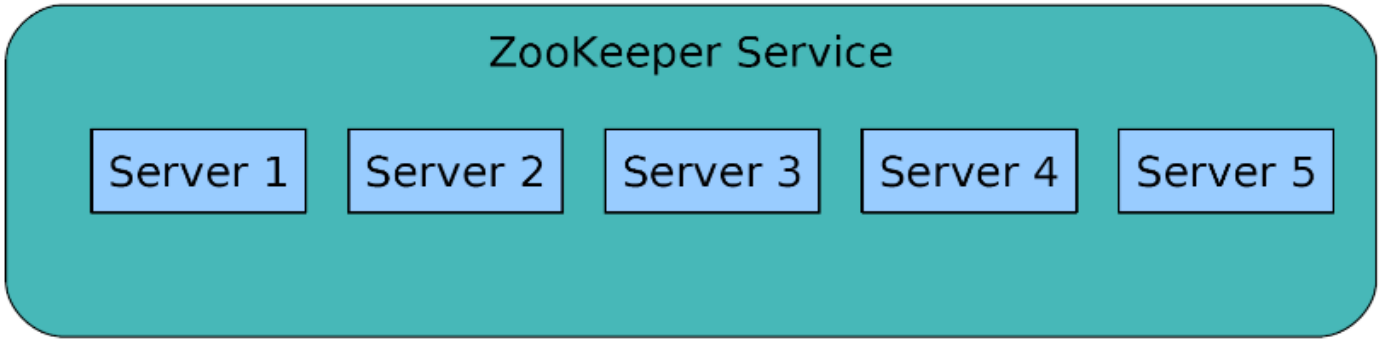
# ZooKeeper Atomic Broadcast

The heart of the ZooKeeper coordination service

# Goals

1) Must be able to tolerate failures
2) Must be able to recover from correlated recoverable failures (power outages)
3) Must be correct
4) **Must be easy to implement correctly**
5) Must be fast (high throughput, low latency)
   - Bursty throughput
   - Homogeneous servers with non homogeneous behavior (some will inevitably be faster than others because of HW or runaway processes etc)

# ZooKeeper Leader Election

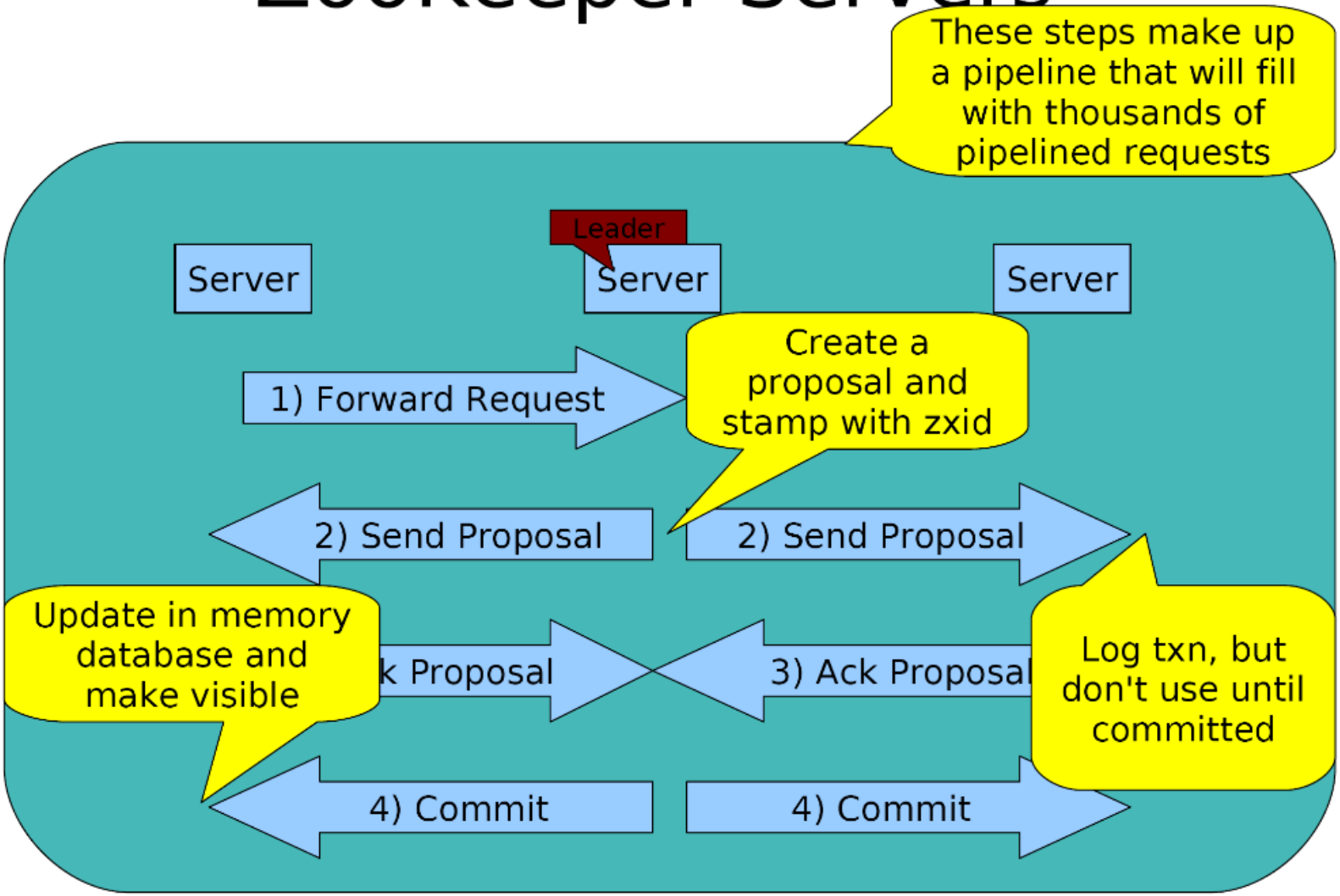



1) UDP or TCP based
2) Server with the highest logged transaction gets nominated
3) Election doesn't have to be absolutely successful, just very likely successful

# Starting assumption

1) Ability to create FIFO channels
   - We use TCP
   - Theoretically not a stronger assumption than classic lossy unordered channel since that is what TCP is built on
2) Crash fail
   - Digests to detect corruption
3) 2f+1 servers to handle f failures
   - Service must be able to recover from correlated recoverable failures (power outages)

# ZooKeeper Servers



# Nice Properties

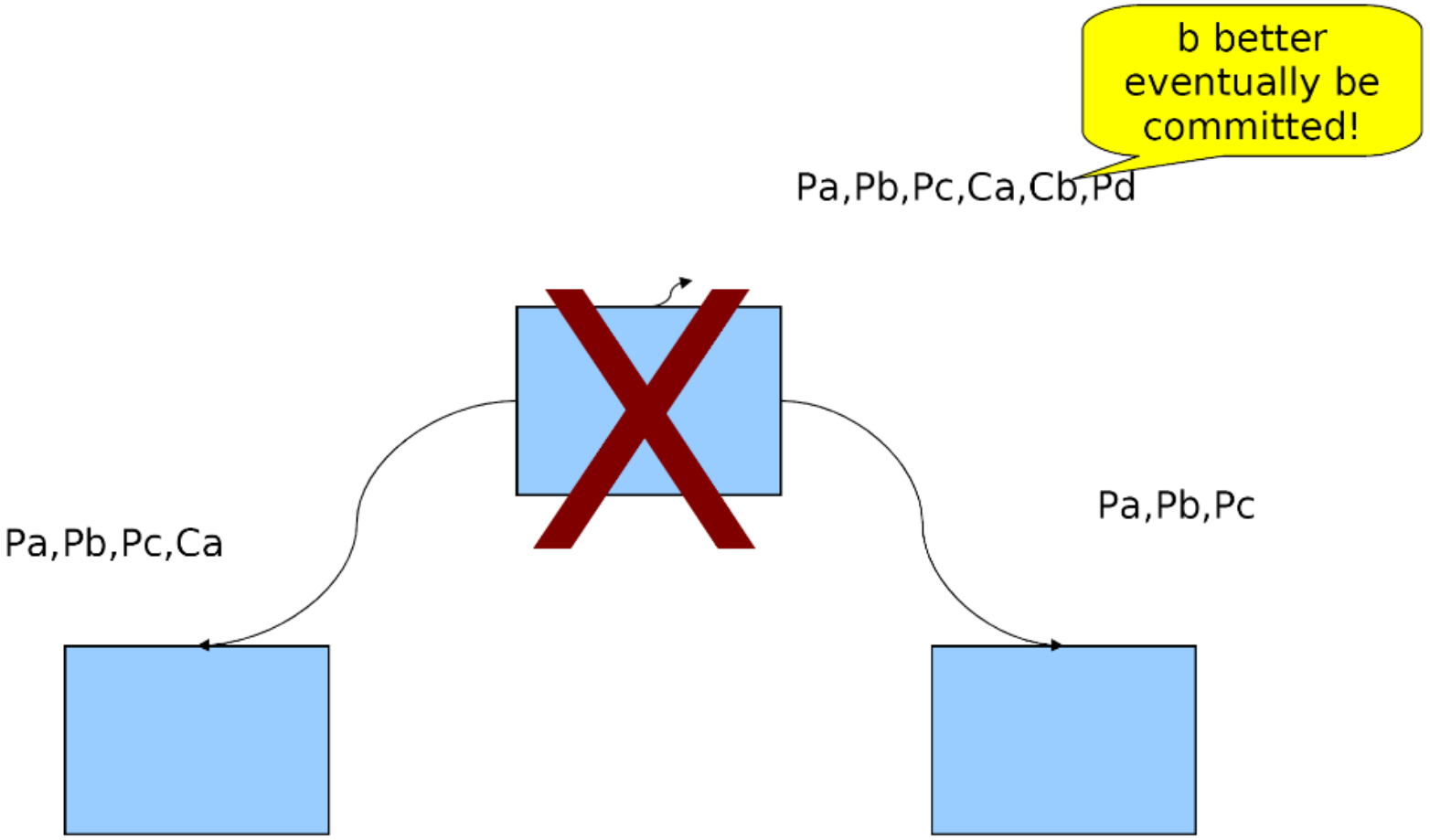
1) Leader always proposes in order
2) Because we use TCP, followers always receive in order
3) Followers process proposals in order
4) TCP means that Leader will get ACKs in order and thus commit in order
5) Followers only need to connect to a single server
6) Leader just waits for connections

# Everything is cool until...

# Leader Failure!

2) Make sure that the what has been delivered to some get delivered to all
3) Make sure that what gets forgotten stays forgotten
4) We get to choose what to do with the stuff in between

# Missed deliveries



# Never forget

1) If we elect the right guy, we will not forget anything
   - A new leader is elected by a quorum of followers
   - Committed messages must be seen by at least someone in the quorum
   - Elect the server that has seen the highest message in a quorum
   - New leader will commit all proposals it has seen from the previous leader

# Letting go

1) We use epochs to make sure that we only recover the last leaders outstanding proposals once.
   - Zxid is a 64-bit number: 32-bit of epoch and 32-bit counter
   - A new leader will increment the epoch
   - A new leader will only start proposing once the previous epoch is cleaned

# Leader Protocol in a nutshell

1)At startup wait for a quorum of followers to connect

2)Sync with a quorum of followers

- Tell the follower to delete any txn that the leader doesn't have (easy since it will only differ in one epoch)
- Send any txns that the follower doesn't have

3)Continually

- Assign and zxid to any message to be proposed and broadcast proposals to followers
- When a quorum has acked a proposal broadcast a commit

# Who uses ZooKeeper?

Companies:

- Yahoo!
- Zynga
- Rackspace
- LinkedIn
- Netflix
- *and many more…*

Projects in FOSS:

- Apache Map/Reduce (Yarn)
- Apache HBase
- Apache Solr
- Neo4j
- Katta
- *and many more…*

Reference: https://cwiki.apache.org/confluence/display/ZOOKEEPER/PoweredBy

# ZooKeeper In Action @Twitter

- Used within Twitter for service discovery
- How?
  - Services register themselves in ZooKeeper
  - Clients query the production cluster for service “A” in data center “XYZ”
  - An up-to-date host list for each service is maintained
  - Whenever new capacity is added the client will automatically be aware
  - Also, enables load balancing across all servers

Reference: http://engineering.twitter.com/

# References


- The Chubby lock service for loosely-coupled distributed systems

  *Google Research (7th USENIX Symposium on Operating Systems Design and Implementation (OSDI), {USENIX} (2006) )*
- ZooKeeper: Wait-free coordination for Internet-scale systems

  *Yahoo Research (USENIX Annual Technology Conference 2010)*
- Apache ZooKeeper Home: http://zookeeper.apache.org/
- Presentations:
  - http://www.slideshare.net/mumrah/introduction-to-zookeeper-trihug-may-22-2012
  - http://www.slideshare.net/scottleber/apache-zookeeper
  - https://cwiki.apache.org/confluence/display/ZOOKEEPER/ZooKeeperPresentations

# Interesting Reads

- The Google File System
- The Hadoop Distributed File System
- MapReduce: Simplified Data Processing on Large Clusters
- Bigtable: A Distributed Storage System for Structured Data
- PNUTS: Yahoo!’s Hosted Data Serving Platform
- Dynamo: Amazon's Highly Available Key-value Store
- Spanner: Google's Globally Distributed Database
- Centrifuge: Integrated Lease Management and Partitioning Cloud Services (Microsoft)
- ZAB: A simple totally ordered broadcast protocol (Yahoo!)
- Paxos Made Simple by Leslie Lamport.
- Eventually Consistent by Werner Vogel (CTO, Amazon)

http://www.highscalability.com/