ZooKeeper: Because building distributed systems is a zoo

Flavio Junqueira
Yahoo! Research Barcelona

Distributed systems

- Large number of processes
- Running on heterogeneous hardware
- Communicate using messages
- Systems are often asynchronous
  - Unbounded amount of time to execute a step
  - Unbounded message delay
  - Makes it difficult to determine if process has failed or is just slow

Real examples

- Search engine
  - Crawling
  - Indexing
  - Query processing

- Large scale data processing
  - Map-reduce jobs
  - *E.g.*, Hadoop

Crawling

- Fetch pages from the web
- Rough estimate
  - 200 billion documents (200 x 10^9)
- If we use a single server…
  - 1s to fetch each page
  - 2 billion seconds if fetching 100 in parallel
  - 63 years!
- More complications
  - Pages are removed
  - Pages change their content
  - Politeness (*e.g.*, crawl-delay directive)

Crawling

- Fetchers
  - Fetch pages from the Web
- Master commands fetchers
  - Distributes work
  - Politeness
- Pool of spare masters for high availability
- Which master process leads?
  - Leader Election

Crawling

- Work assignment
  - Pages to fetch
  - Politeness constraints
  - Metadata (useful when master leader fails)
- Available fetchers
  - Failure detection

Hadoop

- Large-scale data processing
- Map-Reduce
- Large clusters of compute nodes
  - Order of thousands of computers
  - Yahoo!: 13,000+
- Jobs
  - Distribute computation across nodes
  - Yahoo!: hundreds of thousands a month
- An example: WebMap
  - Number of links between pages in the index: **roughly 1 trillion links**
  - Size of output: **over 300 TB, compressed!**
  - Number of cores used to run a single Map-Reduce job: **over 10,000**
  - Raw disk used in the production cluster: **over 5 Petabytes**

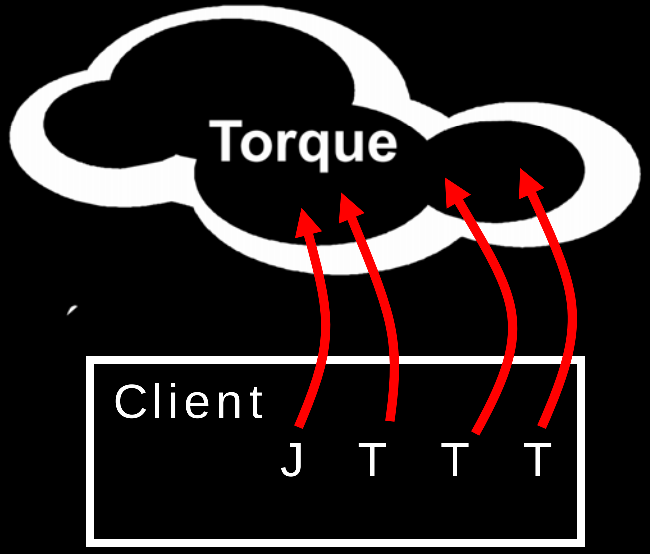
[http://developer.yahoo.net/blogs/hadoop/2008/02/]

Hadoop

- Plain Hadoop
- HDFS + Map-Reduce
- Heads of the system
  - One dedicated machine to the Namenode
    - FS metadata
    - *E.g.*, mapping from data blocks to Datanodes
  - One dedicated machine to Job Tracker
    - Tracks status of tasks
- All other machines are Task Tracker and/or Datanodes

Hadoop

- Hadoop virtual clusters
  - Hadoop on Demand (HOD)
- Rendezvous
  - Address of Job Tracker (J) is not known in advance
  - Task Trackers (T) need to be able to find the Job Tracker
  - Client needs to be able to find the Job Tracker (J)
- Failure detection
  - Task trackers and client need to know if Job Tracker is up and running



Coordination service

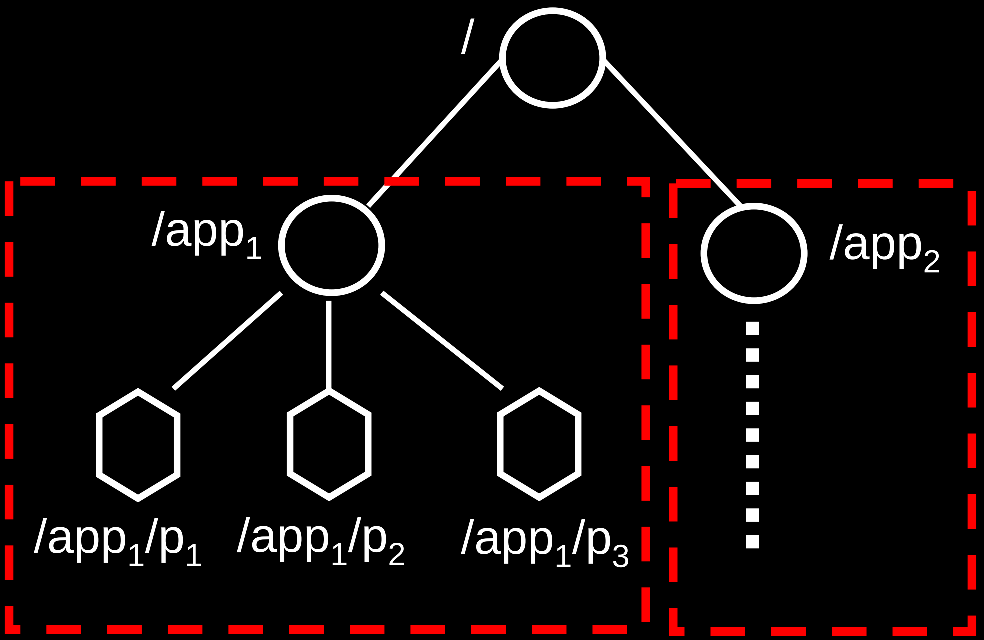
- Coordinate processes of a distributed application
  - Synchronization primitives
  - Metadata
- Why?
  1. Often not the focus of large projects
  2. Distributed algorithms are not trivial to understand and implement
  3. Debugging is difficult and since it is not the focus…
  4. Same functionality implemented (sometimes poorly) over and over again

ZooKeeper

- Coordination service
- A small database of metadata

ZooKeeper

- Shared memory
- Znodes
  - Data objects
  - Organized hierarchically



Applications use different branches.

ZooKeeper: Design

Start with file system API and model, and strip out what we do not need:

1) Rename
2) Partial writes/reads (takes with it open/close/seek)

add what we do need:

- Ordered updates and strong persistence guarantees
- Conditional updates (equivalent to compare-and-swap)
- Watches for data changes
- Ephemeral nodes
- Generated file names

Wait-free synchronization

A wait-free implementation of a concurrent data object is one that guarantees that any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes.

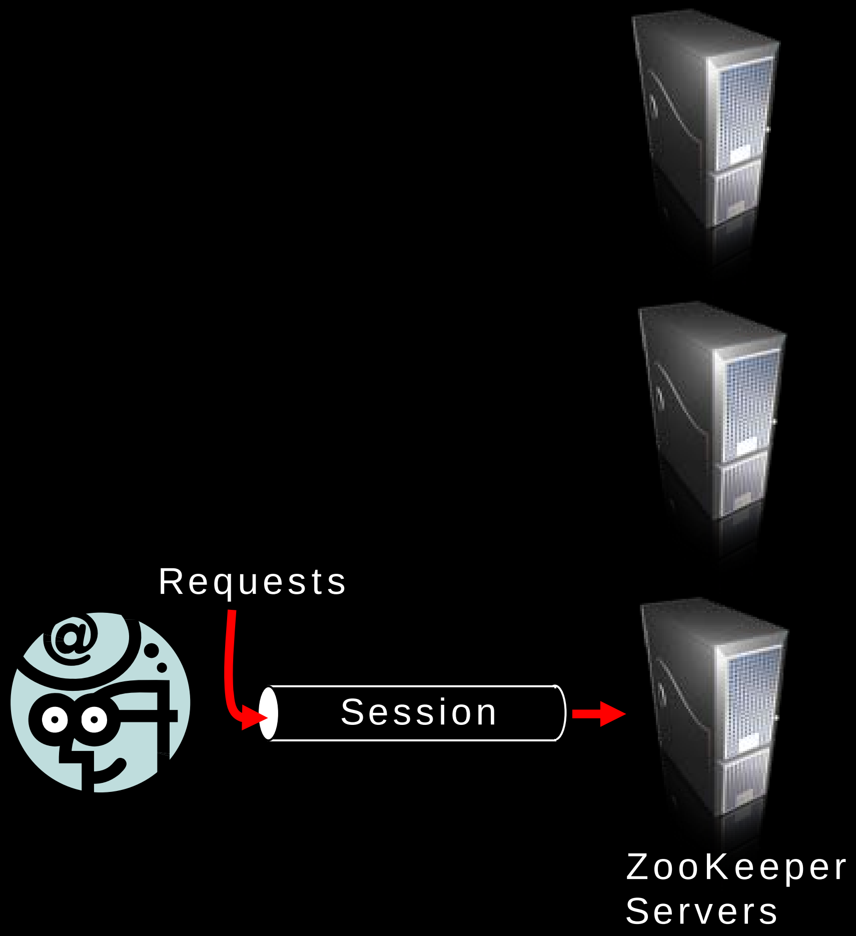
[Herlihy, *ACM TPLS*, Jan 1991]

- Advantages
  - Avoids the convoy effect
  - Convoy effect:

    The speed of the system is driven by the slowest process
  - Performance depends only on ZooKeeper

How it works – 10,000 ft view

- Ensemble of ZooKeeper servers
  - Fault tolerance
  - Throughput
- Clients create a new session with a server
- Clients submit requests
- Programming with ZooKeeper
  - Client library
  - Calls to the ZooKeeper API
  - Callbacks
    - Notifications
    - Changes to the state of client



YAHOO!

ZooKeeper API

```
String create(path, data, acl, flags)

void delete(path, expectedVersion)

Stat setData(path, data, expectedVersion)

(data, Stat) getData(path, watch)

Stat exists(path, watch)

String[] getChildren(path, watch)

void sync(path)
```

ZooKeeper recipes

- Leader election
  - One process eventually arises as the leader out of a group of processes
- Locks
  - Access to critical sessions
  - Mutually exclusive access to resources
- Barriers
  - Points of synchronization
  - Guarantees that processes proceed in a computation in lock-step
- Rendezvous
  - Information to allow client processes to find each other

An example: Wait-free leader election

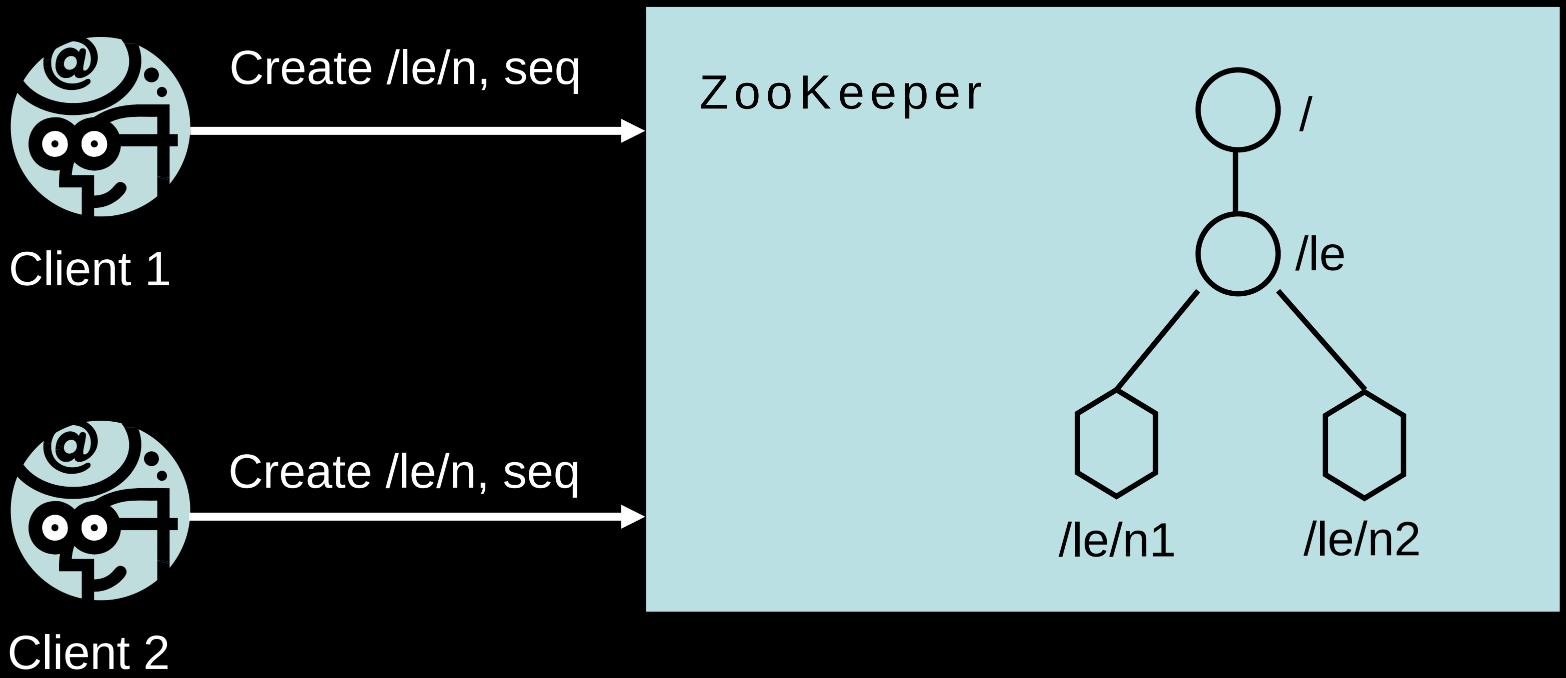
- Algorithm for client *C*
  - Create a sequential | ephemeral node representing the client as a child of "/le"
  - Read the children of "/le"
  - If the sequence number of *C* is the smallest, *C* is the leader

An example: Wait-free leader election

- Why is it wait-free?
  - Client *C* does not have to wait for other bids
  - If there are no other bids, *C* will be the leader
  - If there are concurrent bids, only one will be assigned the smallest sequence number

An example: Wait-free leader election

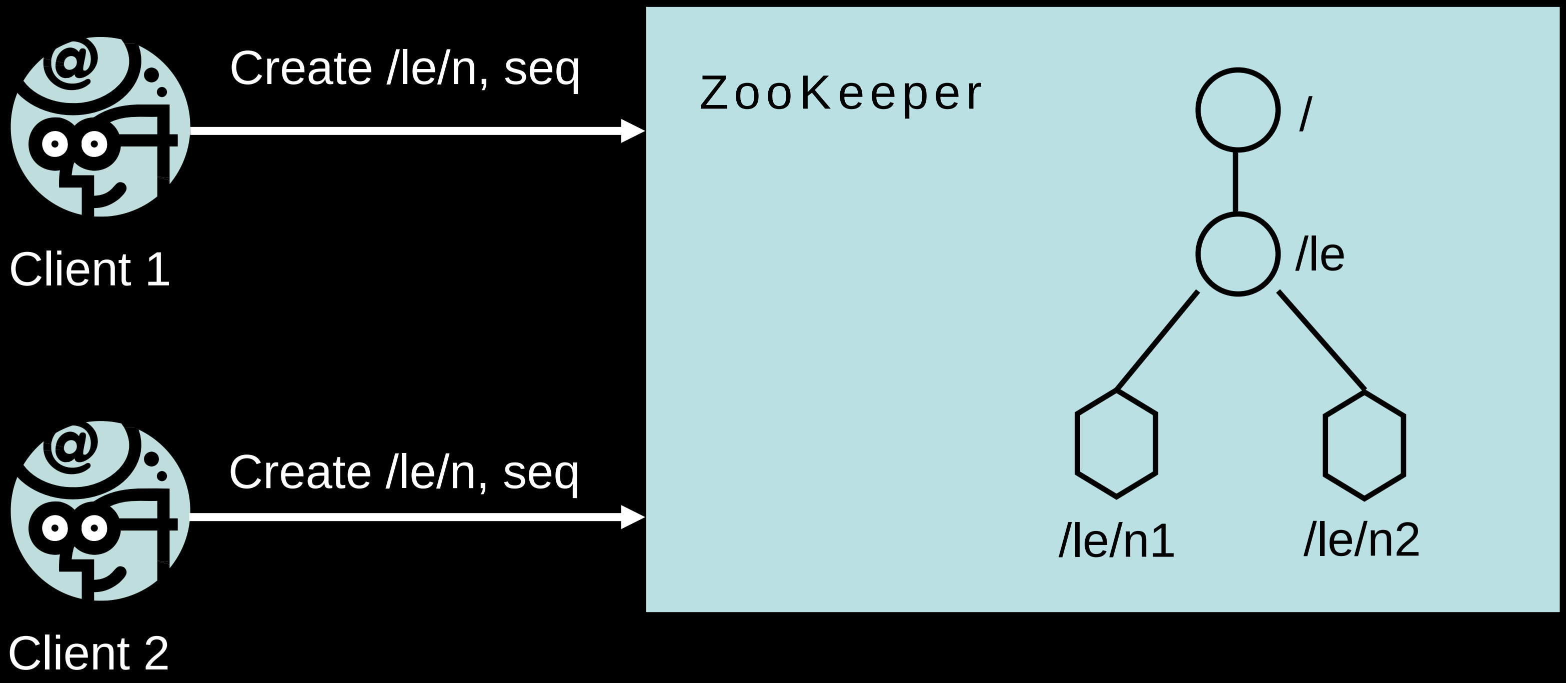
Case 1: Client 1 creates node first



YAHOO!

An example: Wait-free leader election

Case 2: Clients create nodes concurrently



An example: Wait-free leader election

- What if leader fails?

- Client nodes are ephemeral
- If not leader
  - Client watches for the following node in the sequence order
  - If node goes away and there is no preceding node, becomes leader

ZooKeeper Internals

- Updates are totally ordered
  - Leader executes update
  - Atomically broadcast znode state
- Fast read requests served locally
- Advantages
  - Strong consistency guarantees
  - High throughput for read-dominant workloads

- Consistency guarantees
  - History of writes is linearizable
    - Linearizable: sequential + precedence ordering
  - History of reads+writes
    - Serializable, but not linearizable
    - Reads do not satisfy precedence ordering
- Alternative: Slow read requests
  - **sync**( ) + fast read
  - History becomes linearizable

ZAB: ZooKeeper Atomic Broadcast

- Order of updates
- Replica servers
  - Apply the same set of updates in the same order

- The classical *Atomic broadcast* problem
  - Set of processes Π
  - Agreement : If a correct process p delivers m, then a correct process p' also delivers m
  - Order : If both processes p and p' deliver m and m' and p delivers m before m', then p' delivers m before m'

ZAB: ZooKeeper Atomic Broadcast

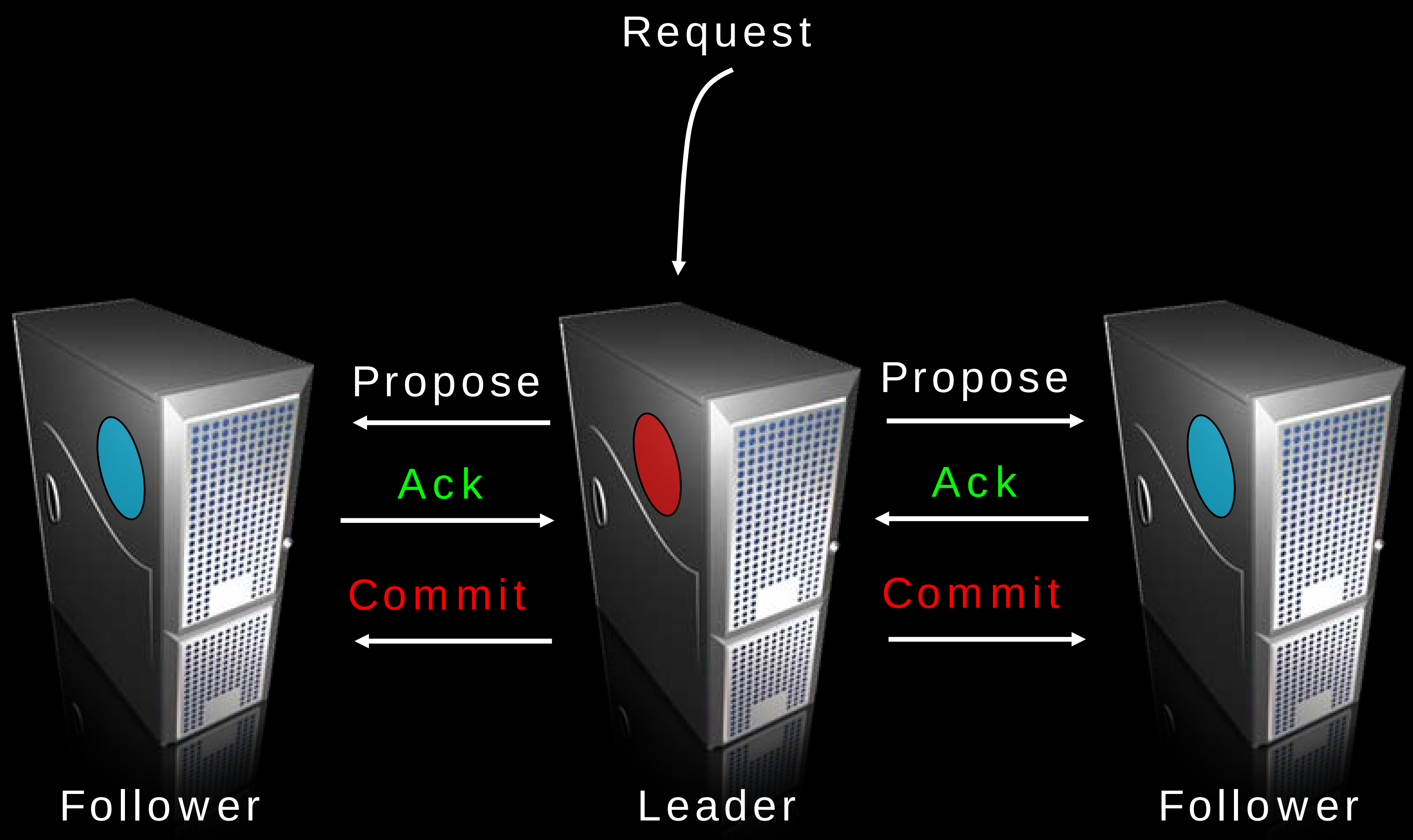
- Sequence of command slots
- Slot identifier is the zxid: $\langle \textit{epoch}, \textit{counter} \rangle$
- Each epoch has a single leader
  - Leader election
  - Different from the LE recipe!
- In a given epoch
  - Leader assigns operations to zxid values sequentially

ZAB: The basic protocol

- Once we have a leader…
- Clients submit requests to servers
- Servers forward requests to leader
- Leader proposes request
- Follower accepts
- Leader
  - Commits upon receiving acks from a quorum
  - Tells followers to deliver (make change of state persistent)

- Requires $n > 2t$ ZooKeeper servers

ZAB: The basic protocol

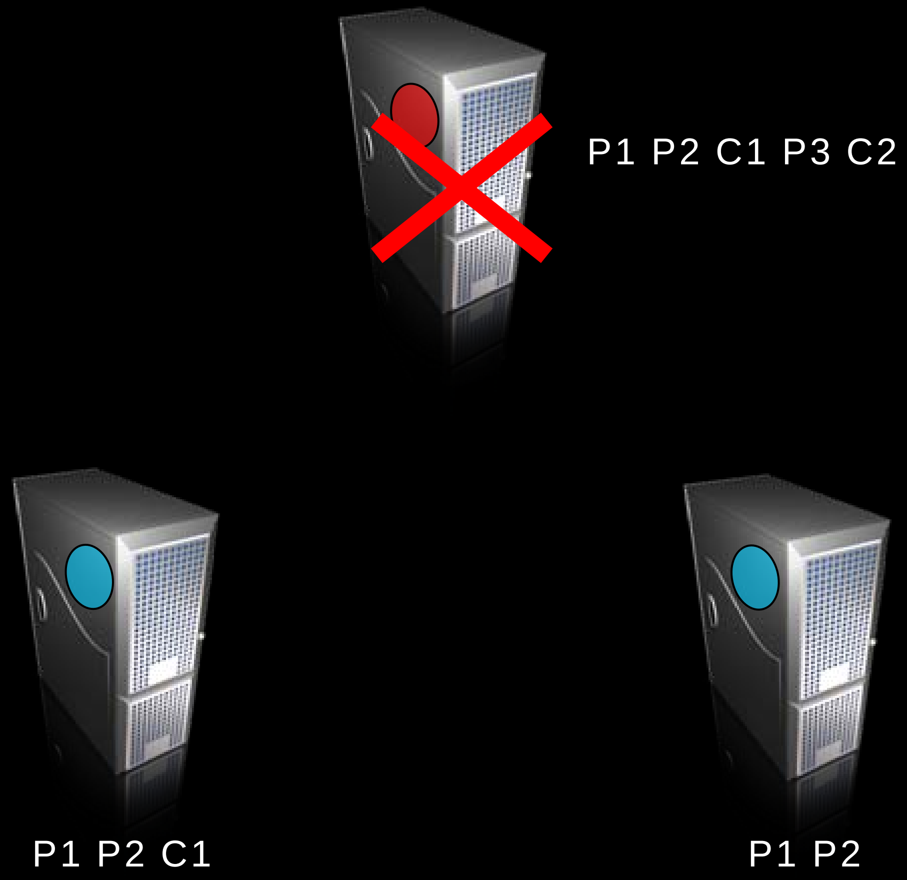


ZAB: Leader failure

- New leader is elected
- ZK server with highest *zxid*
- Role of leader
  - Leader proposes NEWLEADER
    - $zxid = \langle epoch, 0 \rangle$
  - Follower accept after synchronizing
  - Quorum of servers accepts
- During synchronization
  - Can’t forget committed requests
  - Let go of proposals not committed

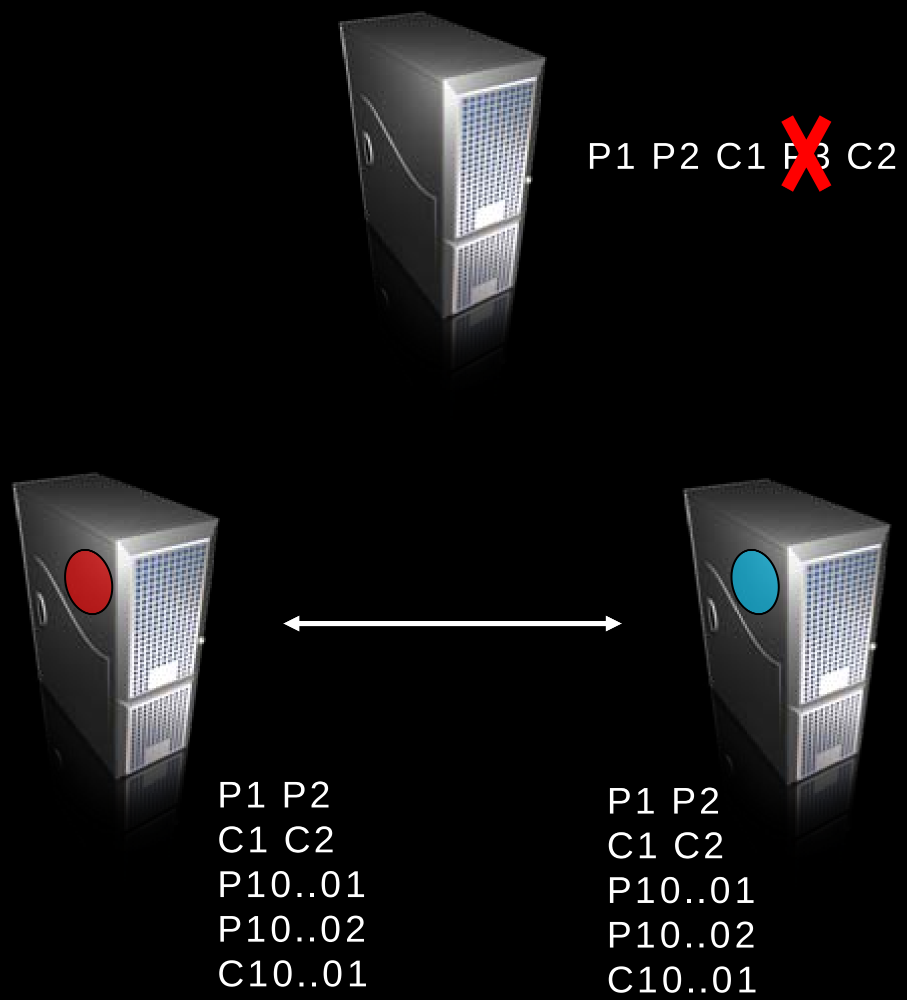
ZAB: Can't forget

- Some process has delivered proposal p
- All processes deliver p
- Leader does not fail:
  - All followers receive commit message
- Leader fails:
  - A quorum of followers has accepted p
  - New leader has accepted such a proposal



ZAB: Let it go

- Some server *s* has accepted proposal *p*
- Server *s* fails and recovers
- Proposal *p* is not committed
- When *s* recovers, it must drop *p*



ZAB: Agreement

- Proof idea
  - Server delivers proposal p by
    - Receiving a commit message
    - Synchronizing with a leader upon a new epoch
  - Servers s_1 and s_2 deliver p with $zxid\ \langle e, c\rangle$ by receiving a commit message
    - Must be the same message
    - Each epoch has at most one leader
  - Server s_1 delivers p with $zxid\ \langle e, c\rangle$ by receiving a commit message but s_2 doesn't
    - Server s_2 must eventually deliver p synchronizing with leader
  - Both servers deliver p with $zxid\ \langle e, c\rangle$ by synchronizing with leader

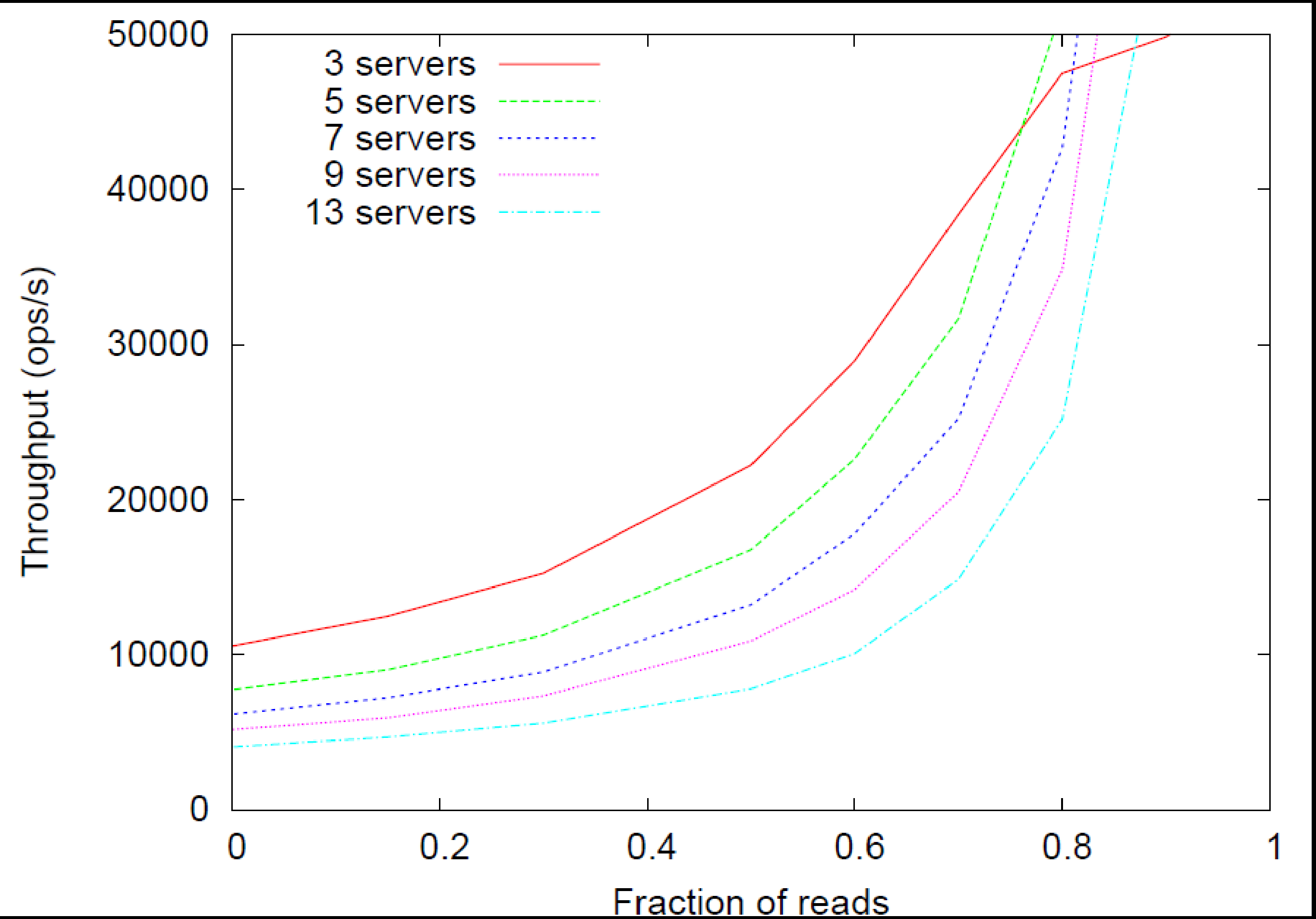
ZAB: Order

- Proof idea
  - Correct followers
    - Receive proposals in order of zxid from leader
  - Unique zxid per proposal
    - Each epoch has a single leader
  - Recovering or new followers
    - Synchronize with leader before accepting new proposals
    - Receive committed proposals in order

Evaluation

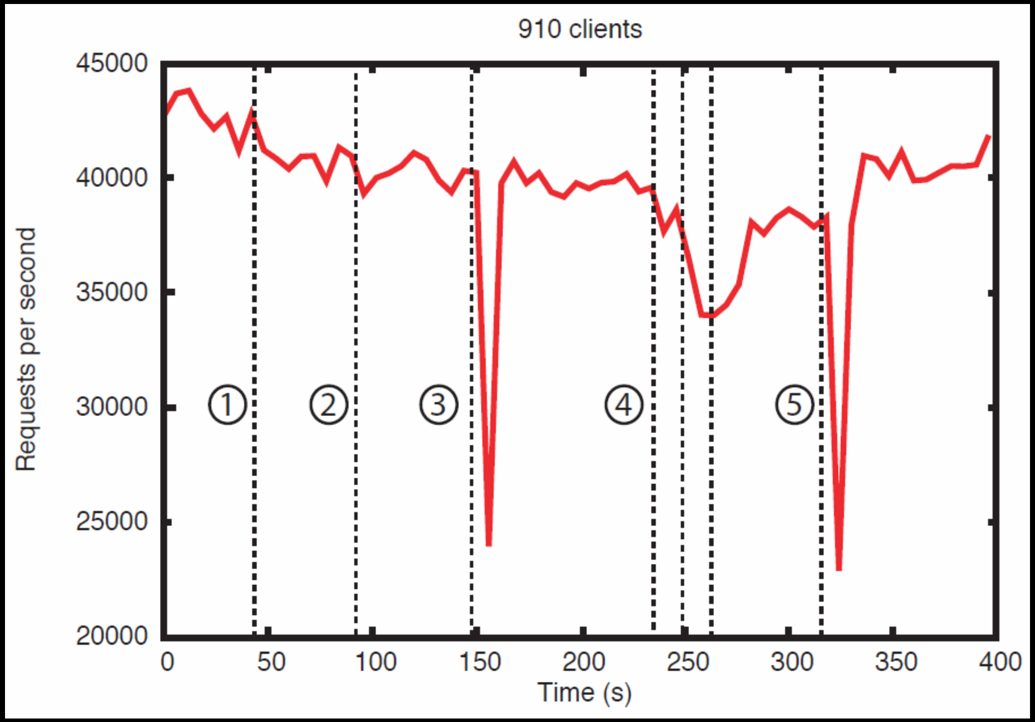
- Cluster of PC servers
- Servers
  - Xeon dual-core 3050 2.13GHz
  - 4GB of RAM
- Network
  - 1 Gbps

Evaluation: Throughput



Evaluation: Series of events

1. Failure and recovery of a follower
2. Failure and recovery of a different follower
3. Failure of the leader
4. Consecutive failures of two followers and recovery of both
5. Failure of the leader

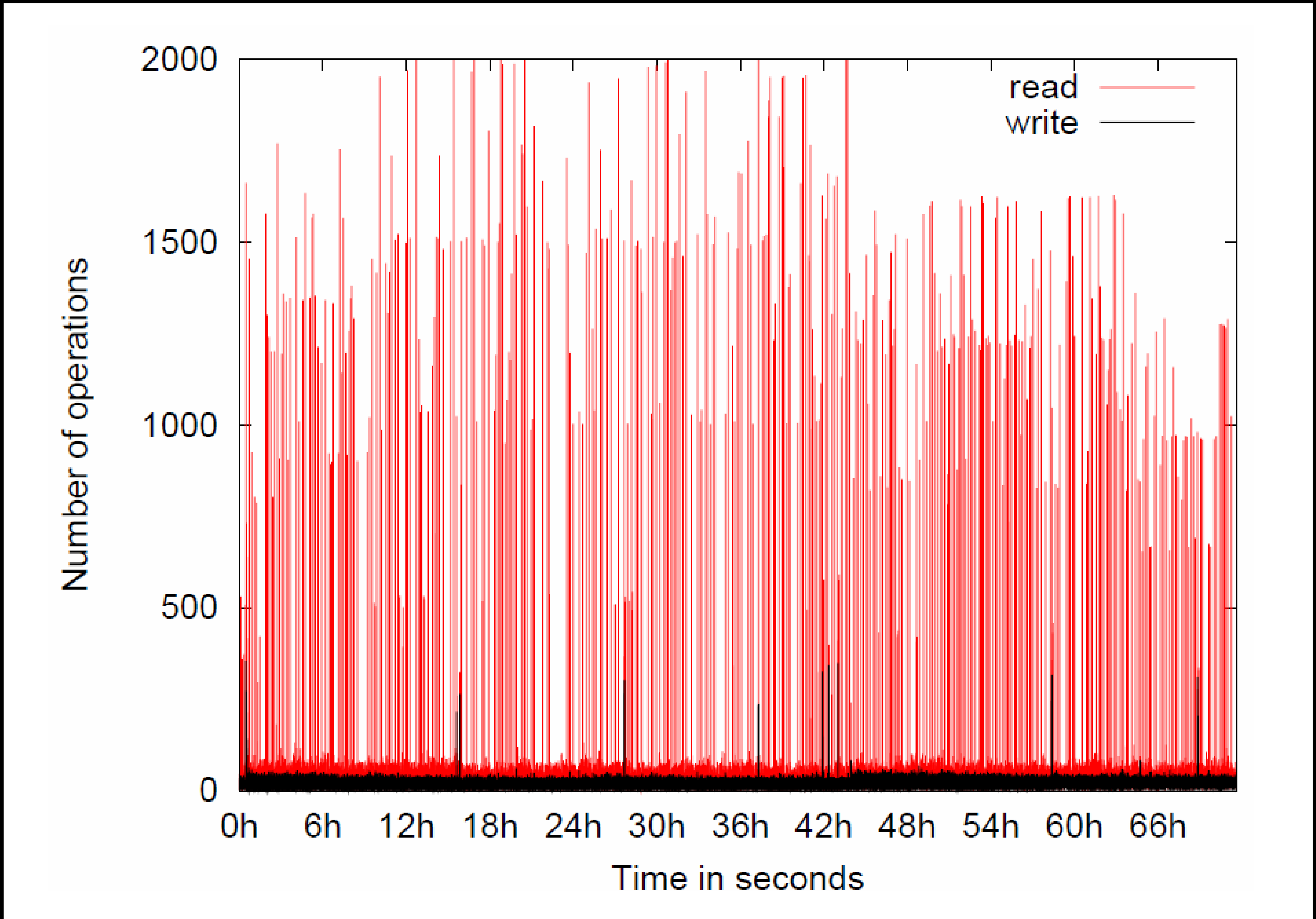


Evaluation: Barriers

- Goal
  - Throughput of primitives
- Double barriers
  - Synchronize in the beginning and at the end
  - Operations `enter()` and `leave()`
- Each client
  - Starts *n* barriers sequentially
  - Leaves barriers sequentially
- Throughput of barrier operations:
  - Roughly 3k ops/s

	# of clients		
# of barriers	50	100	200
200	9.4	19.8	41.0
400	16.4	34.1	62.0
800	28.9	55.9	112.1
1600	54.0	102.7	234.4

ZooKeeper: Fetching service traffic



Related work

- ISIS [Birman and Joseph, ACM SIGOPS Operating System Review, Nov 1987]
  – Toolkit for distributed programming
  – Based on virtual synchrony
- Chubby [Burrows, *USENIX OSDI* 2006]
  – Google's Lock service
- Sinfonia [Aguilera *et al.*, *ACM SOSP* 2007]
  – Minitransactions
  – Application store its data on Sinfonia
- Paxos [Lamport, *ACM TOCS,* May 1998]
  – Algorithm for state-machine replication

Conclusions

- ZooKeeper: Coordination service
  - Synchronization and metadata
  - Mitigates implementing complex synchronization primitives
  - Implemented once, used many times
- Wait-free synchronization
- ZAB: ZooKeeper Atomic Broadcast
  - Implementation simple and efficient
- Evaluation
  - High throughput: sufficient for internal applications
  - Fast recovery upon leader failures

- Distribution: `http://hadoop.apache.org/zookeeper`