Big Data Platforms (5 ECTS)
DATA14003
Lecture 6

Keijo Heljanko

Department of Computer Science
University of Helsinki
keijo.heljanko@helsinki.fi

28.9-2021

Database ACID guarantees

A traditional (centralized) database can guarantee its client ACID (atomicity, consistency, isolation, durability) properties:

- Atomicity: Database modifications must follow an "all or nothing" rule. Each transaction is said to be atomic. If one part of the transaction fails, the entire transaction fails and the database state is left unchanged.
- Consistency (as defined in databases): Any transaction the database performs will take it from one consistent state to another.
- Isolation: No transaction should be able to interfere with another transaction at all.
- Durability: Once a transaction has been committed, it will remain so.

Distributed Databases

In cloud computing we have to implement distributed databases in order to scale to a large number of users sharing common system state.
We define the three general properties of distributed systems popularized by Eric Brewer:

- Consistency (as defined by Brewer): All nodes have a consistent view of the contents of the (distributed) database
- Availability: A guarantee that every database request eventually receives a response about whether it was successful or whether it failed
- Partition Tolerance: The system continues to operate despite arbitrary message loss

Brewer's CAP Theorem

- In a PODC 2000 conference invited talk Eric Brewer made a conjecture that it is impossible to create a distributed system that is at the same time satisfies all three CAP properties:
  - Consistency
  - Availability
  - Partition tolerance
- This conjecture was proved to be a Theorem in the paper: "Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News, 33(2):51-59, 2002."

Brewer's CAP Theorem

Because it is impossible to have all three, you have to choose between two of CAP:

- CA: Consistent & Available but not Partition tolerant
  - A non-distributed (centralized) database system
- CP: Consistent & Partition Tolerant but not Available
  - A distributed database that can not be modified when network splits to partitions
- AP: Available & Partition Tolerant but not Consistent
  - A distributed database that can become inconsistent when network splits into partitions

Example CA Systems

- Single-site (non-distributed) databases
- LDAP - Lightweight Directory Access Protocol (for user authentication)
- NFS - Network File System
- Centralized version control - Svn
- HDFS Namenode

These systems are often based on two-phase commit algorithms, or cache invalidation algorithms.

Example CP Systems

- Distributed databases - Example: Google Bigtable, Apache HBase
- Distributed coordination systems - Example: Google Chubby, Apache Zookeeper

The systems often use a master copy location for data modifications combined with pessimistic locking or majority (aka quorum) algorithms such as Paxos by Leslie Lamport.

Example AP Systems

- Filesystems allowing updates while disconnected from the main server such as AFS and Coda.
- Web caching
- DNS - Domain Name System
- Distributed version control system - Git
- “Eventually Consistent” Datastores - Amazon Dynamo, Apache Cassandra

The employed mechanisms include cache expiration times and leases. Sometimes intelligent merge mechanisms exists for automatically merging independent updates.

System Tradeoffs - CA Systems

- Limited scalability - use when expected system load is low
- Easiest to implement
- Easy to program against
- High latency if clients are not geographically close

System Tradeoffs - CP Systems

- Good scalability
- Relatively easy to program against
- Data consistency achieved in a scalable way
- High latency of updates
- Quorum algorithms (e.g., Paxos) can make the system available if a majority of the system components are connected to each other
- Will eventually result in user visible unavailability for updates after network partition

System Tradeoffs - AP Systems

- Extremely good scalability
- Very difficult to program against - There is no single algorithm to merge inconsistent updates. Analogy: In distributed version control systems there will always be some conflicting updates that can not both be done, and there is no single general method to intelligently choose which one of the conflicting updates should be applied, if any.

System Tradeoffs - AP Systems (cnt.)

- Data consistency sacrificed - Can sometimes be tolerated, see e.g., Web page caching
- Low latency updates possible, always update the “closest copy”, and let the system to eventually propagate changes in the background to other data copies
- Will eventually result in user visible inconsistency of data after network partition

Why AP Systems?

- Much of the Internet infrastructure is AP - Web caching, DNS, etc.
- Works best with data that does not change, or changes infrequently
- Often much simpler to implement in a scalable fashion as CP systems, e.g., just cache data with a timeout
- Needs application specific code to handle inconsistent updates, no generic way to handle this!
- Needed for low latency applications

Example: Amazon Web Store

Most systems are combinations of available and consistent subsystems:

- Shopping Basket: AP - Always available for updates, even during network partitions. In case of two conflicting shopping basket contents - merge takes the union of the two shopping baskets
- In addition CP system for billing and for maintaining master copy of inventory
- Inconsistencies between the two systems are manifested at order confirmation time - items might need to be backordered / cancelled
- Quite often CP system added latency can be hidden in another long latency operation such as emailing the user

Example: Google Gmail

- AP when marking messages read - needs to be available, easy algorithm to merge inconsistent copies (take the union of read messages)
- CP when sending emails - Email will either be sent or not, and this needs to be acknowledged to the user

Example: Hadoop based Web Application

- HDFS Namenode - CA system, a centralized database of filesystem namespace, easy implementation effort
- HBase - CP distributed database, will disallow modifications under network partition, uses Apache Zookeeper for coordination
- Additional (optional) Web Caching layer - AP system, Used to minimize the request load on the background database system

Combinations of AP and CP Systems

- AP system usually used for user interaction due to its better scalability (cheaper to run) and lower latency. Requires application specific code to merge inconsistent updates
- CP system used to store “the master copy” of data, has large latency that needs to be hidden from the user
- Also CA systems used - Often the simplest implementation, needs careful architecture design not to become the bottleneck

Transactions in Distributed Databases

- The most widely used protocol to coordinate transactions in distributed databases is “two phase commit” (`http://en.wikipedia.org/wiki/Two-phase_commit_protocol`)
- In a two phase commit algorithm you have two kinds of servers: A transaction “coordinator”, and database servers called “cohorts”
- The protocol consists of two main phases:
  - A voting phase, where the coordinator asks all cohorts to prepare for a transaction
  - A completion phase, where the coordinator asks all cohorts to either commit (if all cohorts voted for the transaction to proceed) or rollback the transaction (in all other cases)

Transactions in Distributed Databases

- If a cohort crashes during transaction, the system will automatically either rollback or commit the transaction that was in progress for that cohort
- If a coordinator crashes permanently in the middle of the completion phase, some cohorts might be blocked forever waiting for either a commit or a rollback message to arrive
- The problem can not be easily solved without changing the used protocol
- Thus two phase commit can block indefinitely: Two phase commit in distributed databases is not fault tolerant!

The Asynchronous Consensus Problem

- Classic problem in distributed systems, can be used e.g., to decide whether to commit a transaction or not
- System has N processes
- Each process i starts with an initial vote $v_i \in \{0, 1\}$
- Asynchronous (no upper bound on message delivery) but reliable network (all messages are eventually delivered). Note: In particular there is no access to a common clock
- At most one process fails by halting (becoming permanently unresponsive)
- The protocol should terminate in finite time by deciding 0 or 1, and the decided value should be an initial vote of one of the processes (otherwise we could trivially always e.g., decide 0)

The FLP Theorem

- Published in the paper: “Michael J. Fischer, Nancy A. Lynch, Mike Paterson: Impossibility of Distributed Consensus with One Faulty Process. J. ACM 32(2): 374-382 (1985)”
- Impossibility result: There is no algorithm that will solve the asynchronous consensus problem!
- Key intuition: In an asynchronous model it is impossible to distinguish between a failed process and a slow process that has all its messages delayed
- Thus a consensus protocol will have very long (actually infinite) runs where it is assuming some dead process is just slow, making no progress in achieving consensus
- The problem becomes decidable if messages always have an upper bound transmission delay

Detailed Look at FLP Proof

- A detailed look at the proof: FLP proves that any sound fault-tolerant protocol for consensus in an asynchronous setting has runs that never terminate in consensus
- However, in practice these runs consist of an infinite sequence of cases where many different process are repeatedly assumed to have failed because their messages are delayed

FLP and Practical Approaches to Consensus

- In a practical computer networks the probability of infinite runs where the correctly functioning processes repeatedly keep suspecting each other to be faulty due to excessive message delays is diminishingly small
- Thus a practical consensus algorithm can be made that with very high probability quickly solves the asynchronous consensus problem

Paxos Algorithm

- The Paxos algorithm was designed to be a sound algorithm for the asynchronous consensus problem
- It is presented in the paper: “Leslie Lamport: The Part-Time Parliament. ACM Trans. Comput. Syst. 16(2): 133-169 (1998)”
- The guarantee is the following: If the Paxos algorithm terminates, then its output is a correct outcome of a consensus algorithm
- Paxos does not always terminate (FLP Theorem still stands!), but instead it terminates in practice with very high probability

Paxos Algorithm

- One can use Paxos to implement a highly fault tolerant database
- The Paxos algorithm uses $2f + 1$ servers to tolerate f concurrent failures, and still make progress
- Thus if one needs to tolerate two failures, five servers are needed
- Key intuition: If modifications are saved on at least $f + 1$ servers (a majority / a quorum), at least one of them will survive f failures
- For efficiency reasons, the algorithm can use a centralized leader through which all modifications are done
- If the leader is expected to have failed, a new leader can be (with high probability) quickly elected, and modifications sent to it instead

Distributed Coordination Systems

- Consensus under failures protocols are extremely subtle, and should not be done in normal applications code!
- In cloud based systems usually the protocols needed to deal with consensus are hidden inside a “distributed coordination system”

Google Chubby

- One of the most well known systems implementing Paxos is Google Chubby: “Michael Burrows: The Chubby Lock Service for Loosely-Coupled Distributed Systems. OSDI 2006: 335-350”
- Chubby is used to store the configuration parameters of all other Google services, to elect leaders, to do locking, to maintain pools of servers that are up, to map names to IP addresses, etc.
- Because in practical Paxos implementations all writes are sent to a single leader, write throughput of a single leader node can be a bottleneck

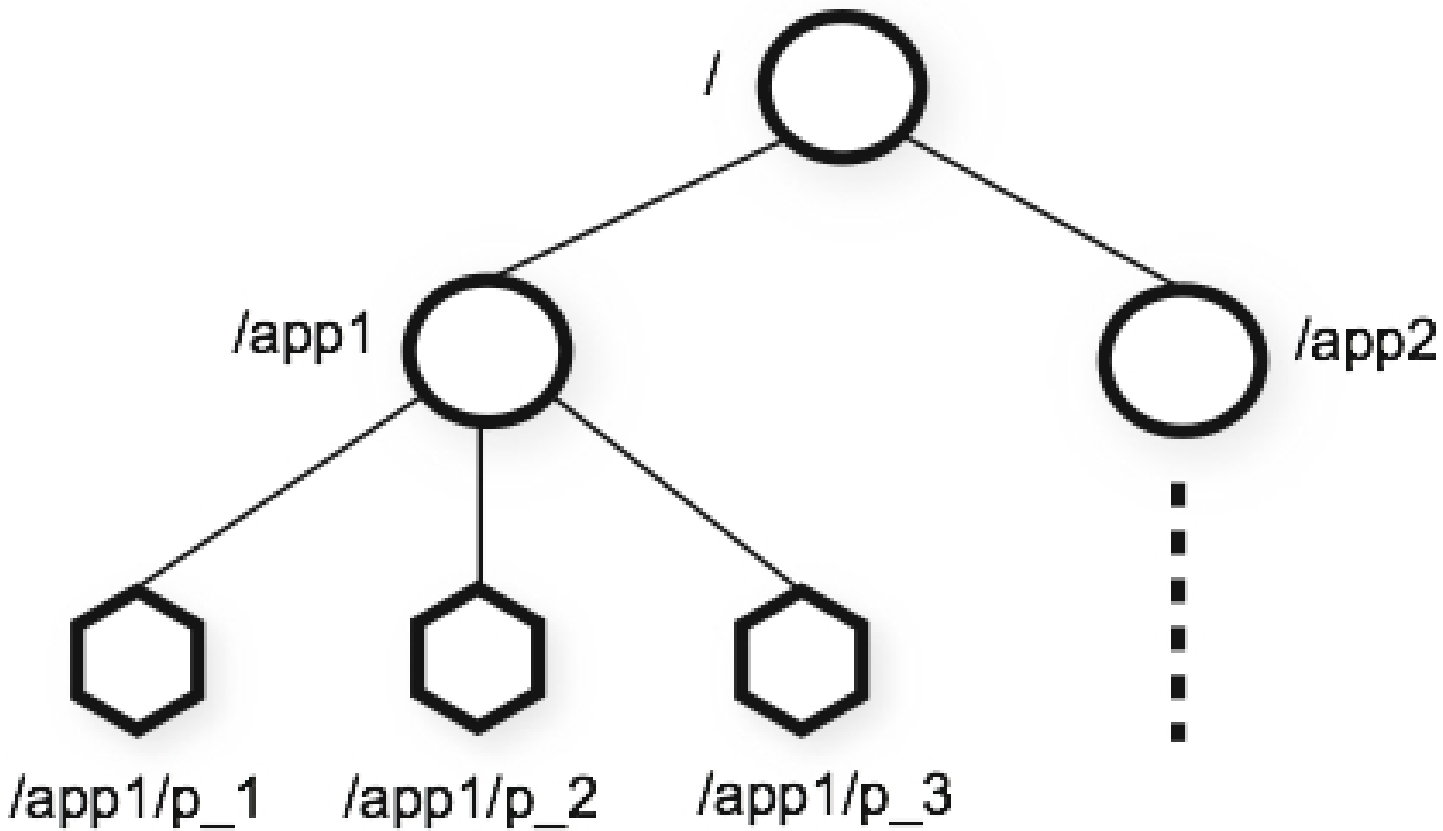
Apache Zookeeper

- An open source distributed coordination service
- Not based on Paxos, protocol outlined in: “Flavio Paiva Junqueira, Benjamin C. Reed: Brief Announcement Zab: A Practical Totally Ordered Broadcast Protocol. DISC 2009: 362-363”
- Shares many similarities with Paxos but is not the same protocol!
- We use Figures from: `http://zookeeper.apache.org/doc/r3.4.0/zookeeperOver.html`

Apache Zookeeper

- Zookeeper gives the clients a view of a small fault tolerant “filesystem”, with access control rights management
- Each “znode” (a combined file/directory) in the filesystem can have children, as well as have upto 1MB of data attached to it
- All znode data reads and writes are atomic, reading or writing all of the max 1MB data atomically: No need to worry about partial file reads and/or writes

Zookeeper Namespace

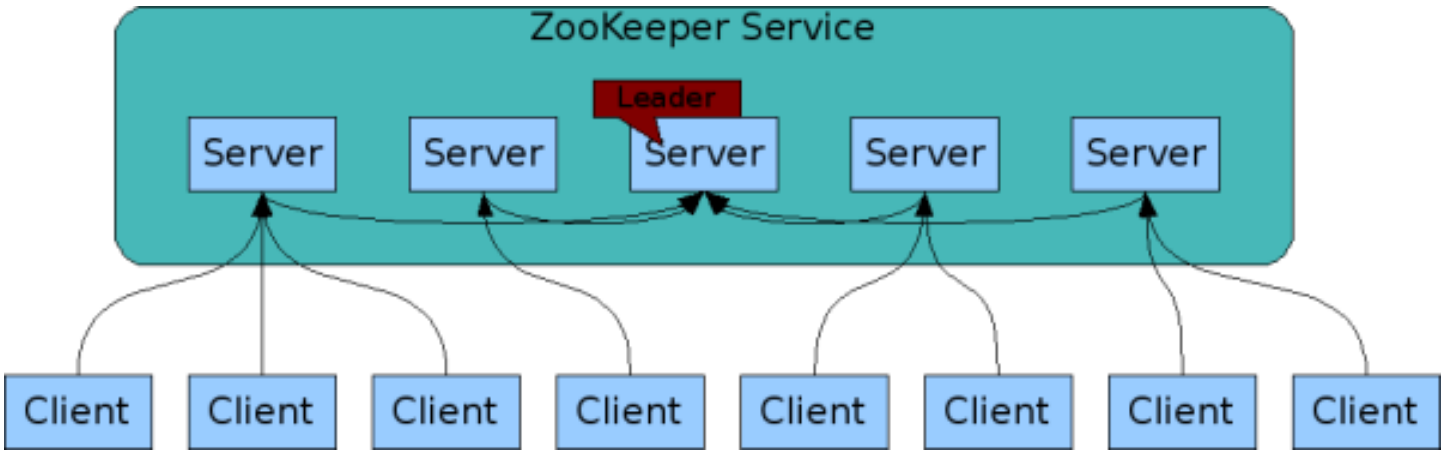


Ephemeral Znodes

- In addition to standard znodes, there are also “ephemeral znodes”, which exist in the system as long as the Zookeeper client who has created them can in timely manner reply to Zookeeper keepalive messages
- Ephemeral nodes are usually used to detect failed nodes: A Zookeeper client that is not able to reply to Zookeeper keepalive messages in time is assumed to have been failed
- Zookeeper clients can also create “watches”, i.e., be notified when a znode is modified. This is very useful to notice, e.g., failures/additions of servers without a continuos polling of Zookeeper

Zookeeper Service

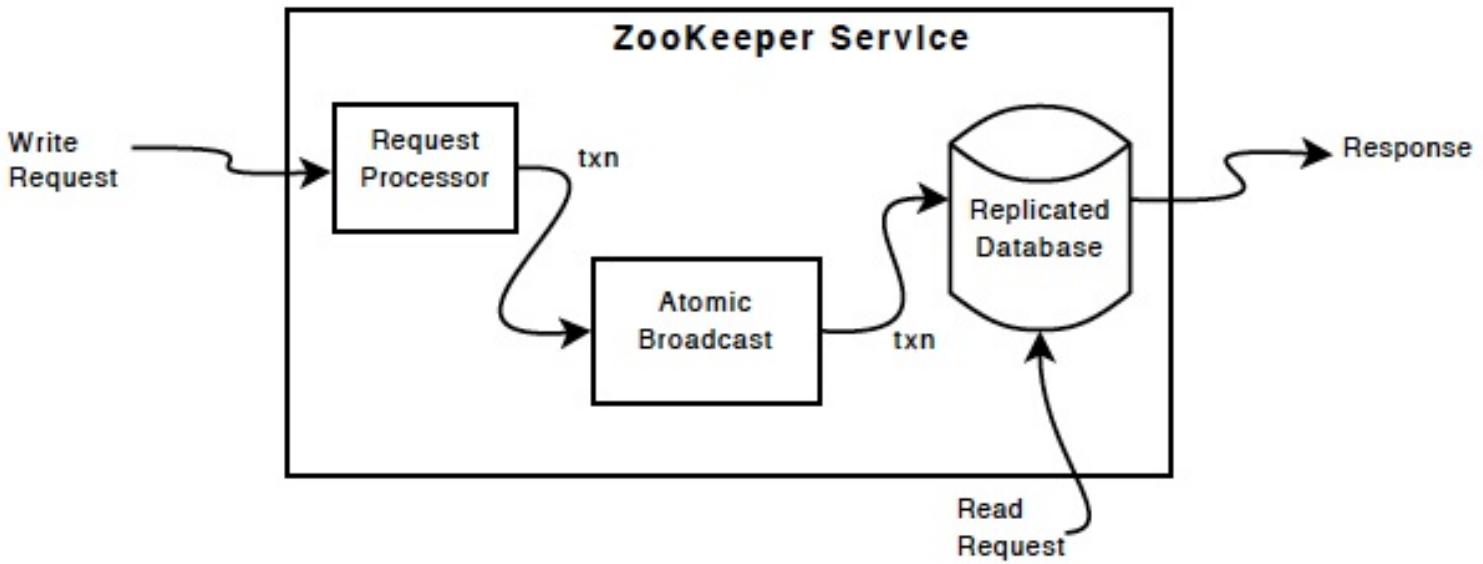
- Each Zookeeper client is connected to a single server
- All write requests are forwarded through a Leader node
- The Leader orders all writes into a total order, and forwards them to the Follower nodes. Writes are acked once a majority of servers have persisted the write



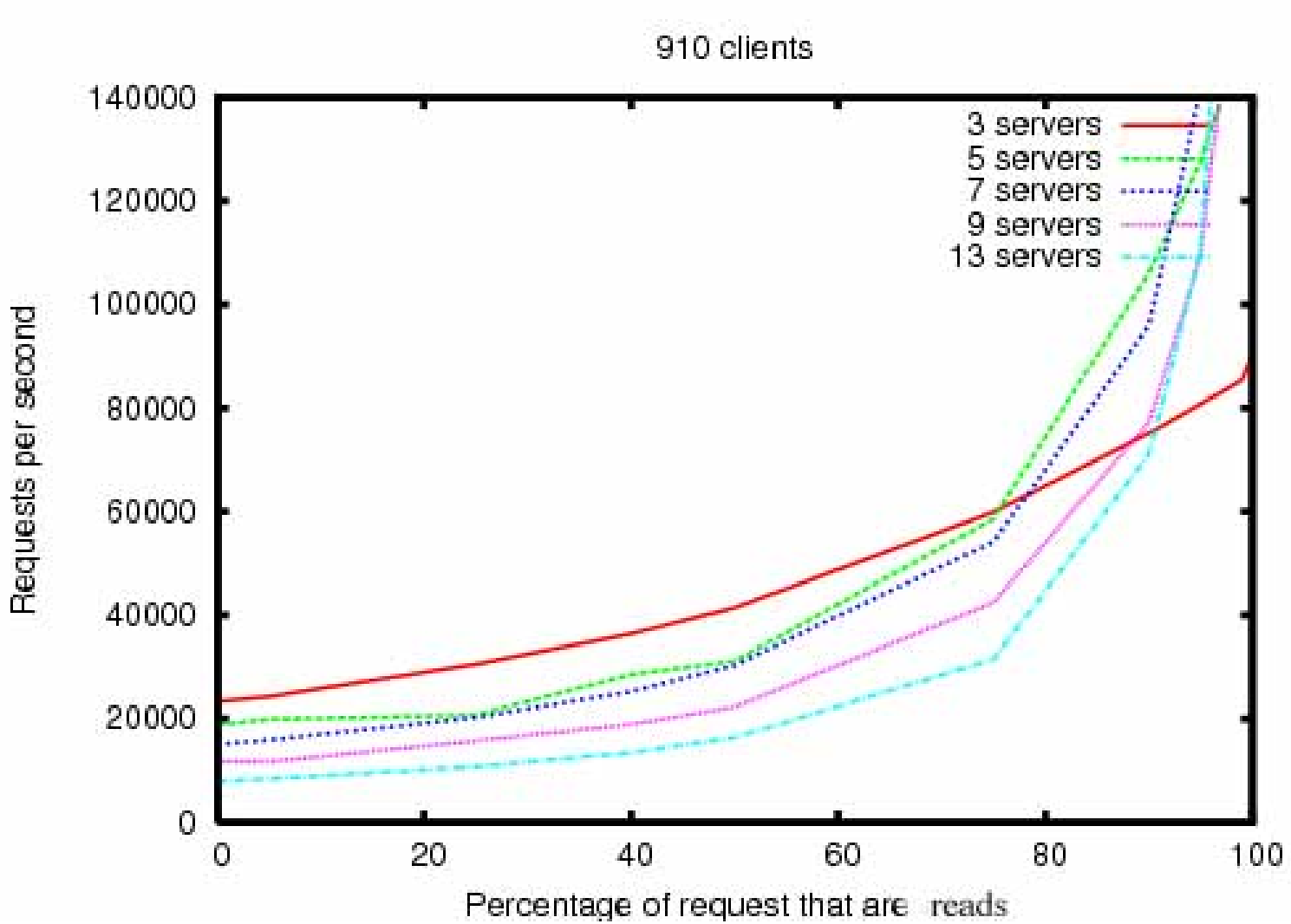
Zookeeper Characteristics

- Reads are served from the cache of the Server the client is connected to. The reads can return slightly out-of-date (max tens of seconds) data.
- There is a `sync()` call that forces a Server to catch up with the Master before returning data to the client
- The Leader will acknowledge a write if a majority of the servers are able to persist it in their logs
- Thus is a minority of Servers fail, there is at least one server with the most uptodate commits
- If the Leader fails, Zookeeper selects a new Leader

Zookeeper Components



Zookeeper Performance



Zookeeper Performance

- Note that for workloads that are mostly reads, adding more servers will improve the Zookeeper performance
- However, if more than roughly 30% of the operations are writes, the minimum three server configuration is the best performing configuration
- Thus Zookeeper's Zab and other Paxos-like algorithms have bad performance for write heavy workloads

Raft

- A widely used alternative to the Paxos and Zab algorithms is the Raft algorithm:
  Diego Ongaro, John K. Ousterhout: In Search of an Understandable Consensus Algorithm. USENIX Annual Technical Conference 2014: 305-319
- Raft is considered slightly easier to understand and implement compared to Paxos
- It is implemented in many different systems and programming languages, see:
  `https://raft.github.io/`
- Example system: `etcd`, which is a Raft based key-value store database used as the fault tolerant core of Kubernetes Container management system

Distributed Coordination Service Summary

- Maintaining a global database image under node failures is a very subtle problem
- One should use centralized infrastructure such as Chubby, Zookeeper, or a Raft implementation to only implement these subtle algorithms only once
- From an applications point of view the distributed coordination services should be used to store global shared state: Global configuration data, global locks, live master server locations, live slave server locations, etc.
- By modularizing the tricky fault tolerance problems to a distributed coordination service, the applications become much easier to program