CS-E4640 - Big Data Platforms
Lecture 10

Keijo Heljanko

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

28.11-2018

Exam

- Exam covers:
  - Lectures 1-10 (unless otherwise noted on the slides)
  - Tutorials (weekly exercises)
  - Home assignments test practical use of Spark, while the exam tests more throughly your knowledge of the course topics
  - Expect both question where you have to calculate (bring a calculator to the exam and learn the formulas needed in the tutorials!), as well as to define/compare concepts, and there might also be a short essay or two. Note: Weekly exercises also cover additional material of the exam area not presented in the Lectures!

Last Lecture and Weekly Exercise

- Last (recap) Lecture is Lecture 11 on Wed 12.12-2017

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

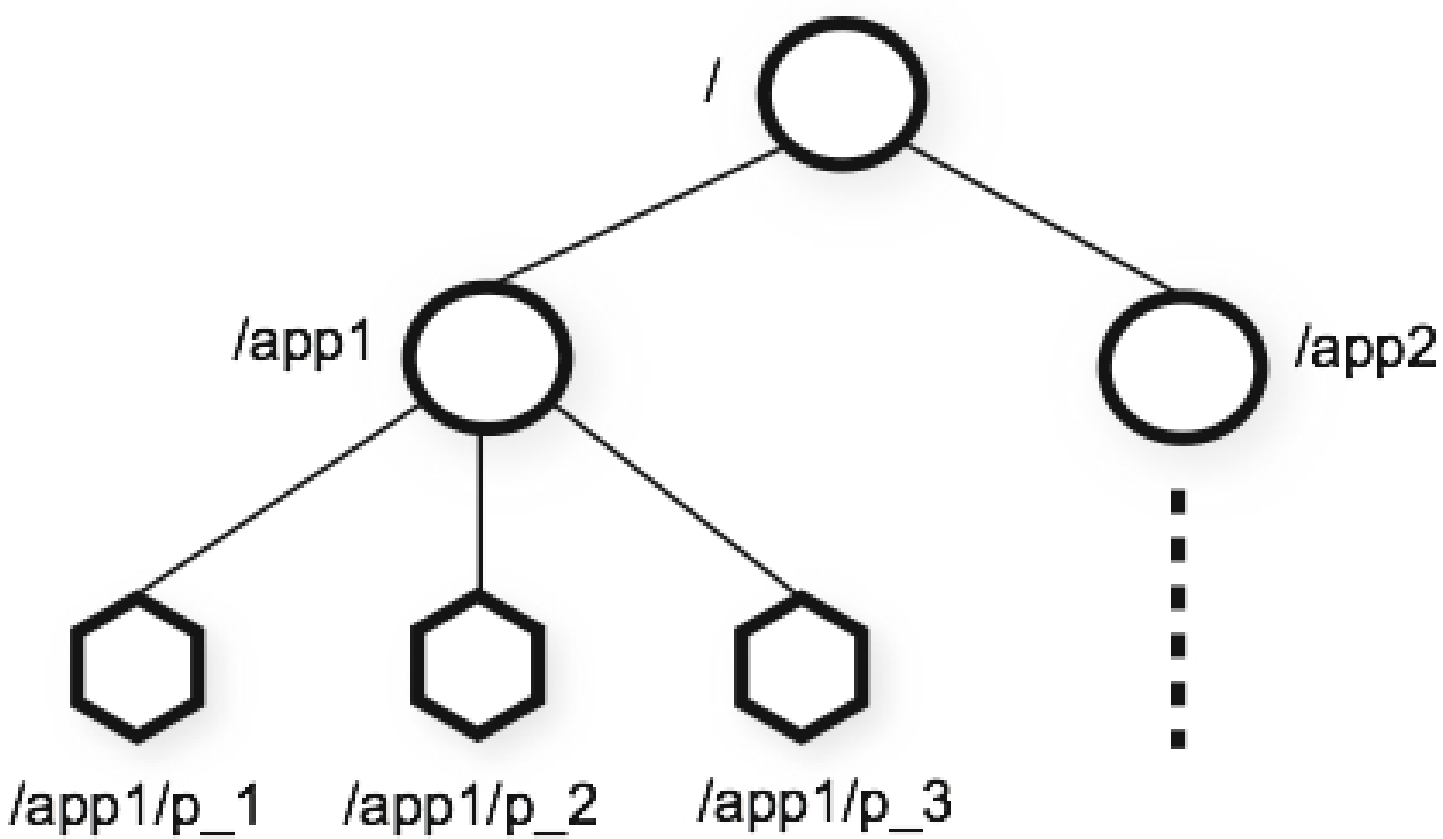
Apache Zookeeper

- 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

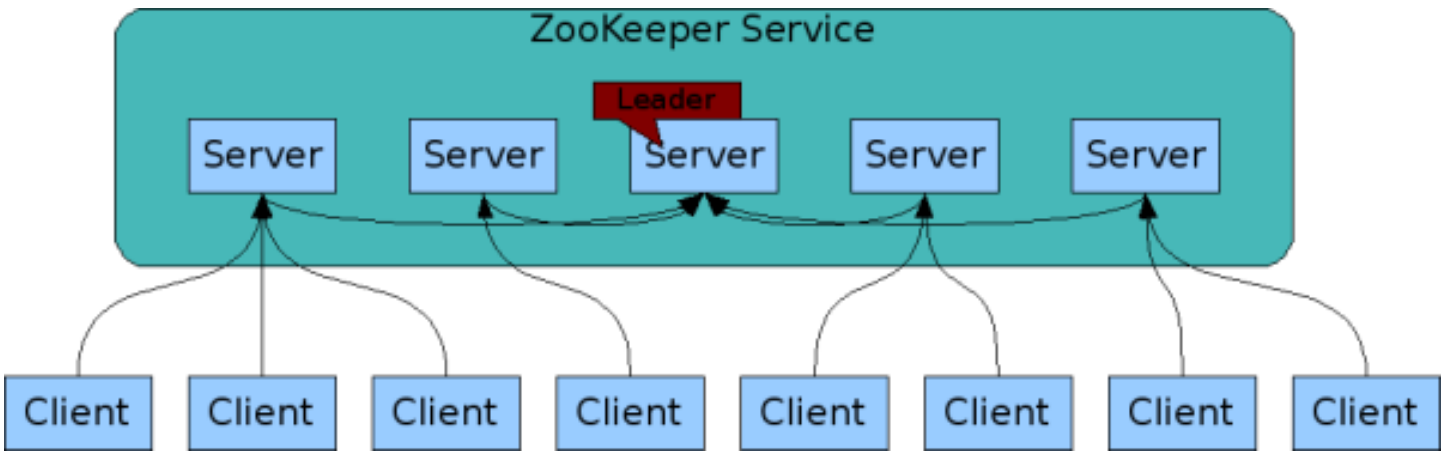


Ephemeral Znodes

- 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

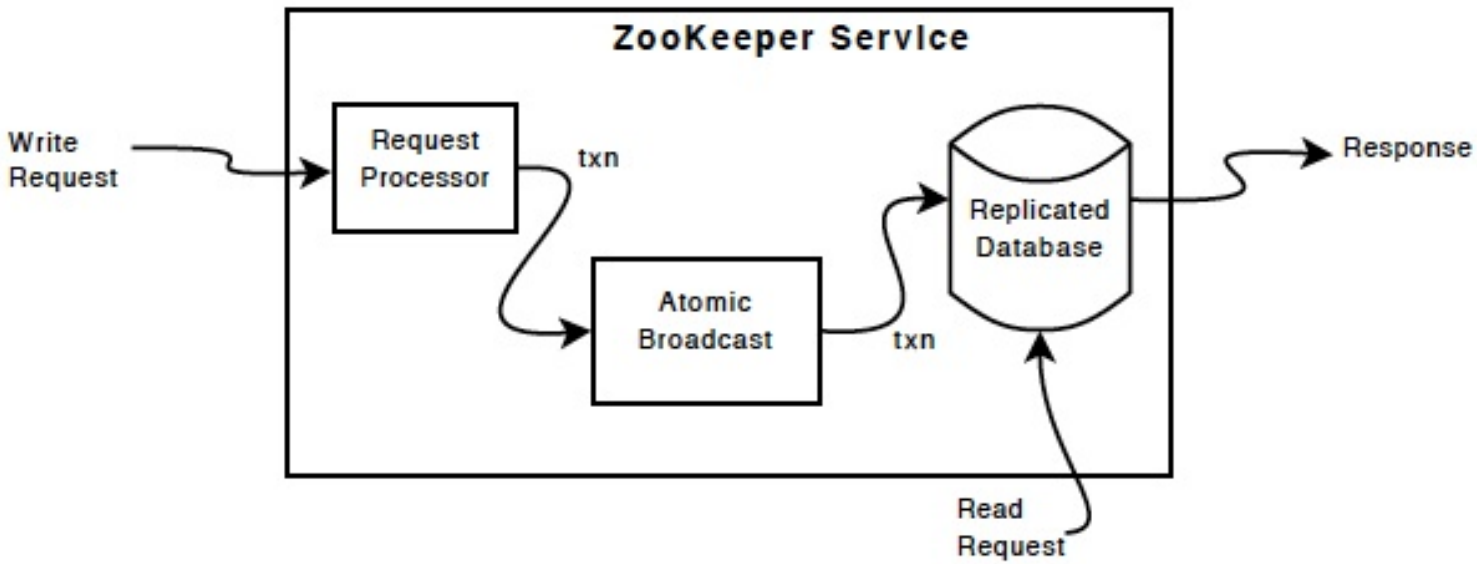
- Each Zookeeper client is connected to a single server
- All write requests are forwarded through a Leader node
- The Leader orders all writes into a total order, and forwards them to the Follower nodes. Writes are acked once a majority of servers have persisted the write



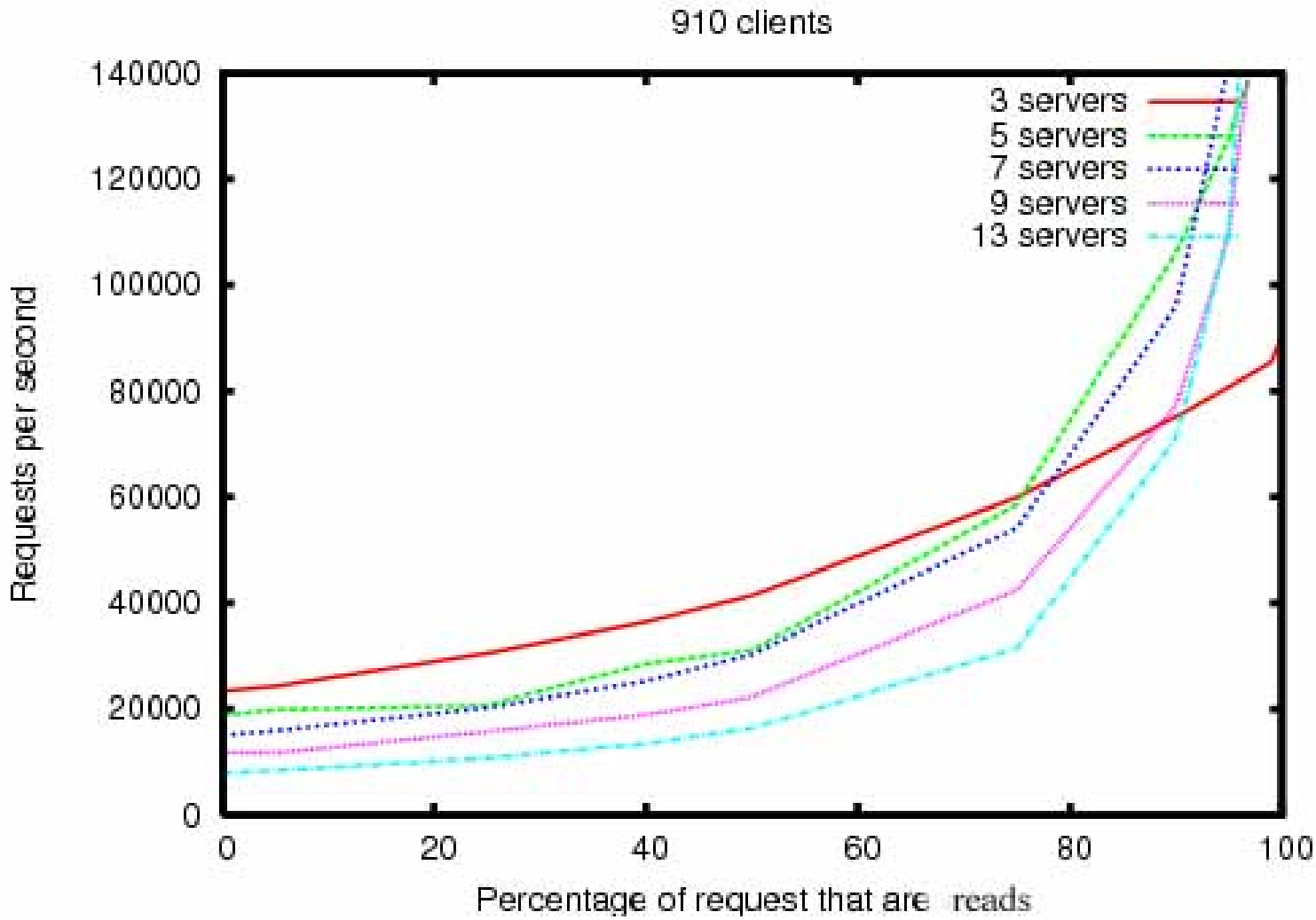
Zookeeper Characteristics

- 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



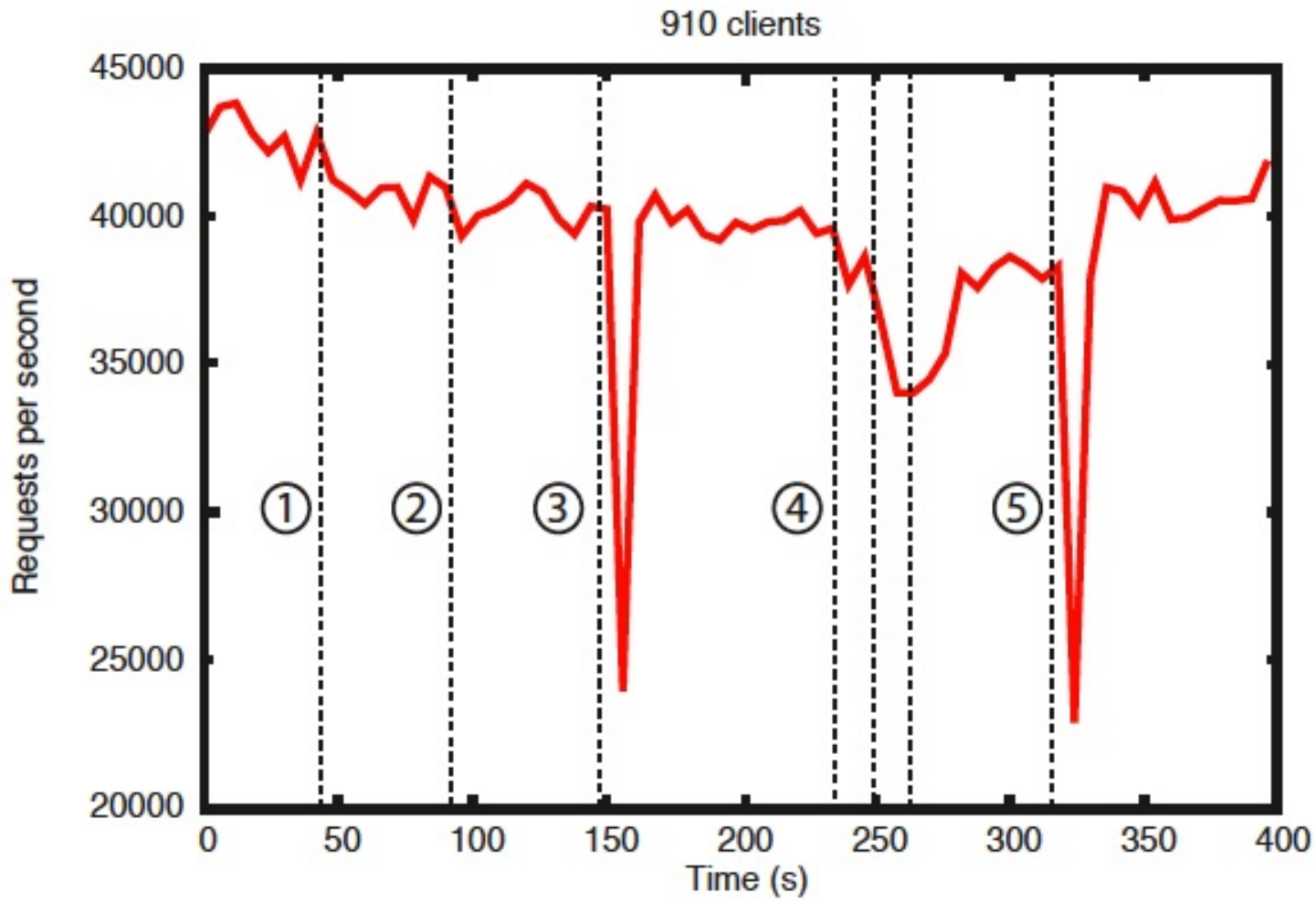
Zookeeper Performance



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

Zookeeper Performance under Failures



Zookeeper Performance under Failures

- Zookeeper survives the crashes of five different servers
- Killing the Zookeeper leader at “3” and and “5” creates a short period of reduced throughput from which the cluster survives
- Killing follower nodes does not change the throughput much

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

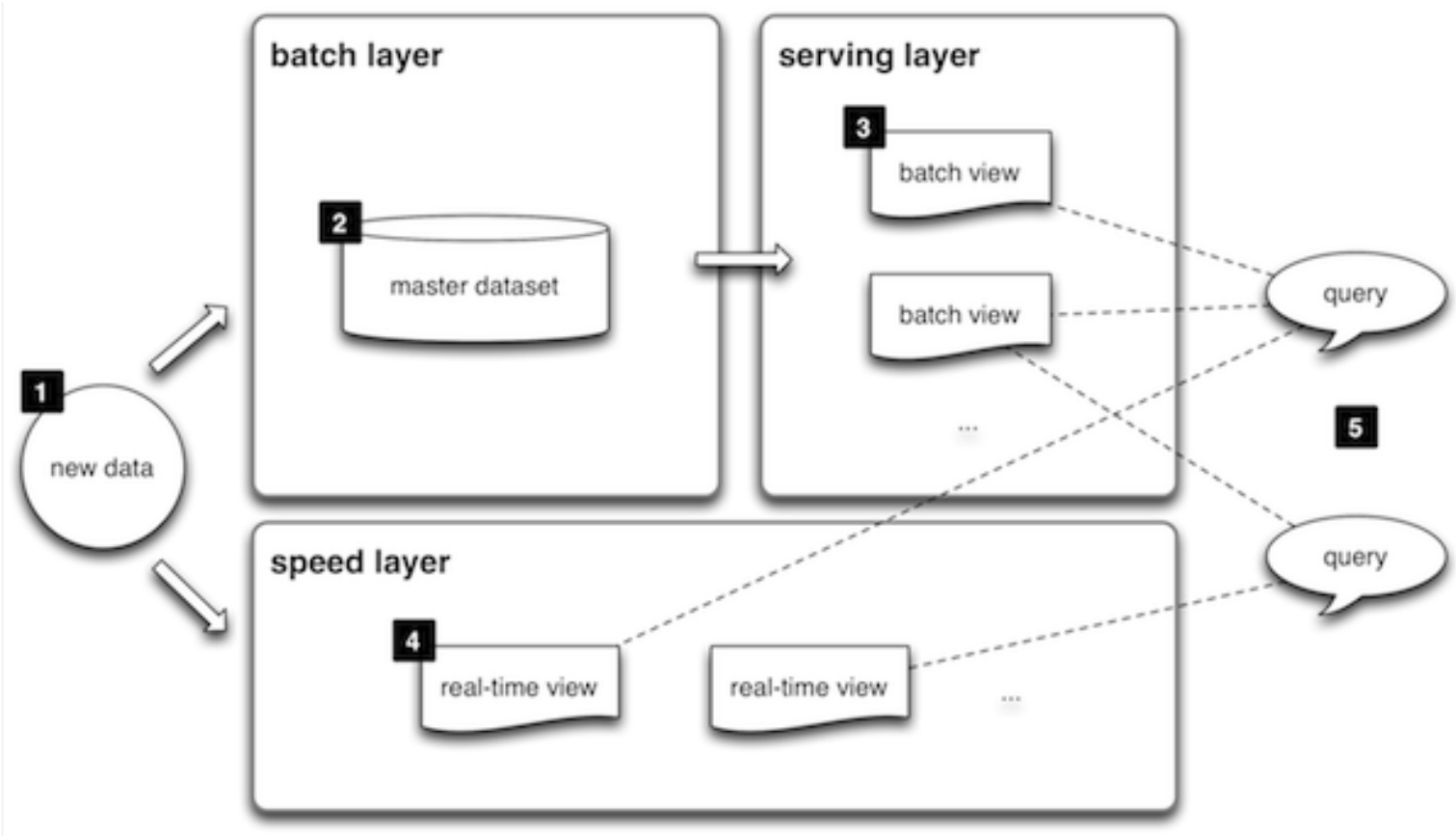
- Maintaining a global database image under node failures is a very subtle problem
- One should use centralized infrastructure such as Chubby, Zookeeper, or a Raft implementation to only implement these subtle algorithms only once
- From an applications point of view the distributed coordination services 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

Lambda Architecture

- Batch computing systems like MapReduce have a large latency in order of minutes to hours but have extremely good fault tolerance - each data item is processed exactly once (CP)
- Sometimes data needs to be processed in real time
- Stream processing systems such as Apache Storm allow for large scale real time processing without minimal fault tolerance - each data item is processed at least once (AP)
- These stream processing frameworks can provide latencies in the order of tens of milliseconds
- Lambda architecture is the combination of consistent / exact but high latency batch processing and available / approximate low latency stream processing

Lambda Architecture (cnt.)

- For more details, see e.g.,:
  `http://lambda-architecture.net/`



Lambda Architecture (cnt.)

- Batch layer - exact computation with high latency, maintaining master dataset, CP system
- Speed layer - approximate computation (minimal fault tolerance) with low latency, only maintaining approximations on recent data that is not yet processed by the batch layer, AP system
- Serving layer - serves the output of the batch layer and speed layer
- Lambda architecture allows for both low latency but also "eventual accuracy" of the approximate results, as these approximations are all eventually overwritten by exact results computed by the batch layer, combination of AP and CP systems

Stream Processing Frameworks

- One of the motivations for the Lambda architecture was the lack of fault tolerance in the Stream Layer
- New Streaming Frameworks have been introduced that have fult tolerance:
  - Spark Streaming - Extension of Spark to stream processing using micro-batching
  - Apache Flink - Open source low latency streaming framework
  - Google Cloud Dataflow
  - Azure Stream Analytics
  - Apache Beam - Open source programming framework for batch and streaming, originally create by Google. Runners for: Google Cloud Dataflow, Spark, Flink, etc.
- Stream processing systems require a fault-tolerant pub-sub messaging system, e.g.: Apache Kafka (Open Source), Amazon Kinesis, Azure Event Hub

Apache Kafka

- An Open Source fault tolerant Publish/Subscribe messaging platform
- Uses Zookeeper for storing slowly changing management and configuration data (`etcd` does the same in Kubernetes)
- For implementing fault tolerance uses a custom replication algorithm similar in spirit to Paxos, Zab, and Raft but differing in technical details
- Uses a very large number of Pub/Sub queues called partitions in a topic. Each one of the queues has a separate leader, thus the leader bottleneck is avoided by using a very large number of leaders.
- Similar ideas are used in many sharded distributed databases, for example Apache Kudu, CockroachDB, and Google Spanner. This does not, however, solve the global transactions problem.