Distributed Storage Systems part 1

Marko Vukolić

Distributed Systems and Cloud Computing

This part of the course (5 slots)

- **Distributed Storage Systems**
  - CAP theorem and Amazon Dynamo
  - Apache Cassandra

- **Distributed Systems Coordination**
  - Apache Zookeeper
  - Lab on Zookeeper

- **Cloud Computing summary**

General Info

- **No course notes/book**
- **Slides will be verbose**
- **List of recommended and optional readings**
  - On the course webpage
    - http://www.eurecom.fr/~michiard/teaching/clouds.html

Today

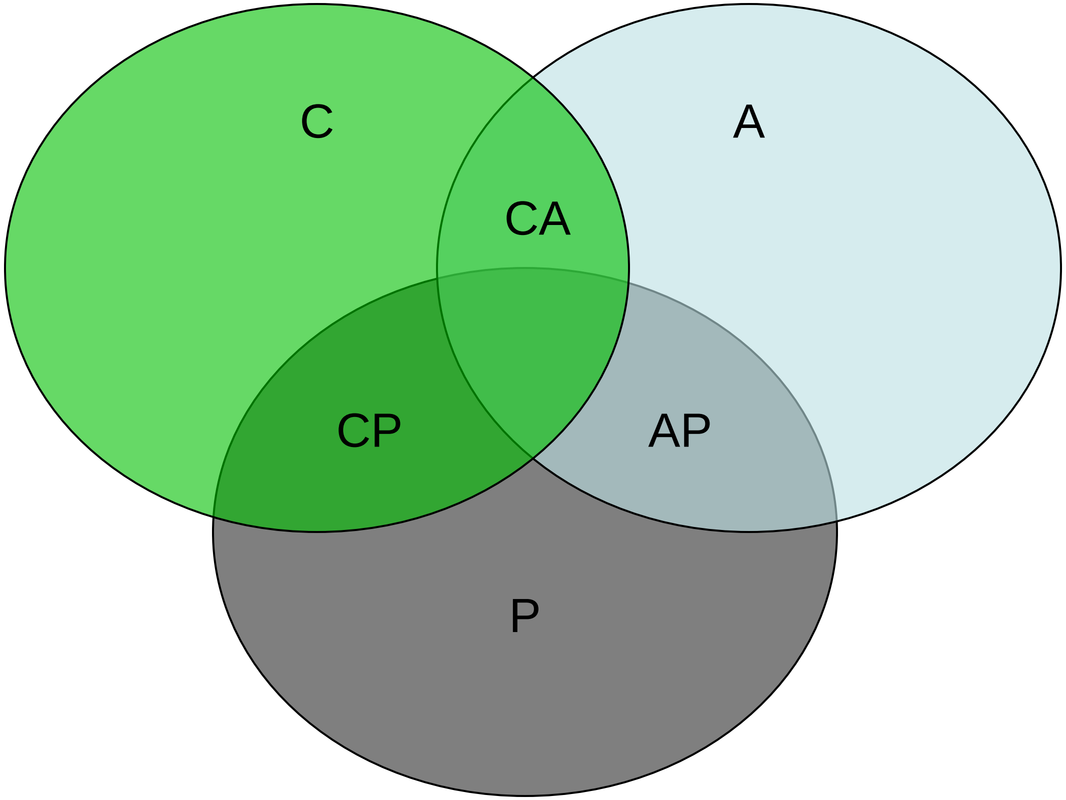
- **Distributed Storage systems part 1**
  - CAP theorem
  - Amazon Dynamo

CAP Theorem

- **Probably the must cited distributed systems theorem these days**
- **Relates the following 3 properties**
  - C: Consistency
    - One-copy semantics, linearizability, atomicity, total-order
    - Every operation must appear to take effect in a single indivisible point in time between its invocation and response
  - A: Availability
    - Every client's request is served (receives a response) unless a client fails (despite a strict subset of server nodes failing)
  - P: Partition-tolerance
    - A system functions properly even if the network is allowed to lose arbitrarily many messages sent from one node to another

CAP Theorem

- **In the folklore interpretation, the theorem says**
  - C, A, P: pick two!



Be careful with CA

- **Sacrificing P (partition tolerance)**
- **Negating**
  - A system functions properly even if the network is allowed to lose arbitrarily many messages sent from one node to another
- **Yields**
  - A system does not function properly even if the network is allowed to lose arbitrarily many messages sent from one node to another
    - This boils down to sacrificing C or A (the system does not work)
  - Or… (see next slide)

Be careful with CA

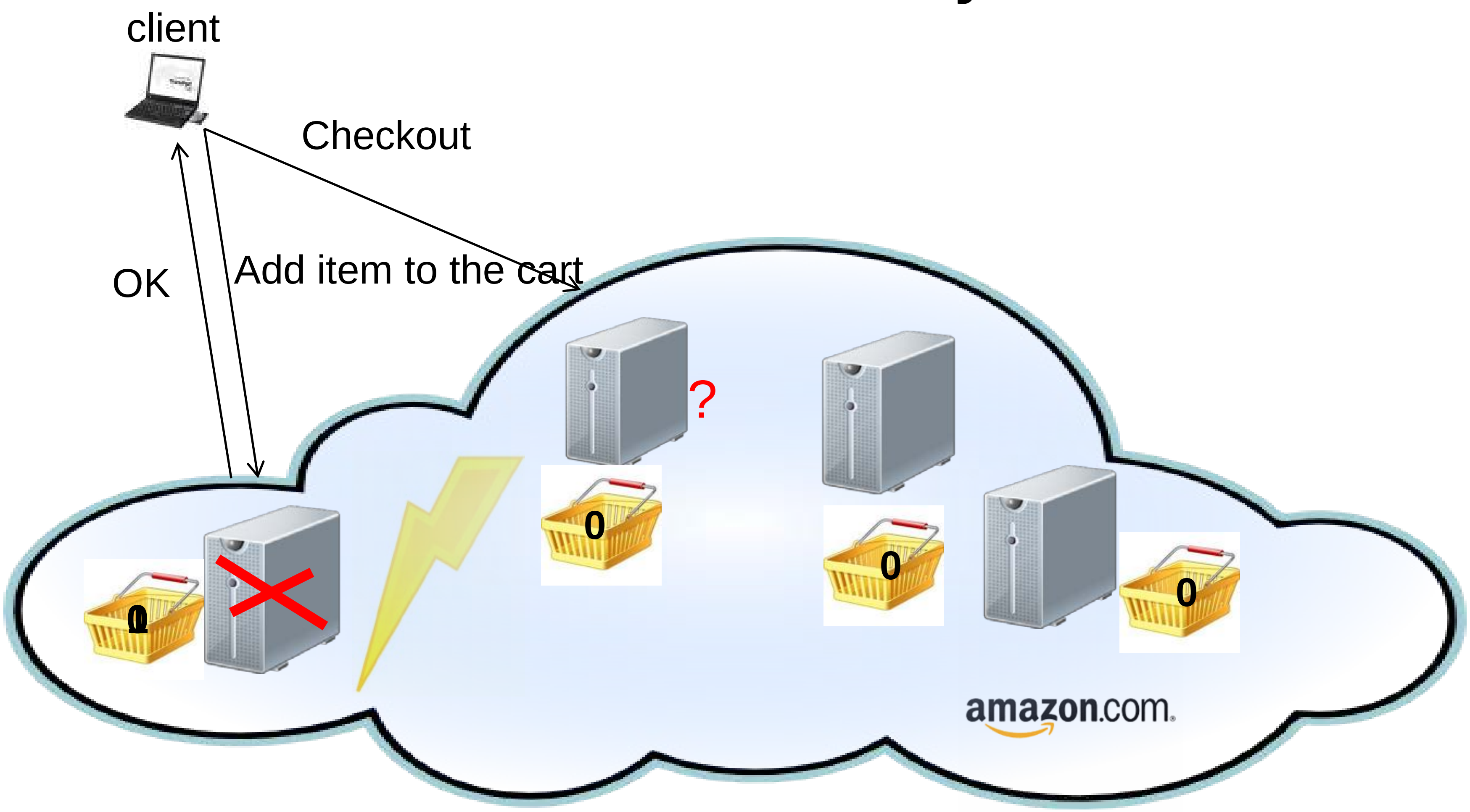
- **Negating P**
  - A system function properly if the network is **not** allowed to lose arbitrarily many messages
- **However, in practice**
  - One cannot choose whether the network will lose messages (this either happens or not)
- **One can argue that not “arbitrarily” many messages will be lost**
  - But “a lot” of them might be (before a network repairs)
  - In the meantime either C or A is sacrificed

CAP in practice

- **In practical distributed systems**
  - Partitions may occur
  - This is not under your control (as a system designer)
- **Designer's choice**
  - You choose whether you want your system in C or A when/if (temporary) partitions occur
  - Note: You may choose neither of C or A, but this is not a very smart option
- **Summary**
  - Practical distributed systems are either in CP or AP

CAP proof (illustration)

- **We cannot have a distributed system in CAP**



CAP Theorem

- **First stated by Eric Brewer (Berkeley) at the PODC 2000 keynote**
- **Formally proved by Gilbert and Lynch, 2002**
  - Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2): 51-59 (2002)
- **NB: As with all impossibility results mind the assumptions**
  - May do nice stuff with different assumptions
- **For DistAlgo students**
  - Yes, CAP is a “younger sibling” of the FLP impossibility

Gilbert/Lynch theorems

- **Theorem 1**

  It is impossible in the **asynchronous** network model to implement a read/write data object that guarantees

  - Availability
  - Atomic consistency

  in all <u>fair</u> executions (including those in which messages are lost)

asynchronous networks: no clocks, message delays unbounded

Gilbert/Lynch theorems

- **Theorem 2**

  It is impossible in the **partially synchronous** network model to implement a read/write data object that guarantees

  - Availability
  - Atomic consistency

  in all executions (including those in which messages are lost)

  partially synchronous networks: bounds on:

  a) time it takes to deliver messages that are not lost and

  b) message processing time,

  exist and are known, but process clocks are not synchronized

EURECOM

Gilbert/Lynch tCA

- **t-connected Consistency, Availability and Partition tolerance can be combined**
- **t-connected Consistency (roughly)**
  - w/o partitions the system is consistent
  - In the presence of partitions stale data may be returned (C may be violated)
  - Once a partition heals, there is a time limit on how long it takes for consistency to return
- **Could define t-connected Availability in a similar way**

CAP: Summary

- **The basic distributed systems/cloud computing theorem stating the tradeoffs among different system properties**

- **In practice, partitions do occur**
  - In pick C or A

- **The choice (C vs. A) heavily depends on what your application/business logic is**

CAP: some choices

- **CP**
  - BigTable, Hbase, MongoDB, Redis, MemCacheDB, Scalaris, etc.
  - (sometimes classified in CA) Paxos, Zookeeper, RDBMSs, etc.

- **AP**
  - Amazon Dynamo, CouchDB, Cassandra, SimpleDB, Riak, Voldemort, etc.

Amazon Dynamo

Amazon Web Services (AWS)

- [Vogels09] **At the foundation** of Amazon's cloud computing are infrastructure services such as
  - Amazon's S3 (Simple Storage Service), SimpleDB, and EC2 (Elastic Compute Cloud)
  - These provide the resources for constructing Internet-scale computing platforms and a great variety of applications.
- **The requirements placed on these infrastructure services are very strict; need to**
  - Score high in security, scalability, availability, performance, and cost-effectiveness, and
  - Serve millions of customers worldwide, continuously.

AWS

- **Observation**
  - Vogels does not emphasize consistency
  - AWS is in AP, sacrificing consistency
- **AWS follows BASE philosophy**
- **BASE (vs ACID)**
  - Basically Available
  - Soft state
  - Eventually consistent

Why Amazon favors availability over consistency?

"even the slightest outage has significant financial consequences and impacts customer trust"

- **Surely, consistency violations may as well have financial consequences and impact customer trust**
  - But not in (a majority of) Amazon's services
  - NB: Billing is a separate story

Amazon Dynamo

- **Not exactly part of the AWS offering**
  - however, Dynamo and similar Amazon technologies **are** used to power parts of AWS (e.g., S3)
- **Dynamo powers internal Amazon services**
- **Hundreds of them!**
  - Shopping cart, Customer session management, Product catalog, Recommendations, Order fullfillment, Bestseller lists, Sales rank, Fraud detection, etc.
- **So what is Amazon Dynamo?**
  - A highly available key-value storage system
  - Favors high availability over consistency under failures

Key-value store

- **put(key, object)**
- **get(key)**
  - We talk also about *writes/reads (the same here as put/get)*
- **In Dynamo case, the put API is put(key, context, object)**
  - where context holds some critical metadata (will discuss this in more details)
- **Amazon services (see previous slide)**
  - Predominantly do not need transactional capabilities of RDBMs
  - Only need primary-key access to data!
- **Dynamo: stores relatively small objects (typically <1MB)**

Amazon Dynamo: Features

- **High performance (low latency)**
- **Highly scalable (hundreds of server nodes)**
- **"Always-on" available (especially for writes)**
- **Partition/Fault-tolerant**
- **Eventually consistent**
- **Dynamo uses several techniques to achieve these features**
  - Which also comprise a nice subset of a general distributed system toolbox

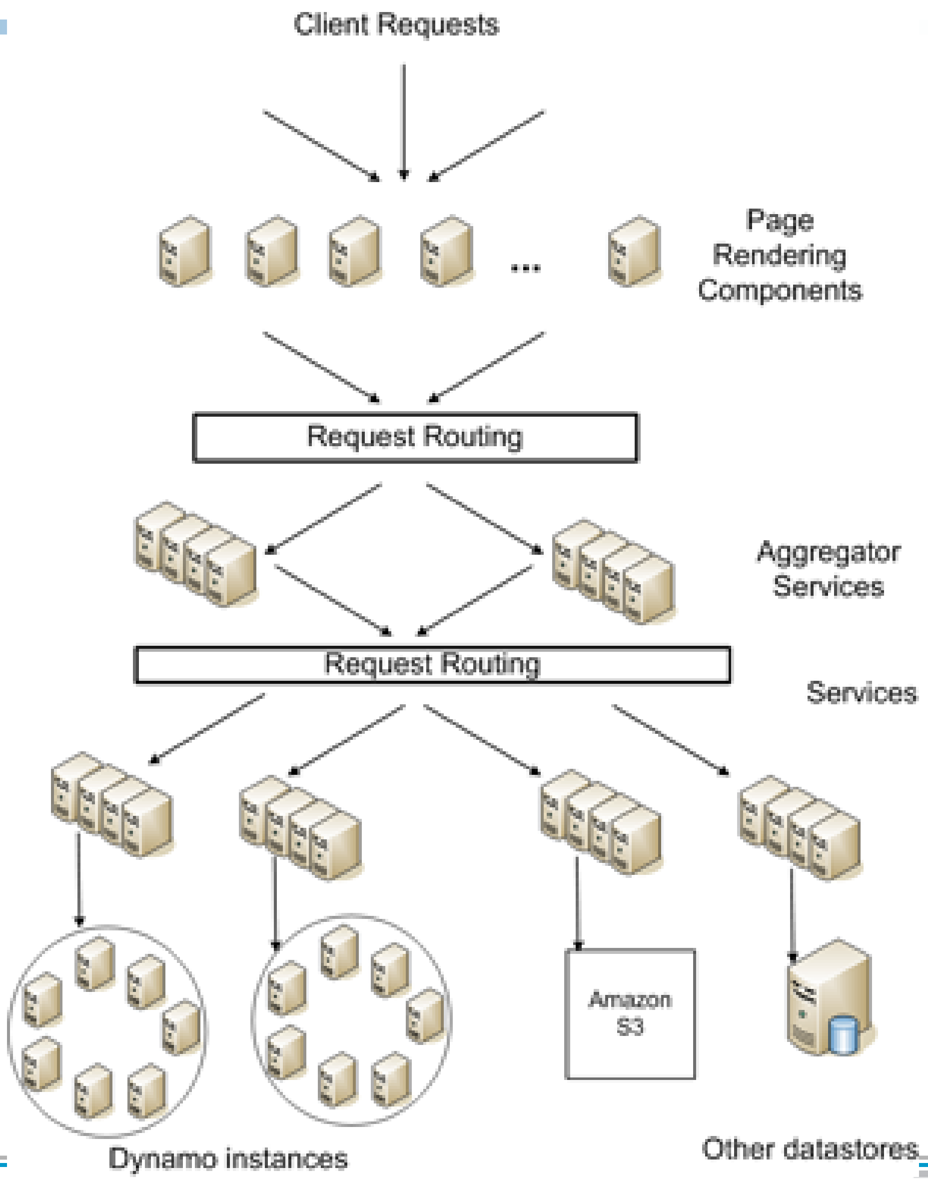
EURECOM

Amazon Dynamo: Key Techniques

- **Consistent hashing [Karger97]**
  - For data partitioning, replication and load balancing
- **Sloppy Quorums**
  - Boosts availability in presence of failures
  - might result in inconsistent versions of keys (data)
- **Vector clocks [Fidge88/Mantern88]**
  - For tracking causal dependencies among different versions of the same key (data)
- **Gossip-based group membership protocol**
  - For maintaining information about alive nodes
- **Anti-entropy protocol using hash/Merkle trees**
  - Background synchronization of divergent replicas

Amazon SOA platform

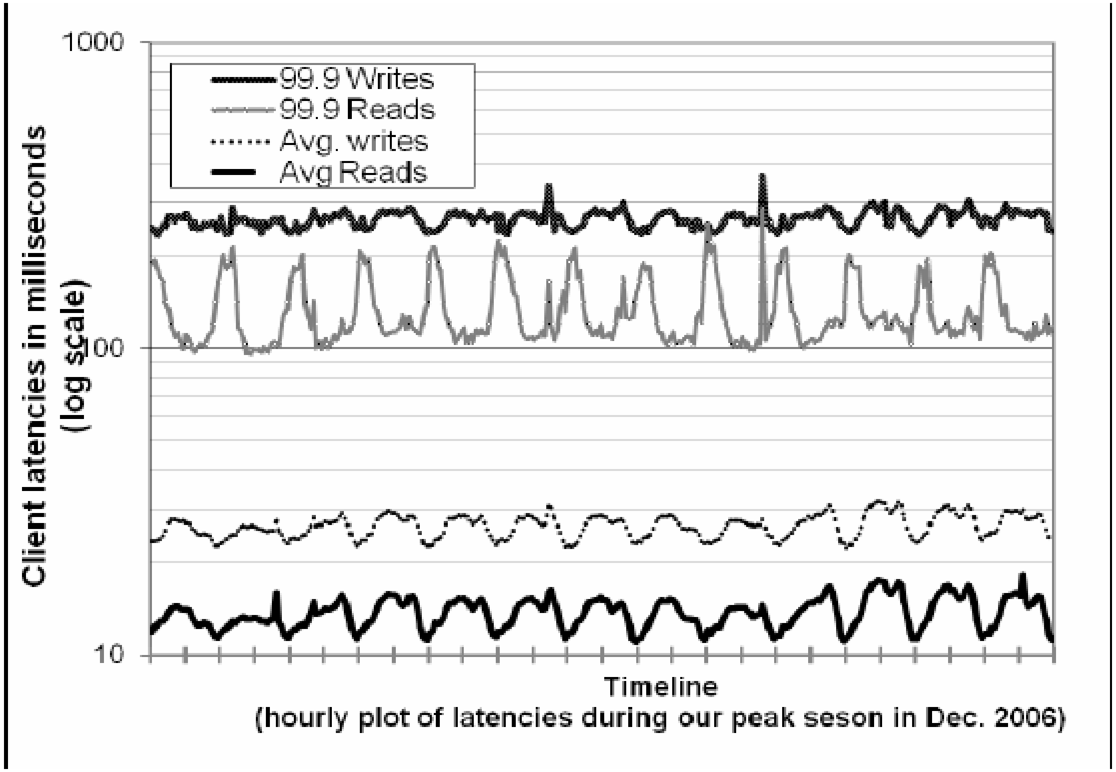
- **Runs on commodity hardware**
  - ➢ NB: This is low-end server class rather than low-end PC
- **Stringent Latency requirements**
  - ➢ Measured at 99.9%
  - ➢ Part of SLAa
- **Every service runs its own Dynamo instance**
  - ➢ Only internal services use Dynamo
  - ➢ No Byzantine nodes



EURECOM

SLAs and three nines

- **Sample SLA**
  - A service XYZ guarantees to provide a response within 300 ms for 99.9% of requests for a peak load of 500 req/s
- **Amazon focuses on 99.9 percentile**



EURECOM

Dynamo design decisions

- **“always-writable” data store**
  - Think shopping cart: must be able to add/remove items
- **If unable to replicate the changes?**
  - Replication is needed for fault/disaster tolerance
  - Allow creations multiple versions of data (vector clocks)
  - Reconcile and resolve conflicts during reads
- **How/who should reconcile**
  - Application: depending on e.g., business logic
    - Complicates programmer’s life, flexible
  - Dynamo: deterministically, e.g., “last-write” wins
    - Simpler, less flexible, might loose some value wrt. Business logic

Dynamo architecture

Dynamo architecture

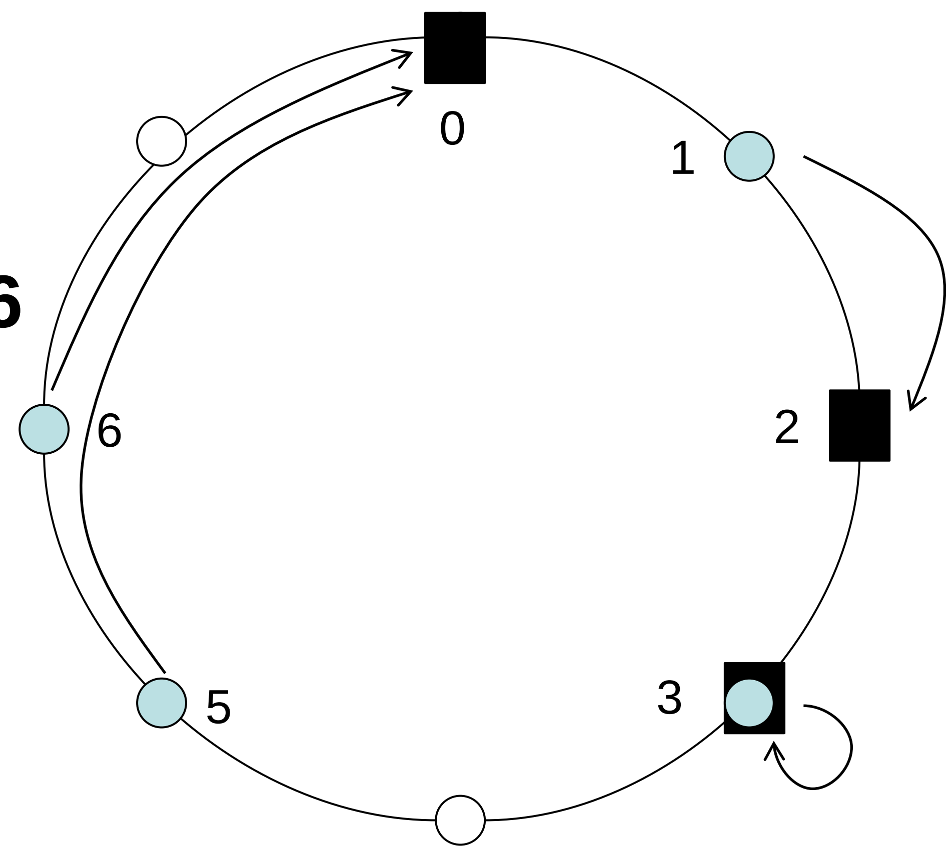
- **Scalable and robust components for**
  - Load balancing, membership/fault detection, failure recovery, replica synchronization, overload handling, state transfer, concurrency, job scheduling, request marshalling, request routing, system monitoring and alarming, configuration management
- **We focus on techniques for**
  - Partitioning, replication, versioning, membership, failure-handling, scaling

Partitioning using consistent hashing

- **Dynamo dynamically partitions a set of *keys* over a set of storage *nodes***
  - Used also in many DHTs (e.g., Chord)
- **Hashes (MD5, can use SHA-1,…) of *keys* (resp., node IP) give key (resp., node) m-bit *identifiers***
- **Consistent hashing**
  - Identifiers are ordered in an identifier circle
- **Partitioning**
  - A key is assigned to the closest **successor node** id
  - i.e., key k is assigned to the first node with id $\geq k$
    - ☞ or if such a node does not exist to the node with smallest id (circle)

Consistent hashing: Example

- **m=3: 3-bit namespace**
- **3 nodes (0,2,3)**
- **4 keys (1,3,5,6)**
- **Node 0 stores keys 5,6**
- **Node 2 stores key 1**
- **Node 3 stores key 3**



Consistent hashing

- **Designed to let nodes enter and leave the network with minimal disruption**
  - Key to incremental scalability

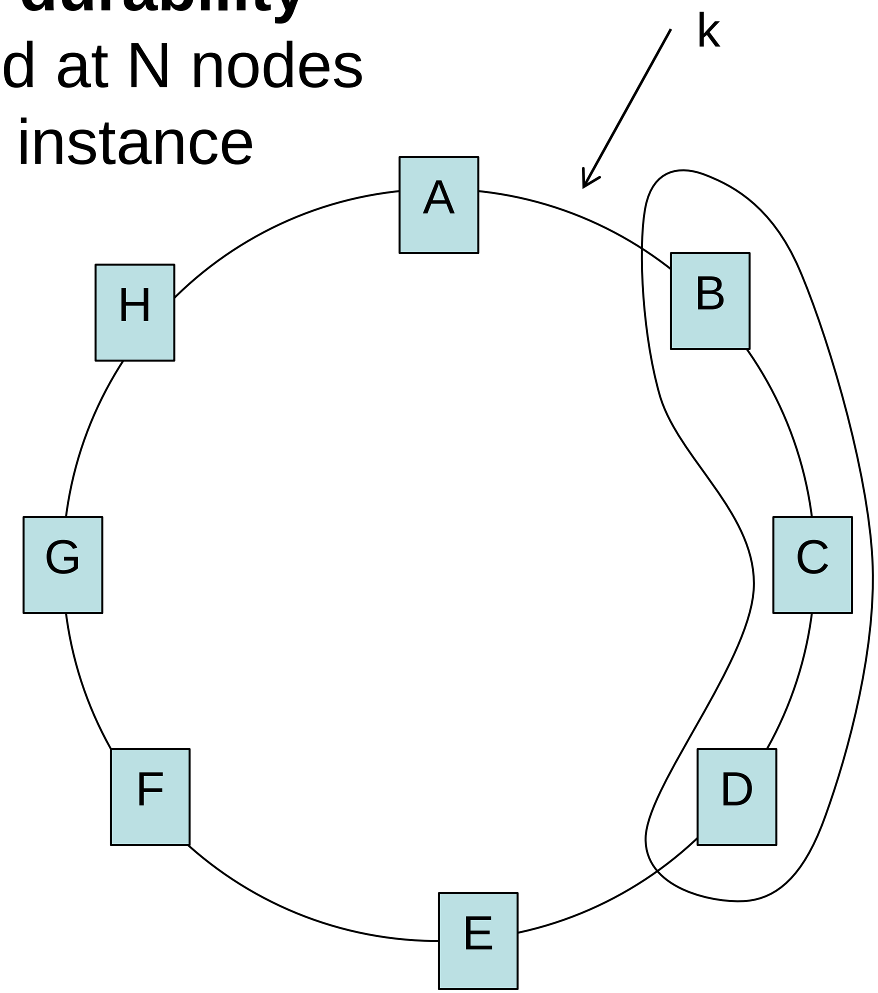
- **Maintainance**
  - When node *n* joins
    - certain keys previously assigned to *n*'s successor now become assigned to *n*.
  - When node *n* leaves
    - all of *n*'s assigned keys are reassigned to *n*'s successor.

Consistent hashing: Properties

- **Assume N nodes and K keys. Then (with high probability) [Karger97]**
  - Each node is responsible for at most $(1+\varepsilon)K/N$ keys
  - When N+1st node joins/leaves, $O(K/N)$ keys change hands (optimal)
- **$\varepsilon = O(\log N)$**
  - Can have $\varepsilon \rightarrow 0$ with "virtual" nodes
- **"Virtual" nodes**
  - Each physical node mapped multiple times to the circle
    - ☞**Load balancing!**
  - Dynamo employs virtual nodes — also in order to leverage heterogeneity among physical nodes

Replication

- **To achieve high availability and durability**
  - Each data item (key) replicated at N nodes
  - N is configurable per Dynamo instance
- **Assume N=3**
  - For key k, B is the 1st successor node (coordinator)
  - B replicates k to N-1 further successor nodes (C and D)
- **B, C and D**
  - are ***preference list*** for k
- **Virtual nodes**
  - Same physical nodes skipped in a preference list



Data versioning

- **Replication performed after a response is sent to a client**
  - This is called *asynchronous replication* (not to be confused with the state machine replication in the asynchronous network model)
  - May result in inconsistencies under partitions
    - ☞Read does not return the last value. **Eventual consistency!**
- **But operations should not be lost**
  - "add to cart" should not be rejected but also not forgotten
  - If "add to cart" is performed when latest version is not available it is performed on an older version
  - We may have different versions of a key/value pair

Data versioning

- **Once a partition heals versions are merged**
  - The goal is not to lose any “add to cart”
- **Most of the time there will be no partitions and the system will be consistent**
  - New versions subsume all previous ones
- **It is vital to understand that the application must know that different versions might exist**
  - This is the Achilles’ heel of eventual consistency (more difficult to reason about, program with)
- **Key data versioning technique: Vector clocks**
  - Capture causality between different versions of an object

Vector clocks in Dynamo

- **Each write to a key k is associated with a vector clock VC(k)**
- **VC(k) is an array (map) of integers**
  - In theory: one entry VC(k)[i] for each node i
- **When node i handles a write of key k it increments VC(k)[i]**
  - VCs are included in the context of the put call
- **In practice:**
  - VC(k) will not have many entries (only nodes from the preference list should normally have entries), and
  - Dynamo truncates entries if more than a threshold (say 10)

Vector clocks in Dynamo

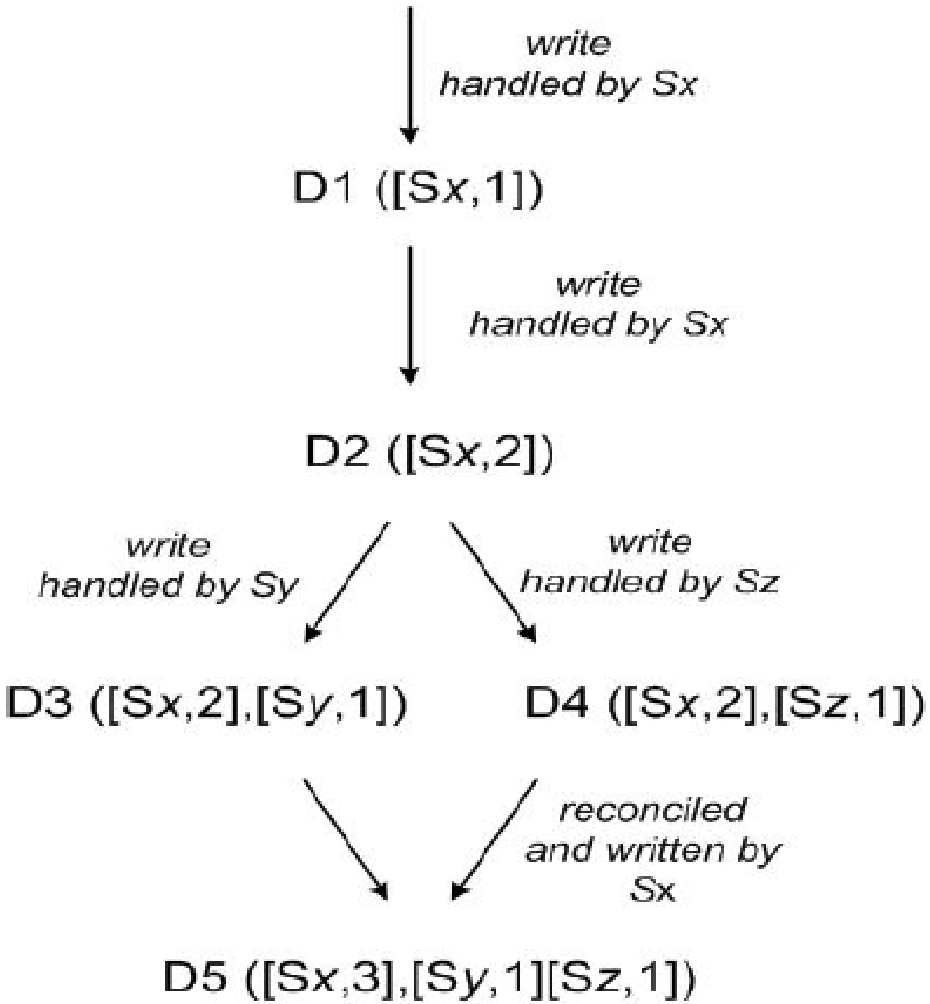


Figure 3: Version evolution of an object over time.

Number of different versions (#DV)

- **These are the evidence of consistency violations (#DV>1)**
- **24h experiment on the shopping cart**
  - #DV=1: 99.94% of requests (all but 1 in cca 1700 req)
  - #DV=2: 0.00057% of requests
  - #DV=3: 0.00047% of requests
  - …
- **Attributed to busy robots (automated client programs)**
  - Rarely visible to humans

Handling puts and gets (failure-free case)

- **Any Dynamo storage node can receive get/put request for any key. This node is selected by**
  - Generic load balancer
  - By a client library that immediately goes to coordinator nodes in a preference list
- **If the request comes from the load balancer**
  - Node serves the request only if in preference list
  - Otherwise, the node routes the request to the first node in preference list
- **Each node has routing info to all other nodes**
  - 0-hop DHT
  - ***Not the most scalable, but latency is critical***

Handling puts and gets

- **Extended preference list**
  - N nodes from preference list + some additional nodes (following the circle) to account for failures
- **Failure-free case**
  - Nodes from preference list are involved in get/put
- **Failures**
  - First N alive nodes from extended preference list are involved

Dynamo’s quorums

- **Two configurable parameters**
  - R number of nodes that need to participate in a get
  - W number of nodes that need to participate in a write
  - R + W > N (a quorum system)
- **Handling put (by coordinator)**   // rough sketch

  Generate new VC, Write new version locally

  Send value, VC to N selected nodes from preference list

  Wait for W-1
- **Handling get (by coordinator)**   // rough sketch

  Send READ to N selected nodes from preference list

  Wait for R

  Select highest versions per VC, return all such versions (causally unrelated)

  Reconcile/merge different versions

  Writeback reconciled version

Of choices of R, W

- **R, W smaller than N**
  - To decrease latency
  - Slowest replica dictates the latency

- **W=1**
  - Always-available for writes
  - Yields R=N (reads pay the penalty)

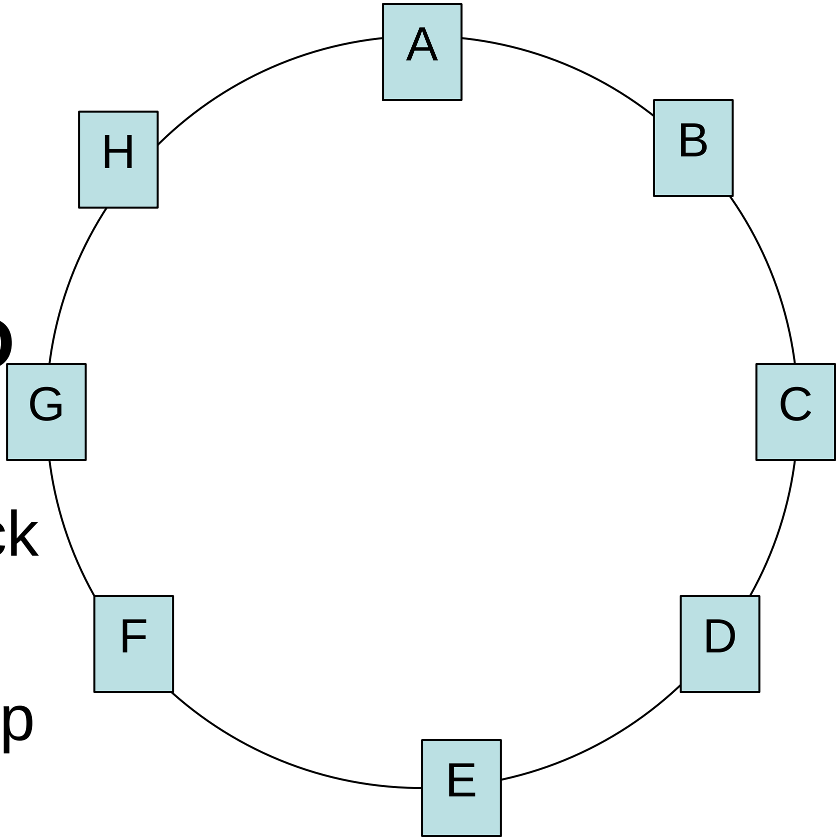
- **Most often in Dynamo (W,R,N)=(2,2,3)**

Handling failures

- **N selected nodes are the first N healthy nodes**
  - Might change from request to request
  - Hence these quorums are “Sloppy” quorums
- **“Sloppy” vs. strict quorums**
  - “sloppy” allow availability under a much wider range of partitions (failures) but sacrifice consistency
- **Also, important to handle failures of an entire data center**
  - Power outages, cooling failures, network failures, disasters
  - Preference list accounts for this (nodes spread across data centers)

Handling temporary failures: hinted handoff

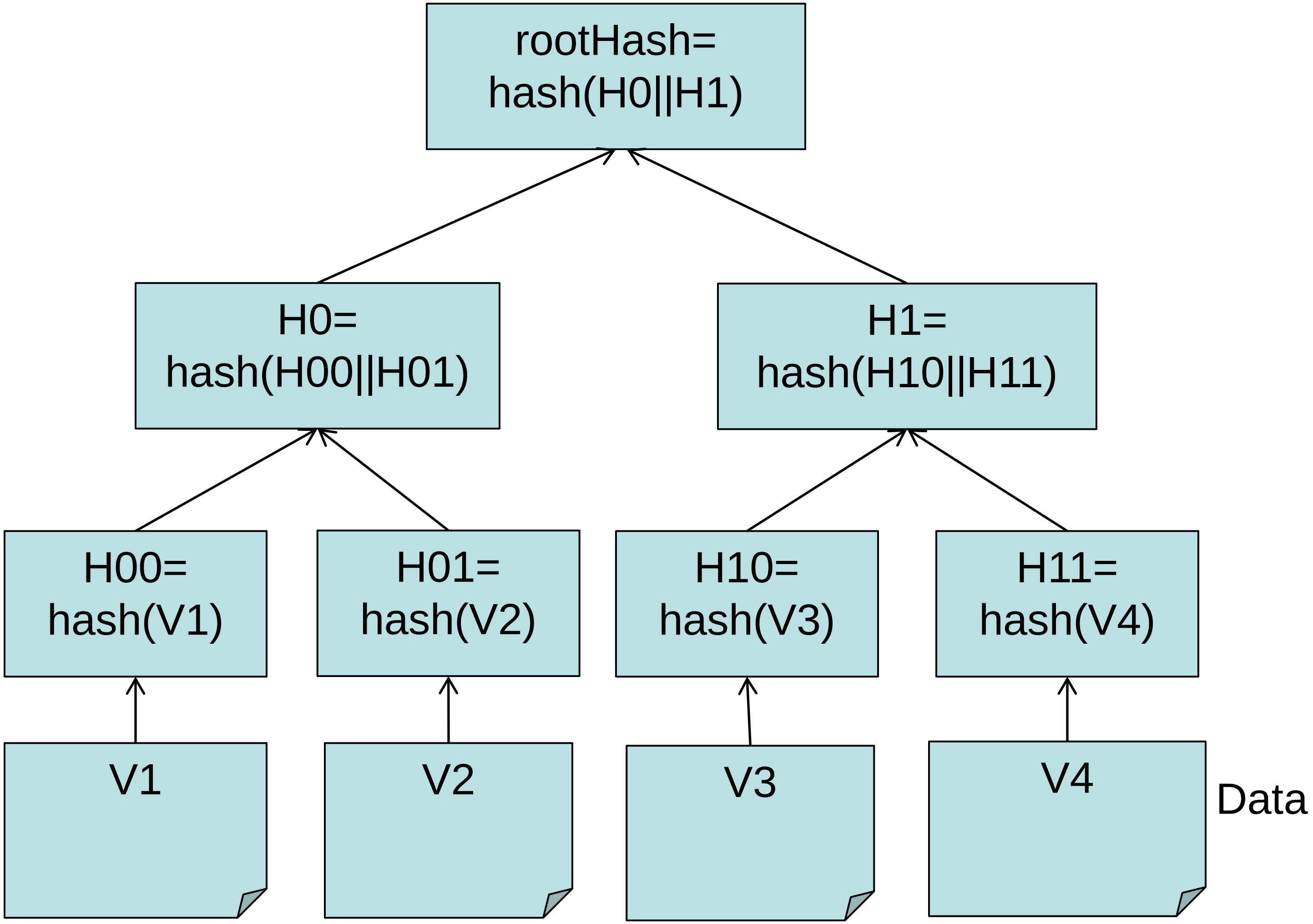
- **If a replica in the preference list is down then another replica is created on a new node**
- **Assume again N=3**
- **A replica A is down**
- **Coordinator will involve D**
  - With a hint that this D substitutes A until A comes back again
  - When D gets info A is back up it hands back the data to A



Anti-entropy synchronization using hash/Merkle trees

- **Each Dynamo node keeps a Merkle tree for each of its key ranges**
  - Remember, one key range per virtual node
- **Compares the root of the tree with replicas**
  - If equal, all keys in a range are equal (replicas in sync)
  - If not equal
    - Traverse the branches of the tree to pinpoint the children that differ
    - The process continues to all leaves
    - Synchronize on those keys that differ

Merkle trees



Membership

- **Node outages temporary**
  - Not considered as permanent leaves
- **Dynamo relies on administrator explicitly declaring joins/leaves on any Dynamo node**
  - This triggers membership changes (with the aid of seeds)
- **Membership info are also eventually consistent — propagated by background gossip protocol**
  - Node contacts a random node every 1s
  - 2 nodes reconcile the membership info
  - This gossip used also for exchanging partitioning/placement metadata

Failure detection

- **Unreliable failure detection (FD)**
  - Used, e.g., to refresh the healthy node info in the extended preference list
- **With steady load node A will find out if node B is unavailable**
  - E.g., if B does not respond to A's messages
  - But this is clearly unreliable, B might be partitioned not faulty
  - Then, A periodically checks on B to see if B recovers
- **In the absence of traffic A might not find out B is unavailable**
  - But this info is does not matter anyway w/o traffic
  - Dynamo has in-band FD, rather than a dedicated component

Dynamo: Summary

- **An eventually consistent highly available key value store**
  - AP in the CAP space
- **Focuses on low latency, SLAs**
  - Very low latency writes, reconciliation in reads
- **Key techniques used in many other distributed systems**
  - Consistent hashing, (sloppy) quorum-based replication, vector clocks, gossip-based membership, Merkle-tree based synchronization

Further reading (recommended)

Seth Gilbert, Nancy A. Lynch: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2): 51-59 (2002)

DeCandia et al. Dynamo: Amazon's highly available key-value store. SOSP 2007: 205-220 (2007)

Further Reading (optional)

- Eric A. Brewer: Pushing the CAP: Strategies for Consistency and Availability. IEEE Computer 45(2): 23-29 (2012)

- Seth Gilbert, Nancy A. Lynch: Perspectives on the CAP Theorem. IEEE Computer 45(2): 30-36 (2012)

- Marko Vukolić: Quorum Systems with Applications to Storage and Consensus. Morgan&Claypool (2012)

- Ion Stoica et al: Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Trans. Netw. 11(1): 17-32 (2003)