INGI2145: CLOUD COMPUTING (Fall 2015)

Cloud Basics & Storage

1 October 2015

Certain lecture slides adapted from Upenn NETS212 by A. Haeberlen, Z. Ives, Eurecom DSCC by P. Michiardi, M. Vukolic and from Cornell CS5412 by K. Birman. Reproduced with permission

Announcements

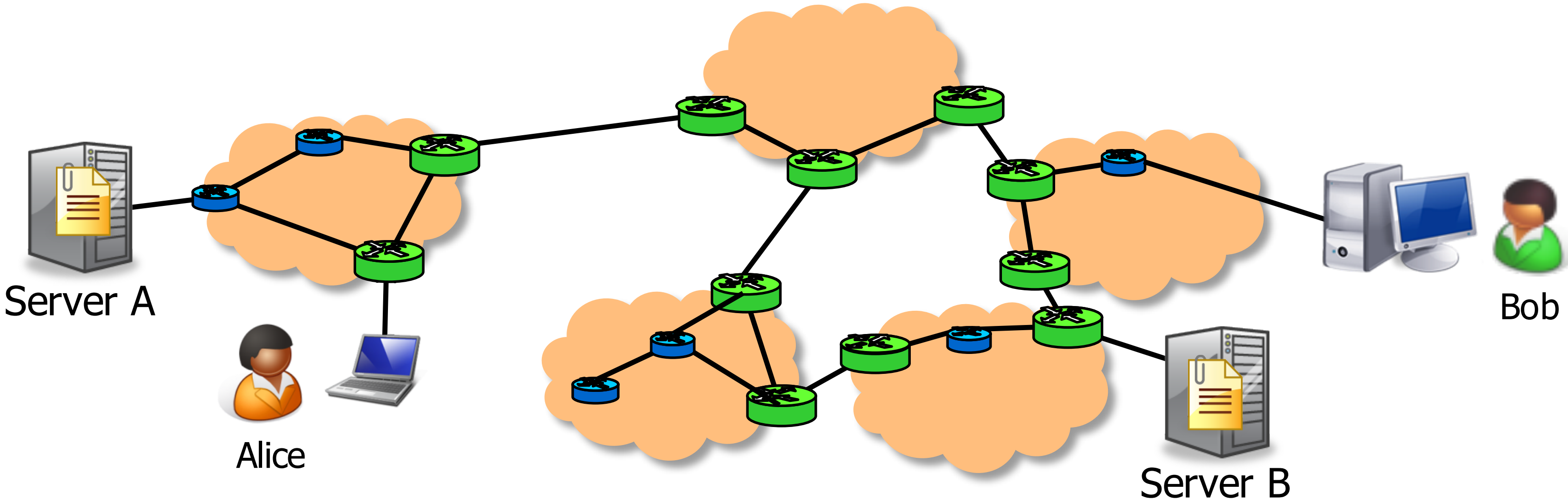
- First homework assignment will be announced next week
  - You will need to be setup with AWS
  - Next week's lab (9 Oct) will cover important background for this assignment
- Tomorrow's lab session is about Amazon Storage Services
  - Bring your own laptop
  - Lab will be in BARB 91
- Deadline of readings' quizzes extended to Thursdays at noon
  - Will post quizzes online 6 days in advance
- Will improve organization of information by having links on the website and a FAQ

Plan for today

- Distributed programming and its challenges
  - Faults, failures, and what we can do about them
  - Network partitions, CAP theorem, relaxed consistency
- Cloud basics
  - Anatomy of Cloud applications
  - Scaling: stateless, caching, and sharding
- Cloud storage
  - Overview
  - KVS and current systems
- Amazon Dynamo

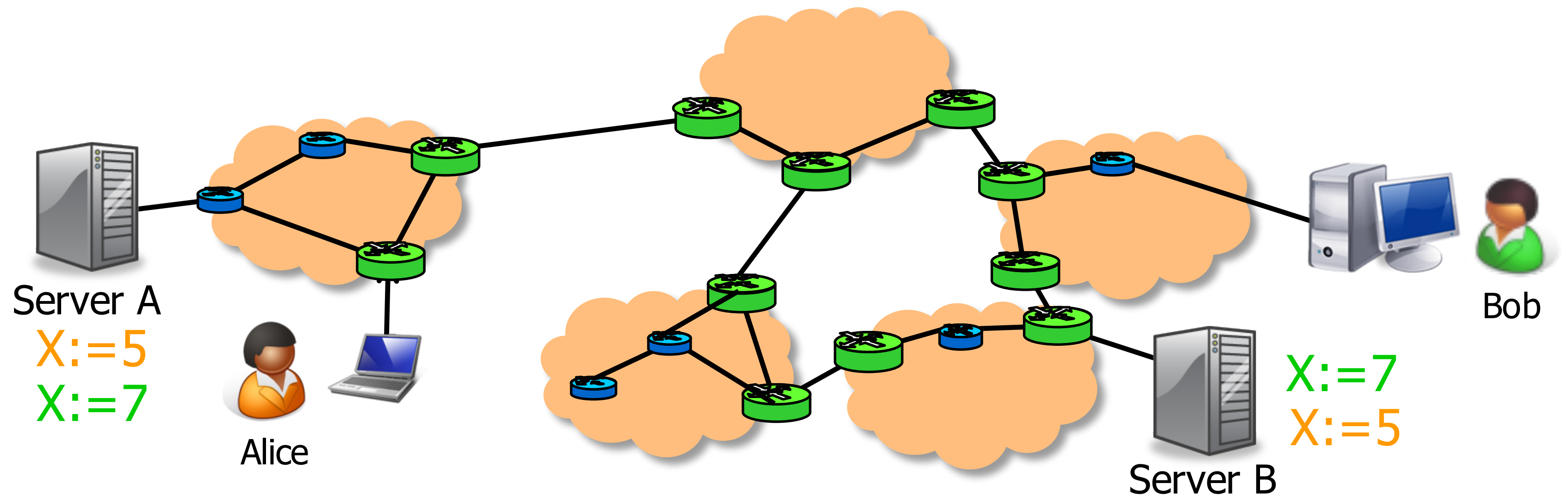
NEXT

Masking faults with replication



- Alice can store her data on both servers
- Bob can get the data from either server
  - A single crash fault on a server does not lead to a failure
  - Availability is maintained
  - What about other types of faults, or multiple faults?

Problem: Maintaining consistency



- What if multiple clients are accessing the same set of replicas?
  - Requests may be ordered differently by different replicas
  - Result: Inconsistency! (remember race conditions?)
  - For what types of requests can this happen?
  - What do we need to do to maintain consistency?

Types of consistency

- Strong consistency
  - After an update completes, any subsequent access will return the updated value
- Weak consistency
  - Updated value not guaranteed to be returned immediately, only after some conditions are met (inconsistency window)
- Eventual consistency
  - A specific type of weak consistency
  - If no new updates are made to the object, eventually all accesses will return the last updated value

Eventual consistency variations

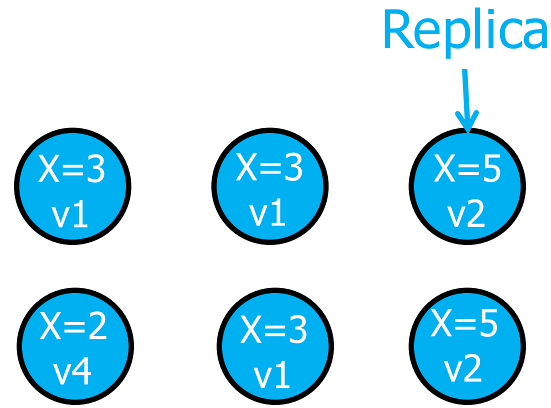
- Causal consistency
  - If client A has communicated to client B that it has updated a data item, a subsequent access by B will return the updated value, and a write is guaranteed to supersede the earlier write. Client C that has no causal relationship to client A is subject to the normal eventual consistency rules
- Read-your-writes consistency
  - Client A, after it has updated a data item, always accesses the updated value and will never see an older value
- Session consistency
  - Like previous case but in the context of a session, for as long as the sessions remains alive

Eventual consistency variations

- Monotonic read consistency
  - If client A has has seen a particular value for the object, any subsequent accesses will never return any previous values
- Monotonic write consistency
  - In this case the system guarantees to serialize the writes by the same process
  - Systems that do not guarantee this level of consistency are notoriously hard to program
- Few consistency properties can be combined
  - monotonic reads + read-your-writes most desirable for eventual consistency. Why?

Example: Storage system

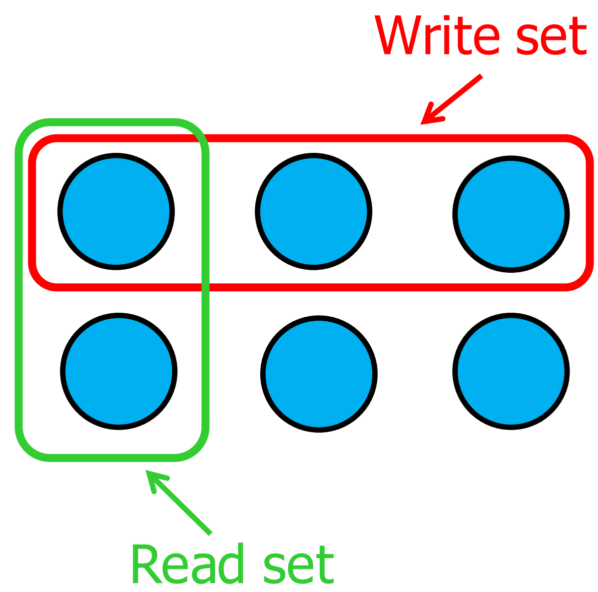
- Scenario: Replicated storage
  - We have N nodes that can store data
  - Data contains a monotonically increasing timestamp



- To write a value:
  - Pick W replicas and write the value to each, using a fresh timestamp (say, the current wallclock time)
- To read a value:
  - Pick R replicas and read the value from each
  - Return the value with the highest timestamp
  - If any replicas had a lower timestamp, send them the newer value

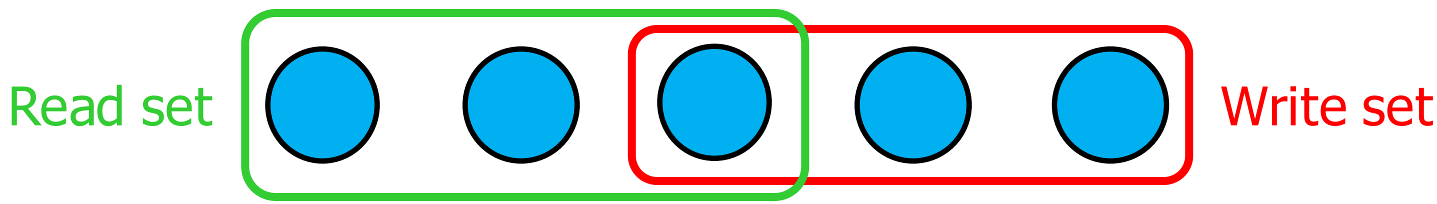
How to set N, R, and W

- For strong consistency?
  - What happens otherwise?
  - Will the data ever become consistent again?
- To avoid conflicting writes?
- To make reads fast? Writes?
- To minimize the risk of data loss?
- Let's do some examples!
  - N=2, W=2, R=1
  - N=2, W=1, R=1



Strong consistency: Quorum principle

- Majority quorum
  - Always write to and read from a majority of nodes
    - At least one node knows the most recent value
  - Pro: tolerate up to $\lceil N/2 \rceil - 1$ crashes
  - Con: have to read/write $\lfloor N/2 \rfloor + 1$ values

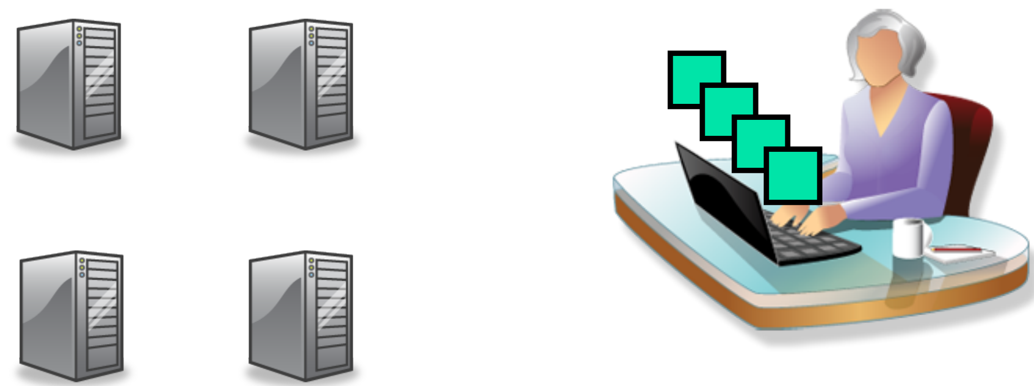


- Read/write quorums
  - Read R nodes, write W nodes, s.t. $R + W > N$
  - Pro: adjust performance of reads/writes
  - Con: availability can suffer

Consensus

- Replicas need to agree on a single order in which to execute client requests
  - How can we do this?
  - Does the specific order matter?
- Problem: What if some replicas are faulty?
  - Crash fault: Replica does not respond; no progress (bad)
  - Byzantine fault: Replica might tell lies, corrupt order (worse)
- Solution: Consensus protocol
  - Paxos (for crash faults), PBFT (for Byzantine faults)
  - Works as long as no more than a certain fraction of the replicas are faulty (PBFT: one third)

How do consensus protocols work?

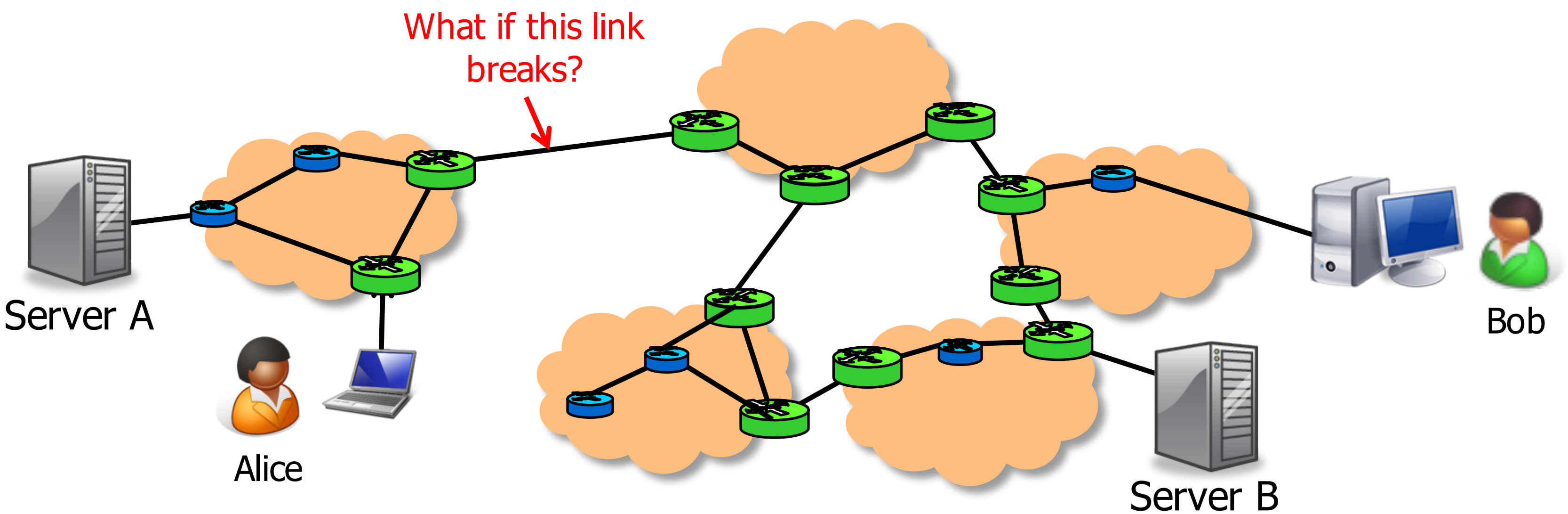


- Idea: Correct replicas 'outvote' faulty ones
  - Clients send requests to each of the replicas
  - Replicas coordinate and each return a result
  - Client chooses one of the results, e.g., the one that is returned by the largest number of replicas
  - If a small fraction of the replicas returns the wrong result, or no result at all, they are 'outvoted' by the other replicas

Plan for today

- Distributed programming and its challenges
  - Faults, failures, and what we can do about them
  - Network partitions, CAP theorem, relaxed consistency NEXT
- Cloud basics
  - Anatomy of Cloud applications
  - Scaling: stateless, caching, and sharding
- Cloud storage
  - Overview
  - KVS and current systems
- Amazon Dynamo

Network partitions



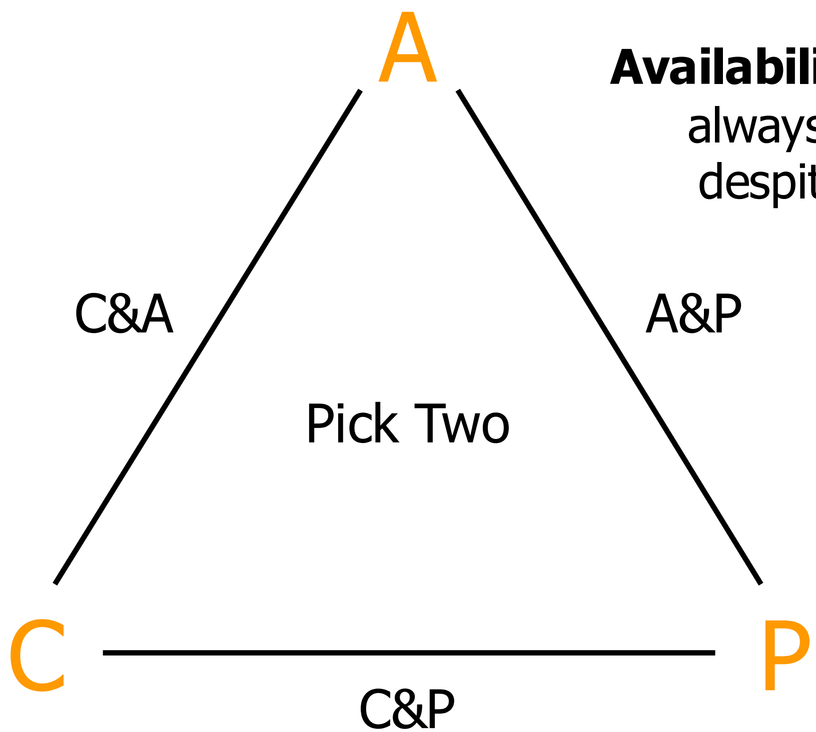
- Network can partition
  - Hardware fault, router misconfigured, undersea cable cut, ...
  - Result: Gobal connectivity is lost
  - What does this mean for the properties of our system?

The CAP theorem

- What we want from a web system:
  - Consistency: All clients see up-to-date copy of the data, even in the presence of concurrent updates
  - Availability: Every request (including updates) received by a non-failing node in the system must result in a response, even when faults occur
  - Partition-tolerance: Consistency and availability hold even when the network partitions

- Can we get all three?
  - CAP theorem: We can get at most two out of the three
    - Which ones should we choose for a given system?
  - Conjecture by Brewer; proven by Gilbert and Lynch

Visual CAP



Availability: Each client can always read and write despite node failures

Consistency: All clients always have the same view of the data at the same time

Partition-tolerance: The system continues to operate despite arbitrary message loss

Common CAP choices

- Example #1: Consistency & Partition tolerance
  - Many replicas + consensus protocol
  - Do not accept new write requests during partitions
  - Certain functions may become unavailable

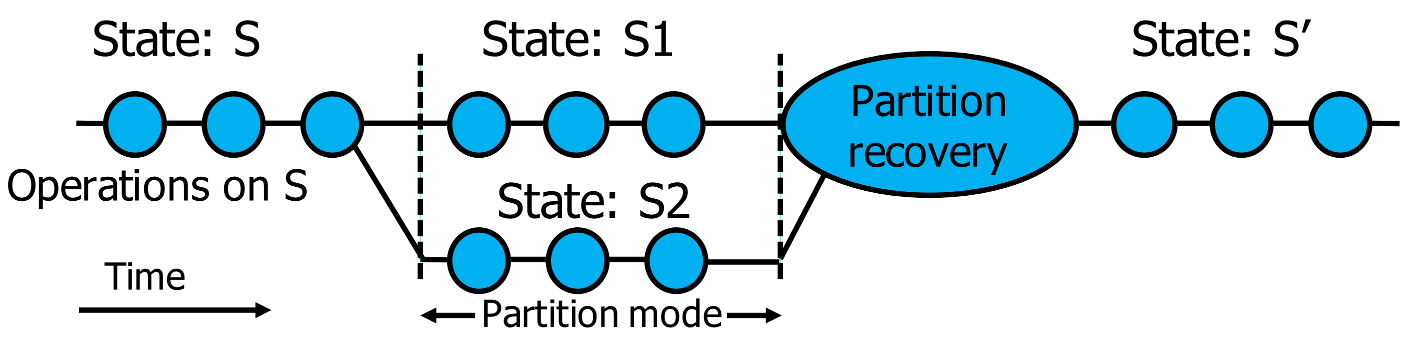
- Example #2: Availability & Partition tolerance
  - Many replicas + relaxed consistency
  - Continue accepting write requests
  - Clients may see inconsistent state during partitions

“2 of 3” view is misleading

- Meaning of C&A over P is unclear
  - If a partition occurs, the choice must be reverted to C or A
  - No reason to forfeit C or A when system is not partitioned
- Choice of C and A can occur many times within the same system at fine granularity
- Three properties are more of a continuous
  - Availability is 0 to 100
  - Many levels of consistency
  - Disagreement within the system whether a partition exists
- The modern CAP goal should be to maximize application-specific combinations of C and A

Dealing with partitions

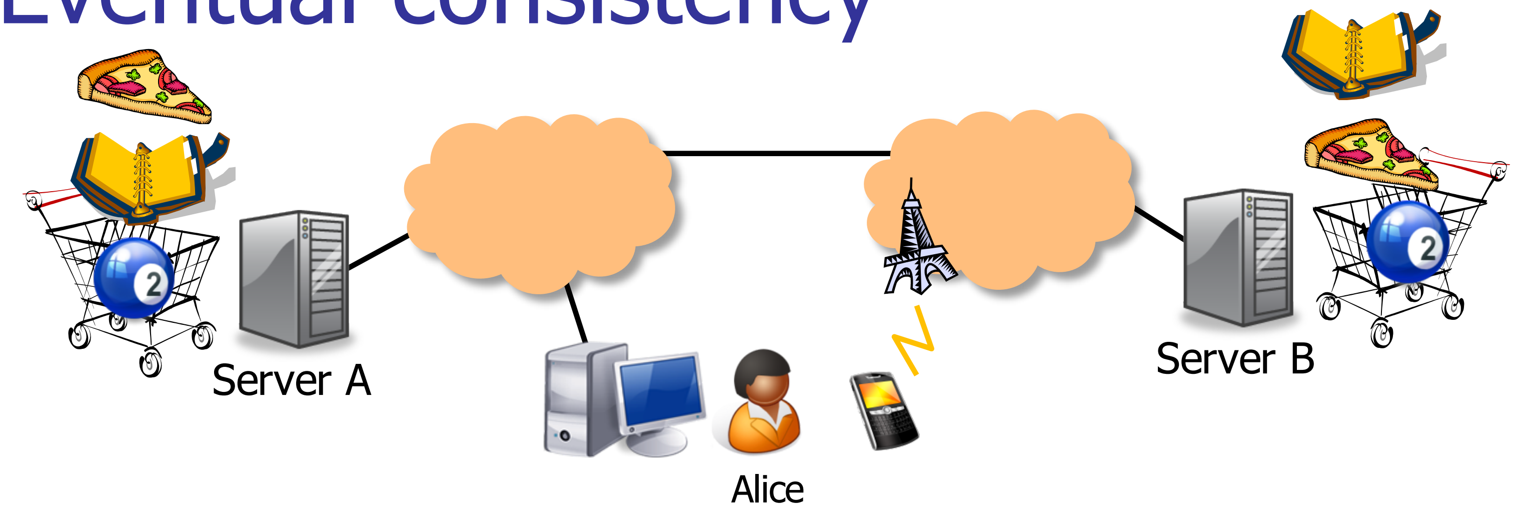
- Detect partition
- Enter an explicit partition mode that can limit some operations
- Initiate partition recovery when communication is restored
  - Restore consistency and compensate for mistakes made while the system was partitioned



Which operations should proceed?

- Depends primarily on the invariants that the system intends to maintain
- If an operation is allowed and turns out to violate an invariant, the system must restore the invariant during recovery
  - Example: 2 objects are added with the same (unique) key; to restore, we check for duplicate keys and merge objects
- If invariant cannot be violated, system must prohibit or modify the operation (e.g. record the intent and execute it after)
  - Example: delay charging the credit card; user does not see system is not available

Eventual consistency



- Idea: Optimistically allow updates
  - Don't coordinate with ALL replicas before returning response
  - But ensure that updates reach all replicas eventually
    - What do we do if conflicting updates were made to different replicas?
  - Good: Decouples replicas. Better performance, availability under partitions
  - (Potentially) bad: Clients can see inconsistent state

Partition recovery

- State on both sides must become consistent
- Compensation for mistakes during partition
- Start from state at the time of the partition and roll forward both sets of operations in some way, maintaining consistency
- The system must also merge conflicts
  - constraint certain operations during partition mode so that conflicts can always be merged automatically
  - detect conflicts and report them to a human
  - use commutative operations as a general framework for automatic state convergence
    - commutative replicated data types (CRDTs)

Compensate for mistakes

- Tracking and limitation of partition-mode operations ensures the knowledge of which invariants could have been violated
  - trivial ways such as “last writer wins”, smarter approaches that merge operations, and human escalation
- For externalized mistakes typically requires some history about externalized outputs
- System could execute orders twice
  - If the system can distinguish two intentional orders from two duplicate orders, it can cancel one of the duplicates
  - If externalized, send an e-mail explaining the order was accidentally executed twice but that the mistake has been fixed and to attach a coupon for a discount

Relaxed consistency: ACID vs. BASE

- Classical database systems: ACID semantics
  - Atomicity
  - Consistency
  - Isolation
  - Durability

- Modern Internet systems: BASE semantics
  - Basically Available
  - Soft-state
  - Eventually consistent

Recap: Consistency and partitions

- Use replication to mask limited # of faults
  - Can achieve strong consistency by having replicas agree on a common request ordering
  - Even non-crash faults can be handled, as long as there are not too many of them (typical limit: 1/3)

- Partition tolerance, availability, consistency?
  - Can't have all three (CAP theorem)
  - Typically trade-off between C and A
  - If service works with weaker consistency guarantees, such as eventual consistency, can get a compromise (BASE)

Plan for today

- Distributed programming and its challenges ✓
  - Faults, failures, and what we can do about them ✓
  - Network partitions, CAP theorem, relaxed consistency ✓
- Cloud basics
  - Anatomy of Cloud applications ⟵ NEXT
  - Scaling: stateless, caching, and sharding
- Cloud storage
  - Overview
  - KVS and current systems
- Amazon Dynamo

Recap: Cloud benefits

- Elastic, just-in-time infrastructure
- More efficient resource utilization
- Pay for what you use
- Potential to reduce processing time
  - Parallelization
- Leverage multiple data centers
  - High availability, lower response times

- How do applications exploit these benefits?

Today’s Cloud applications

- Web applications
  - Client/server paradigm
  - Request/response messaging pattern
  - Interactive communication
- Processing pipelines
  - Examples: Indexing, data mining, image processing, video transcoding, document processing
- Batch processing systems
  - Example: report generation, fraud detection, analytics, backups, automated testing

Many styles of system

- Near the edge of the application focus is on vast numbers of clients and rapid response
- Inside we find data-intensive services that operate in a pipelined manner, asynchronously
- Deep inside the application we see a world of virtual computer clusters that are scheduled to share resources and on which applications like MapReduce (Hadoop) are very popular

Example: Obama for America AWS

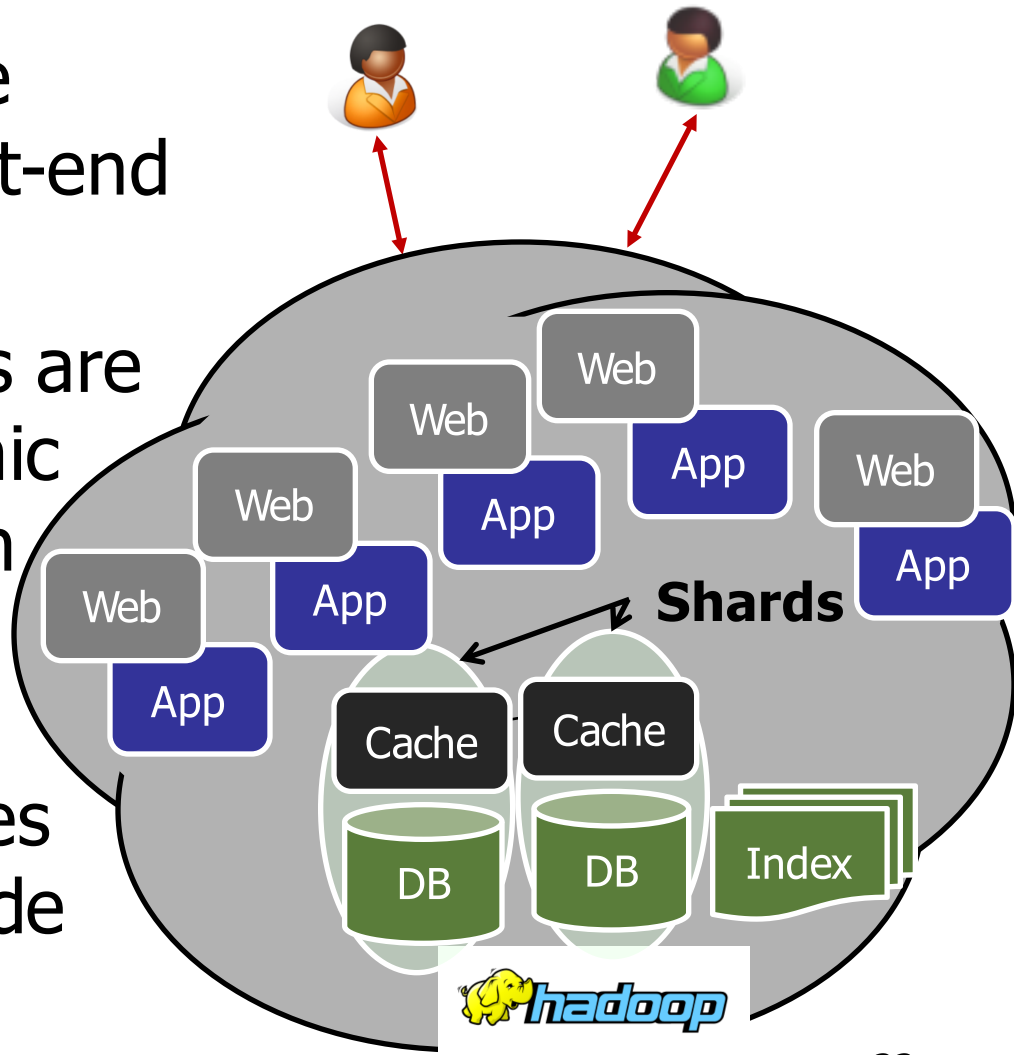
http://awsofa.info/

How are Cloud apps structured?

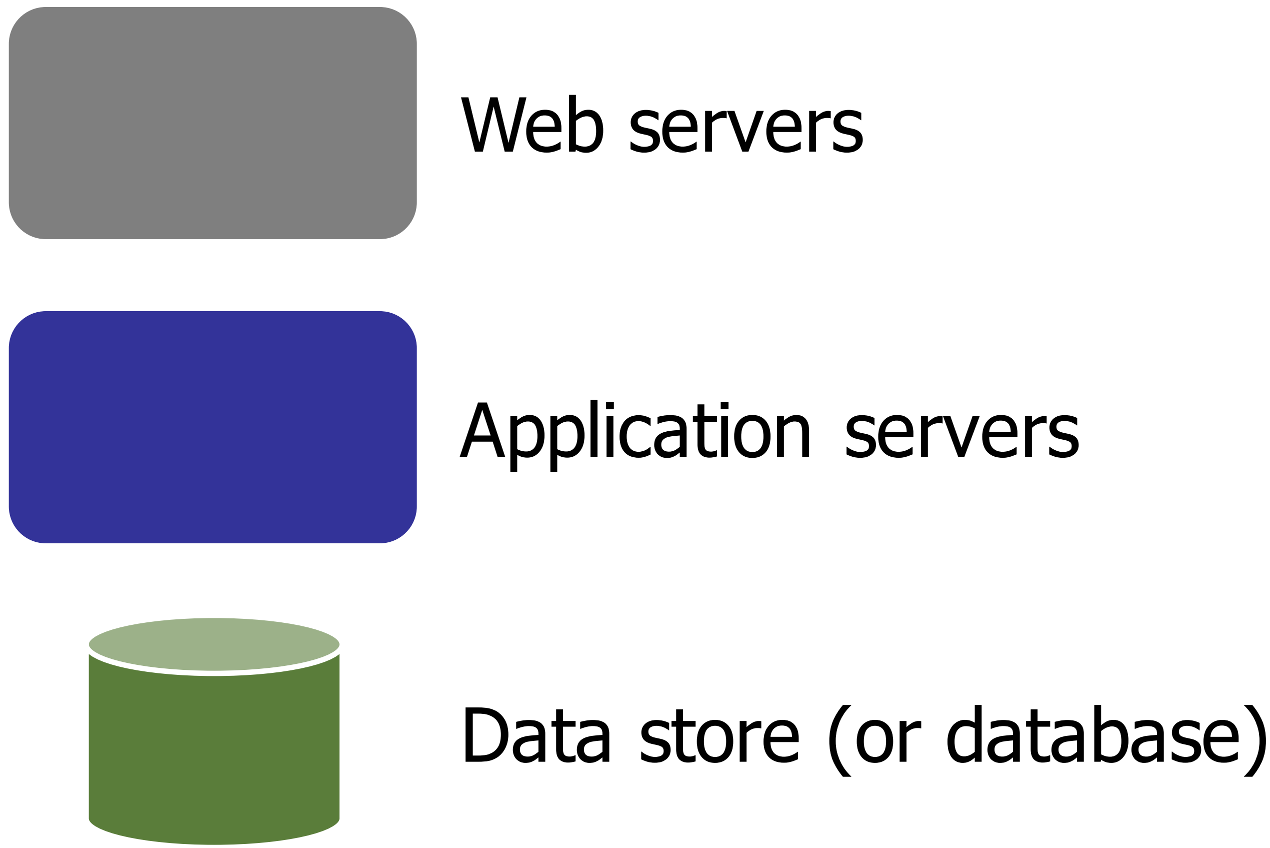
- Clients talk to application using Web browsers or the Web services standards
  - But this only gets us to the outer "skin" of the data center, not the interior
  - Consider Amazon: it can host entire company web sites (like Netflix.com), data (S3), servers (EC2), databases (RDS) and even virtual desktops!

Big picture overview

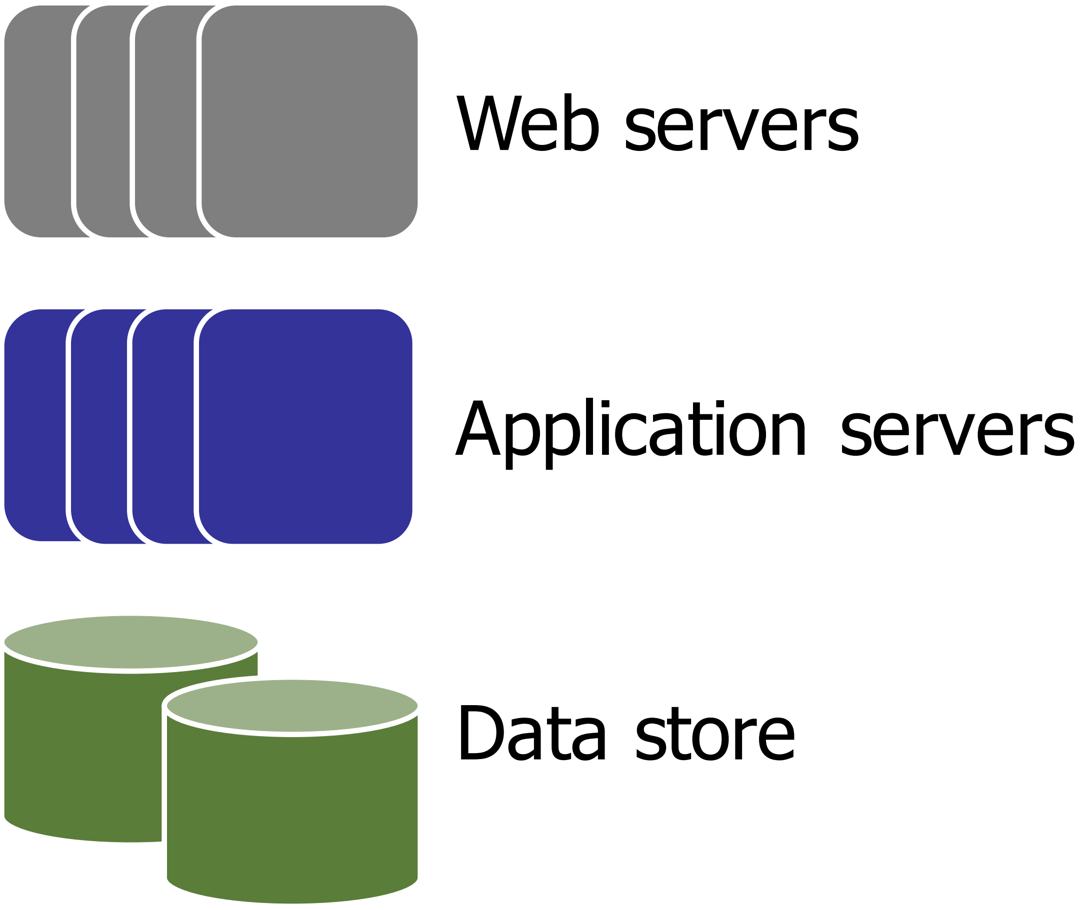
- Client requests are handled in by front-end Web servers
- Application servers are invoked for dynamic content generation and run app logic
  - PHP, Java, Python, …
- Back-end databases manage and provide access to data



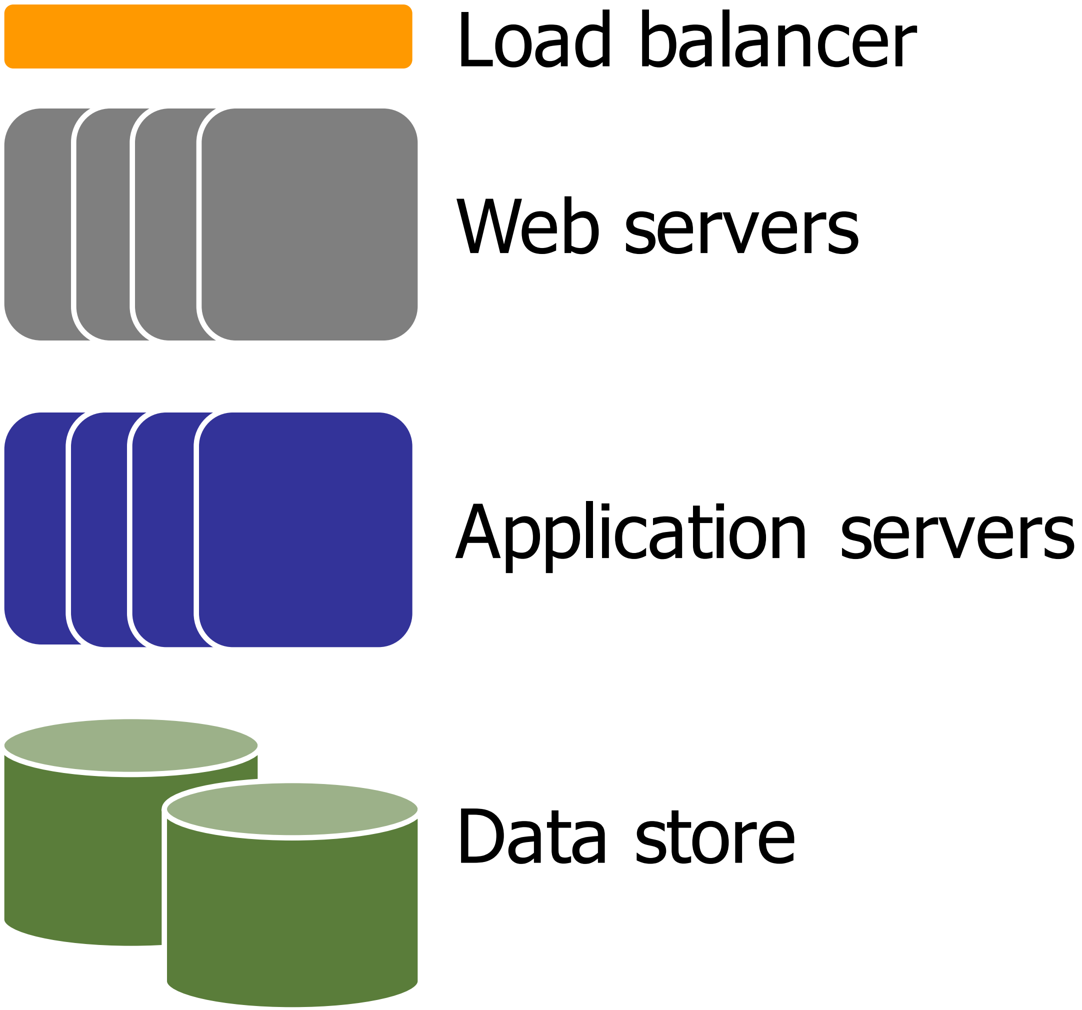
Applications with multiple tiers



Redundancy at each tier



Load balancer

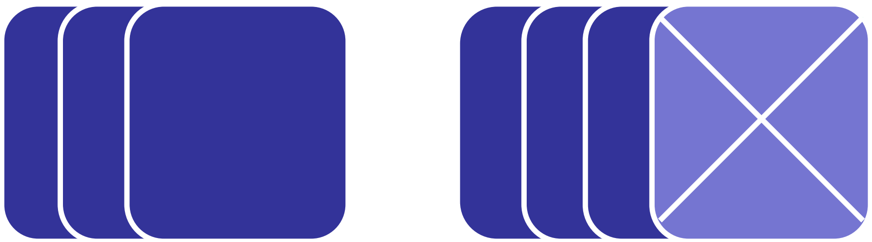


Plan for today

- Distributed programming and its challenges
  - Faults, failures, and what we can do about them
  - Network partitions, CAP theorem, relaxed consistency
- Cloud basics
  - Anatomy of Cloud applications
  - Scaling: stateless, caching, and sharding NEXT
- Cloud storage
  - Overview
  - KVS and current systems
- Amazon Dynamo

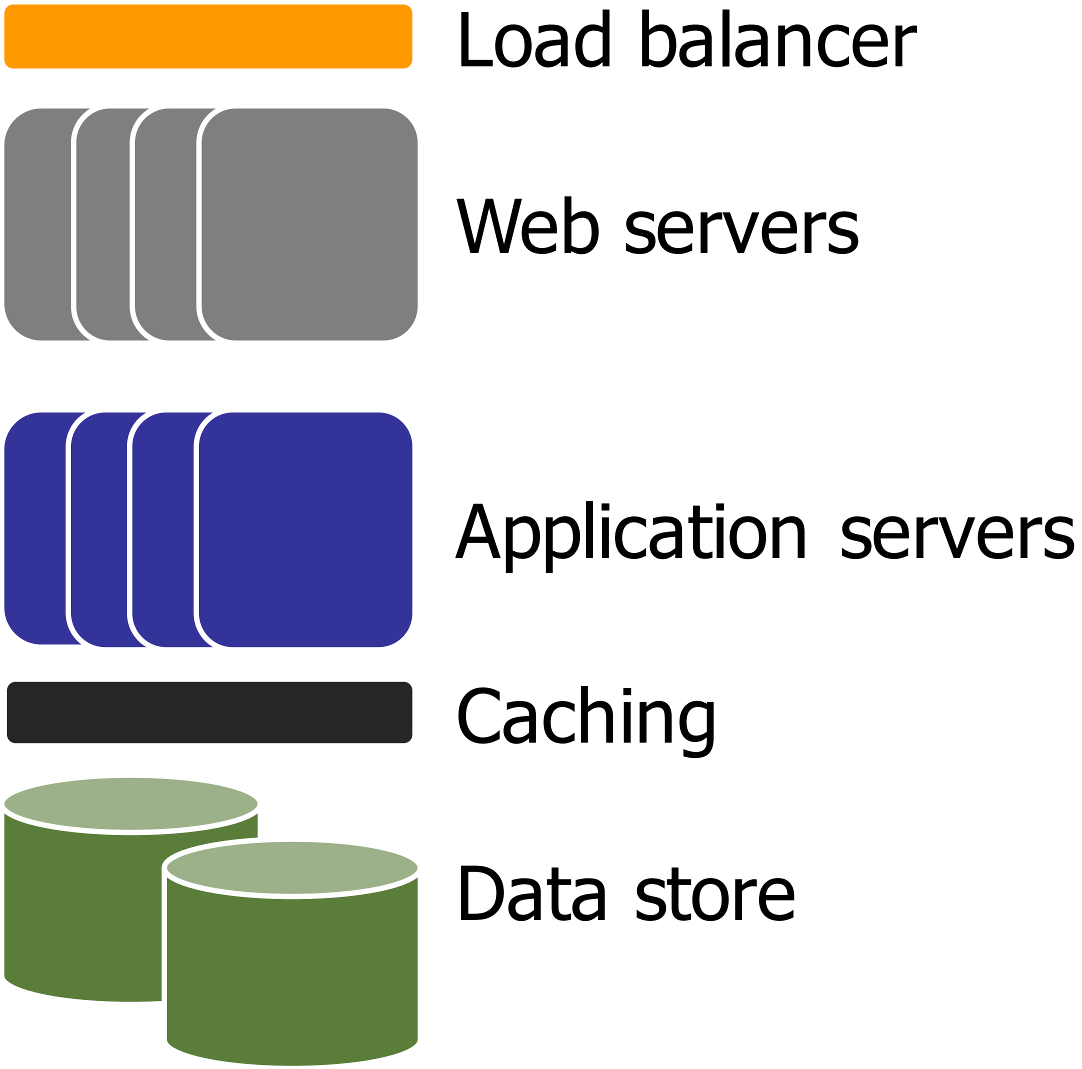
Stateless servers are easiest to scale

- Views a client request as an independent transaction and responds to it
- Advantages:
  - Simpler and easier to scale: does not maintain state
  - More robust: tolerating instance failures does not require overheads restoring state



Stateless servers

Caching

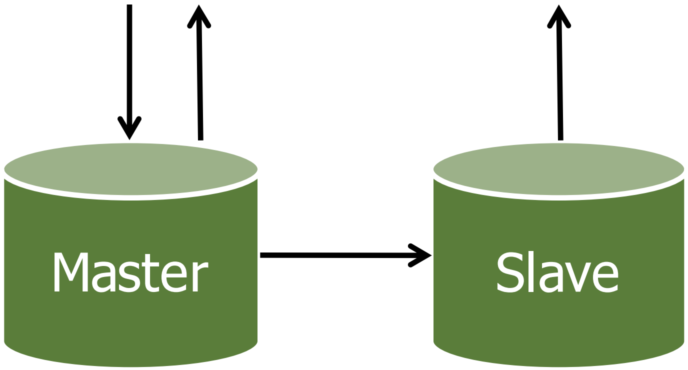

Load balancer
Web servers
Application servers
Caching
Data store

Caching

- Caching is central to responsiveness
  - Basic idea is to always used cached data if at all possible, so the inner services (data stores) are shielded from “online” load
  - Caching is only temporary storage, hence it is stateless
  - We can add multiple cache serves to spread loads
- Must think hard about patterns of data access
  - Some data needs to be heavily replicated to offer very fast access on vast numbers of nodes
  - In principle the level of replication should match level of load and the degree to which the data is needed

Stateful servers require attention

- Scaling a relational database is challenging
- Traditional approach is replication
  - Data is written to a master server and then replicated to one or more slave servers (synchronously or asynchronously)
  - Read operations can be handled by the slaves
  - All writes happen on the master



Data store

Stateful servers require attention

- Scaling a relational database is challenging
- Traditional approach is replication
  - Data is written to a master server and then replicated to one or more slave servers (synchronously or asynchronously)
  - Read operations can be handled by the slaves
  - All writes happen on the master
- Cons:
  - Master becomes the write bottleneck
  - Master is a single point of failure
  - As load increases, cost of replication increases
  - Slaves may fall behind and serve stale data

Sharding

- Data partitioning strategy
- Basic idea: split data between multiple machines and have a way to make sure you always access data from the right place
  - Typically define a sharding key and create a shard mapping (e.g., shard_idx = hash(key) mod N)
  - Other partitioning schemes exist: e.g., allocate whole tables on the same machine

Benefits of sharding

- Increased read and write throughput
- High availability
- Possibility of doing more work in parallel within the application server

- Challenge: picking a good partitioning scheme
  - Otherwise risk of having hotspots in the system due to load imbalance

Sharding used in many ways

- Sharding is not only for partitioning data within a database
- Applies essentially to every application tier
  - Notion of sharding is cross-cutting

- Example: partition data across caching servers
- Two popular in-memory caching systems:
  - memcached: distributed object caching system
  - redis: distributed data structure server (also works as store)

And it isn’t just about updates

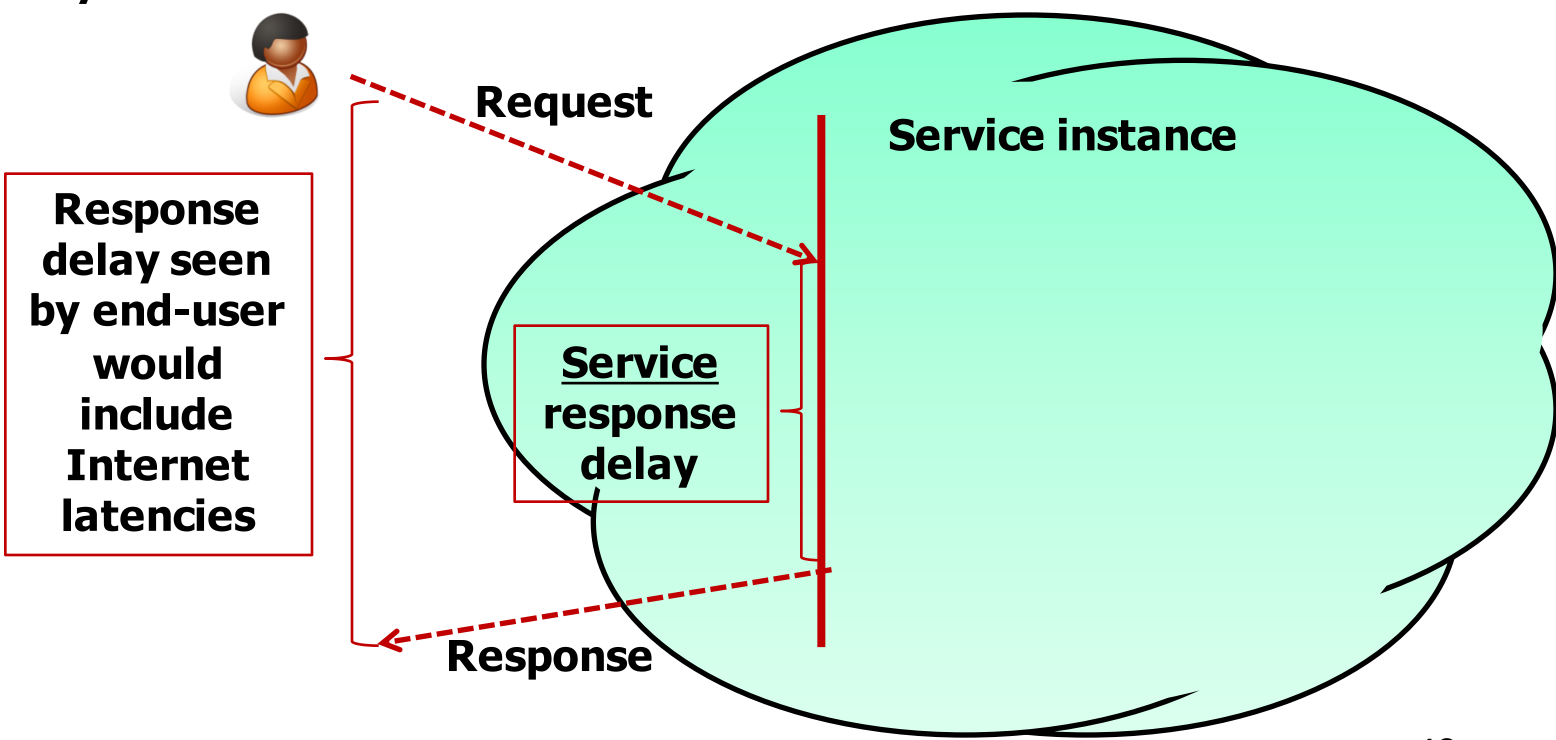
- Should also be thinking about patterns that arise when doing reads (“queries”)
  - Some can just be performed by a single representative of a service
  - But others might need the parallelism of having several (or even a huge number) of machines do parts of the work concurrently
- The term sharding is used for data, but here we might talk about “parallel computation on a shard”

First-tier parallelism

- Parallelism is vital for fast interactive services
- Key question:
  - Request has reached some service instance X
  - Will it be faster…
    - … For X to just compute the response
    - … Or for X to subdivide the work by asking subservices to do parts of the job?
- Glimpse of an answer
  - When you make a search on Bing, the query is processed in parallel by even 1000s of servers that run in real-time on your request!
- Parallel actions must focus on the critical path

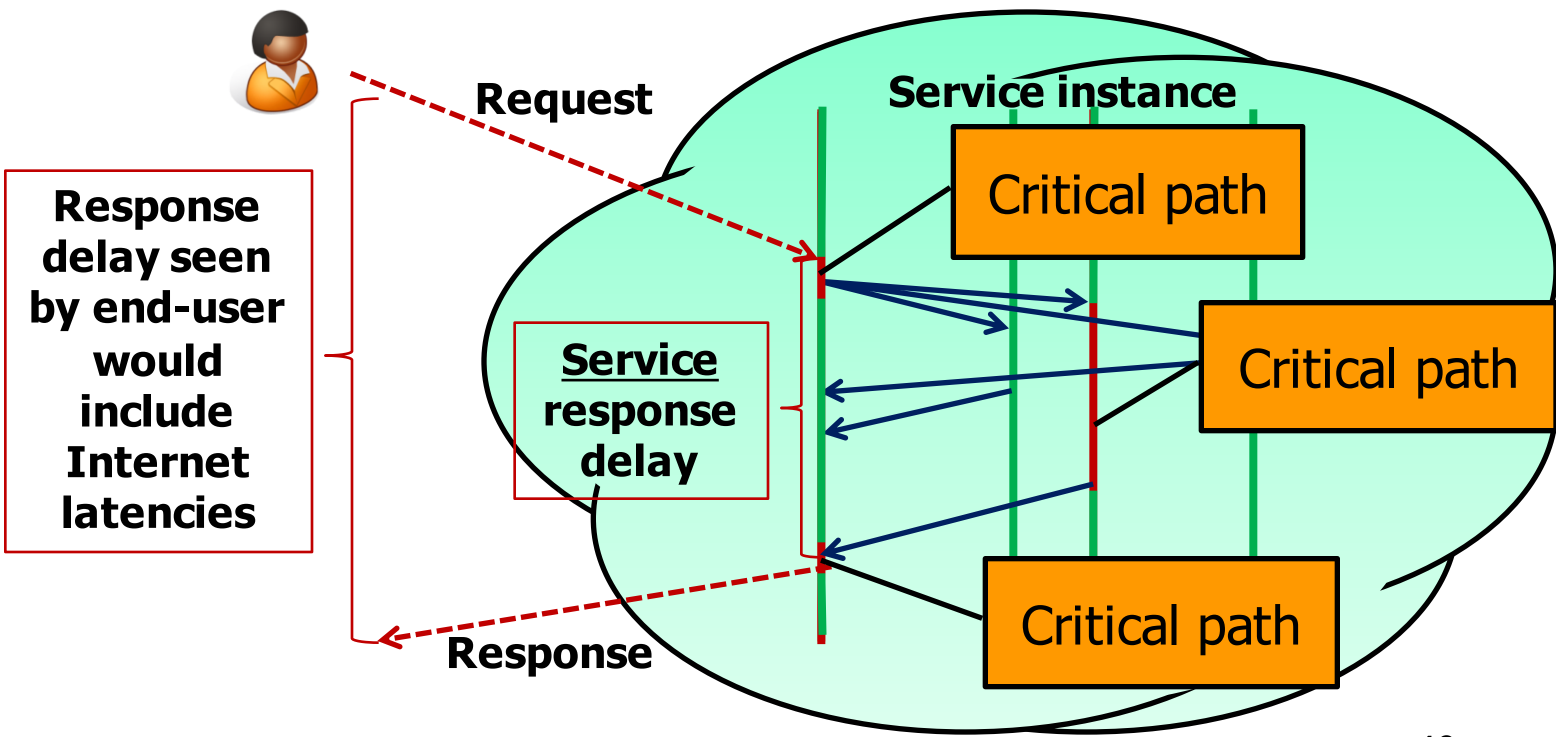
What does “critical path” mean?

- Focus on delay until a client receives a reply
- Critical path are actions that contribute to this delay

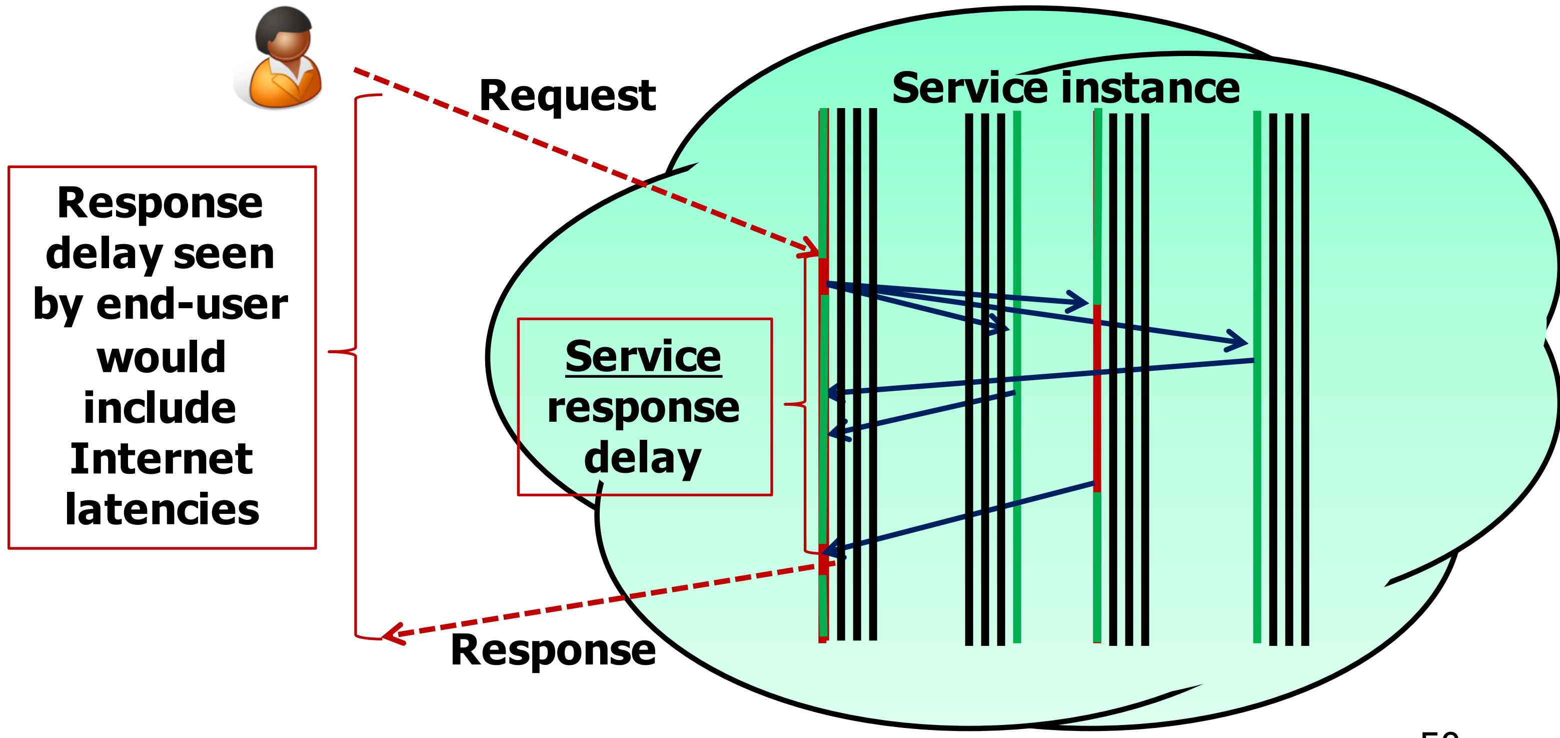


Parallel speedup

- In this example of a parallel read-only request, the critical path centers on the middle "subservice"



With replicas we just load balance



What if a request triggers updates?

- If updates are done "asynchronously" we might not experience much delay on the critical path
  - Cloud systems often work this way
  - Avoids waiting for slow services to process the updates but may force the tier-one service to "guess" the outcome
  - For example, store in the master database and replicate to the slave in the background
- Many cloud systems use these sorts of "tricks" to speed up response time

What if we send updates without waiting?

- Several issues now arise
  - Are all the replicas applying updates in the same order?
    - Might not matter unless the same data item is being changed
    - But then clearly we do need some “agreement” on order
  - What if the leader replies to the end user but then crashes and it turns out that the updates were lost in the network?
    - Data center networks can be surprisingly lossy at times
    - Also, bursts of updates can queue up
- Such issues result in *inconsistency*

Is inconsistency a bad thing?

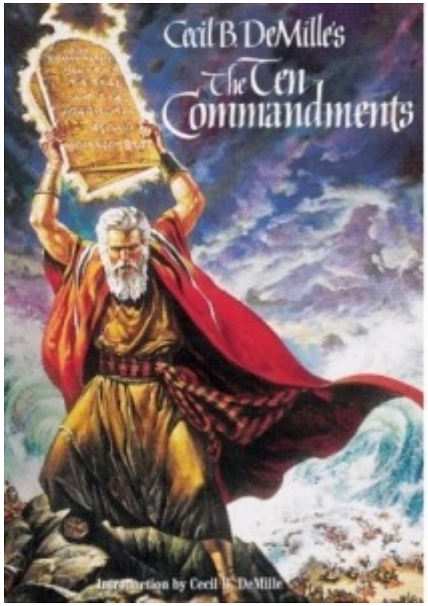
- How much consistency is really needed in the first tier of the cloud?
  - Think about YouTube videos. Would consistency be an issue here?
  - What about the Amazon "number of units available" counters. Will people notice if those are a bit off?
- Puzzle: can you come up with a general policy for knowing how much consistency a given thing needs?

eBay’s Five Commandments

- As described by Randy Shoup at LADIS 2008

Thou shalt…

1. **Partition Everything**
2. **Use Asynchrony Everywhere**
3. **Automate Everything**
4. **Remember: Everything Fails**
5. **Embrace Inconsistency**



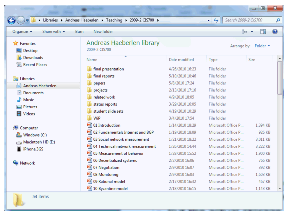
Recap

- Cloud applications are multi-tiered systems
- Caching can enable significant speedups for read-heavy workloads
- Sharding provides opportunities for parallelization and improve read/write throughputs
- Asynchronous operations decouple systems and enable quicker responses at the expense strong consistency

Plan for today

- Distributed programming and its challenges
  - Faults, failures, and what we can do about them
  - Network partitions, CAP theorem, relaxed consistency
- Cloud basics
  - Anatomy of Cloud applications
  - Scaling: stateless, caching, and sharding
- Cloud storage
  - Overview NEXT
  - KVS and current systems
- Amazon Dynamo

Complex service, simple storage



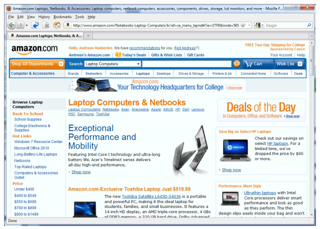

Variable-size files
- read, write, append
- move, rename
- lock, unlock
- ...

Operating system

Fixed-size blocks
- read
- write

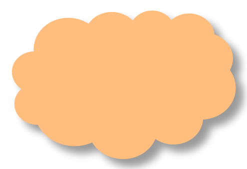
- PC users see a rich, powerful interface
  - Hierarchical namespace (directories); can move, rename, append to, truncate, (de)compress, view, delete files, ...
- But the actual storage device is very simple
  - HDD only knows how to read and write fixed-size data blocks
- Translation done by the operating system

Analogy to cloud storage



Shopping carts
Friend lists
User accounts
Profiles
...

Web service



Key/value store
- read, write
- delete

- Many cloud services have a similar structure
  - Users see a rich interface (shopping carts, product categories, searchable index, recommendations, ...)
- But the actual storage service is very simple
  - Read/write 'blocks', similar to a giant hard disk
- Translation done by the web service

What's Wrong with Relational DBs?

- Most applications interact through a database
- Recall RDBMS:
  - Manage data access, enforce data integrity, control concurrency, support recovery after a failure
- Many applications push traditional RDBMS solutions to the limit by demanding:
  - High scalability
  - Very large amounts of data
  - Minimal latency
  - High availability
- Solution is far from ideal

Ideal data stores on the Cloud

- Many situations need hosting of large data sets
  - Examples: Amazon catalog, eBay listings, Facebook pages, …
- Ideal: Abstraction of a 'big disk in the clouds', which would have:
  - Perfect durability – nothing would ever disappear in a crash
  - 100% availability – we could always get to the service
  - Zero latency from anywhere on earth – no delays!
  - Minimal bandwidth utilization – we only send across the network what we absolutely need
  - Isolation under concurrent updates – make sure data stays consistent

The inconveniences of the real world

- Why isn't this feasible?

- The "cloud" exists over a physical network
  - Communication takes time, esp. across the globe
  - Bandwidth is limited, both on the backbone and endpoint
- The "cloud" has imperfect hardware
  - Hard disks crash
  - Servers crash
  - Software has bugs

- Can you map these to the previous desiderata?

Finding the right tradeoff

- In practice, we can't have everything
  - ... but most applications don't really need 'everything'!
- Some observations:
  1. Read-only (or read-mostly) data is easiest to support
     - Replicate it everywhere! No concurrency issues!
     - But only some kinds of data fit this pattern – examples?
  2. Granularity matters: "Few large-object" tasks generally tolerate longer latencies than "many small-object" tasks
     - Fewer requests, often more processing at the client
     - But it's much more expensive to replicate or to update!
  3. Maybe it makes sense to develop separate solutions for large read-mostly objects vs. small read-write objects!
     - Different requirements → different technical solutions

Many situations need hosting of large data sets

Examples: Amazon catalog, eBay listings, Facebook pages, …

- General trend:

From performance at any cost to …
reliability at the lowest possible cost

Plan for today

- Distributed programming and its challenges
  - Faults, failures, and what we can do about them
  - Network partitions, CAP theorem, relaxed consistency
- Cloud basics
  - Anatomy of Cloud applications
  - Scaling: stateless, caching, and sharding
- Cloud storage
  - Overview
  - KVS and current systems NEXT
- Amazon Dynamo

Key-value stores

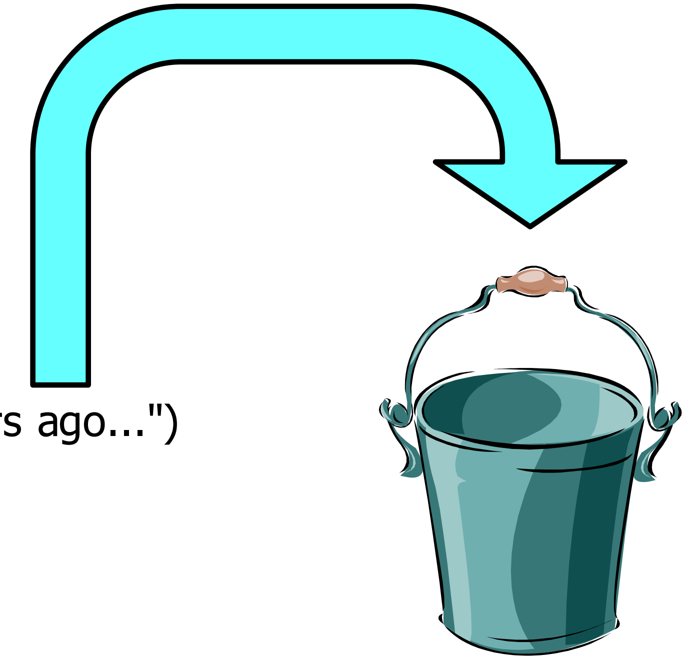
Keys ↓ Values ↓

(bob, bschmitt@foo.com)

(gettysburg, "Four score and seven years ago...")

(29ck2dxa1, 0128ckso1$9#*!!8349e)

(windows, )



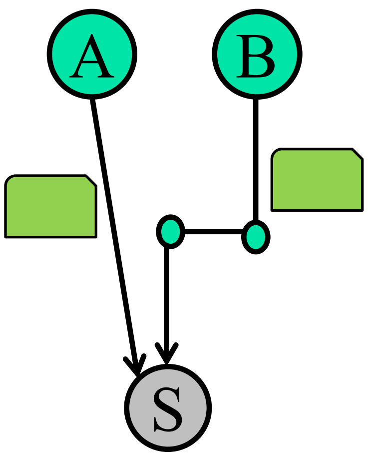
- The key-value store (KVS) is a simple abstraction for managing persistent state
  - Data is organized as (key,value) pairs
  - Only three basic operations:
    - PUT(key, value)
    - GET(key) → value
    - Delete(key)

Examples of KVS

- Where have you seen this concept before?
- Conventional examples outside the cloud:
  - In-memory associative arrays and hash tables – limited to a single application, only persistent until program ends
  - On-disk indices (like BerkeleyDB)
  - "Inverted indices" behind search engines
  - Database management systems – multiple KVSs++
  - Distributed hashtables
    - Decentralized distributed systems inspired by P2P (see LSINF2345)
    - Examples: Chord/Pastry

Supporting an Internet service with a KVS

- We'll do this through a central server, e.g., a Web or application server



- Two main issues:
  1. There may be multiple concurrent requests from different clients
     - These might be GETs, PUTs, DELETEs, etc.
  2. These requests may come from different parts of the network, with message propagation delays
     - It takes a while for a request to make it to the server!
     - We'll have to handle requests in the order received (why?)

Managing concurrency in a KVS

- What happens if we do multiple GET operations in parallel?
  - ... over different keys?
  - ... over the same key?

- What if we do multiple PUT operations in parallel? or a GET and a PUT?

- What is the unit of protection (concurrency control) that is necessary here?

Concurrency control

- Most systems use locks on individual items
  - Each requestor asks for the lock
  - A lock manager processes these requests (typically in FIFO order) as follows:
    - Lock manager grants the lock to a requestor
    - Requestor makes modifications
    - Then releases the lock when it's done

Limitations of per-key concurrency control

- Suppose I want to transfer credits from my WoW account to my friend's?
  - … while someone else is doing a GET on my (and her) credit amounts to see if they want to trade?
- This is where one needs a database management system (DBMS) or transaction processing manager (app server)
  - Allows for "locking" at a higher level, across keys and possibly even systems (see LINGI2172 for more details)
- Could you implement higher-level locks within the KVS? If so, how?

No longer one-size-fits-all solution

Specialized data stores

- Example: Amazon's solutions

- Dynamo [SOSP'07]
  - Many services only store and retrieve data by primary key
    - Examples: user preferences, shopping cart, best seller lists
  - Don't require querying and management RDBMS functionality
- Simple Storage Service (S3)
  - Need to store large objects that change infrequently
    - Examples: virtual machines, pictures

Specialized data stores

- Example: Google's solutions

- The Google File System [SOSP'03]
  - Distributed file system for large data-intensive applications
  - No POSIX API; focus on multi-GB files divided in fixed-size chunks (64 MB); mostly mutated by appending new data
  - Single master node maintains all file metadata
- Bigtable [OSDI'06]
  - Distributed storage system for structured data
  - Data model is a sparse multi-dimensional sorted map indexed by row and column keys and a timestamp
  - Each value in the map is opaque to the storage system

Specialized data stores

- Example: Facebook's solutions
- Cassandra [Ladis'09]
  - A distributed storage system for large sets of structured data
  - Optimized for very high write throughput; no master nodes
- Haystack [OSDI'10]
  - Object store system optimized for photos
  - In 2010, over 260 billion images; 20 PB of data; 60 TB/week
  - Data written once, read often, never modified, rarely deleted
- TAO [ATC'13]
  - A read-optimized graph data store to serve the social graph
  - Sustains 1 billion reads/s on a changing data set of many PBs
  - Explicitly favors availability over consistency

Specialized data stores

- Example: LinkedIn’s solutions

- Kafka [NetDB’11]
  - A high-throughput distributed messaging system
  - Pub/sub architecture designed for aggregating log data
  - Messages are persisted on disk for durability and replicated for fault tolerance; guarantees at-least-once delivery
- Voldemort
  - A distributed key-value store supporting only get/put/delete
  - Inspired by Amazon’s Dynamo: tunable consistency, highly available

Plan for today

- Distributed programming and its challenges
  - Faults, failures, and what we can do about them
  - Network partitions, CAP theorem, relaxed consistency
- Cloud basics
  - Anatomy of Cloud applications
  - Scaling: stateless, caching, and sharding
- Cloud storage
  - Overview
  - KVS and current systems
- Amazon Dynamo NEXT

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 fulfillment, 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

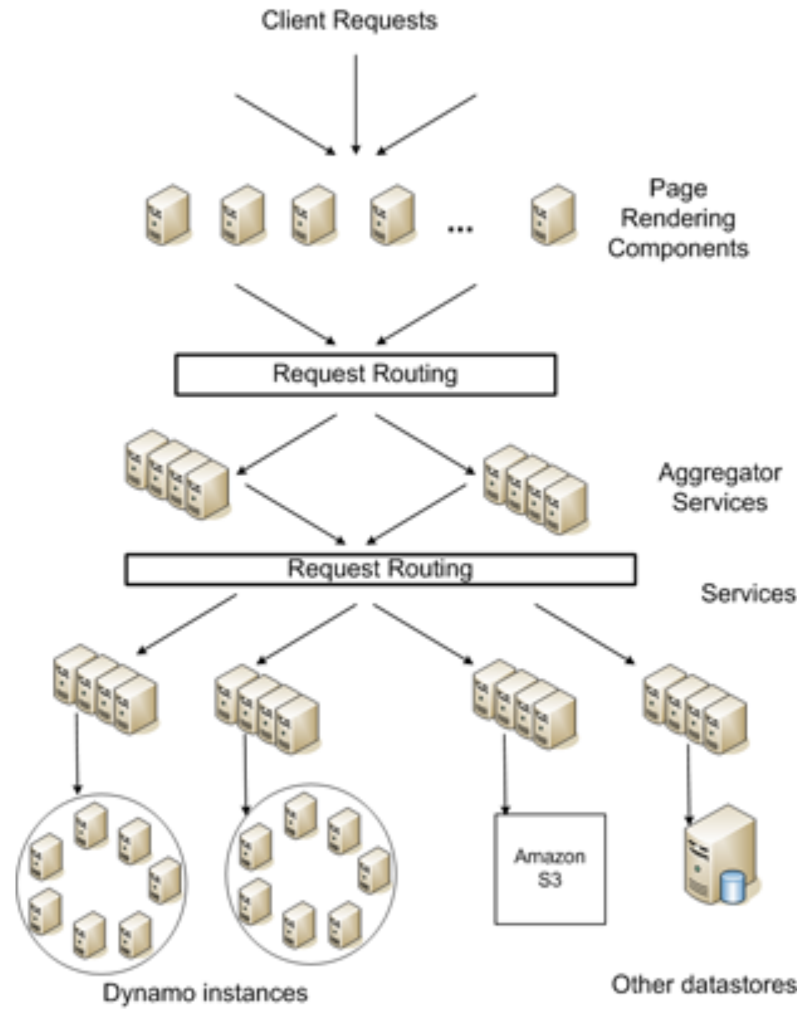
- High performance (low latency)
- Highly scalable (hundreds of server nodes)
- "Always-on" available (especially for writes)
- Partition/Fault-tolerant
- Eventually consistent
- Dynamo uses several techniques to achieve these features
  - Which also comprise a nice subset of a general distributed system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

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

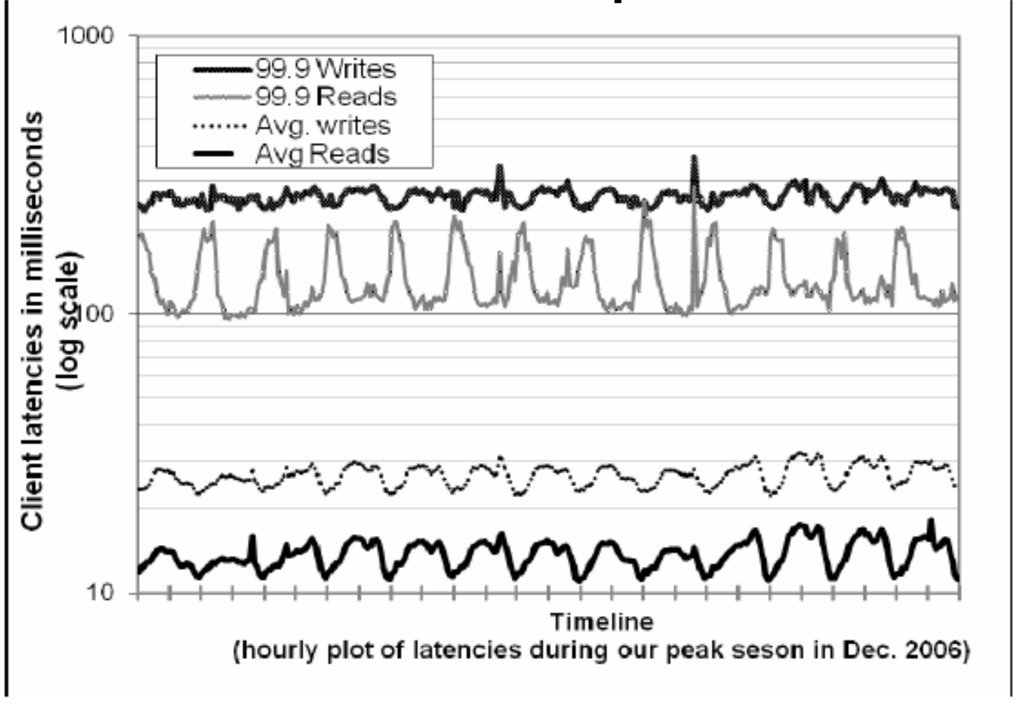
Amazon SOA platform

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



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



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

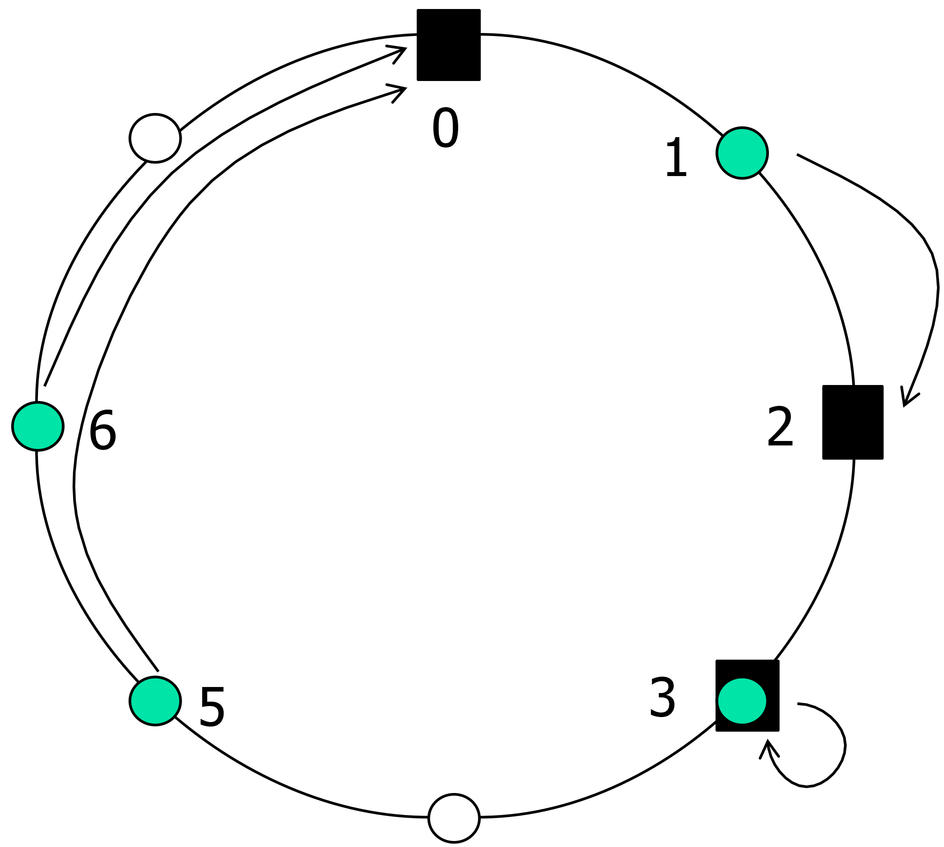
- Scalable and robust components for
  - Load balancing, membership/fault detection, failure recovery, replica synchronization, overload handling, state transfer, concurrency, job scheduling, request marshalling, request routing, system monitoring and alarming, configuration management
- We focus on 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 of *keys* give key m-bit *identifiers*
  - (MD5, can use SHA-1, …)
- 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

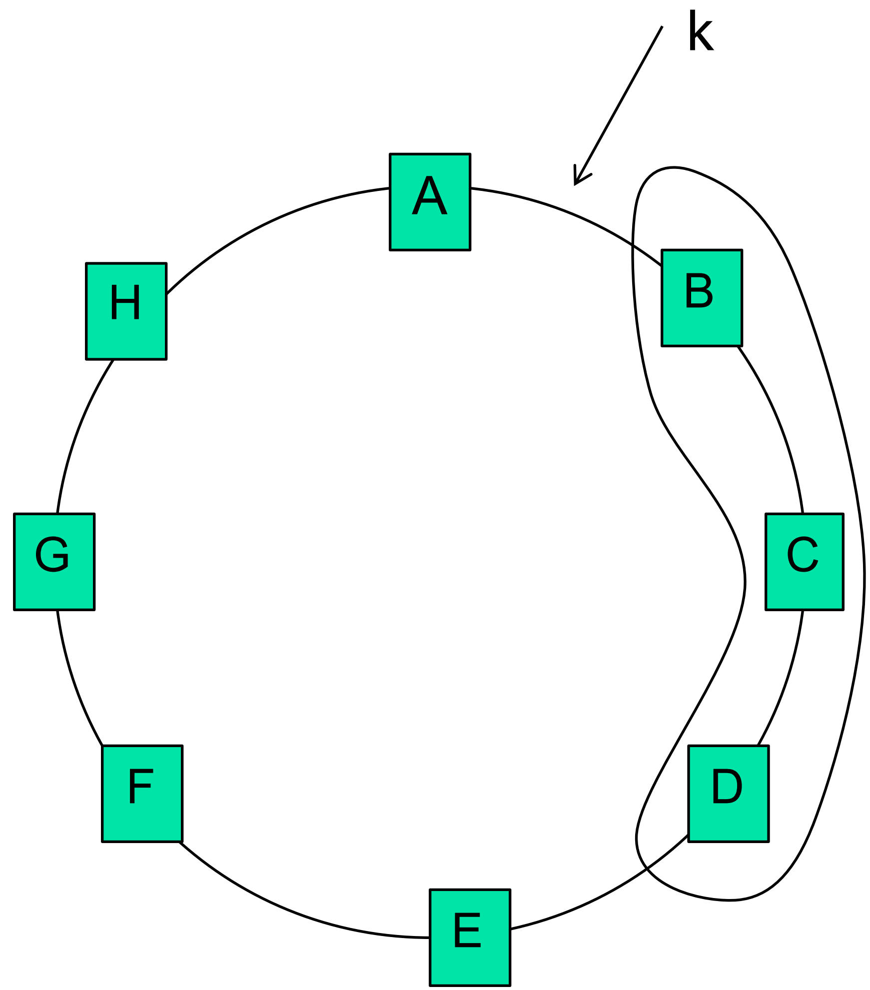
- Maintenance
  - When node *n* joins
    - certain keys previously assigned to *n*'s successor now become assigned to *n*.
  - When node *n* leaves
    - all of *n*'s assigned keys are reassigned to *n*'s successor.

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+1[st] 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*
  - 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

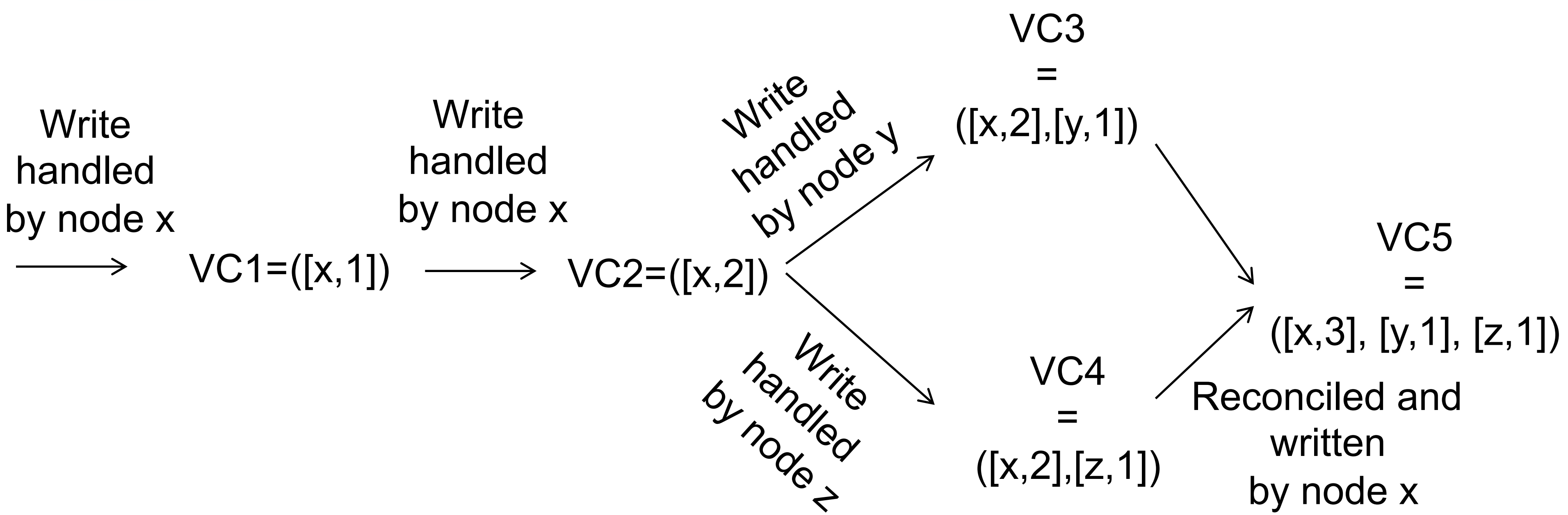
Data versioning

- 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



NB: one VC per key

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
  - Any node can coordinate a read request
  - For a write request, the node routes the request a node in the key's 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

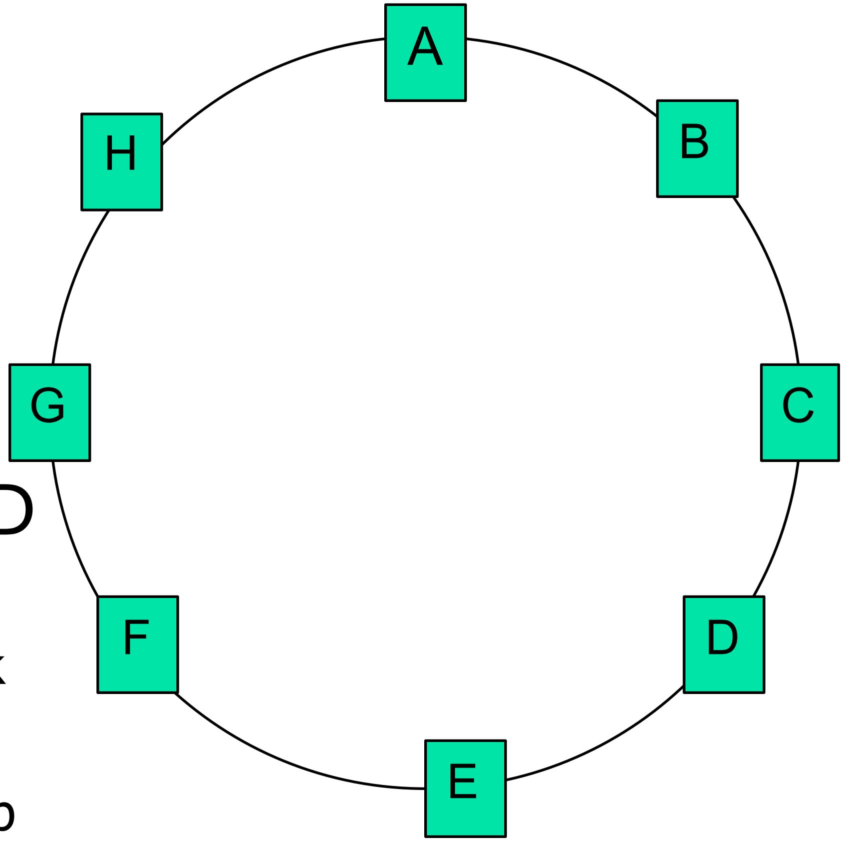
- R, W smaller than N
  - To decrease latency
  - Slowest replica dictates the latency
- W=1
  - Always-available for writes
  - Yields R=N (reads pay the penalty)
- Most often in Dynamo (W,R,N)=(2,2,3)

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

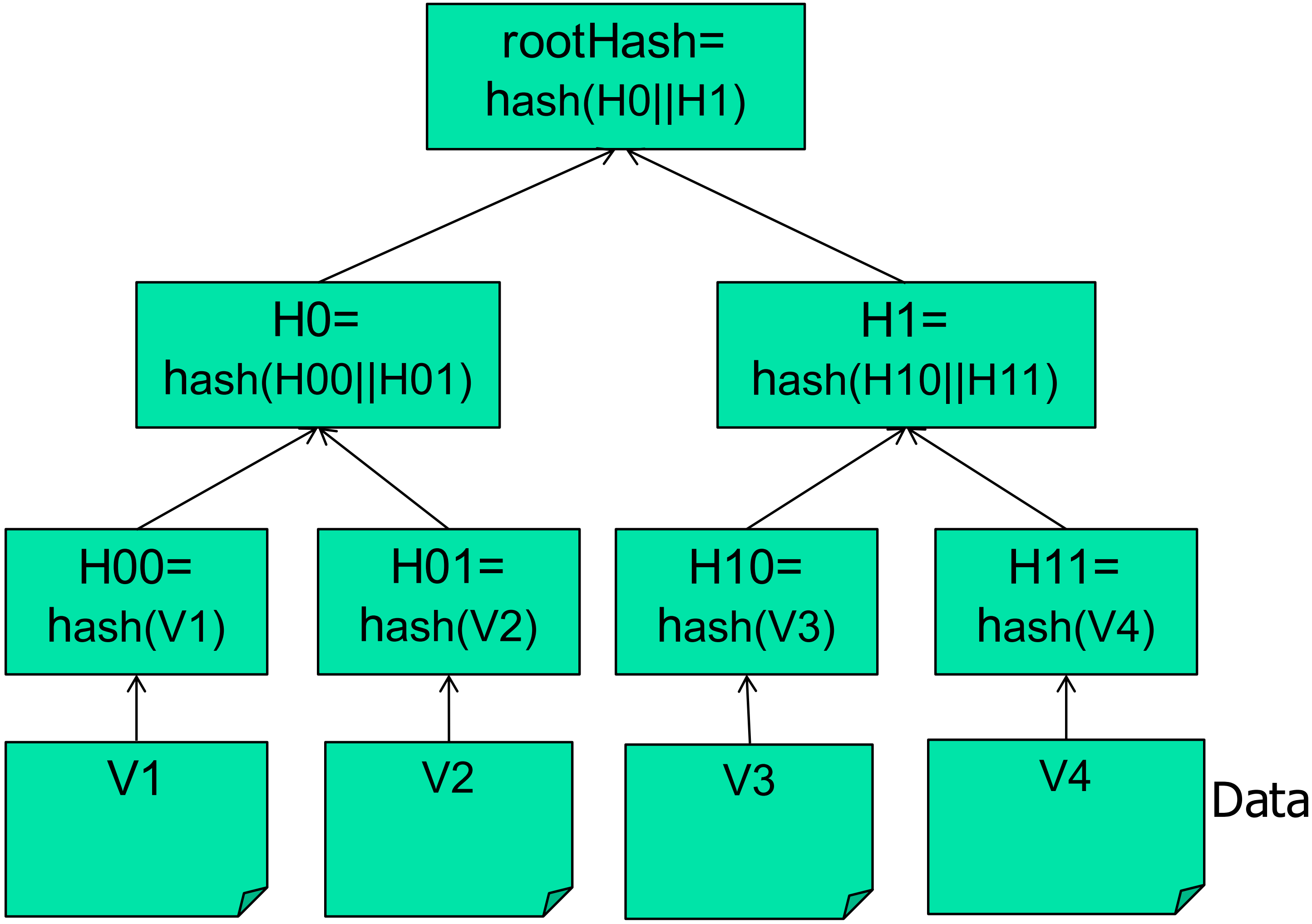
- If a replica in the preference list is down then another replica is created on a new node
- Assume again N=3
- A replica A is down
- Coordinator will involve D
  - With a hint that this D substitutes A until A comes back again
  - When D gets info A is back up it hands back the data to A



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 anyway does not matter 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

Stay tuned

Next time you will learn about:

A programming model for the Cloud