CS5412: ANATOMY OF A CLOUD

Lecture VII

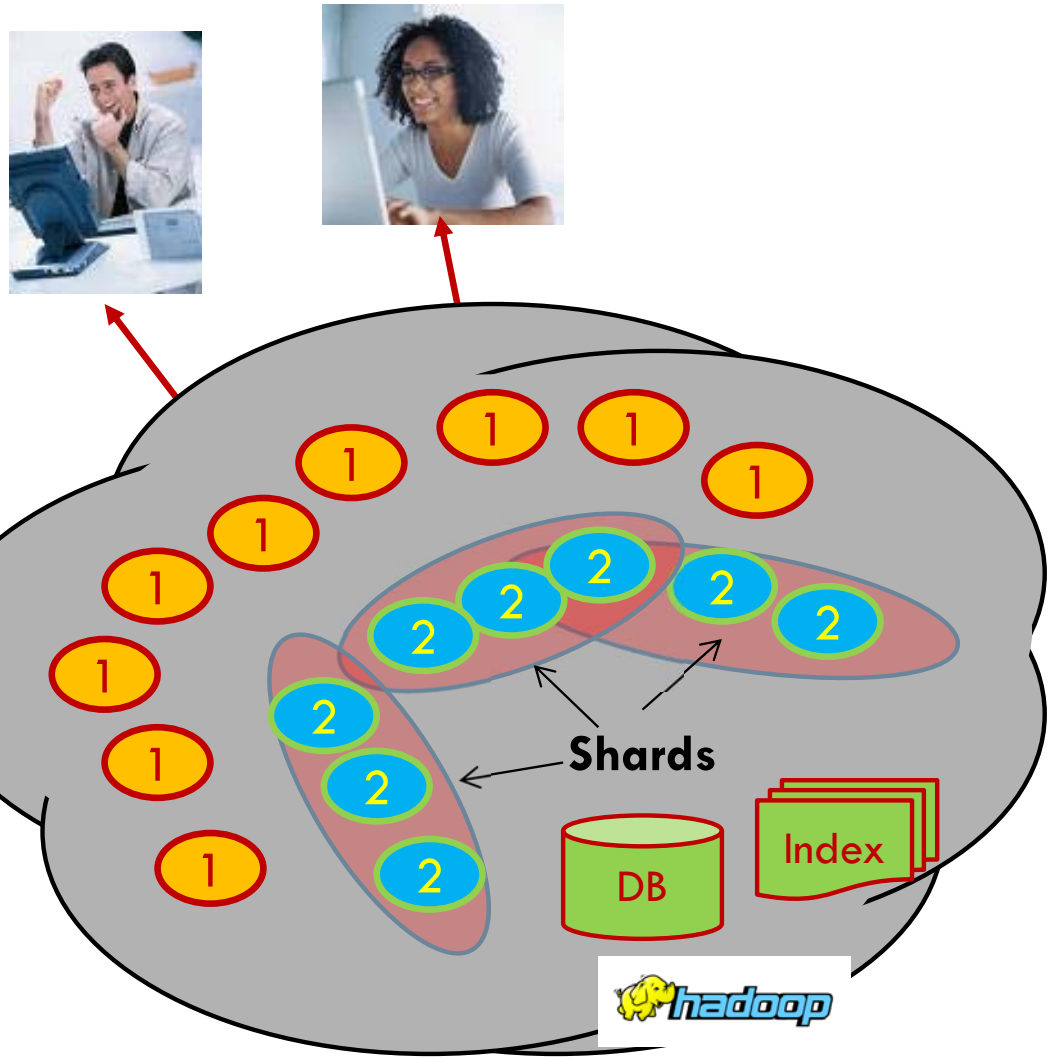
Ken Birman

How are cloud structured?

- Clients talk to clouds using web browsers (increasingly more) or the web services standards (increasingly less)
  - But this only gets as far as the outer "skin" of the cloud data center, not the interior
  - Consider Amazon: it can host entire company web sites (like Target.com or Netflix.com), data (AC3), servers (EC2) and even user-provided virtual machines!

Big picture overview

- Client requests are handled in the "first tier" by
  - E.g. PHP or ASP pages
  - And associated logic
- These lightweight services are fast and very nimble
- Much use of caching: the second tier



Many styles of system

- Near the edge of the cloud, focus is on
  - Vast numbers of clients
  - Rapid response
- Inside, we find high-volume services that operate in a pipelined manner, asynchronously
- Deep inside the cloud, we see a world of virtual computer clusters scheduled to share computing resources, on which massively-parallel applications like MapReduce (Apache Hadoop) are very popular

In the outer tiers replication is key

- We need to replicate
  - Processing: each client has what seems to be a private, dedicated server (for a little while)
  - Data: as much as possible, that server has copies of the data it needs to respond to client requests without any delay at all
  - Control information: the entire structure is managed in an agreed-upon way by a decentralized cloud management infrastructure

What about the "shards"?

- The caching components running in tier two are central to the responsiveness of tier-one services
  - Basic idea: to always use cached data if at all possible, so the inner services (here, a DB and a search index stored in a set of files) are shielded from "online" load
  - We need to replicate data within our cache to balance load and provide fault tolerance
  - But not everything needs to be fully replicated. Hence we often use shards with just a few replicas

Sharding used in many ways

- Tier two could be any of a number of caching services
  - Memcached: a sharable **in-memory** key-value store
  - Distributed Hash Tables that use key-value APIs
  - Dynamo: A service created by Amazon as a scalable way to represent the shopping cart and similar data
  - BigTable: A very elaborate key-value store created by Google and used not just in tier-two but throughout their "GooglePlex" for sharing information
- The notion of sharding is cross-cutting
  - Most of these systems replicate data to some degree

Do we *always* need to shard data?

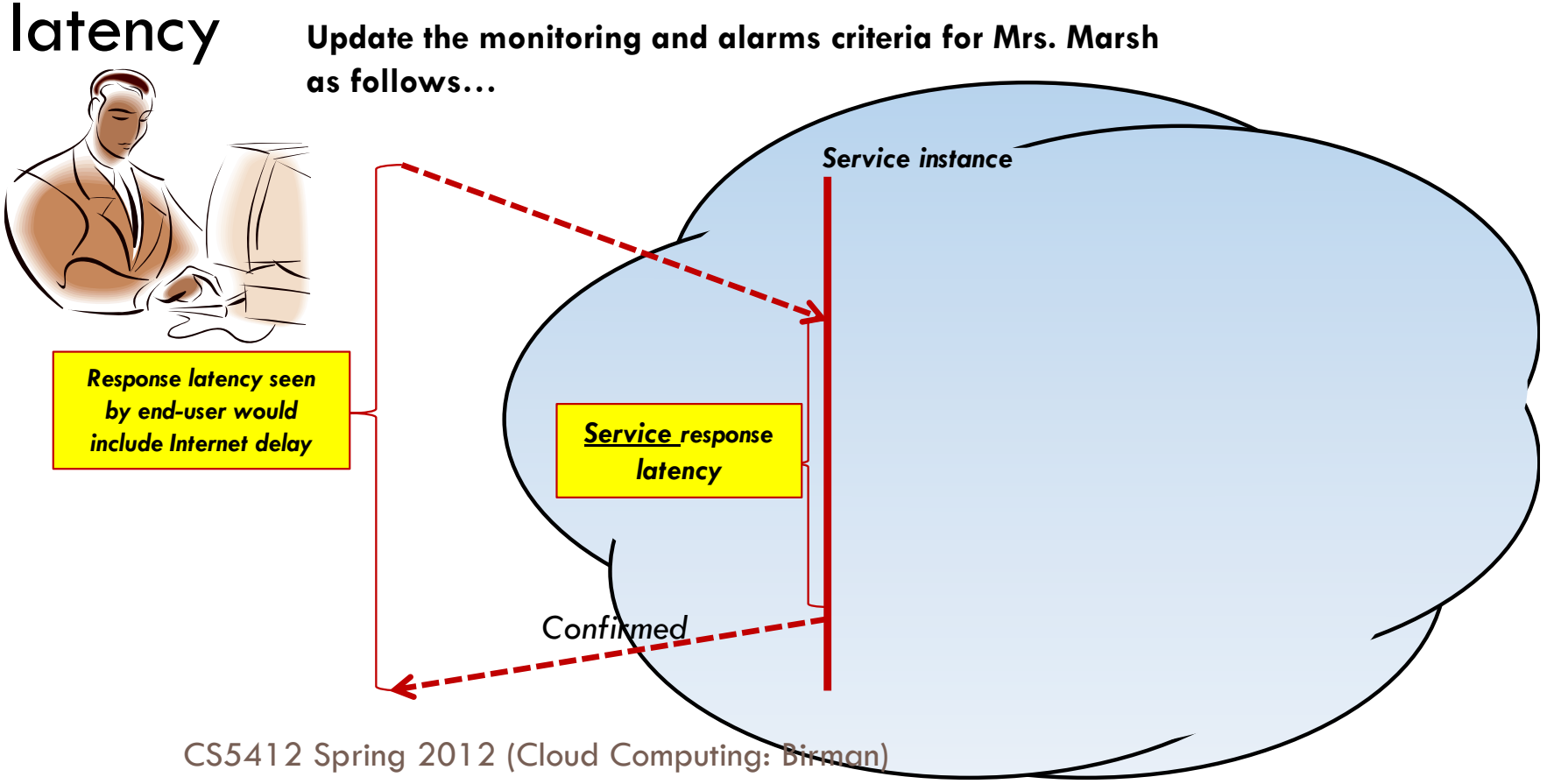
- Imagine a tier-one service running on 100k nodes
  - Can it ever make sense to replicate data on the entire set?
- *Yes*, **if** some kinds of information might be so valuable that almost every external request touches it
- Must think hard about patterns of data access and use
  - *Some* information needs to be heavily replicated to offer blindingly fast access on vast numbers of nodes
  - We want the level of replication to match the level of load and the degree to which the data is needed on the **critical path**

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
  - Others might need that several (perhaps a huge number of) machines undertake parts of the work in parallel
- The term sharding is used for data, but here we might talk about "parallel computation on a shard"

What does "critical path" mean? 1/2

- Focus on the latency of the reply to the client
- Critical path is formed by actions that contribute to this latency



What if a request triggers updates?

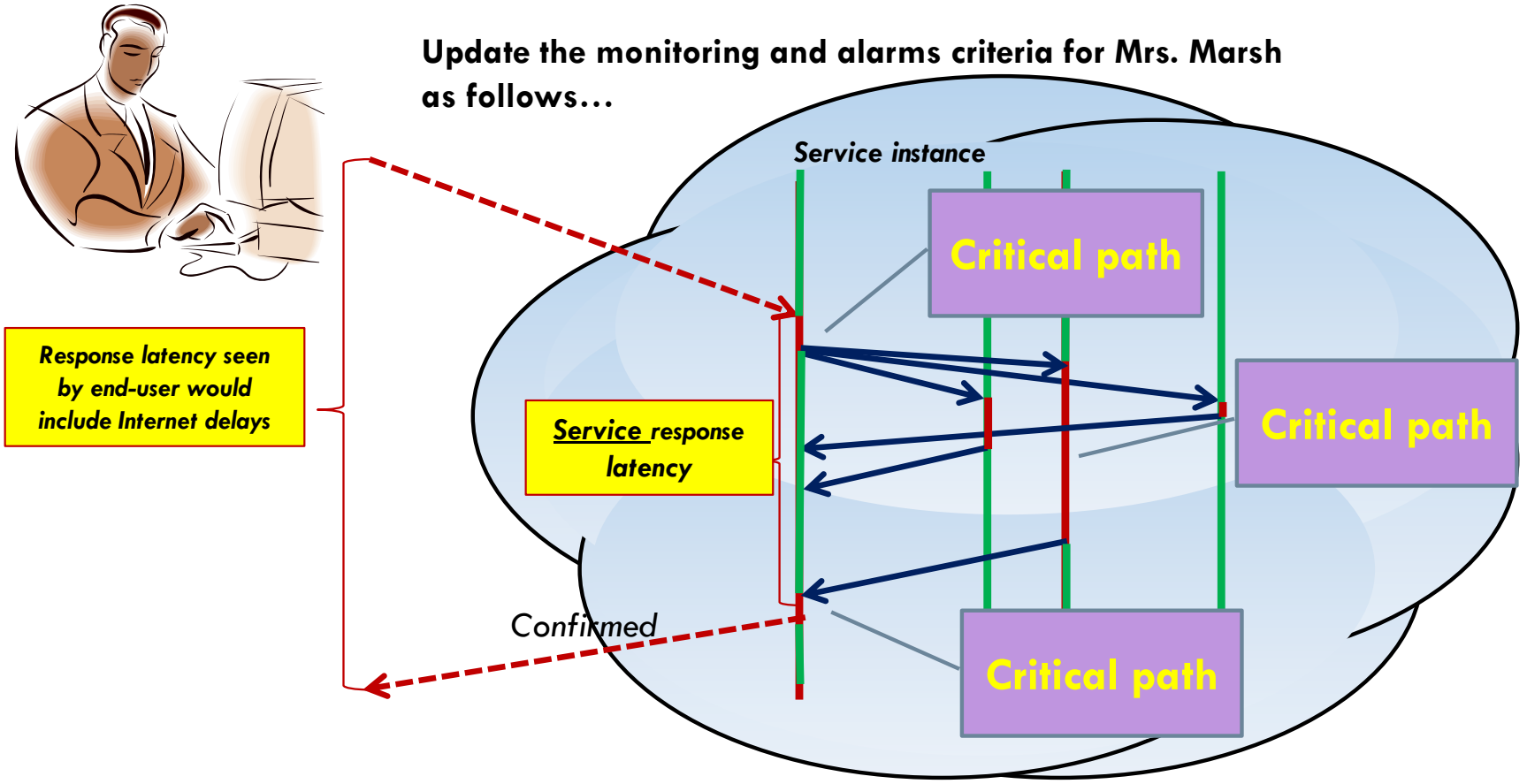
- If the updates are done **asynchronously** we might not experience much delay on the critical path
  - Cloud systems often work this way
  - Avoids waiting for slow services to process the updates but may force the tier-one service to "guess" the outcome
  - For example, could optimistically apply update to value from a cache and just hope this was the right answer
- Many cloud systems use these sorts of "tricks" to speed up response time

Tier-one parallelism

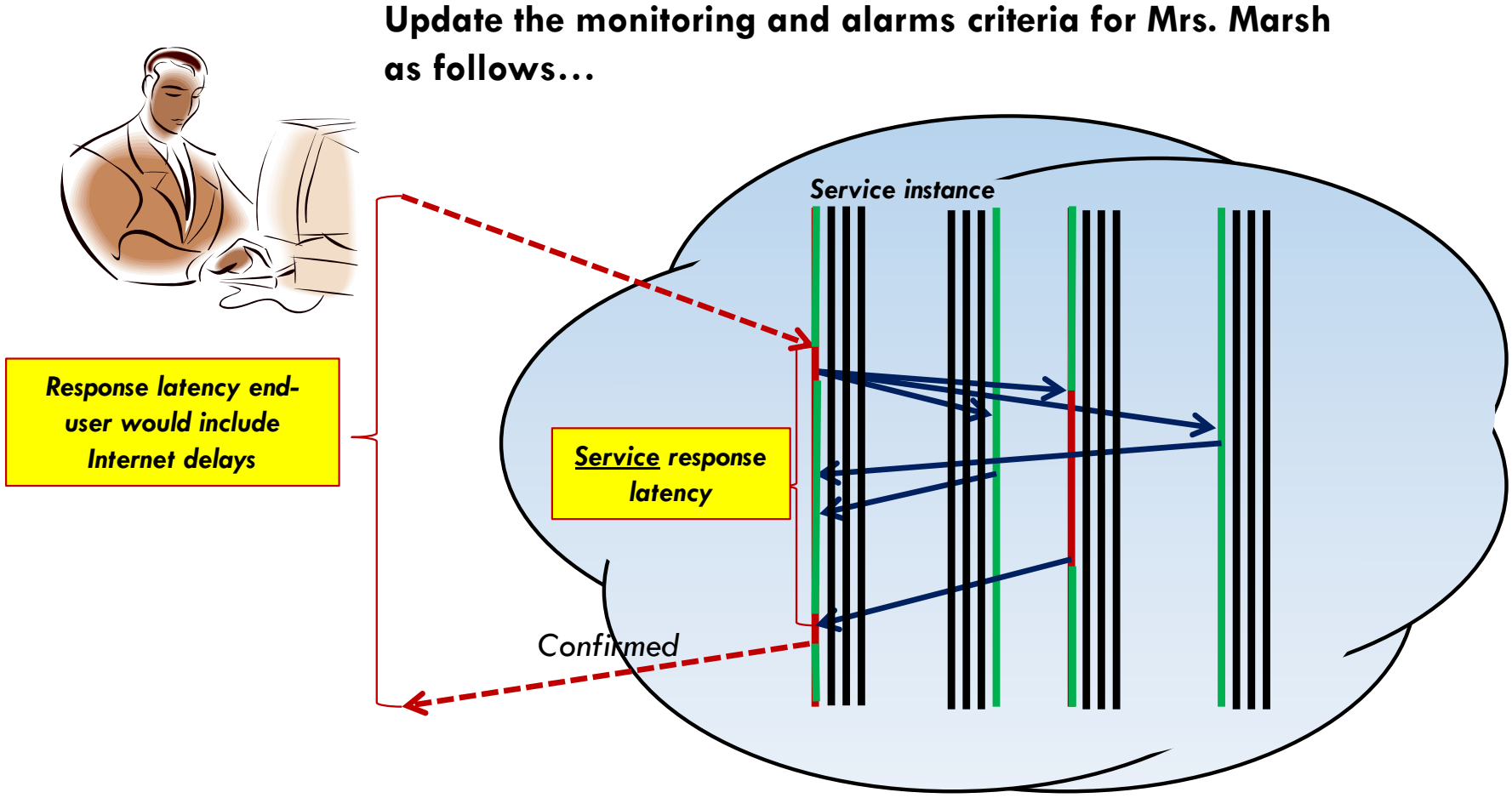
- Parallelism is vital to speeding up tier-one services
- Key question
  - Request has reached some service instance X
  - Will it be faster…
    - … For X to just compute the response
    - … Or for X to subdivide the work by asking subservices to do parts of the job?
- Glimpse of an answer
  - Werner Vogels, CTO at Amazon, noted in a talk that many Amazon pages have content from 50 or more parallel subservices that run, in real-time, on your request!

What does "critical path" mean? 2/2

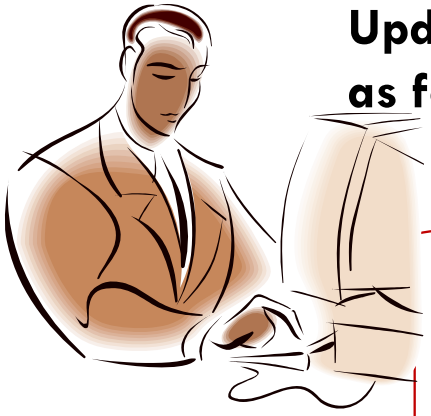
- In this example of a parallel read-only request, the critical path centers on the middle "**subservice**"



With replicas we just load balance



But when we add updates….



Update the monitoring and alarms criteria for Mrs. Marsh as follows…

Execution timeline for an individual tier-one replica

A B C D

Soft-state tier-one service

Send

Send

Send

…

Response latency seen by end-user would also include Internet delays

The delay associated with waiting for the multicasts to finish could impact the critical path even in a single service

Confirmed

What about updating w/o 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 are surprisingly lossy at times
    - Also, bursts of updates can queue up
- Such issues result in *inconsistency*

Eric Brewer's CAP theorem

- In a famous 2000 keynote talk at ACM PODC, Eric Brewer proposed that "*you can have just two from Consistency, Availability and Partition Tolerance*"
  - Data centers need very snappy response, hence availability is paramount
  - They should be responsive even if a transient fault makes it hard to reach some service
    - They should use cached data to respond faster even if the cached entry can't be validated and might be stale!
- Conclusion: weaken consistency for faster response

Clarification intermission /1

- **Consistency (C)**
  - There must be a total order on all operations
    - Equivalent to centralization with run-to-completion semantics
  - Each operation looks as if it were completed at a single instant
  - Each update is applied to all relevant replicas at the same logical time
  - Consistency that is both instantaneous and global is simply impossible

Clarification intermission /2

- **Availability (A)**
  - Every request received by a non-failing node must yield a response
  - Every request processing must terminate even under severe network failures
- **Partition Tolerance (P)**
  - Under PT, the network may lose arbitrarily many messages sent from one node to another
  - When a network is partitioned, all messages sent from nodes in one component of the partition to nodes in another component are lost
  - Any node failure can be seen as a network partition

Clarification intermission /3

- The probability that any one node fails (causing a network partition) rises exponentially with the number of nodes

$$P(failure) = 1 - P(individual\_node\_not\_failing)^{number\_of\_nodes}$$

- **Choosing C over A in the presence of partitions**
  - The system will preserve the guarantees of atomic reads and writes, and reject some requests
- **Choosing A over C**
  - The system will respond to all requests, potentially returning stale reads and accepting conflicting writes
  - Some of the conflicts may be resolved by Lamport's like algorithms

Clarification intermission /4

- Brewer proposed the useful notions of Yield and Harvest
- **Yield**
  - The probability of completing a request
  - More useful metric than *uptime*: being down at peak or off-peak times generates the same uptime but vastly different yields
- **Harvest**
  - The fraction of data reflected in the response
    - Which reflects the completeness of the answer to the query

$$\frac{data\_available}{total\_data}$$

- System design and implementation can influence whether faults impact Yield, Harvest or both
  - Replicated systems map faults to *reduced Yield* at peak times
  - Partitioned systems map faults to *reduced Harvest* for the same Yield

Is inconsistency a bad thing?

- How much consistency does tier one need?
  - YouTube videos: would consistency be an issue here?
  - Amazon's "number of units available" counters: will people notice if those are a bit off?
- Marvin Theimer's advice
  - Avoid costly guarantees that aren't even needed
  - But sometimes you just need to guarantee something
  - Then, be clever and engineer it to scale
  - And expect to revisit it each time you scale out 10x
- Performance-intensive scalability scenarios require looking closely at this tradeoff
  - Cost of stronger guarantee, versus
  - Cost of being faster but offering weaker guarantee

Properties we might want

- **Consistency**: Updates in an agreed order
- **Durability**: Once accepted, won't be forgotten
- **Responsiveness**: Replies with bounded delay
- **Security**: Only permit authorized actions by authenticated parties
- **Privacy**: Won't disclose personal data
- **Resilience**: Failures can't prevent the system from providing desired services
- **Coordination**: actions won't interfere with one another

Cloud services and their properties

Service	Properties it guarantees
Memcached	No special guarantees
Google's GFS	File is current if locking is used
BigTable	Shared key-value store with many consistency properties
Dynamo	Amazon's shopping cart: eventual consistency
Databases	Snapshot isolation with log-based mirroring (a fancy form of the ACID guarantees)
MapReduce	Uses a "functional" computing model within which offers very strong guarantees
Zookeeper	Yahoo! file system with sophisticated properties
PNUTS	Yahoo! database system, sharded data, spectrum of consistency options
Chubby	Locking service… very strong guarantees

THE WISDOM OF THE SAGES

eBay's Five Commandments

- As described by Randy Shoup at LADIS 2008

Wkrx#kdoo

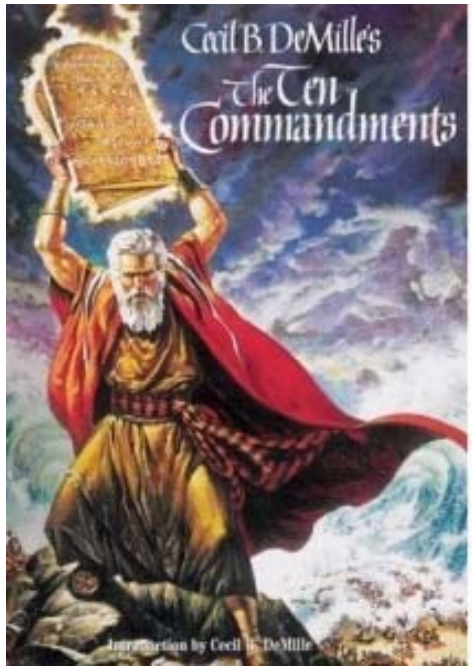
41#Sdwwhuq#Hyhu|wkbj

51#Xvh#Dv|qfkurq|#Hyhu|zkhuh

61#Dxwrp dwh#Hyhu|wkbj

71#Jhp hp ehu#Hyhu|wkbj#Idlov

81#Hp eudfh#Lqfrqvlwhqf|



Vogels at the Helm

- Werner Vogels, CTO at Amazon.com …
- Involved in building a new shopping cart service …
  - The old one used strong consistency for replicated data
  - New version was built over a DHT, like **Chord**, and has weak consistency with eventual convergence
    - Chord: a scalable P2P lookup service

- This weakens guarantees … but
  - ***Speed matters more than correctness***

James Hamilton's advice

- Key to scalability is decoupling, loosest possible synchronization
- *Any* synchronized mechanism is a risk
  - His approach: create a committee
  - Anyone who wants to deploy a highly consistent mechanism needs committee approval (cf. Lamport's Paxos)

…. They don't meet very often