SE252:Lecture 23/24, Apr 9/14

ILO5: Performance & Consistency *(CAP Theorem)*

Yogesh Simmhan

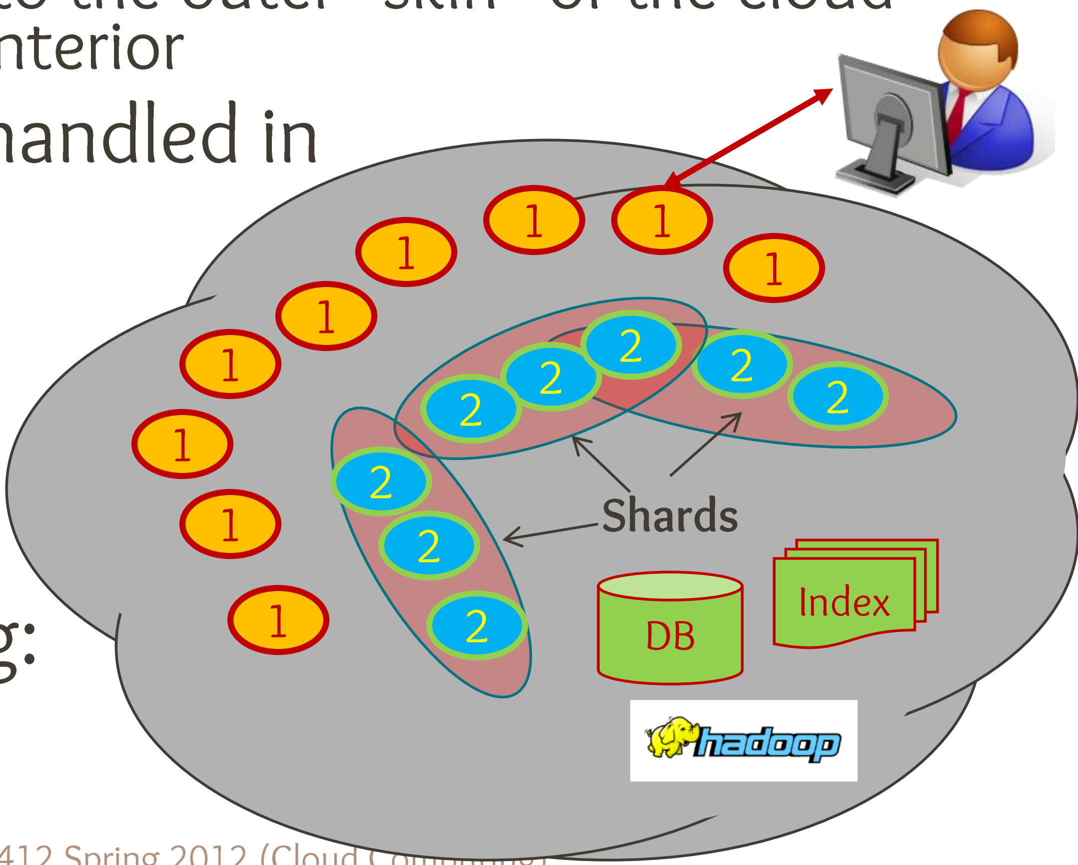
Today's Lecture based on © Ken Birman's CS5412 Spring 2012 (Cloud Computing), Lecture 7: Anatomy of a Cloud

ILO 5: Performance & Consistency on Clouds

- *Describe* and *compare* different performance metrics for evaluating Cloud applications and ✓
- *demonstrate* their use for application measurement. ➲Project
- *Explain* the distinctions between Consistency, Availability and Partitioning (CAP theorem), and
- *discuss* the types of Cloud applications that exhibit these features.

Client+Cloud Interaction

- Clients talk to clouds using web browsers/services
  - But this only gets us to the outer "skin" of the cloud data center, not the interior
- Client requests are handled in the "first tier" by
  - PHP/ASP + Biz logic
- These lightweight services are fast and very nimble
- Much use of caching: the second tier



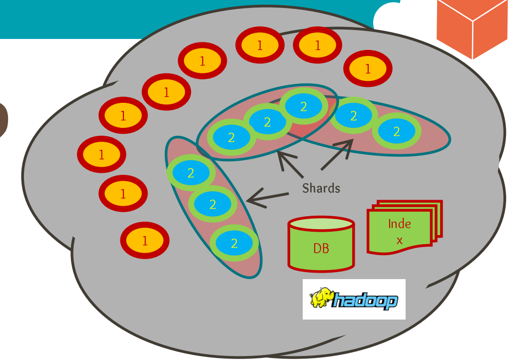
Many styles of system

- *Near the edge* of the cloud focus is on vast numbers of clients and 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 resources
  - Applications like MapReduce (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 is to *always used cached data if at all possible*, so the inner services (here, a database and a search index stored in a set of files) are shielded from "online" load
  - We need to replicate data within our cache to spread loads and provide fault-tolerance
  - But not everything needs to be "fully" replicated.
  - *Hence we often use "shards" (horizontal partitions) with just a few replicas*

Sharding used in many ways

- The second tier could be any of a number of caching services:
  - **Memcached:** a sharable in-memory key-value store
  - Other kinds of **DHTs** 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
- 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 make sense to replicate data on the entire set?*

?

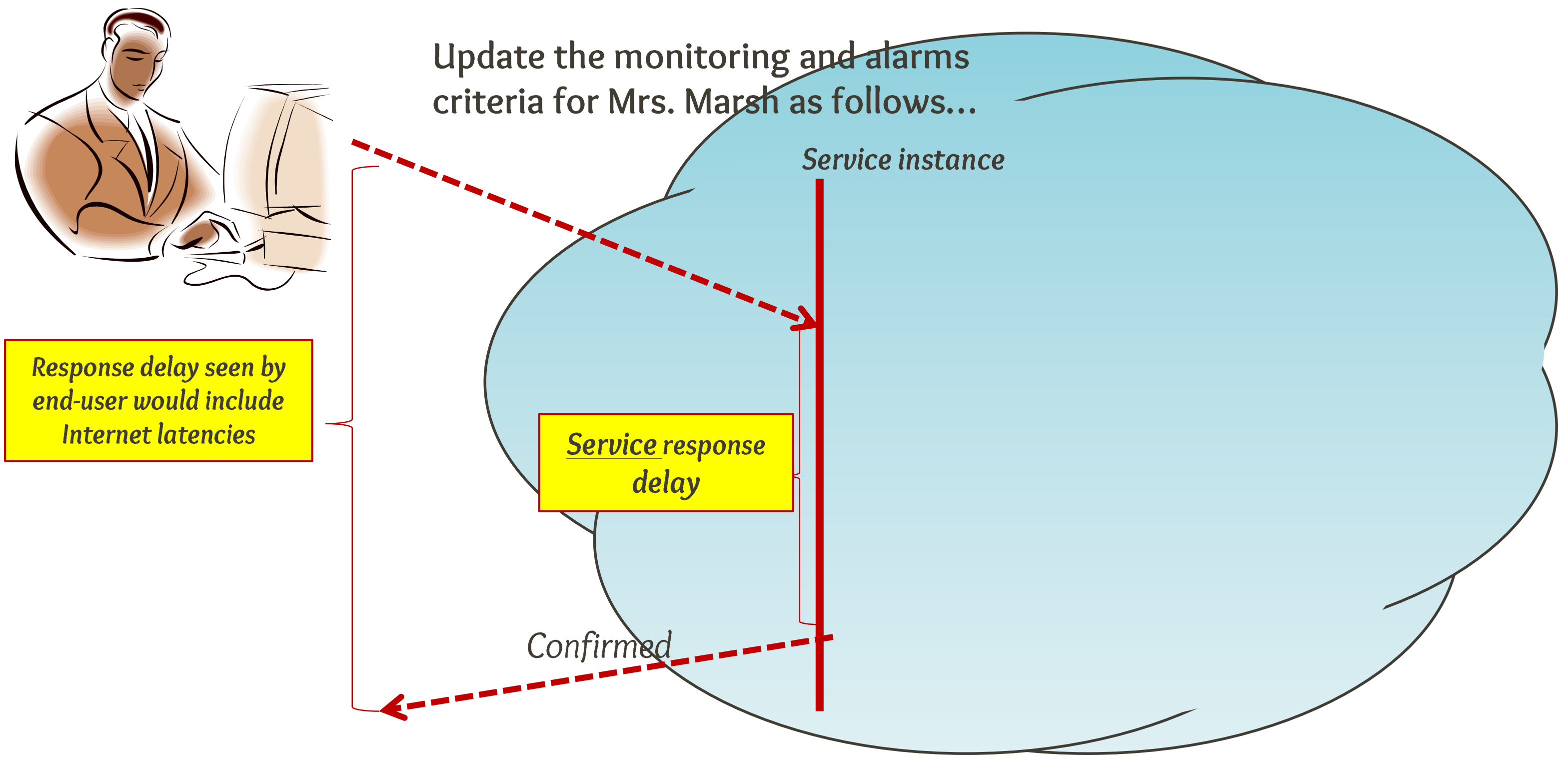
- *Yes, if some information might be so valuable that almost every external request touches it.*
  - Must think hard about patterns of data access and use
  - Some information needs to be heavily replicated to offer blindingly fast access on vast numbers of nodes
  - The principle is similar to the way Beehive operates.
    - » Even if we don't make a dynamic decision about the level of replication required, the principle is similar
    - » We want the level of replication to match 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
  - 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"*

What does "critical path" mean?

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



What if a request triggers updates?

- If updates are done "asynchronously" we might not experience much delay on 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
  - E.g. 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

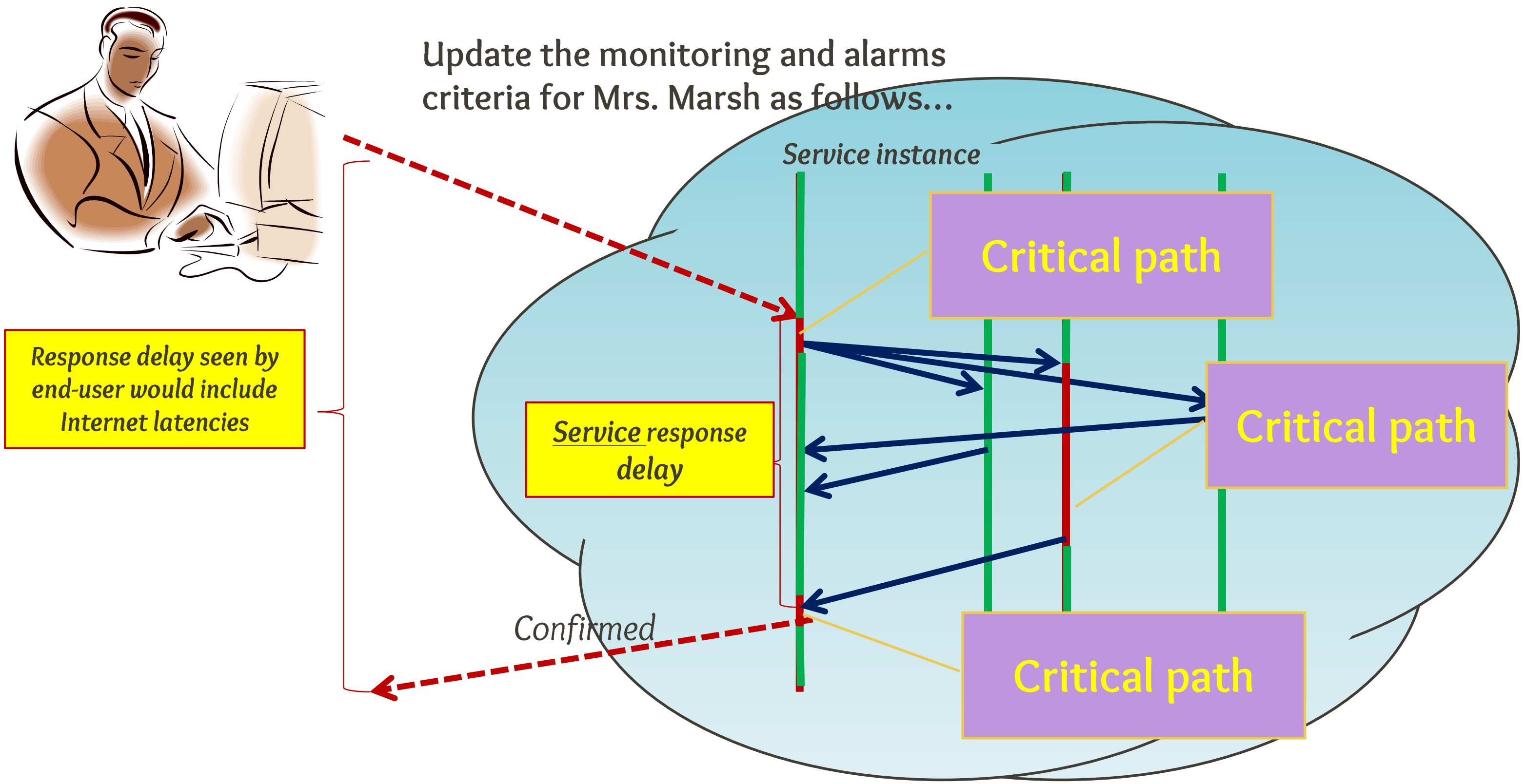
First-tier parallelism

- *Parallelism is vital to speeding up first-tier 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, commented in one talk that many Amazon pages have content from 50 or more parallel subservices that ran, in real-time, on your request!*

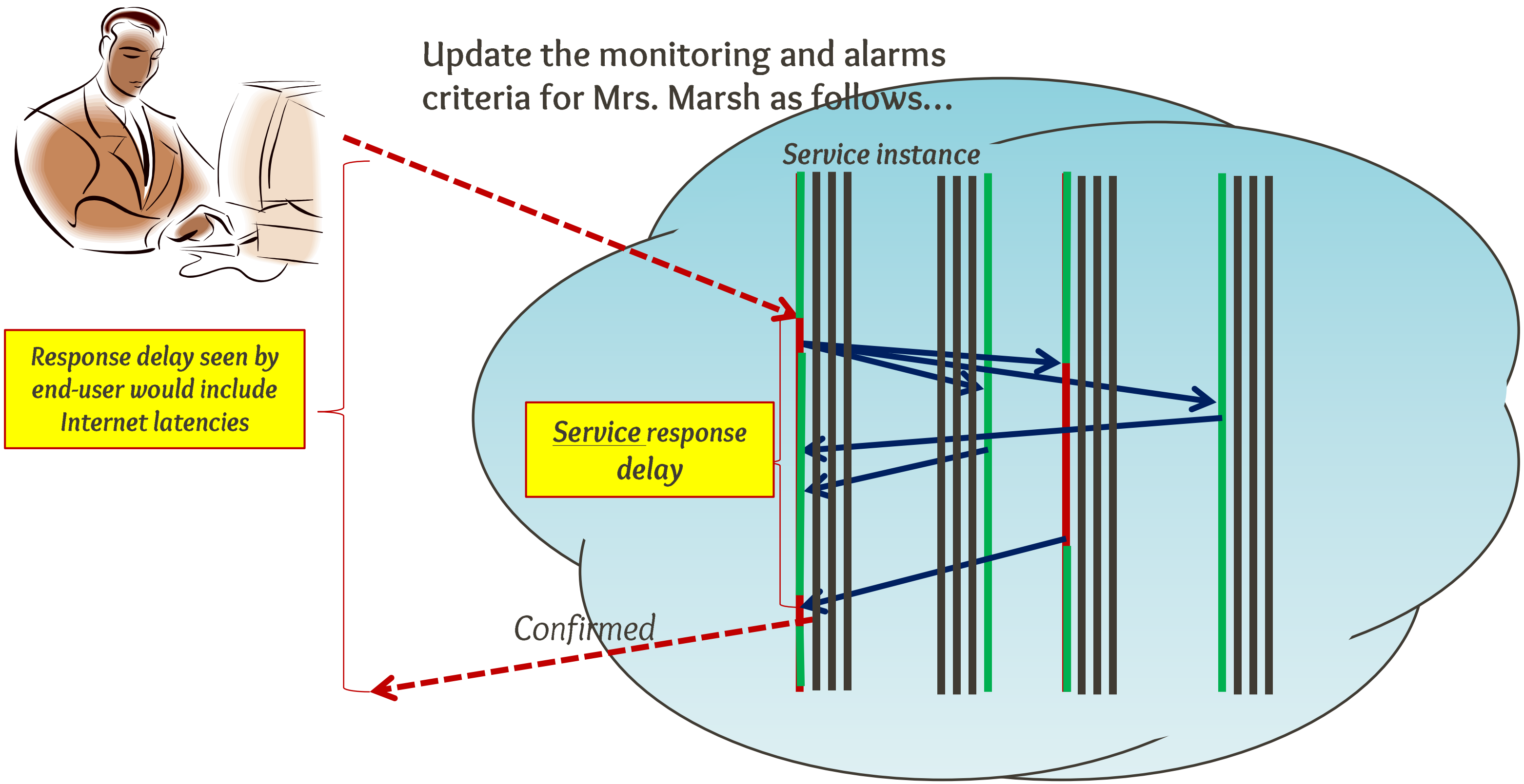
?

What does "critical path" mean?

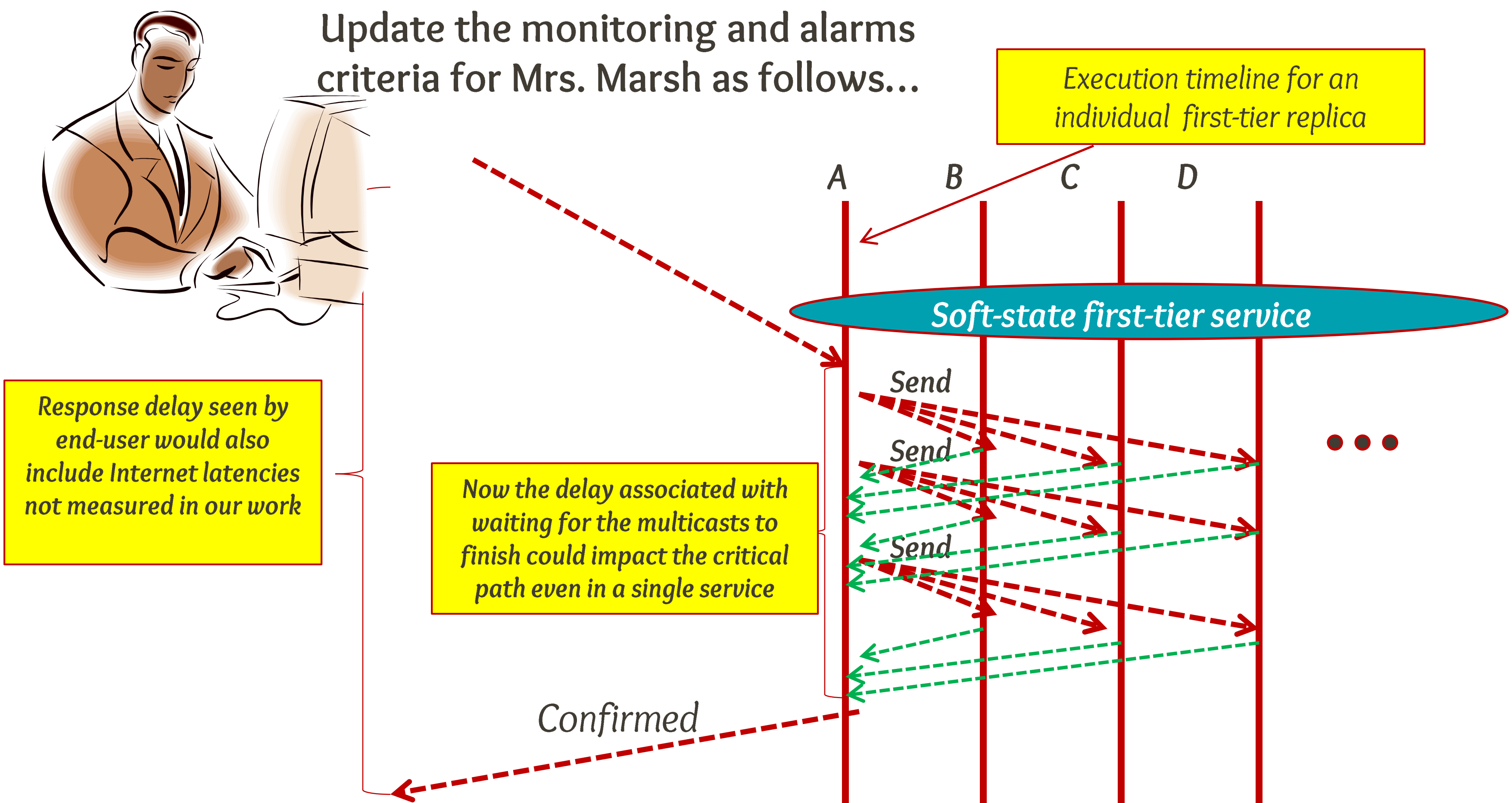
- E.g. in this parallel read-only request, the critical path is the middle "subservice"



With replicas we just load balance



But when we add updates….



What if we send updates without waiting?

- Several issues now arise
  - Are all the replicas applying updates in the same order?
    - » May 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"*
  - He argues that data centers need very snappy response, hence availability is paramount
  - And they should be responsive even if a transient fault makes it hard to reach some service.
  - So they should use cached data to respond faster even if the cache can't be validated and might be stale!
- Conclusion: **weaken consistency for faster response**

CAP Twelve Years Later: How the "Rules" Have Changed, Eric Brewer, *IEEE Computer*, FEBRUARY 2012

CAP theorem

- A proof of CAP was later introduced by MIT's Seth Gilbert and Nancy Lynch
  - Suppose a data center service is active in two parts of the country with a network link between them
  - We temporarily cut the link ("partitioning" the network)
  - And present the service with conflicting requests
- The replicas can't talk to each other so can't sense the conflict
- If they respond at this point, inconsistency arises

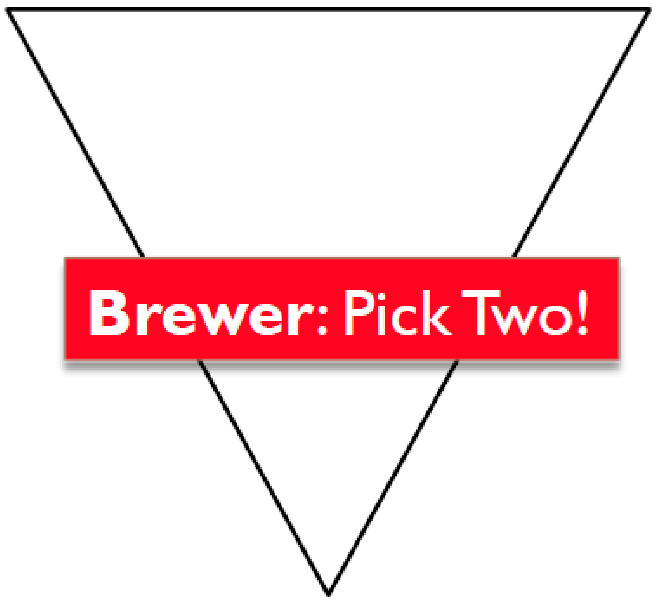
Perspectives on the CAP Theorem,
Seth Gilbert & Nancy A. Lynch, *IEEE Computer*, FEBRUARY 2012

Atomic/Linearizable Consistency

Exist a total order of all Operations such that each operation looks as if it were completed at a single instant

Availability

Every request received by a non-failing node must result in a response

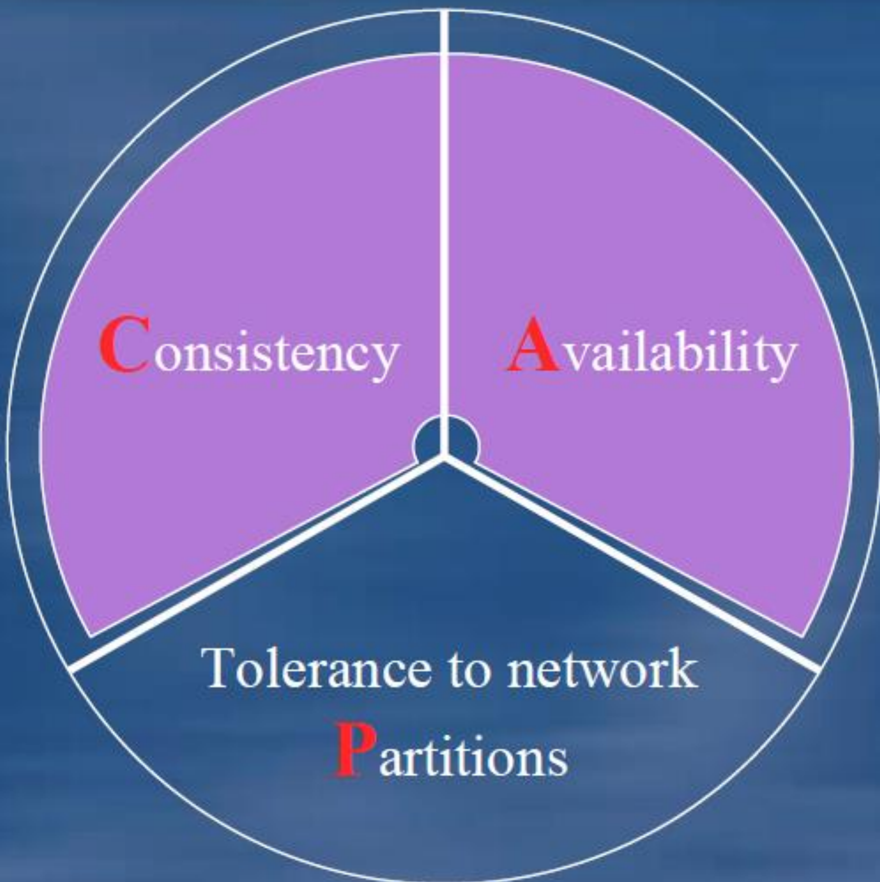


Partition-tolerance

No set of failures less than total network failure Is allowed to cause the system to response incorrectly

Forfeit Partitions

Inktomi



Examples

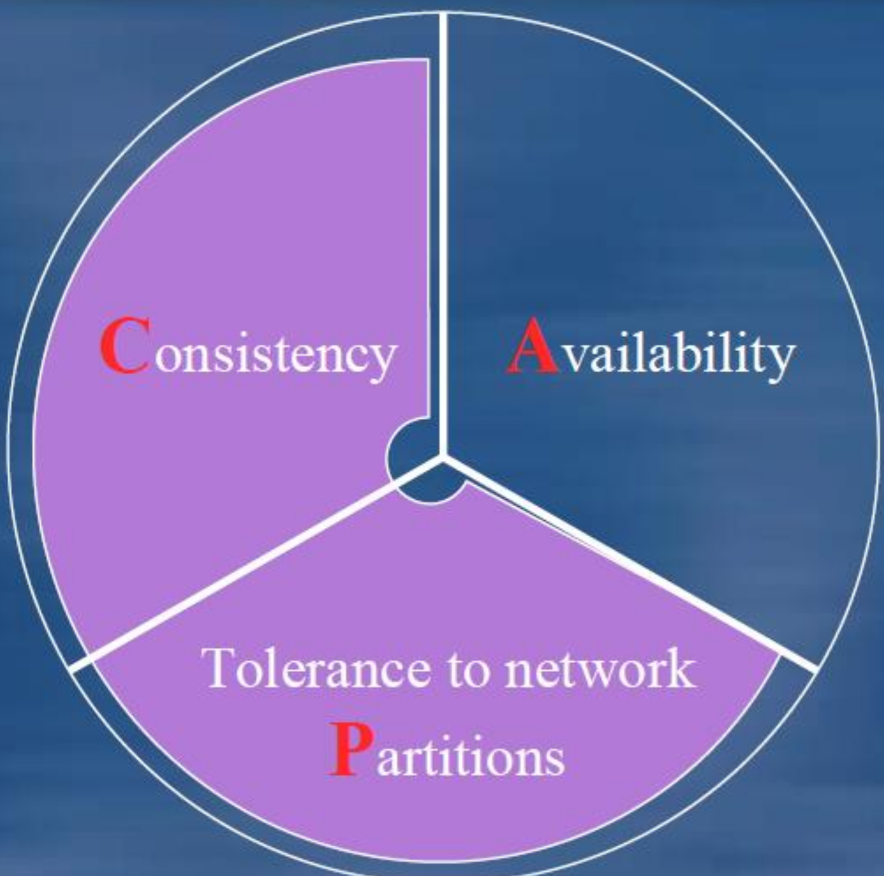
- Single-site databases
- Cluster databases
- LDAP
- xFS file system

Traits

- 2-phase commit
- cache validation protocols

Forfeit Availability

Inktomi



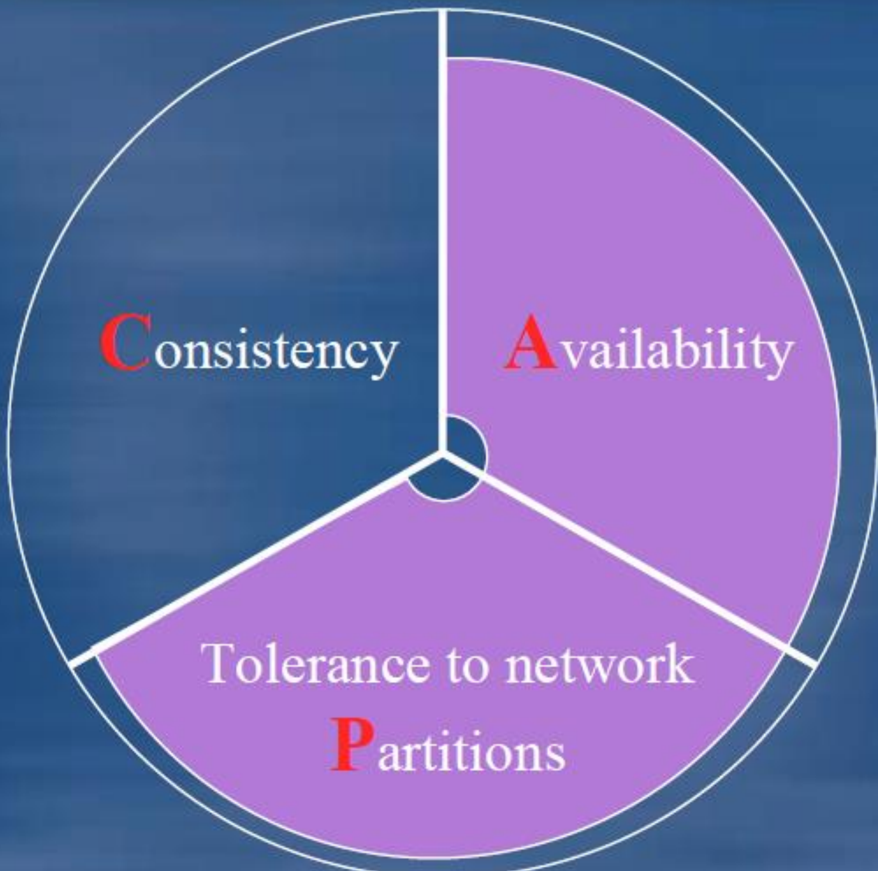
Examples

- Distributed databases
- Distributed locking
- Majority protocols

Traits

- Pessimistic locking
- Make minority partitions unavailable

Forfeit Consistency



Examples

- Coda
- Web caching
- DNS

Traits

- expirations/leases
- conflict resolution
- optimistic

Is inconsistency a bad thing?

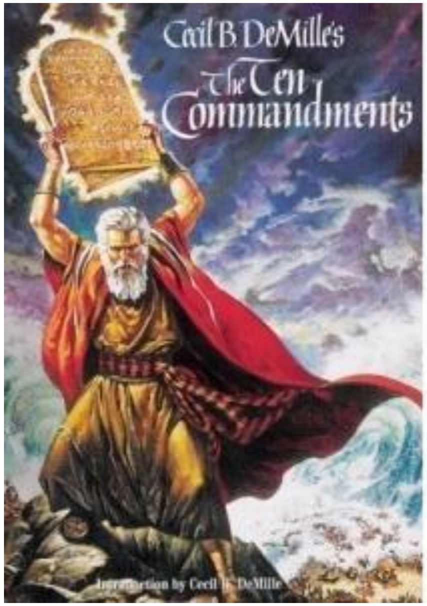
- How much consistency is really needed in the first tier of the cloud?
  - *Think about YouTube videos. Would consistency be an issue here?*
  - *What about the Amazon "number of units available" counters. Will people notice if those are a bit off?*
- **Puzzle**: Can you come up with a general policy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



Vogels at the Helm

- Werner Vogels is CTO at Amazon.com…
- He was involved in building a new shopping cart service
  - The old one used strong consistency for replicated data
  - New version was build over a DHT, like Chord, and has weak consistency with eventual convergence

- 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*

…. They don't meet very often

Consistency

Consistency technologies just don't scale!

But inconsistency brings risks too!

My rent check bounced?
That can't be right!

- Inconsistency causes bugs
  - Clients would never be able to trust servers… a free-for-all

- Weak or "best effort" consistency?
  - Strong security guarantees demand consistency
  - Would you trust a medical electronic-health records system or a bank that used "weak consistency" for better scalability?

Puzzle: Is CAP valid in the cloud?

- **Facts:** data center networks don't normally experience partitioning failures
  - Wide-area links do fail
  - But most services are designed to do updates in a single place and mirror read-only data at others
  - *So the CAP scenario used in the proof can't arise*
- Brewer's argument about not waiting for a slow service to respond does make sense
  - Argues for using any single replica you can find
  - But does this preclude that replica being consistent?

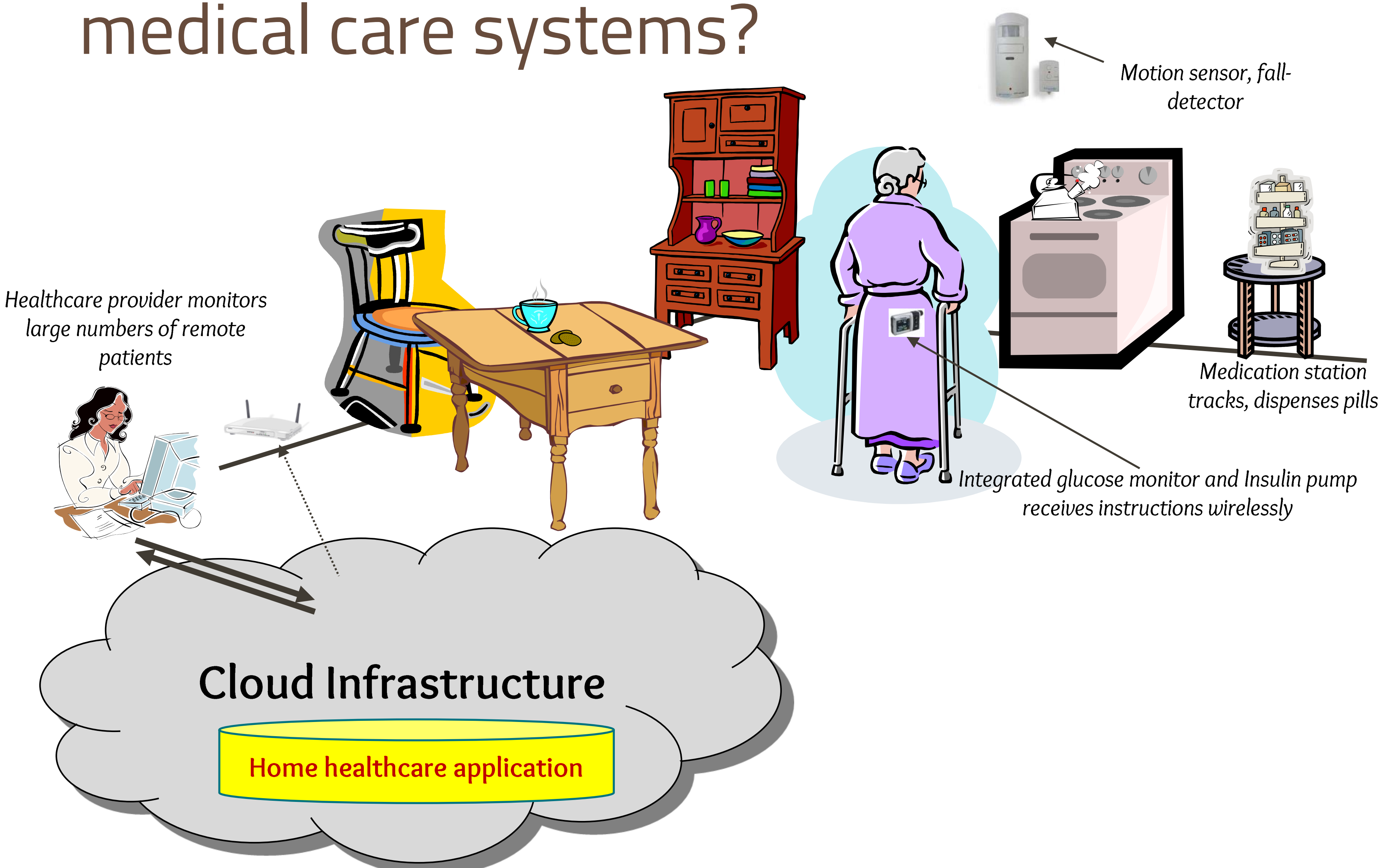
What does "consistency" mean?

- We need to pin this basic issue down!

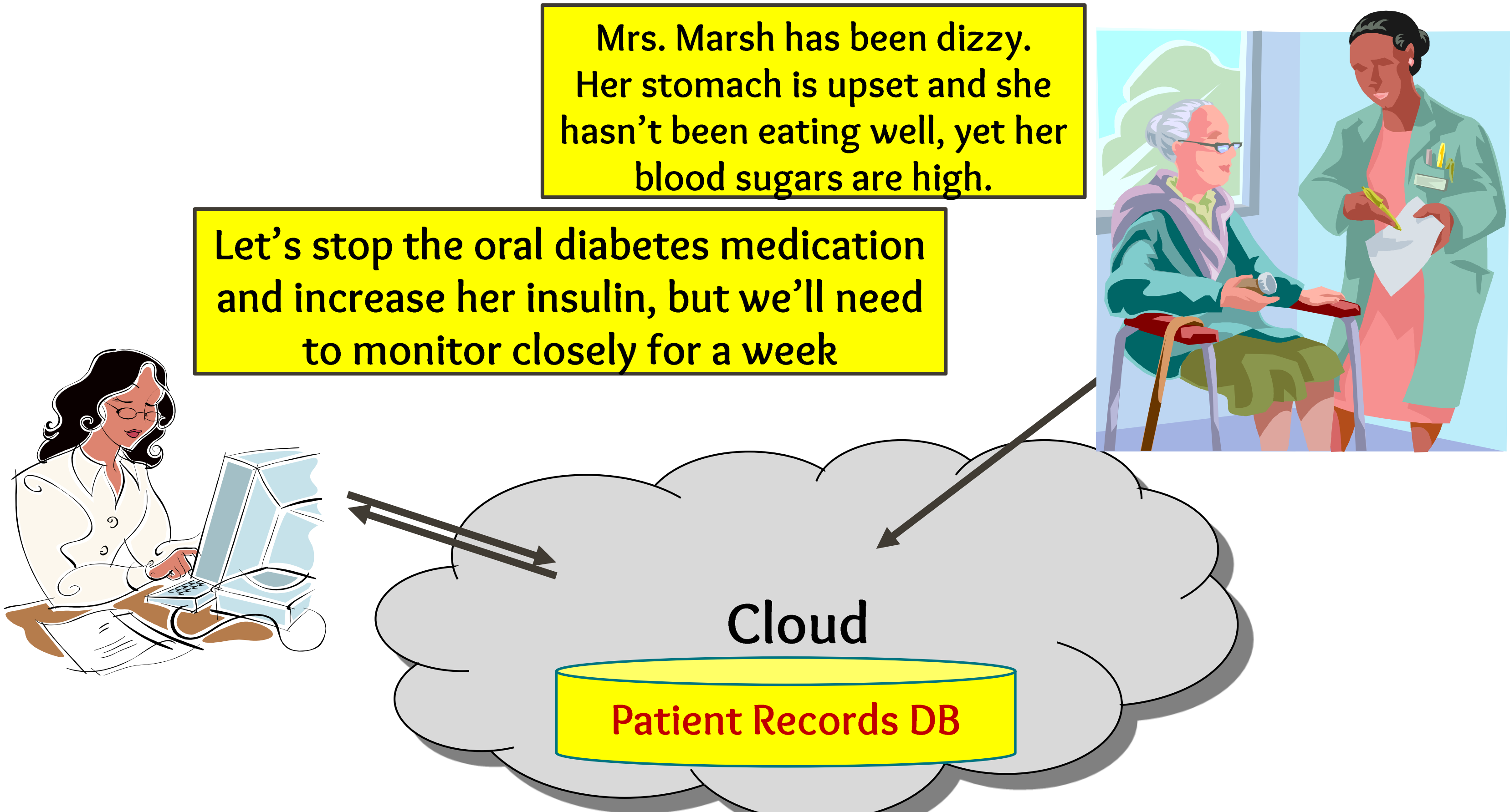
- As used in CAP, consistency is about two things
  1. First, that *updates* to the *same data item* are applied in *some agreed-upon order*
  2. Second, that once an *update is acknowledged* to an external user, it *won't be forgotten*

- Not all systems need both properties

What properties are needed in remote medical care systems?

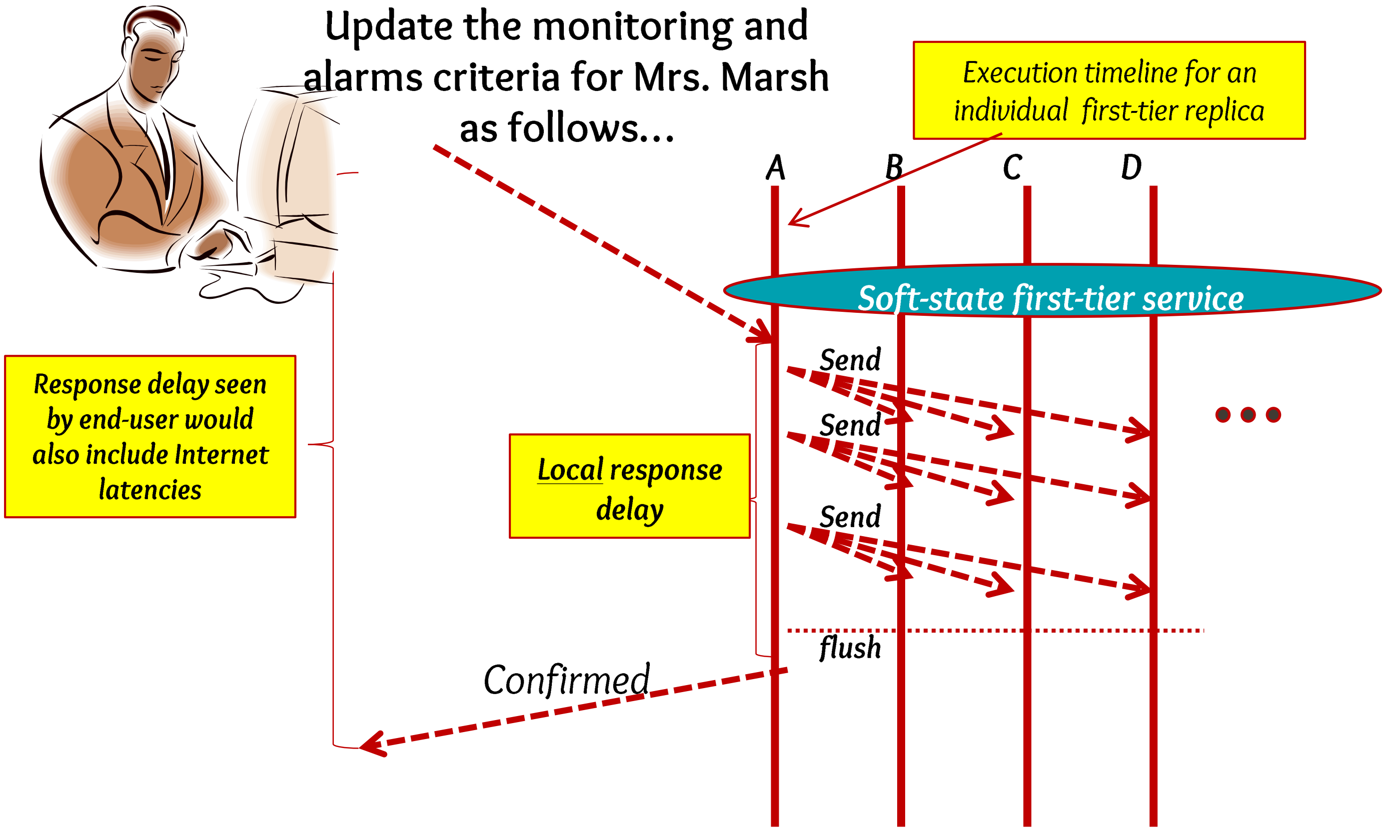


Which matters more: fast response, or durability of the data being updated?



- Need: Strong consistency ***and*** durability for data

What if we were doing online monitoring?



- Online monitoring may focus on real-time response & value consistency, yet be **less concerned** with durability

Why does monitoring have weaker needs?

- When a monitoring system goes "offline" the device turns a red light or something on.
  - Later, on recovery, the monitoring policy may have changed and a node would need to reload it
  - Moreover, with in-memory replication we may have a strong enough guarantee for most purposes
- *If durability costs enough to slow us down, we might opt for a weaker form of durability in order to gain better scalability and faster responses!*
- *E.g. Sensors in IoT for Water Management*

This illustrates a challenge!

- *Cloud systems just can't be approached in a one-size fits all manner*
- For performance-intensive scalability scenarios we need to look closely at tradeoffs
  - Cost of stronger guarantee, versus
  - Cost of being faster but offering weaker guarantee
- *If systems builders blindly opt for strong properties when not needed, we just incur other costs!*
  - Amazon: Each 100ms delay reduces sales by 1%!

Properties we might want

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

Does CAP apply deeper in the cloud?

- The principle of wanting speed and scalability certainly is universal
- But many cloud services have strong consistency guarantees that we take for granted but depend on
- Marvin Theimer at Amazon explains:
  - *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*

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

Is there a conclusion to draw?

- One thing to notice about those services…
  - Most cost $10's or $100's of millions to create!
  - Huge investment required to build strongly consistent and scalable and high performance solutions
  - Oracle's current parallel database: billions invested
- CAP isn't about telling Oracle how to build a database product…
  - *CAP is a warning to you that strong properties can easily lead to slow services*
  - *But thinking in terms of weak properties is often a successful strategy that yields a good solution and requires less effort*

Core problem?

- When can we safely sweep consistency under the rug?
  - If we weaken a property in a safety critical context, something bad can happen!
  - Amazon and eBay do well with weak guarantees *because many applications just didn't need strong guarantees to start with!*
  - By embracing their weaker nature, we reduce synchronization and so get better response behavior
- *But what happens when a wave of high assurance applications starts to transition to cloud-based models?*

Proposition

- High assurance cloud computing is just around the corner!
  - Experts already doing it in a plethora of services
  - The main obstacle is that typical application developers can't use the same techniques
- As we develop better tools and migrate them to the cloud platforms developers use, options will improve