University of Bologna

Dipartimento di Informatica –
Scienza e Ingegneria (DISI)
Engineering Bologna Campus

Class of
Infrastructures for
Cloud Computing and **Big Data M**

Cloud Support and Global Strategies

Antonio Corradi

Academic year 2019/2020

CLOUD DATA CENTER

Let us have a look at the **cloud internal organization**…
The Cloud means **a big data center**, **federated with other ones**, and capable of giving g**ood and fast answers** to users

There are several **(even contrasting) requirements** a DC must obey to:
- **Fast answers to users**, so not to let them wait for too much
- Giving **correct & safe answers**, without taking answer time in mind

There are two main tendencies for customer satisfaction :

- **Real time in answering**
- **Safety of data in the architecture**

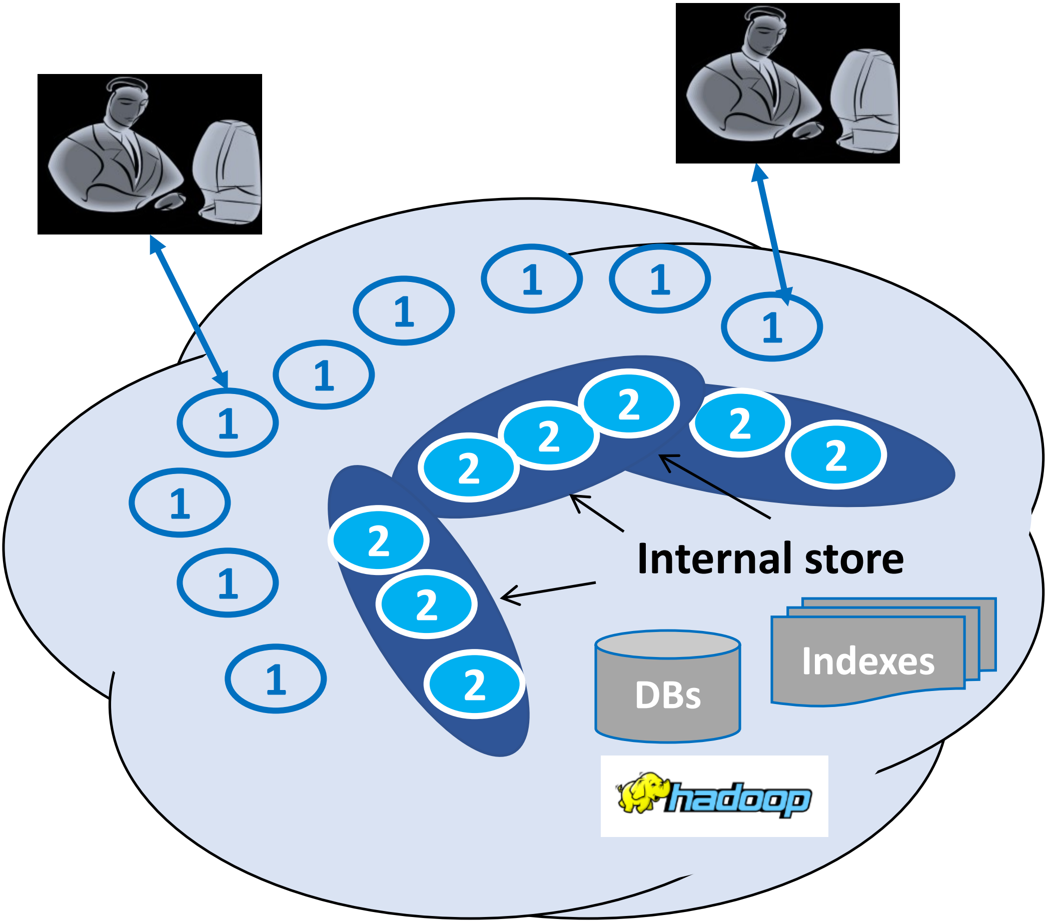
In global environments, where many copies are present and reside in many geographical locations

Cloud Data Center

Let us have a look at the **cloud internal organization**…
The Cloud means **a big data center**, **federated with other ones**, and capable of giving good and fast answers to users.

It consists of **two levels** of service in a **two-level** architecture:

- The **first level** is the **one interacting with users** that requires **fast and prompt answers**
- The **second, deep level** is the one in charge of **deep data** and their **correct** and **persistent values**

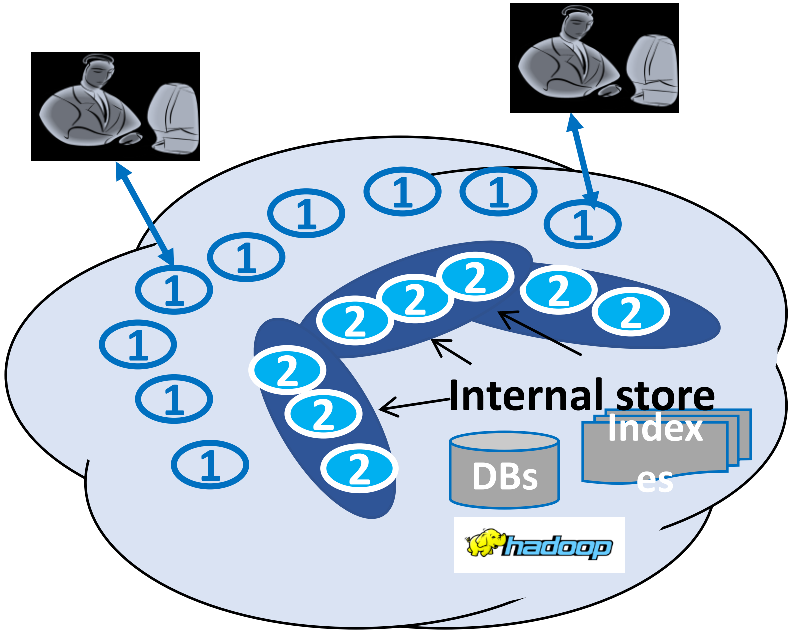


CLOUD EDGE LEVEL 1

The **level 1** of the Cloud **is the layer very close to the client** in charge of the **fast answer to user needs**

This level is called **“CLOUD edge”** and must give very prompt answers to **many possible client requests, even concurrent ones with user reciprocal interaction**

The edge level has the first requirement of **velocity** and **should return fast answers**: for read operations, no problem; for write operations some **problems may arise** and **updates are tricky**



Easy **guessing model**: try to **forecast the update outcome** and **answer fast**, but operate in background with the level 2

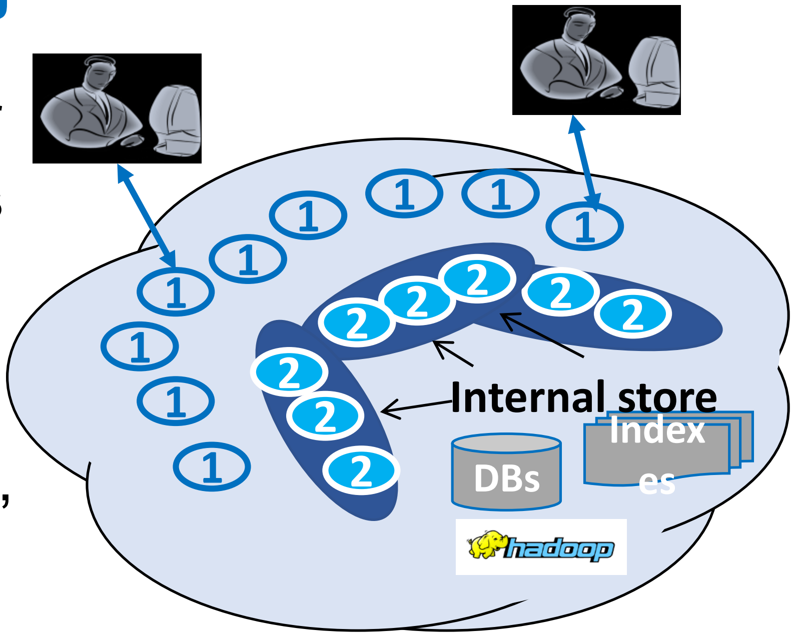
LEVEL 2: CLOUD INTERNAL

The **level 2** of the **Cloud** is the layer responsible for stable **answers** to the **users given by level 1 and of their responsiveness.**

This level is **CLOUD internal**, hidden from users and away from **online duties of fast answer.**

Level 2 is in charge of **replicating data and keeping caches** to favor user answers. Replication of course can provide several copies to provide fault tolerance and to spread loads

Replication policies do not require **replication of everything**, but only some **significant parts** are replicated (called **shard** or 'important' pieces of information **dynamically decided**)



Cloud Two Levels

Let us have a deeper look at the replication in a **cloud data center**

Any level replicates

- The **first edge level** proposes an architecture based on **replication** tailored to **user needs. Resources are replicated for user demands and to answer with the negotiated SLA within deadlines**
- The **second internal level** must support the **user long-term strategies** (also **user interactions and mutual replication**) and its design meets that requirements for different areas. The second level **optimizes data in term of smaller pieces called shards**, and also supports **many forms of caching services** (memcached, dynamo, bigtable, …)

Any level replicates in a dynamic way, according to current necessity

Cloud Replication – Level 1

Replication is used extensively in Cloud, at any level of the data center, with different goals

At the **level edge 1**, users expect a good support for their needs
Replication is the key for an efficient support and for prompt answer (often transparently to the final user)

- **Processing**: any client must own **an abstraction of at least one dedicated server activity** (even more in some case for specific users)
- **Data**: the server must organize **copies of data to give efficient and prompt answers**
- **Management**: the Cloud control must be capable of **controlling resources in a previously negotiated way, toward the correct SLA**

Replication is not user-visible and transparent

Cloud Replication – Level 2

Cloud replication at the level 2 has the goal of supporting safe answers to user queries and operations, but it is **separated as much as possible from the level 1**
Replication here is in the use of **fast caches that separate the two levels and make possible internal deep independent policies**

Typically the level 2 tends to use replication in a more **systemic perspective**, so driven by the **whole load** and **less dependent on single user requirements.**
In general, nothing is **completely and fully replicated** (too expensive), and **Cloud identifies smaller dynamic pieces** (or shards) that are **small enough** not to clog the system and changing based on **user dynamic needs**

SHARDING is used at any level of Datacenters

Cloud Shards

The definition of **smaller contents for replication** (**sharding**) is very important and cross-cutting

- Interesting systems replicates data but must define the proper pieces of data to replicate (shards), both to achieve **high availability** and to **increase performance**
- Depending on **use** and **access to data** we replicate **most requested pieces** and **adapt them** to changing requirements
- Data cannot be **entirely** replicated with a **high** degree of replication

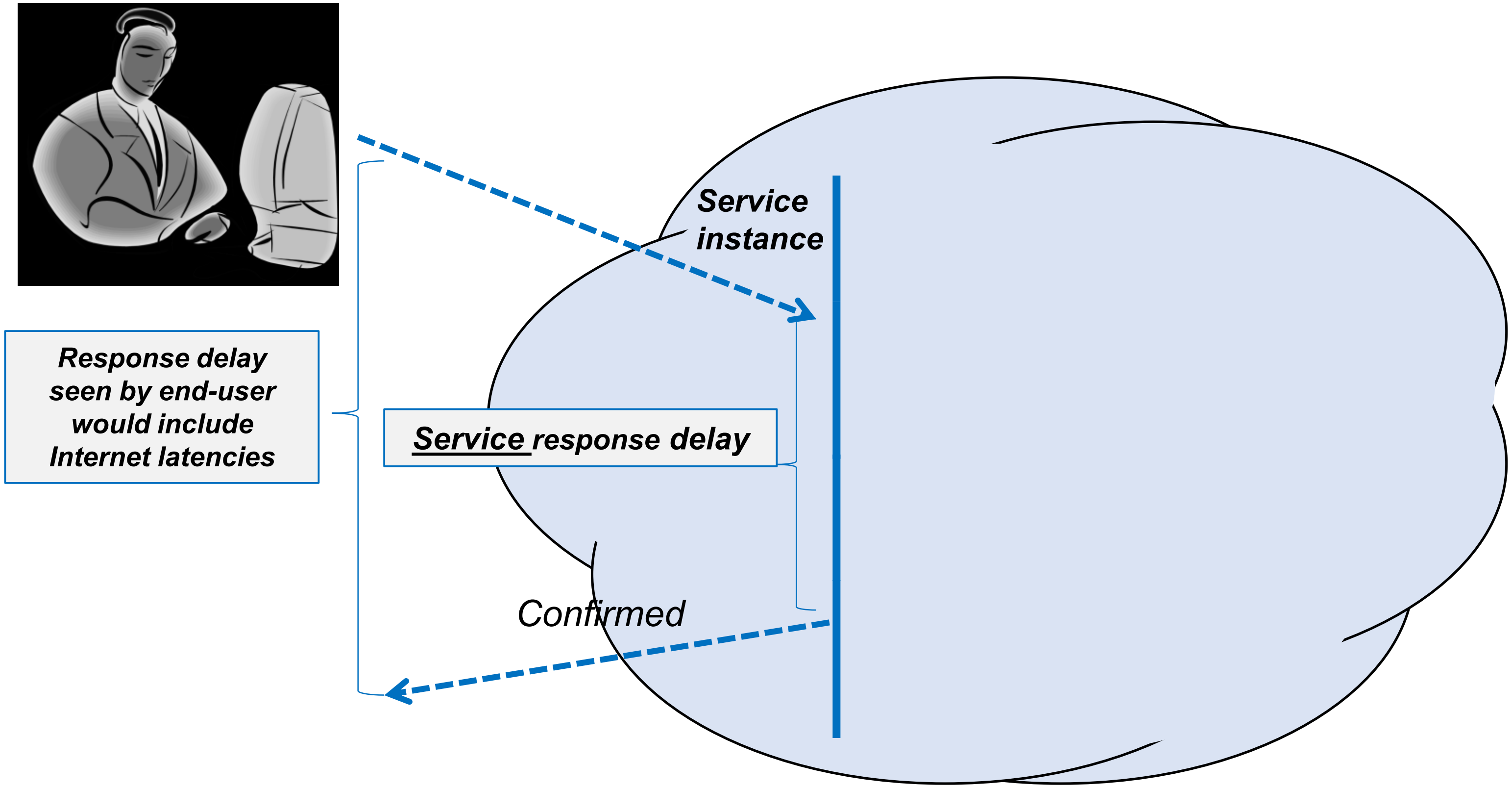
Shards may be very different depending on current usage

*If a piece of **data is very critical**, it is replicated in **more and more copies** to support the operations*

*Another critical point is when the same data is operated upon by several processes: the **workflow must be supported not to introduce bottlenecks** (so parallelism of access can shape data shard)*

Cloud Services Time

- **Users expects a very fast answer** and some operations accordingly.
- **The system must give fast answers** but must operate with dependability (reliability and availability).



Service instance

Response delay seen by end-user would include Internet latencies

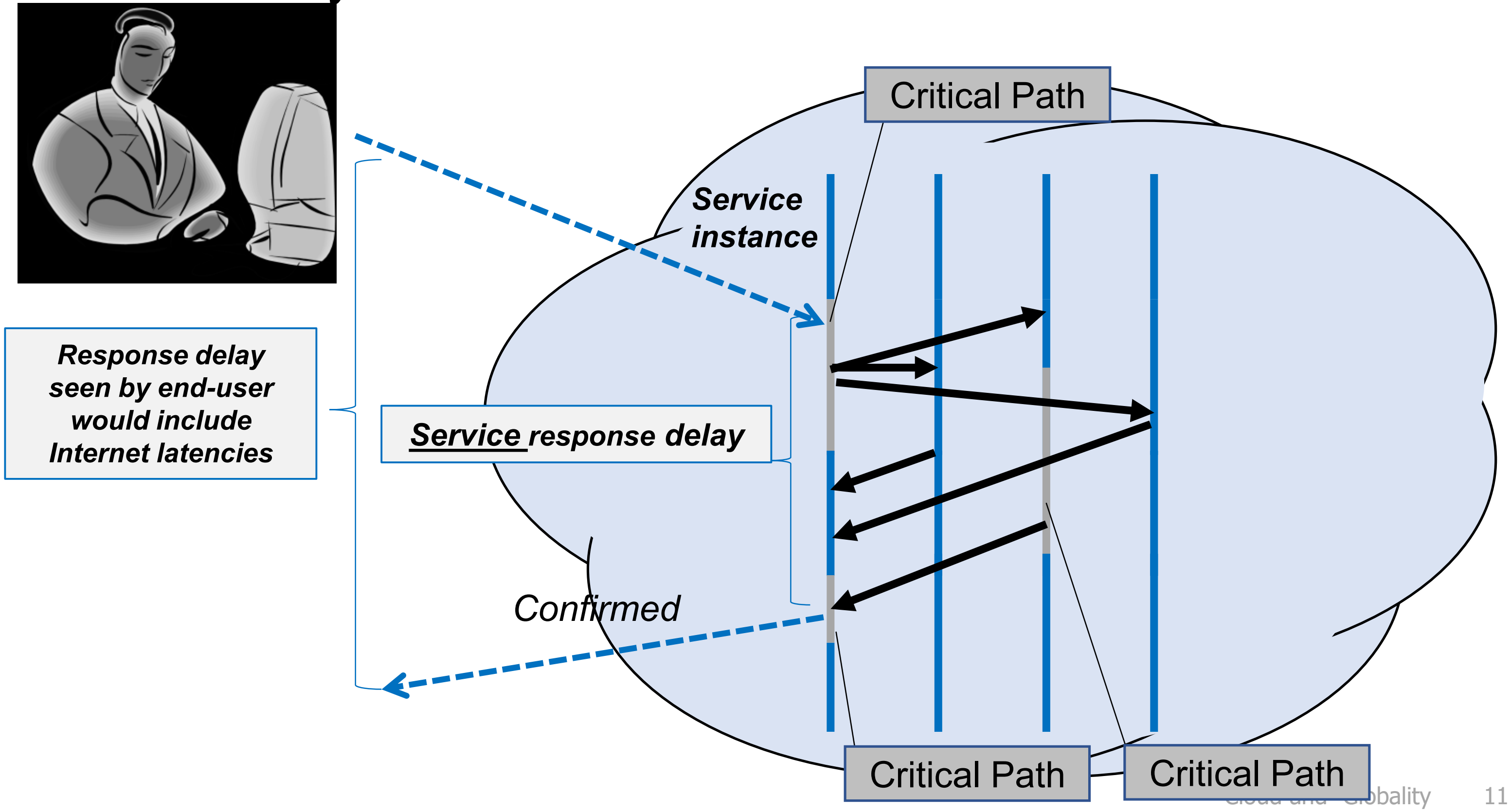
Service response delay

Confirmed

Cloud Critical Paths

The fast answers are difficult for working synchronously if you have subservices

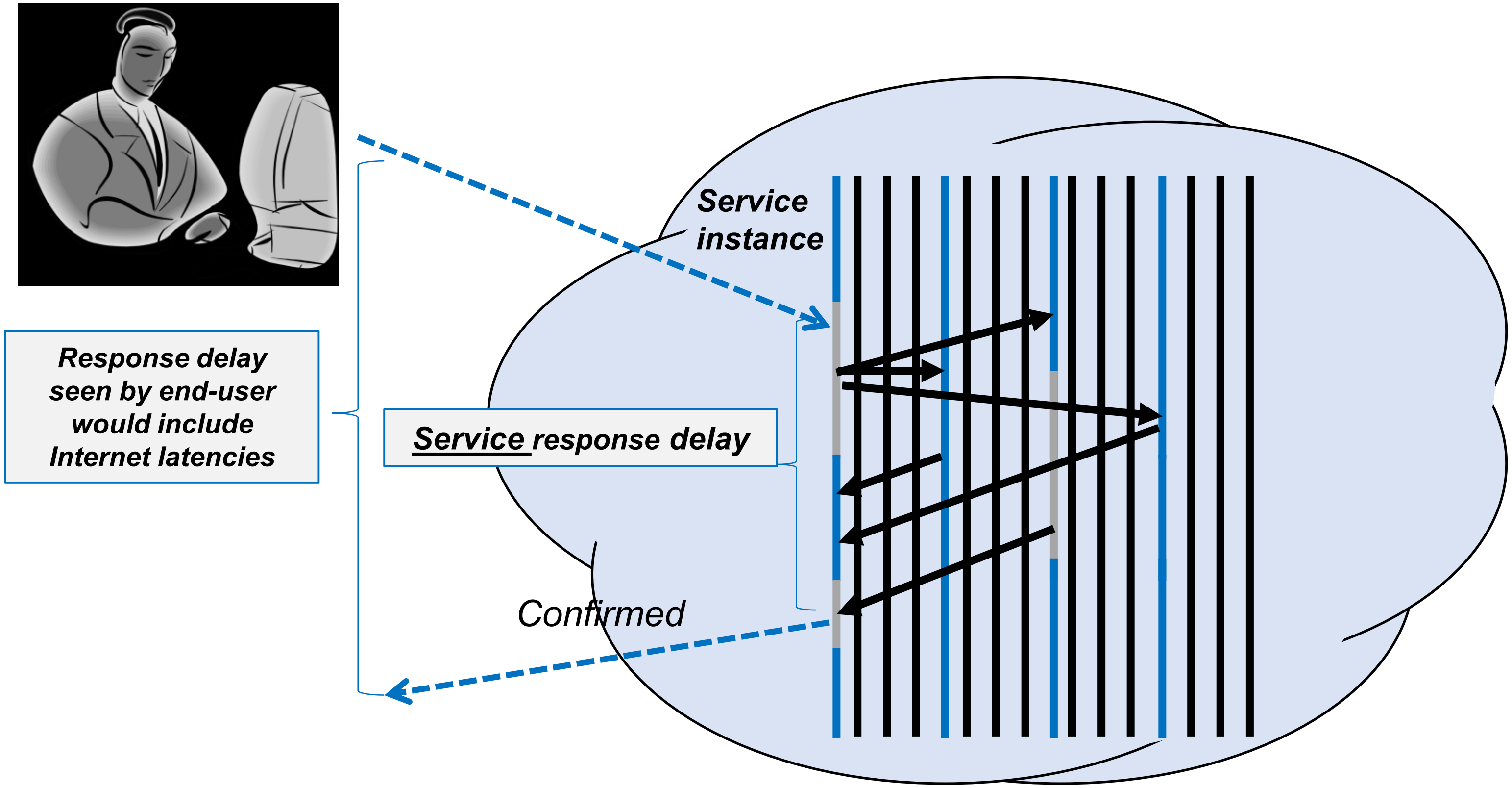
- Waiting for slow services with many updates forces to defer the confirmation and worsening the service time
- In this case the delay is due to middle subservice



Cloud Replicas and Parallelism

Fast answers can stem from replicas and parallelism

- With replication you can favor parallel execution only for **read operations**
- **That can speed up the answer**



Response delay seen by end-user would include Internet latencies

Service response delay

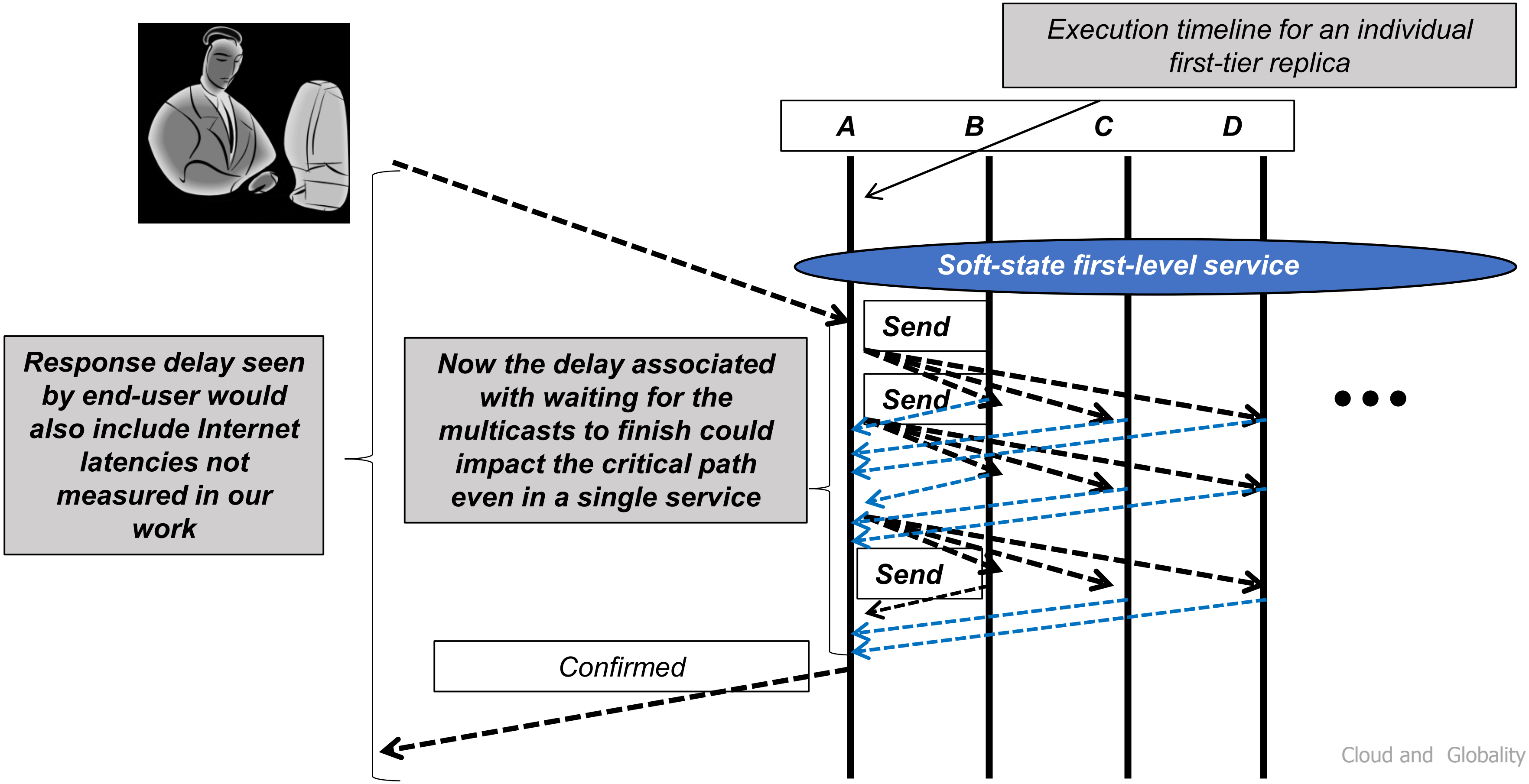
Service instance

Confirmed

Cloud Updates vs Inconsistency

If we have several copy update, we need multicast, but we may have more inconsistency

If we do not want to wait, the order in which the operations are taken on copies can become a problem



Execution timeline for an individual first-tier replica
A
B
C
D
Soft-state first-level service
Send
Send
Send
• • •
Response delay seen by end-user would also include Internet latencies not measured in our work
Now the delay associated with waiting for the multicasts to finish could impact the critical path even in a single service
Confirmed

CLOUD ASYNCHRONOUS EFFECT

If we use extensively internal replicas and parallelism to limit user expected delays, we need to go **less synchronous**, and adopt extensively an **asynchronous strategy**

Answer are given back when the **level 2 has not completed the actions** (write actions) **and those actions can fail …**

- If the replicas receive the operations **with a different schedule, their final state can be different and not consistent**
- If some replicas fail**, the results cannot be granted (specially if the leader fails)** and the **given back answer is incorrect**
- **Some agreements between different copies must be achieved at level 2 (eventually)**

All above issues contribute to **inconsistency that clashes with safety and correctness**

INCONSISTENCY IS DEVIL

We tend to be very concerned about correctness and safety
⇒ **So we feel that inconsistency is devil**

We tend to be very consistent in our small world and confined machine
But let us think to **specific CLOUD environments**:
do we really need a strict consistency any time?

- **Videos on YouTube**. Is consistency a real issue for customers?
- **Amazon counters** of “number of units available” provided real time to clients

  Customers can really feel the difference with small variations?

So, **there many cases in which you do not need a real correct answer**, but some **approximations to it** (of course, the closer the better) is more than enough

CAP THEOREM OF ERIC BREWER

Eric Brewer argued that:

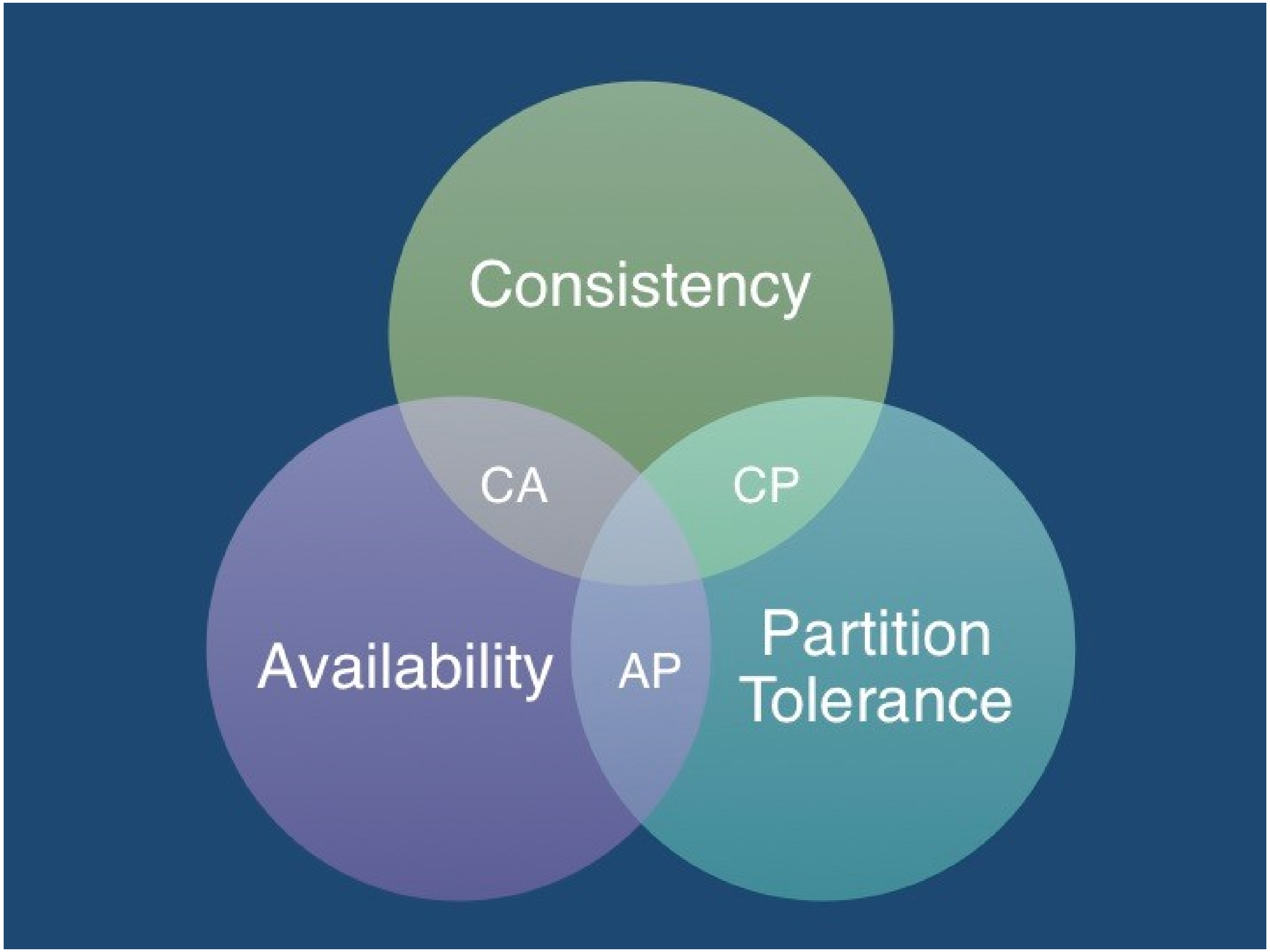
“*you can **have just two from the three properties***
(2000 keynote at ACM PODC)
Consistency**, **Availability**, and **Partition Tolerance”.

- **strong Consistency**: all clients see the **same view**, even in the presence of updates
- **high Availability**: all clients can find anytime **at least one replica** of the data, even in the presence of failures
- **Partition-tolerance**: the system properties still hold even when the **system is partitioned** and **must works on**

Brewer argued that since **availability** is paramount to grant fast answers, and **transient faults** often makes impossible to reach all copies, **caches must be used even if they are stale**

The **CAP** conclusion is to **weaken consistency for faster response (AP, neglecting and even losing C)**

CAP theorem of Eric Brewer



TWO PERSPECTIVE

Pessimist: "You know you have one problem when the crash of a computer you have never heard of stops you from getting any work done" (***Lamport***)

Optimist: A distributed system is a collection of independent computers that appears to its users as a single coherent system

Academic-like point of view:

Clean abstractions, Strong semantics, Things that are formally provable and that are smart

User-like point of view:

Systems that work (most of the time), Systems that scale well, Consistency not important per se

ACID PROPERTIES (PESSIMISTIC)

The idea of granting the **maximum of consistency** is embodied by the **ACID properties** typically considered in

- Concurrent execution of multiple transactions
- Recovery from failure

- **Atomicity**: Either all operations of the transaction are properly reflected in the database (commit) or none of them are (abort)
- **Consistency**: If the database is in a *consistent* state before starting a transaction, it must be in a *consistent* state at the end of the transaction
- **Isolation**: Effects of ongoing transactions are not visible to transactions that executed concurrently. Basically "we'll hide any concurrency"
- **Durability**: Once a transaction commits, updates can not be lost nor their effects rolled back

ACID Properties

A **“serial” ACID execution** is one where there is **at most one transaction running at a time**, and it completes via commit or abort before another one starts: **“serializability” is the “illusion” of a serial execution but with (increasingly) heavy costs**

Let us think to two extreme cases:

- **Embarrassingly easy ones**: transactions that do not conflict at all (like Facebook updates by a single owner to a page that others only read and never change – meteorological systems)
- **Conflict-prone ones**: transactions sometimes interfere and replicas could be left in conflicting states, if no attention is paid to order the updates. **Scalability for this case is terrible**

The costs of transactional **ACID model on replicated global data** can be surprisingly high in some settings

Solutions must involve ad-hoc solutions, such as sharding and coding ad-hoc transactions

CAP STRATEGY

Brewer's **CAP** theorem:

"*you cannot use transactions at large scale in the cloud*"

…or in other large dimension systems …

We saw that the **real issue is mostly in the highly scalable and elastic outer tier** ("stateless tier") close to the users and does **not impact on the second inner layer**

In reality, Cloud systems use transactions all the time, but **they do so in the "back end",** and they shield away that layer as much as possible **from users**, **to avoid overload** and **not to create bottlenecks**

BASE Motivation

eBay researchers:

- found that many eBay employees came from **transactional database** backgrounds and were used to the transactional style of “thinking”
- but the resulting applications **did not scale well** and **performed poorly on their cloud**

Goal was to guide that kind of programmers to a cloud solution that performs much better by giving new guidelines in designing internal applications:

- **BASE is the solution that reflects experience with real cloud applications** and provide a new workflow
- “Opposite” of **ACID**

BASE PROPERTIES

Basically Available: the goal is to provide fast responses. Since in data centers **partitioning faults** are very rare, they are mapped into **crash failures** by forcing **the isolated machines to reboot…**But we may **need rapid responses** even when **some replicas can not be contacted on the critical path**

Basically Available: Fast response even if some replicas are slow or crashed
Soft State Service: Runs in first tier cannot store any permanent data and restarts in a "clean" state after a crash

To maintain data, either **replicate in memory in enough copies to never lose all** in any crash (active copies in memory) or pass it to **some other service that keeps "hard state"**

- **and E?**

MORE BASE PROPRIETIES

Basically Available: Fast response even if some replicas are slow or crashed.

Soft State Service: No durable memory.

Eventual Consistency: shorten return path by send "**optimistic**" answers to the external client

- Could use **cached data** (without checking for staleness)
- Could **guess** at what the **outcome** of an update will be
- Might **skip locks**, hoping that no conflicts will happen (optimistic approach)
- Later, if eventually needed, **correct any inconsistencies** with an **offline cleanup activity**

SOME IMPLEMENTATIONS

Use **transactions**, but **removing Begin/Commit points**
- **Fragment it into "steps" that can be done in parallel**, as much as possible
- Ideally each step is associated with a single event that triggers that step, by using **delivery of a multicast**

The **transaction Leader** stores these events in a **MOM middleware** system:
- Like an email service for programs
- Events are delivered by the message queuing system
- To provide a sort of 'all-or-nothing' behavior

The idea is **Sending the reply to the user** **before finishing the operation**
Modify the end-user application to mask any **asynchronous side-effects that might be noticeable**, by "weakening" the semantics of the operation and coding the application to work properly anyhow

BASE EFFECT

Before **BASE**, the code was often **too slow and scaled poorly**, so end-users waited a long time for responses

With **BASE:**

- Code itself is **more concurrent**, hence **faster**
- **Eliminate locking, with early responses,** all make end-user experience snappy and positive
- But we do **sometimes see oddities when we look hard**

Suppose an eBay **auction running fast and furious**

Does every single bidder necessarily see every bid? And do they see them in the identical order?

Clearly, everyone needs to see the winning bid, but slightly different bidding histories should not hurt much, and that makes eBay 10x faster

The achieved speed may be worth the slight change in behavior!

ACID VS. BASE

ACID	BASE
Strong consistency for transactions highest priority	**Availability and scaling** highest priorities
Availability less important	**Weak consistency**
Pessimistic	**Optimistic**
Rigorous analysis	**Best effort**
Complex mechanisms	**Simple and fast**

ACID + BASE = CAP

What goals you might want from a large organization support system for sharing data globally

Consistency, Availability, Partition tolerance

- **Strong Consistency**: all clients see the same view, even in presence of updates;
- **High Availability**: all clients can find some replicas of the data, even in presence of failures;
- **Partition-tolerance**: the system properties hold even when the system is partitioned, and the work can go on without interruption

You can **obtain** only **two out of the three properties**

The choice of which feature to discard determines the **nature of your system**

CONSISTENCY AND AVAILABILITY

Providing transactional semantics requires all functioning nodes to be in contact with each other (and no partition is allowed).

When a **partition occurs, no work can go on and the reconnection must be awaited**.

Examples:

- Single-site and clustered databases
- Other cluster-based designs

Typical Features:

- **Two-phase commit**
- **Cache invalidation** protocols
- **Classic DB** style

AVAILABILITY AND PARTITION-TOLERANCE

If you neglect consistency, life is much better and easy…

You **can work in case of a partition and give answers**, then you will grant reconciliation afterwards**.**

Examples:

- DNS
- Web caches
- Practical distributed systems for mobile environments are choosing like that (eBay as the pioneer)

Typical Features:

- **Optimistic updating** with conflict resolution
- That is the **Internet philosophy**
- **TTLs** and **lease cache** management

SEVERAL CONSISTENCIES

Strict: updates must happen instantly everywhere.

- A read must return the result of the latest write on that data: instantaneous propagation are not so realistic

Linearizable: updates appear to happen instantaneously at some point in time

- Like "Sequential" but operations ordered by a **global clock**
- Primarily used for formal verification of concurrent programs

Sequential: all updates occur in the same order everywhere

- **Every client sees the writes in the same order**
  - Order of writes from the same client is preserved
  - Order of writes from different clients may not be preserved
- Equivalent to Atomicity + Consistency + Isolation

Eventual consistency: when all updating stops, then eventually all replicas will converge to the identical values

- **Equivalent to CAP**

EVENTUAL CONSISTENCIES

When all updating stops, then eventually all replicas will converge to the identical values

Write propagation can be implemented with two steps

All writes **eventually** propagate to all replicas

Writes, when they arrive, are written **to a log and applied in the same order at all replicas** (timestamps and “undo-ing”)

Update propagation in two phases:

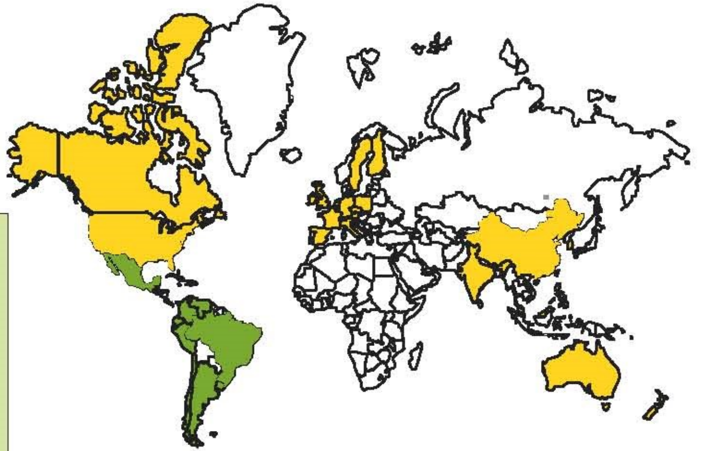
1. **Epidemic stage**: Attempt to spread an update quickly willing to tolerate incomplete coverage for reduced traffic overhead
2. **Correcting omissions**: this phase grants that all replicas that were not updated during the first stage get the update

TECHNOLOGY TOOLS CAP

Service	Guaranteed properties and introduced requirements
Memcached	No special guarantees
Google GFS	File is current if locking is used
BigTable	Shared key-value store with many consistency properties
Dynamo	Amazon 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

Challenges at Internet Scale

- eBay manages …
  - Over 276,000,000 registered users
  - Over 2 Billion photos
  - eBay users trade $2040 in goods every second -- $60 billion per year
  - eBay averages over 2 billion page views per day
  - eBay has roughly 120 million items for sale in over 50,000 categories
  - eBay site stores over 2 Petabytes of data
  - eBay Data Warehouse processes 25 Petabytes of data per day



- In a dynamic environment
  - 300+ features per quarter
  - We roll 100,000+ lines of code every two weeks
- In 39 countries, in 8 languages, 24x7x365

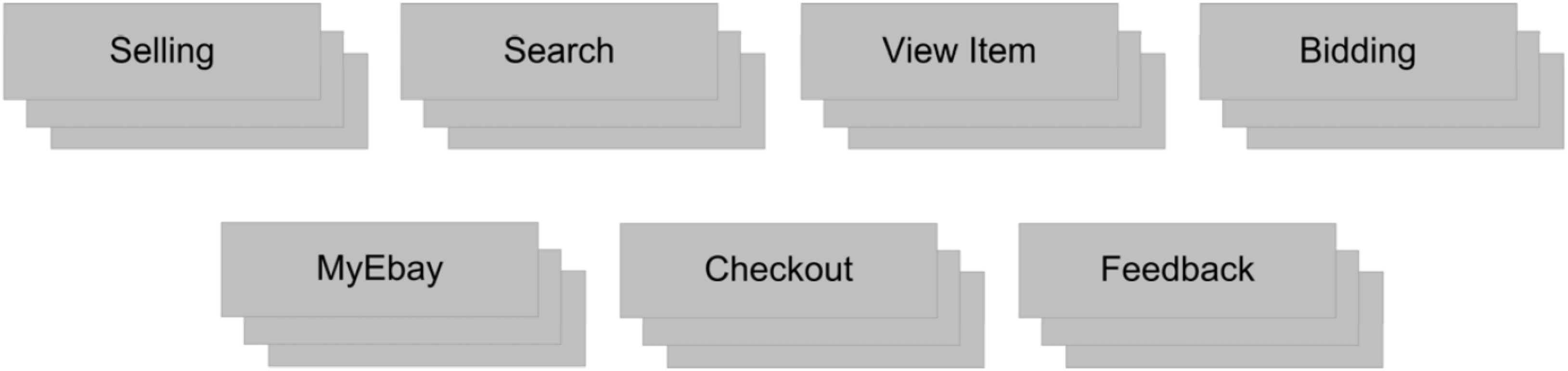
>48 Billion SQL executions/day!

1 - PARTITION EVERYTHING

Pattern: **Functional Segmentation**

Segment processing into pools, services, and stages

Segment data along **usage boundaries**



- You should split anything you can in **separated localities**
- No **large components** (to be kept consistent)

1 - Partition Everything

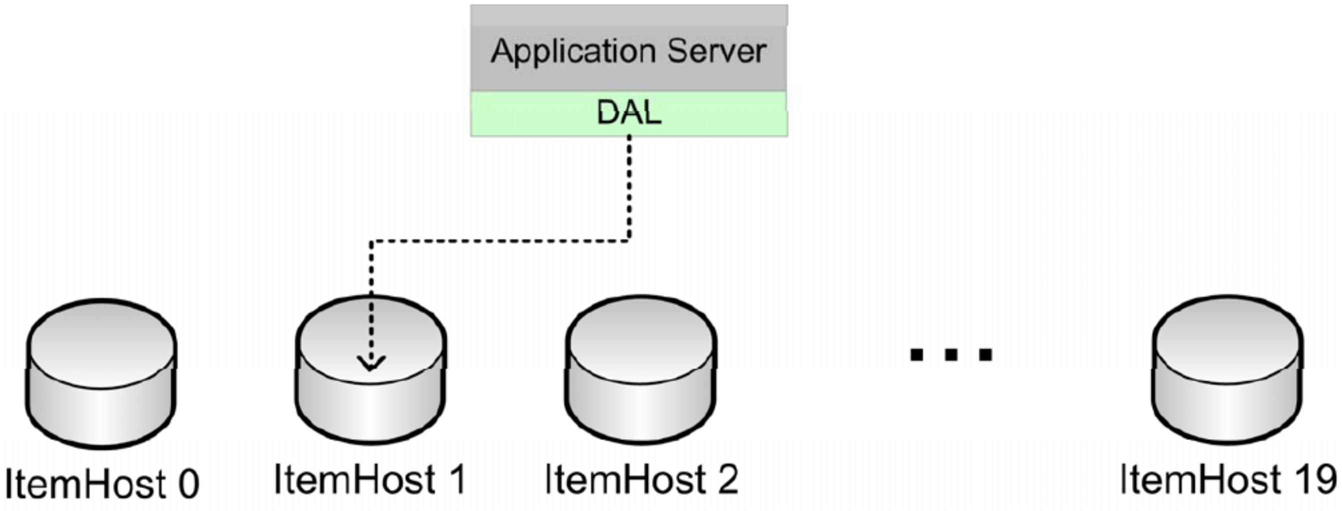
Pattern: Horizontal Split.

Load-balance processing.

- all servers are created **equal** within a pool.

Split (or "shard") **data** along **primary access path:**

- partition by range, modulo of one key, lookup, etc.
- in the data access layer



1 - Partition Everything

The principle suggests to simplify the management

Corollary: No Database Transactions

- Absolutely no client side transactions, two-phase commit, ...
- Auto-commit for vast majority of DB writes
- Consistency is not always required or possible

Corollary: No Session State

- User session flow moves through multiple application pools
- Absolutely no session state in application tier

Keep it simple (and short in time)

2 - ASYNCHRONY EVERYTHING

Prefer Asynchronous Processing

Move as much processing as possible to asynchronous flows

Where possible, integrate disparate components asynchronously

Requirements

- **Scalability**: can scale components independently
- **Availability**: can decouple availability state and retry operations
- **Latency**: can significantly improve user experience latency at cost of data/execution latency
- **Cost**: can spread peak load over time
  Asynchronous Patterns, Message Dispatch, Periodic Batch.

2 - ASYNCHRONY EVERYTHING

Pattern: **Event Queue** or **Streams decoupling**

Primary use-case **produces event transactionally** such as

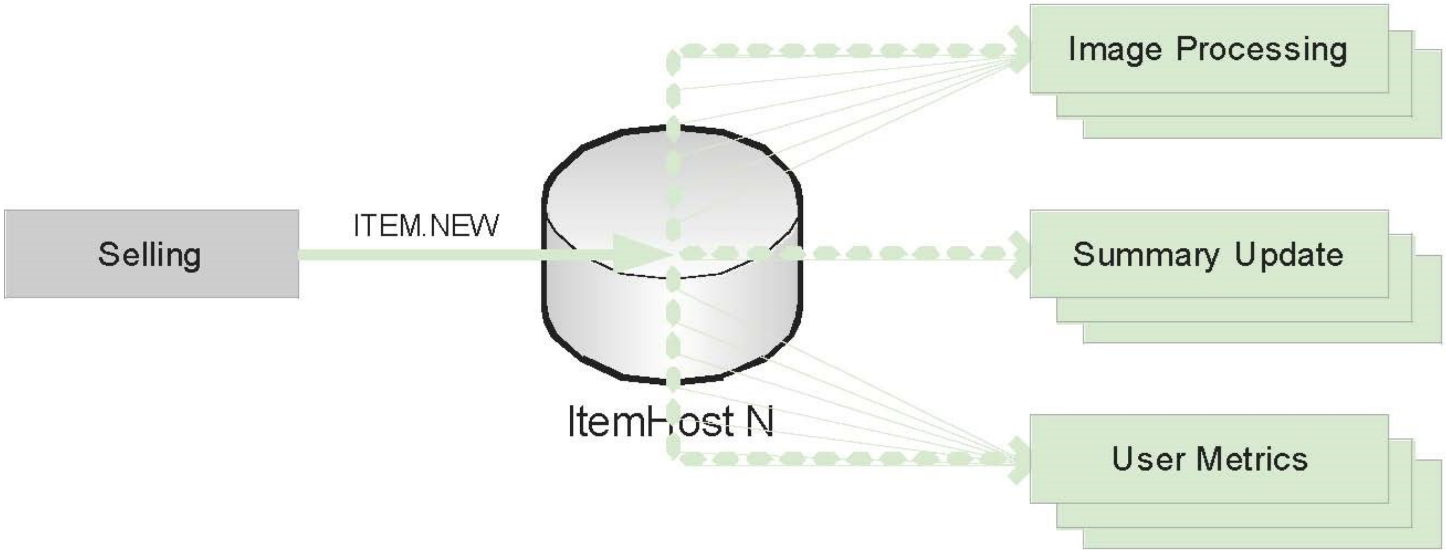
Create event (ITEM.NEW, ITEM.SOLD)

with primary insert/update

Consumers subscribe **to events**

At least once delivery, No guaranteed order

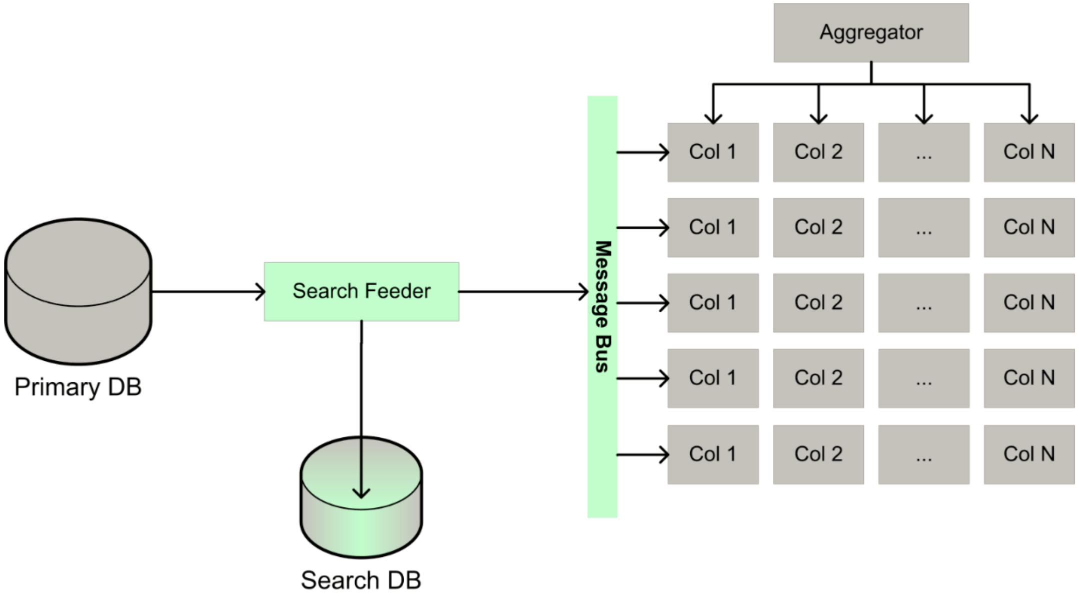
with idempotency and readback



2 - ASYNCHRONY EVERYTHING

Pattern: **Message Multicast**

- **Search Feeder** publishes item **updates**, by reading item updates from primary database, and it publishes sequenced updates via Scalable Reliable Multicast-inspired protocol
- **Nodes listen** to assigned subset of messages, by the update of **in-memory index in real time** and **request recovery** (NAK) when messages are missed



2 - ASYNCHRONY EVERYTHING

Pattern: **Periodic Batch**

Scheduled **offline batch processes**

Most appropriate for:

- Infrequent, periodic, or scheduled processing (once per day, week, month)
- Non-incremental computation (no “Full TableScan”)

Examples

- Import third-party data (catalogs, currency, etc.)
- Generate recommendations (items, products, searches, etc.)
- Process items at end of auction

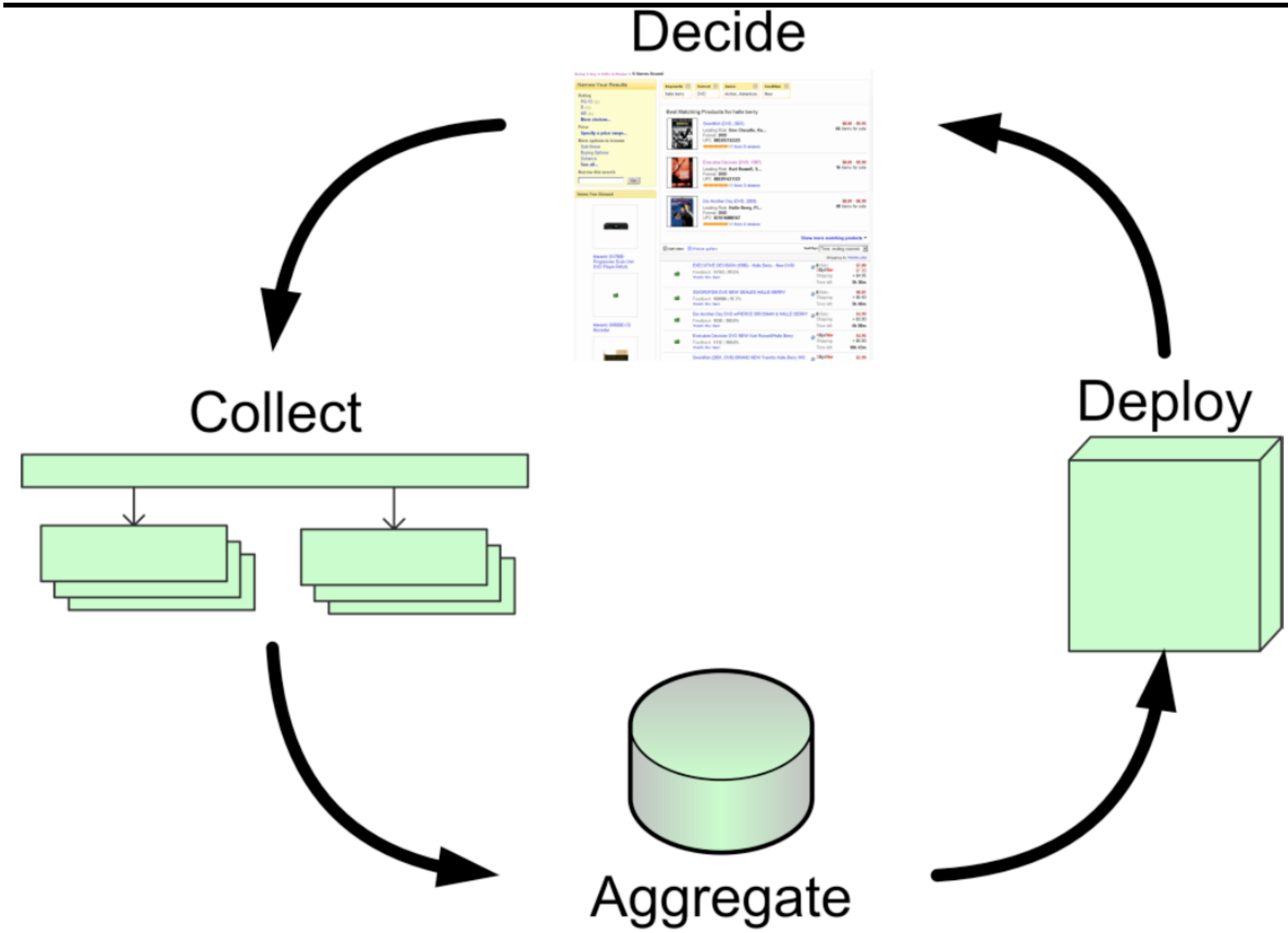
Often drives further **downstream processing** through **Message Dispatch**

3 - Automate Everything

Prefer Adaptive / Automated Systems to Manual Systems:

- **Scalability**: to scale with machines, not humans
- **Availability / Latency** to fast adapt to changing environment
- **Cost** Machines are far less expensive than humans and it is easy to learn / improve / adjust over time without manual effort
- **Functionality** Easy to consider more factors in decisions and explore solution space more thoroughly and quickly
- **Automation Patterns**
- **Adaptive Configuration**
- **Machine Learning**

3 - Automate Everything



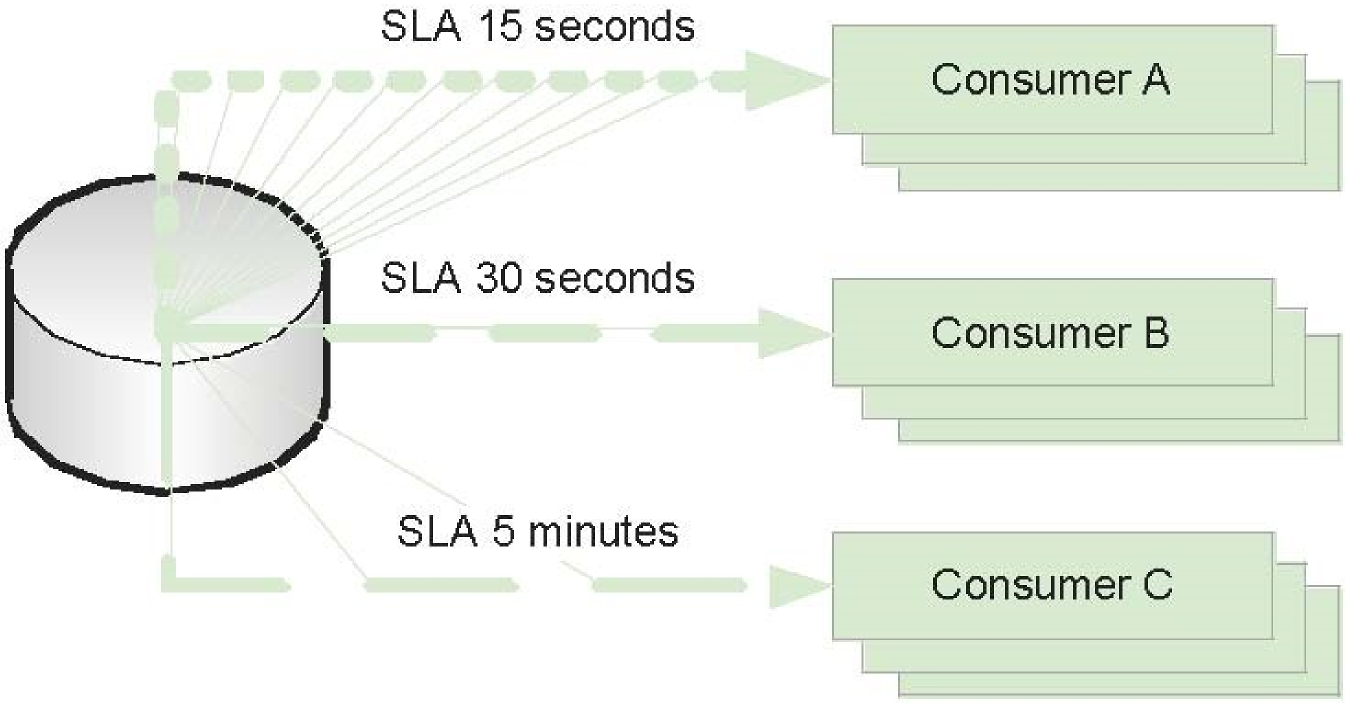
3 - Automate Everything

Pattern: **Adaptive Configuration**

define **SLA for a given logical** consumer

such as: 99% of events processed in 15 seconds

dynamically adjust configuration to meet defined SLA



3 - AUTOMATE EVERYTHING

Pattern: **Machine Learning**

Dynamically adapt search experience

- Determine best inventory and assemble optimal page for that user and context

Feedback loop enables system to learn and improve over time

- Collect user behavior
- Aggregate and analyze offline
- Deploy updated metadata
- Decide and serve appropriate experience

Perturbation and dampening

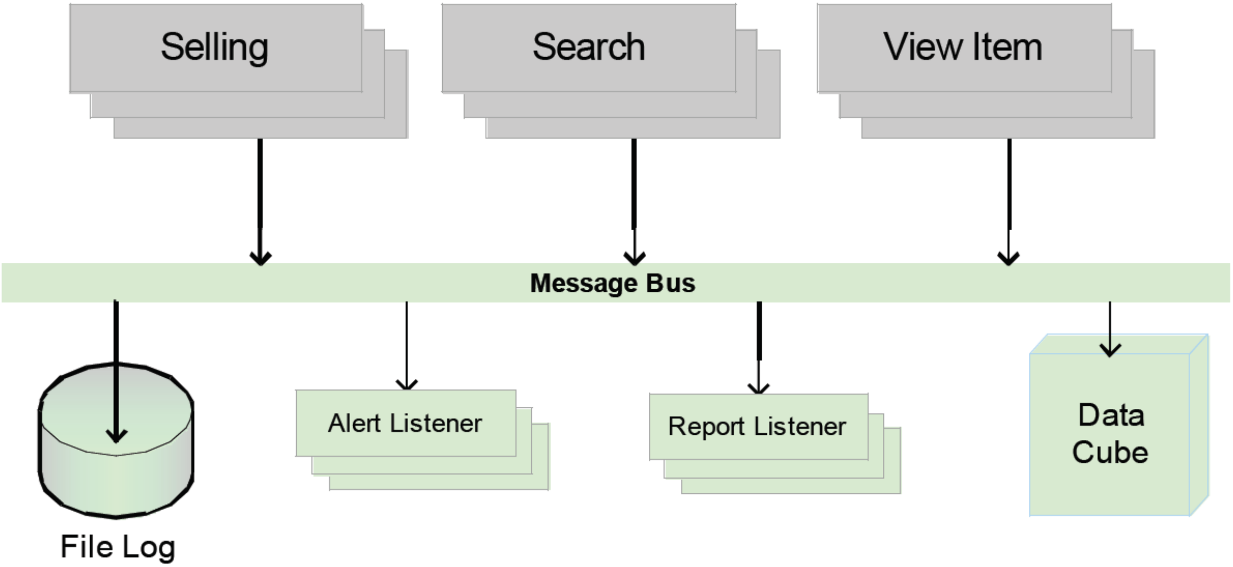
4 – EVERYTHING FAILS

Pattern: **Failure Detection**

Servers **log all requests**

- Log all application activity, database and service calls on multicast message bus
- More than **2TB** of log messages per day

Listeners **automate failure detection and notification**



4 – Everything Fails

Pattern: **Rollback**

Absolutely no changes to the site that **cannot be undone (!)**

- The system **does not take any action** in case **irreversible actions are to be taken**

Every feature has **on / off state** driven by central configuration

- Feature can be immediately turned off for operational or business reasons
- Features can be deployed “**wired-off**” to unroll dependencies

4 – Everything Fails

Pattern: **Graceful Degradation**

Application “marks down” an **unavailable or distressed resource**

- Those resources are dealt with specifically

Non-critical functionality is removed or ignored

- All unneeded functions are neither considered nor generally supported

Critical functionality is retried or deferred

- All critical points are dealt with specifically and in case of success no problem
- in case of a failure, retried until completed

5 – Embrace Inconsistency

Choose Appropriate Consistency Guarantees

- According with Brewer's CAP Theorem prefer **eventual consistency** to **immediate consistency**
- To guarantee **availability** and **partition-tolerance**, we trade off **immediate consistency**

Avoid Distributed Transactions

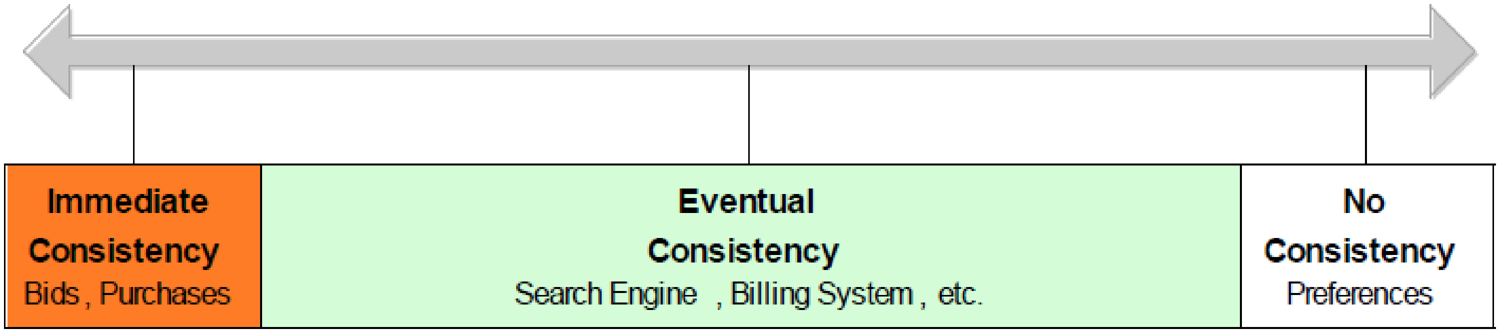
- eBay does absolutely **no distributed transactions** – **no two-phase commit**
- minimize inconsistency through **state machines and careful ordering of operations**
- **eventual consistency** through **asynchronous event or reconciliation batch**

EBAY PRINCIPLES

Randy Shoup described Five Commandments with many details along the lines of **high Scalability**, **high Availability**, **low Latency**, **high Manageability**, **low Cost**

And all those requirements can be met by following the first four commandment that can push toward many design lines details, about shard, asynchronicity, adaptive configuration, failure detection and graceful degradation

The final point is **releasing consistency**, depending on application areas, **looking at consistency as a spectrum and not a specific position**



Immediate Consistency
Bids , Purchases

Eventual Consistency
Search Engine , Billing System , etc.

No Consistency
Preferences

MOVE ALONG THAT LINE

Randy Shoup described **eBay Five Commandments for their system organization**

Thou shalt…

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