



University of Bologna

Dipartimento di Informatica –
Scienza e Ingegneria (DISI)

Engineering Bologna Campus

Class of **Computer Networks M** or
**Infrastructures for Cloud
Computing and Big Data**

Dependability and new replication strategies

Antonio Corradi

Academic year 2016/2017

Dependability 1

Replication to tolerate faults

Models and some definitions related to faults

failure any behavior not conforming with the requirements

error any problem that can generate an incorrect behavior or a *failure*

fault set of events in a system that can cause *errors*

An application can fail and it can cause a wrong update on a database

fault is the concrete causing occurrence (several processes entering at the same time), **error** is the sequence of events (mutual exclusion has not been enforced)

these can generate the visible effect of **failures (to be prevented)**

fault ⇒ **transient, intermittent, permanent ones**

Bohrbug repeatable, neat failures, and often easy to be corrected

Eisenbug less repeatable, hard to be understood failures, hard to correct

Eisenbug often tied to specific runs and events, so not easy to be corrected

Dependability 2

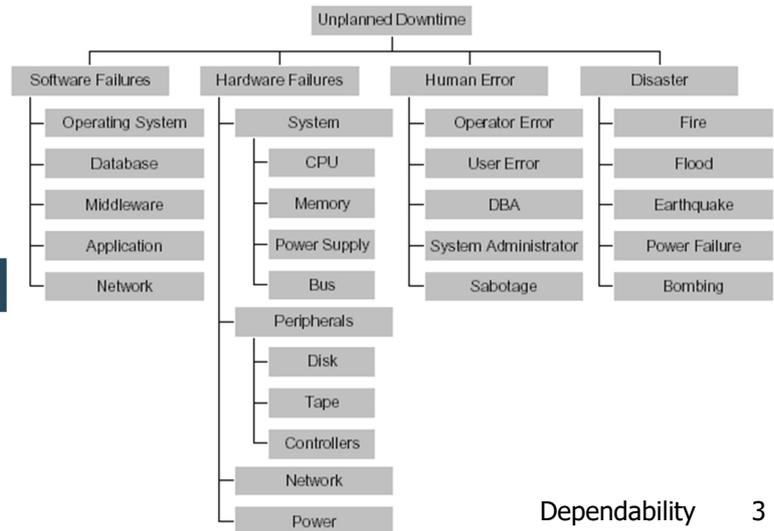
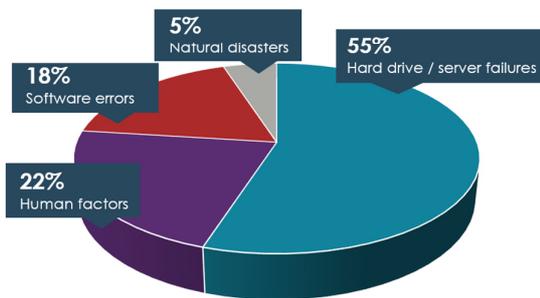
SERVICE UNAVAILABILITY

Any system may become unavailable for some time, for several reasons, so to recover and make it correct again

Causes of unavailability can stem from many different reasons, either planned ones or not planned

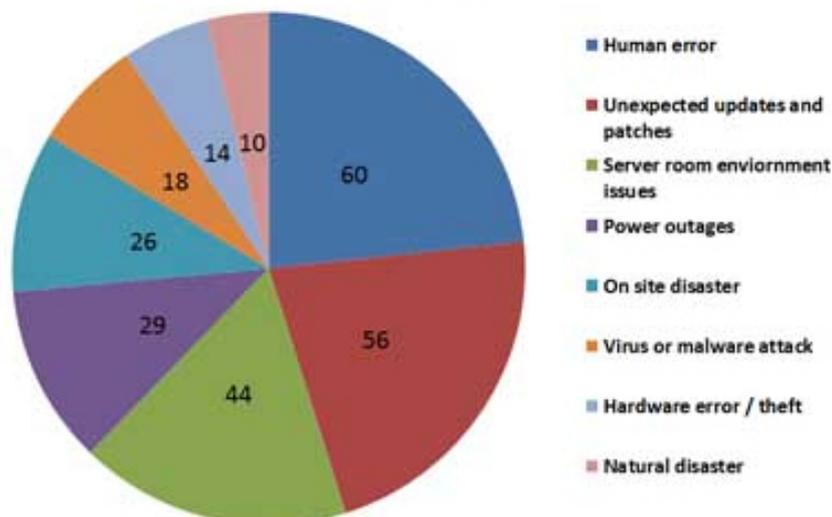
We need phases of fault/error **IDENTIFICATION** and **RECOVERY** to go back to normal operations and requirement conformance

Common Causes of Downtime



DOWNTIME CAUSES

What is the most common cause of system downtime?

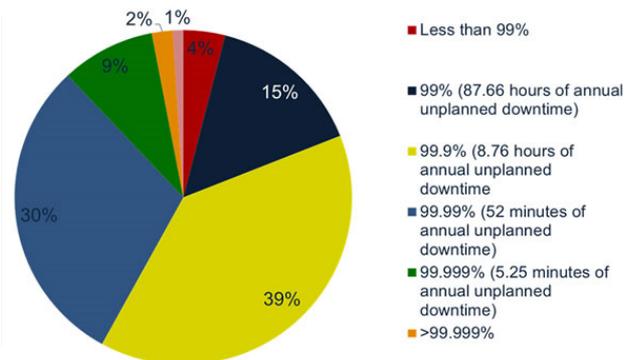


SERVICE UNAVAILABILITY INDICATORS

If a system crashes with a specified probability, at those times we experience unavailability periods (**downtime**) that may be very different and must be measured

Often we use the **number of 9s** to measure availability
 That indicator expresses not only the *frequency of crashes* and the *percentage of uptime*, but also the *capacity of fast recovery*, because the **uptime depends not only from fatale failure occurrences but also from the capacity of recovering**
 The indicators averaged over one year time

Uptime (%)	Downtime (%)	Downtime (year)
98%	2%	7.3 days
99%	1%	3.65 days
99.8%	0.2%	17h,30'
99.9%	0.1%	8h, 45'
99.99%	0.01%	52,5'
99.999%	0.001%	5.25'
99.9999%	0.0001%	31.5"



FAILURE COSTS

Again any area has **downtime costs**, very different because of the different impact on the society or on the customers, due to the importance and the interests in the service

Of course a true and precise evaluation is very difficult

Business Consequences of Outages

Industrial Area	Loss/h
Financial (broker)	\$ 6.5M
Financial (credit)	\$ 2.6M
Manufacturing	\$ 780K
Retail	\$ 600K
Avionic	\$ 90K
Media	\$ 69K



More Definitions

DEPENDABILITY **FAULT TOLERANCE (FT)**

The customer has a complete confidence in the system
Both in the sense of hardware, software, and in general any aspect
Complete confidence in any design aspect

RELIABILITY **(reliance on the system services)**

The system must provide **correct answers**
(the stress is on correct responses)
A disk can save any result ⇒ but cannot grant a fast response time

AVAILABILITY **(continuity of services)**

The system must provide **correct answers in a limited time**
(the stress is on correct response timing)
Replication with active copies and service always available

RECOVERABILITY **(recovery via state persistency)...**

Consistency, Safety, Security, Privacy, ...

Fault Identification & Recovery in C/S

C/S play a reciprocal role in control & **identification**

the client and the server control each other
the client waits for the answer from the server synchronously
the server waits for the answer delivery verifying it
messages have timeout and are resent

Fault identification and recovery strategies

Faults that can be tolerated without causing failure
(at any time, all together and during the recovery protocol)

Number of repetitions ⇒ possible fault number

The design can be vary hard and intricate →

Fault assumptions simplify the complex duty

SINGLE FAULT ASSUMPTION

Fault assumptions simplify the management and system design

single fault assumption (one fault at a time)

The **identification** and **recovery** must be less than
(TTR Time To Repair and MTTR Mean TTR)

the interval between two faults

(TBF Time Between Failure and MTBF Mean TBF)

In other words, during recovery we assume that no fault occurs and the system is safe

With **2** copies, we can **identify one fault** (identification *via some invariant property*), and, even if fault caused the block, we can continue with the residual correct copy (in a degraded service) **with single fault assumption**

With **3** copies, we can **tolerate one fault**, and **two can be identified**

In general terms,

with 3t copies, we can tolerate **t** faults for a replicated resource
(without any fault assumption)

FAULT ASSUMPTIONS FOR COMMUNICATING PROCESSORS

We can work with computing resources, with executing and communicating processors

FAIL-STOP

one processor fails by stopping (halt) and all other processors can verify its failure state

FAIL-SAFE (CRASH or HALT assumption)

one processor fails by stopping (halt) and all other processors **cannot** verify its failure state

BYZANTINE FAILURES

one processor can fail, by exhibiting any kind of behavior, with passive and active malicious actions (see byzantine *generals and their baroque strategies*)

DISTRIBUTED SYSTEMS ASSUMPTIONS

More advanced fault assumptions

SEND & RECEIVE OMISSION

one processor fails by receiving/sending only some of the messages it should have worked on correctly

GENERAL OMISSION

one processor fails by receiving/sending only some of the messages it should have worked on correctly or by halting

NETWORK FAILURE

the whole interconnection network does not grant correct behavior

NETWORK PARTITION

the whole interconnection network does not work by partitioning the systems in two parts that cannot communicate with each other

Replication as a strategy to build dependable components

HIGH LEVEL GOALS

Availability and Reliability measured in terms of

MTBF Mean Time Between Failures system availability

MTTR Mean Time To Repair system unavailability

Availability $A = \text{MTBF} / (\text{MTBF} + \text{MTTR})$

It defines the percentage of **correct services** in time (number of 9s)

It can also be different for read and write operations

If we consider more copies, the read can be answered also if only one copy is available, and others ones are not (action that does not modify)

Reliability probability of an available service depending on time and based on a period of Δt

$R(\Delta t)$ = reliable over time Δt

$R(0) = A$, as a general limit

RELATED PROPRIETIES

Formal properties

Correctness - Safety guarantees that there are no **problems**
all invariants are always met

Vitality - Liveness achieving goals with **success**
the goal is completely reached

A system without safety & liveness does give any guarantee for any specific fault (no tolerance)

A system with safety e liveness can tolerate occurring faults

A system with **safety** without **liveness** operates **always correctly and can give results, without guarantee of respecting timing constraints**

A system without **safety** with **liveness** always provides **a result in the required time, even if the results maybe incorrect** (e.g., an exception)

In any case, to grant any of those the solutions should consider replication either in time or space

FAULT-TOLERANCE ARCHITECTURES

Use of replicated components that introduce added costs and require new **execution models**

Hw replication, but replication also propagates at any level

Differentiated execution: several copies either all active or not, over the same service, or working on different operations

- **One component only** executes and produces the result, all the others are there as **backups**
- **All components are equal and play the same role**, by executing different services at the same time and give out different answers (max throughput)
- **All components are equal in role and execute the same operation** to produce a **coordinated unique result** (maximum guarantee of correctness: algorithm diversity)

Those architectures are typically metalevel organizations, because they introduce parts that control the system behavior and manage replication

STABLE MEMORY

Stable Memory uses replication strategies (persistence **on disk**) to grant not to lose any information

Limiting fault assumption: *we consider a support system in which there is a low and negligible probability of multiple faults over related memory components (single fault over connected blocks of memory)*

In general, the fault probability during a possible recovery must be minimal, to mimic the **single fault assumption**

Memory with correct blocks

any error is converted in an **omission** (a control code is associated to the block and the block is considered correct or faulty, in a clear way)

Blocks are organized in **two different** copies over different disks, with a really low *probability of simultaneous **fault (or conjunct fault)***: the two copies contain the same information

Replication in degree of two

STABLE MEMORY - SUPPORT PROTOCOLS

Any operation (either **read** or **write**) operates on both copies: if one is incorrect, a recovery protocol starts

Any **action** proceeds starting from one of the copies and then to the other one

Any **action** from an **incorrect block** is considered an omission fault and starts a recovery protocol

The **recovery** protocol has the goal of recovering both copies to a safe state, even by working for a long time.... repeating actions

*if both copies are **equal and consistent**, no action*

*If only **one** of the copies is **correct**, the protocol copies the correct value over the wrong copy*

if both copies are correct but inconsistent, the consistency is established (one content is enforced)

(if copies have a time/version indicator it is used to choose the correct)

High cost of implementation, especially in terms of timing
(how to limit the recovery time?)

TANDEM

Special-purpose system with online data (not disk)

TANDEM (bought and adopted by Compaq e HP)

replication via two copies of any system component (two processors, two buses, two disks, ...) and the system works in a perfectly synchronous approach

The goal: **fail-safe** system dependable with single fault assumption

*Any error is identified via component replication and the **double** approach can tolerate it*

The stable memory approach is implemented via the access to the double bus to a doubled disk with double data replicated

The system cost is high and makes it special purpose (banks)

Replicated copies can push to two strategies

to make the actions twice, in any **component** or

to make actions only **once** and use the other copy as a back up

Tandem has a high cost, both in terms of resources and timing

Dependability 17

Redundant Array of Inexpensive Disks

General-purpose organization of disks with a replication goal but low-cost intention

well spread and used market solution

RAID Redundant Array of Inexpensive Disk

A set of **low cost** disks coordinated toward common actions with different goals in shared common actions to achieve different standard objectives

Commercial low cost off the shelf systems

The initial goal of **RAID** was to offer **low response time**, via **data striping**, so that a content is **split among different disks** to be read / written in parallel

Then **some standards** extended to consider **data replication**

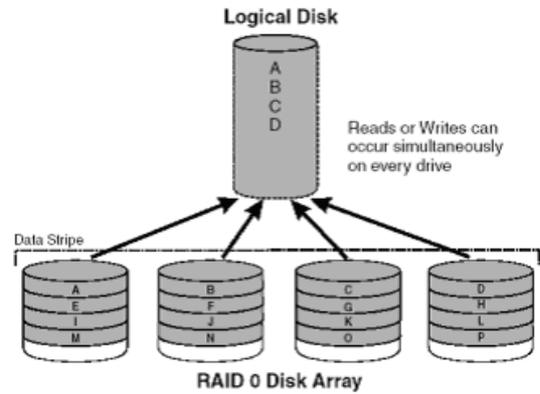
Some classes consider different organizations for different standard goal

Dependability 18

RAID

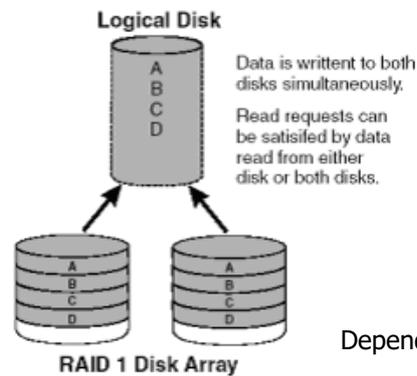
Raid 0: simple striping

parallel I/O but no redundancy, suitable for I/O intensive applications but worse **MTBF**



Raid 1: mirroring – maximum redundancy

for high availability even if higher cost
good performances in reading and less in writing

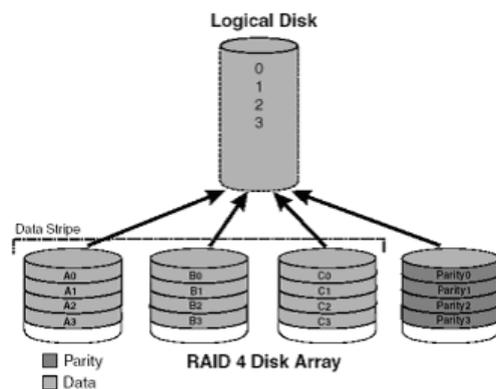


Dependability 19

RAID

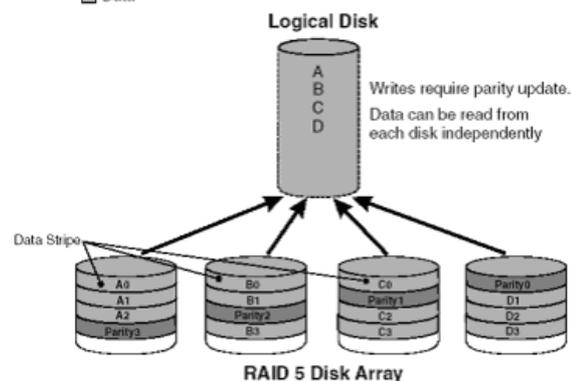
Raid 3 & 4: striping with dedicated parity disk

High speed to support operations on large contents (images)
one I/O operation at a time, for the contention on the parity disk

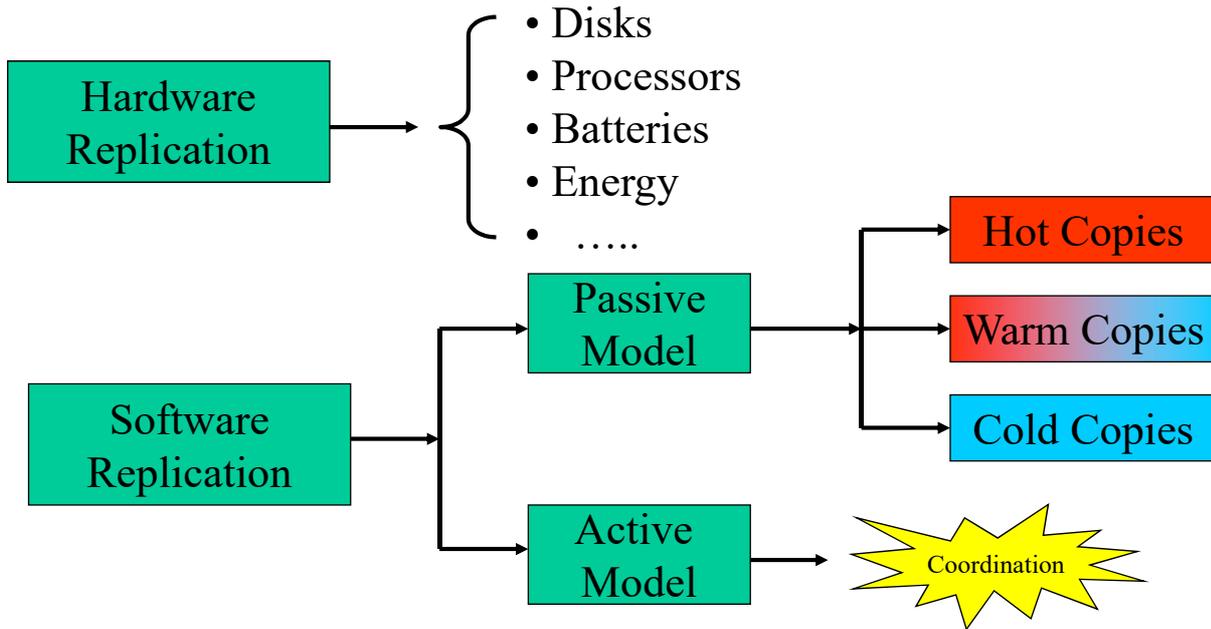


Raid 5: striping without dedicated parity disk

the distributed parity check achieves good speed in case of many readings for small contents and
good writing operations for large contents



REPLICATION FORMS



Hot copies, continuous updating

Cold copies, no update actions

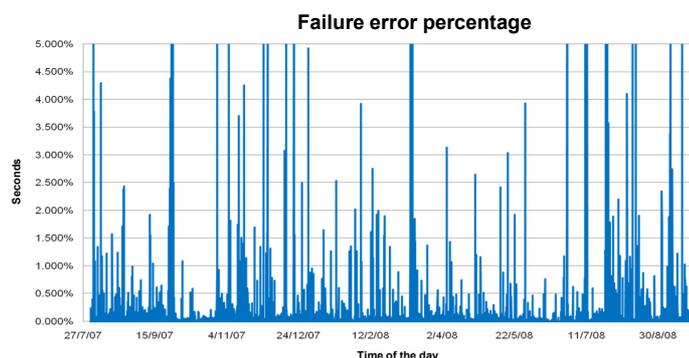
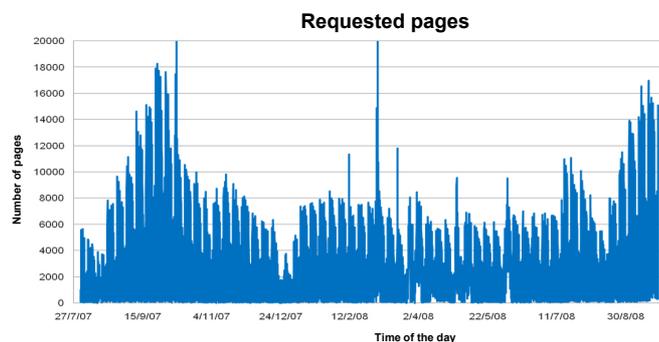
Warm copies, some update actions, but not continuous

A SMALL SCALE EXAMPLE

Alma ICT has a small problem in answering many requests in a short time for a specific Web Service

A better solution than a **simple server** has to be devised to grant **limited answer times** with **no errors**

and some fault tolerance to single fault occurrence



ALMA WEBSERVICE ARCHITECTURE

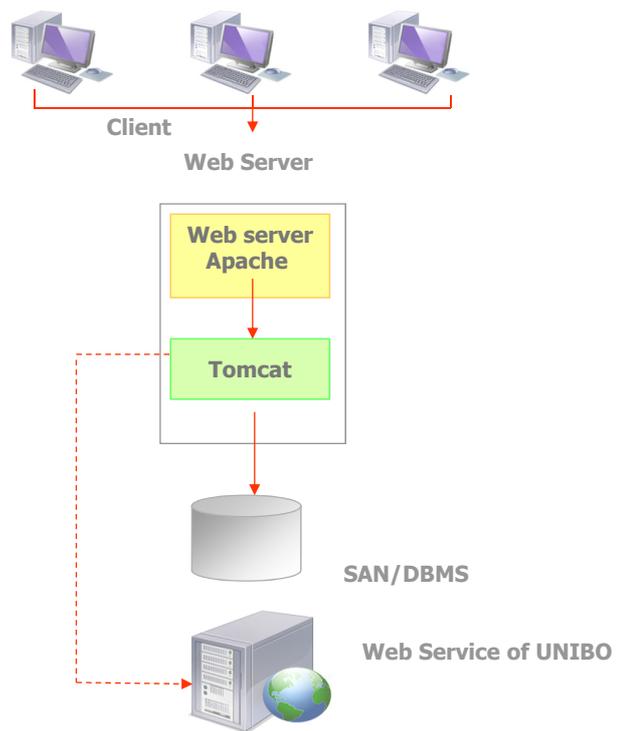
Devise a minimal cost solution

Users are interested in getting **Web server answers**

after invoking a Web service that interacts asking to a back end database

Requests arrive both from **final portal users** and also from external **programs** and other **internal UNIBO applications**

The **correct answers** are very crucial



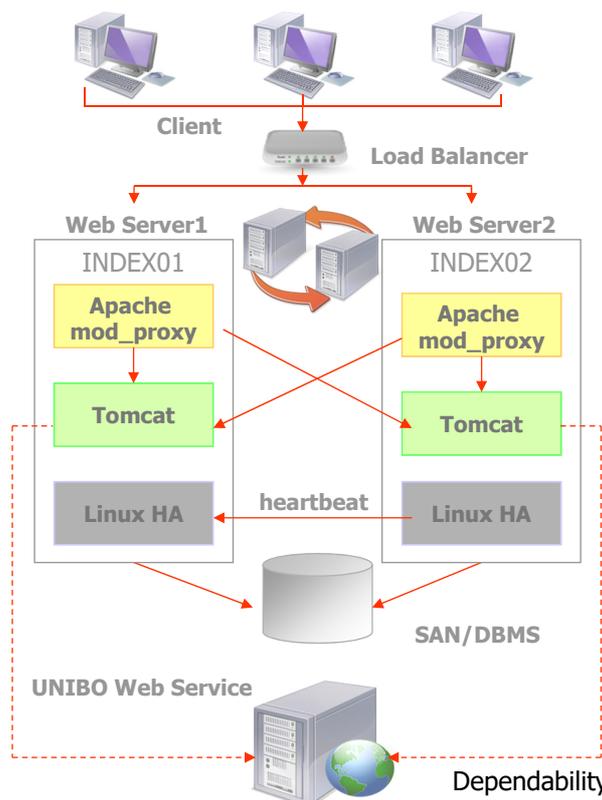
Dependability 23

PROTOTYPE ARCHITECTURE

- Load balancing via a hardware balancer as a front end of the **two main servers**

- **INDEX01 & INDEX02: two Web Server in a cluster**
 - two Tomcat instances managed by an Apache proxy

Reliability granted by a module of High Availability Linux master-slave with a heartbeat



Dependability 24

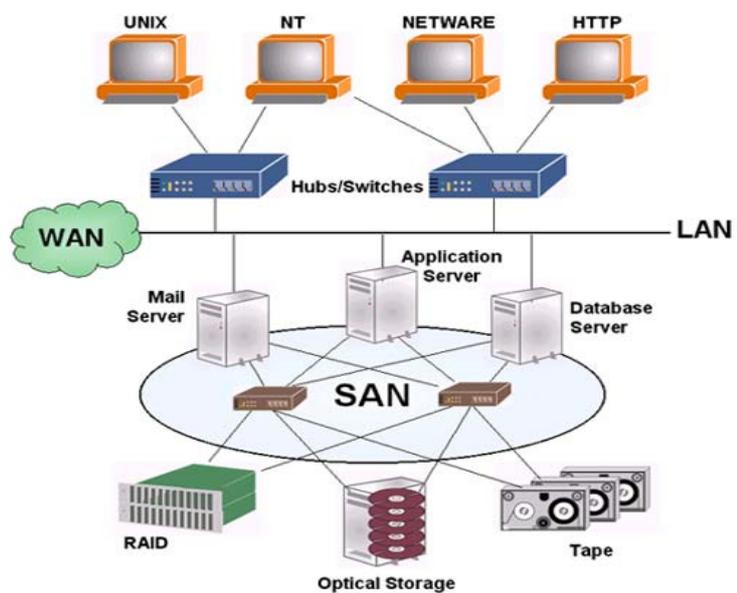
STORAGE AREA NETWORK (SAN)

A Storage Area Network is a **set of interconnected resources with several QoS** to grant the storage service with the best suitability for different users

Users can employ SAN to get the **storage resource** they need **without any interference** and ideally **without any capacity limit** and with **minimal delay**

In Cloud, the SAN can offer **Storage as-a-Service**

Storage Area Networks



Source: allSAN Report 2001 Copyright © 2000 allSAN.com Inc. allSAN.com

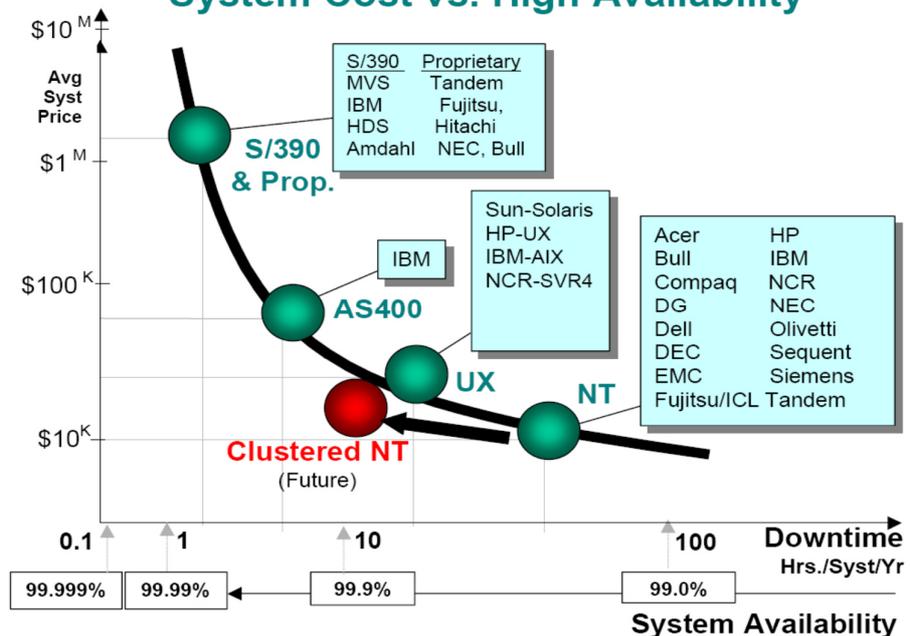
HIGH AVAILABILITY (HA)

High availability costs tend to decrease and to get better service

Low cost solutions are more and more common with a **better QoS** and **better dependability**

Solutions are more and more **off-the-shelf**

System Cost vs. High Availability



HIGH AVAILABILITY CLUSTERS

Cluster have different motivations

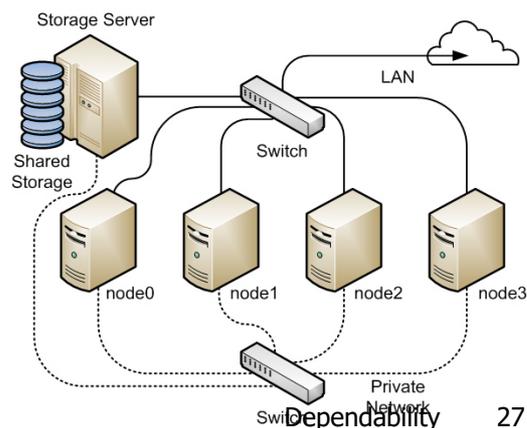
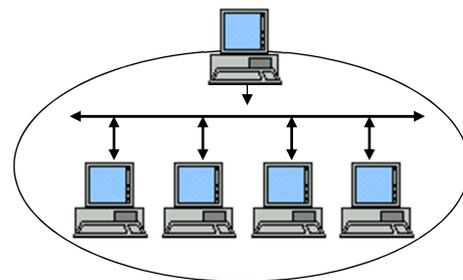
- **high availability**
- **high performance**
- **load balancing**

A cluster for **high availability**

Consists of a **set of independent nodes** that cooperate to provide a dependable **service, always on 24/7**

The cluster are a good off-the-shelf solution for **high availability**:

- **robust and reliable**
- **cost-effective** (easy to buy off-the-shelf hardware and support)
- **typically one Front-end**



27

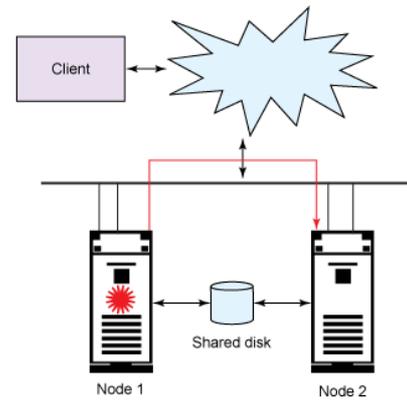
CLUSTER SUPPORT OPERATIONS

The cluster support must provide:

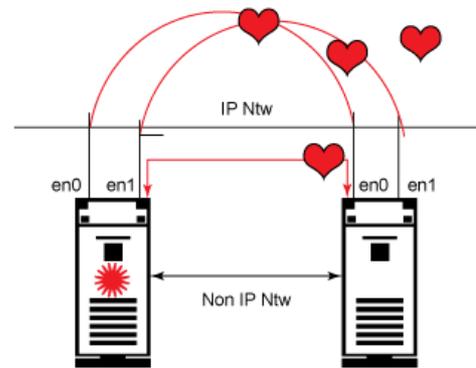
- Service **monitoring**
To dynamically ascertain the current QoS (*final and perceived*)
- **Failover (service migration)**
the failover is a hot migration of a service immediately after the crash, whichever the cause. The failover must take place very fast to limit service unavailability
Typically should automatic, fast, and transparent (the sooner the better)
- **Heartbeat (node state monitoring)**
The heartbeat is the protocol to check node state to monitor and ascertain any copy failure
Exchange of are-you-alive messages with low intrusion
Some cluster can also work in case of partitioning and allows to go on and support reconciliation when reconnected

CLUSTER: FAILOVER & HEARTBIT

In case of failover, the data must be available to the new node of the cluster via **shared component** over the cluster



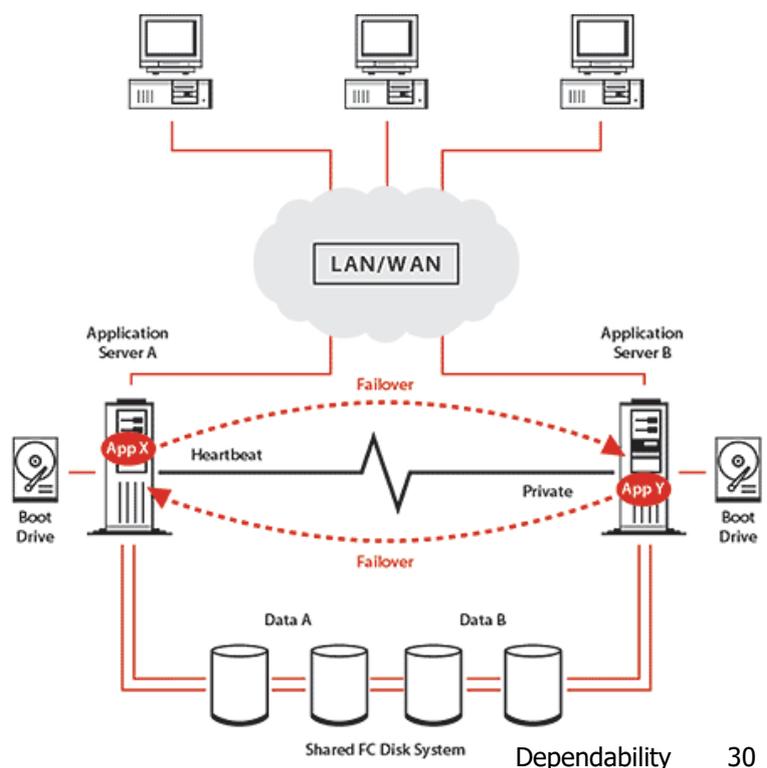
The detection of problem is via a **lightweight heartbeat protocol**



Red Hat CLUSTER

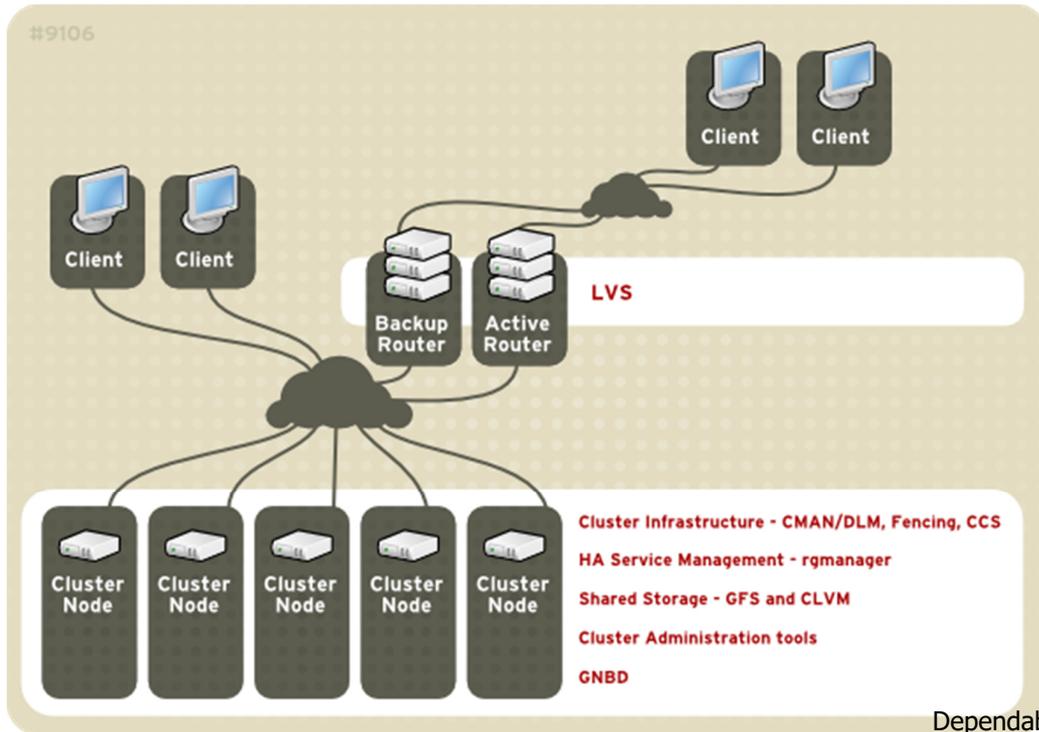
Red Hat Cluster suite
(open source)

The figure has a replication degree of two
It comprises also some shared disks to share data



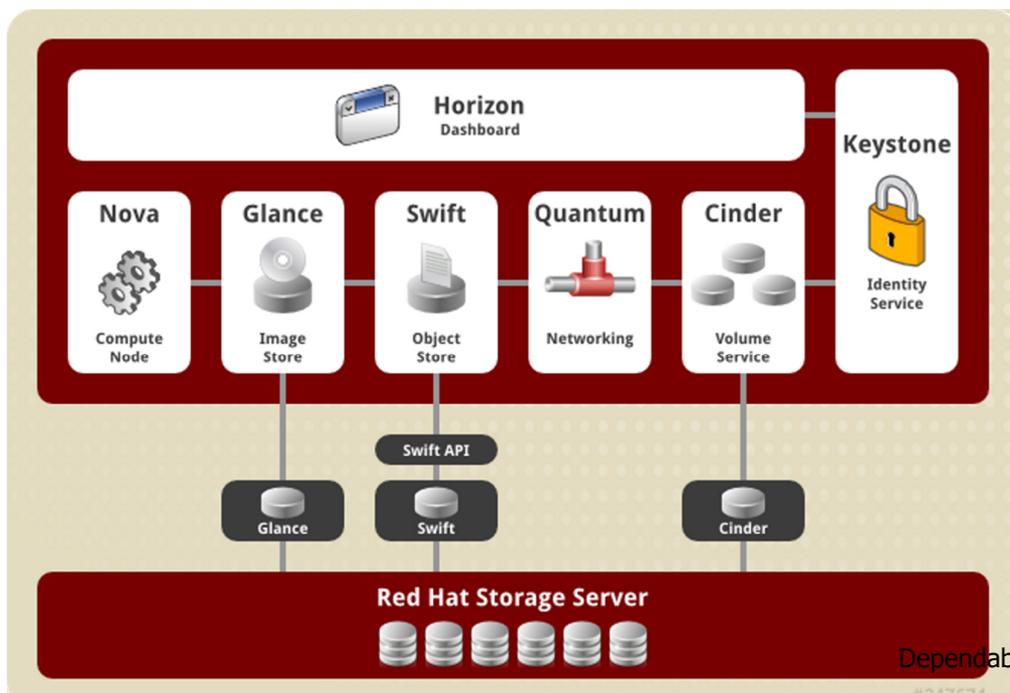
Red Hat CLUSTER

Red Hat Cluster suite evolved a lot and is off-the-shelf



Red Hat CLUSTER & ...

Red Hat Cluster can coexist with most widespread architectures... Here Openstack



FAULT-TOLERANT SUPPORT

FAULT TOLERANCE requires support, resources, protocols
protocols are expensive in term of required resources
complexity and length of the algorithms
implementation of the algorithm (and their correctness)

The **(hw & sw) support to grant dependability** is a challenge to be answered

There is no unique strategy for always accepted solutions because it is strictly interconnected with system requirements and dependability is a non functional property with many facets

In general terms, the recovery protocol must be more reliable than the application itself (*it is a problem how to grant it*)

Special-purpose systems ⇒
ad hoc resources even with better QoS

General-purpose systems ⇒
fault tolerance support insists on user resources

Dependability 33

FAULT-TOLERANT ARCHITECTURES

Minimal intrusion principle applies to any solution

To limit **the cost of the dependability support, by organizing the resource engagement (overhead)** at any support and system level

The minimal intrusion principle is an engineering one, so it should be considered in any design of systems, to answer with the requested SLA any need

Special-purpose systems ⇒

These systems can achieve dependability via an added ad-hoc architecture completely separated from the application one
Costs are high and the design is complex (formal proofs?)

General-purpose systems ⇒

User resources are the only one available. The fault tolerance support must economize on its design so not to get too much from the resources for the application levels

Dependability 34

HIGH REPLICATION COSTS

Dependability costs are generally high in the two senses and dimensions

space in terms of required **resource** available (multiple copies)

time in terms of **time, answer and service timing**

Often the fault assumptions can make the system more or less complex and viable the cost of solutions

Cost may depend on many different factors

Memory and persistency costs

Communication overhead

Implementation complexity

what to replicate, how many copies, where to keep them, how to coordinate? etc.

The general trend is in the sense of optimizing protocols, supports, infrastructures

Dependability 35

RESOURCE MANAGEMENT

We can consider **replicated resources in distributed systems** with an obvious **need of coordination of them toward a common goal (also software fault-tolerance)**

Replicated resources

Multiple resource copies on different nodes with several replication degrees

Partitioned resources

Multiple resource copies on different nodes (without any replication degree) to work independently

Redundancy can organize architectures to get a better QoS

replication of processes and data

Dependability 36

AN ABSTRACT UNIQUE RESOURCE

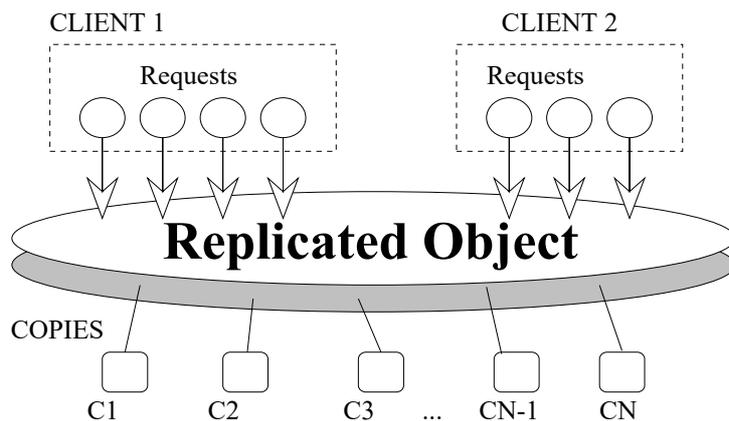
Replication degree and FT models

Number of copies of the entity to replicate

The greater the **number of copies**, the greater the **redundancy**

The better the **reliability & availability**

The greater the cost and the overhead



Two extreme models of FT architectures

- **One only** executes (master-slave)
- **All** execute (copies are active and peer)

With variations

Dependability

37

REPLICATION ARCHITECTURES

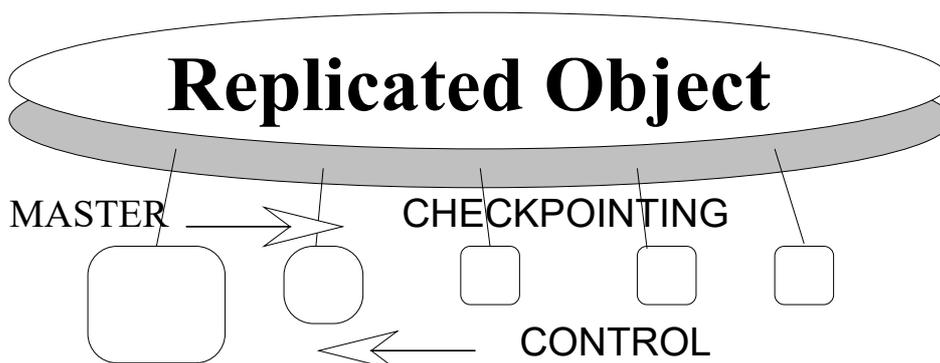
Passive model (also called *Master-Slave*)

Only one copy executes, the others are back-ups

This is the first replication model well spread in industrial plants

*The **master** is externally visible and manages the whole resource*

The slaves must control the master for errors and faults



Dependability

38

ACTIVE REPLICATION

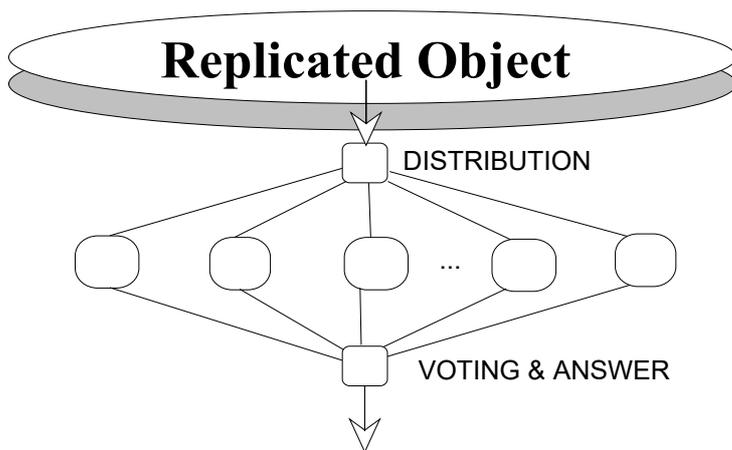
Active Model

All copies execute all operations in a more or less synchronous way and with some forms of coordination among copies

In **TMR** (Triple Modular Redundancy) **three** copies

We can tolerate on faults and can identify up to two faults

In software FT different copies can use different algorithms toward the goal



Dependability 39

PASSIVE REPLICATION MODEL

The two extreme FT models are

Master Slave (passive model)

Active Copies (active model)

The passive model or master/slave or primary/backup

Has one **active process only (the master or primary) actively executes over data**, the other copies (passive or back-ups) become operating only in case of failure of the master

only one copy is fresh and updated, the other can also be obsolete in state and not updated (cold or hot copies)

This mode can produce a possible conflict between the state of the **master** and the state of the **slaves**

In case of a failure and cold copies, one must start repeating from the previous state, to produce the updated state

Dependability 40

CHECKPOINT - SLAVE UPDATE

In general, the master updates the slave states via **checkpointing** ⇒ *the updating action also made in a chain: the master updates the first slave that updates the second, ...*

The management **policies** can distinguish *the required actions to grant a correct response*

*update of the **primary** copy (first slave)
from successive actions (less crucial)*

*update of the **secondary** copies (other slaves)*

The strategy of those can achieve different policies and different state updating costs and quality

the client gets the answer with **less delay** if the master answers before the state has been **updated in all copies** (but only a part) or even in **no slave copy** at all (**prompt but not safe**)

In the other case, the **delay is more**, but we grant more **consistency on the internal resource state** (**safe less prompt**)

Master – Slave: CHECKPOINT

CHECKPOINT

The update of the state and its establishment over the slaves

- **Periodic action** **time-driven**
- **Event action** **event-driven**

When to do it?

In case of a **sequential** resource,

the state is more clear and easy to identify and establish

In case of a **parallel** resource, all the parallel actions should be taken into account and considered toward the state saving

the state subjected to more **concurrent actions** is **less easy to isolate** and the state is harder to identify and distinguish

*In other words, the **checkpoint of a resource with several operations going on at the same time** is more complex to deal with and to complete correctly, because of the **sharing of data between concurrent activities***

Checkpoint at entrance/exit and in specific decision points

MASTER - SLAVE: FAILURE RECOVERY

Who identifies the fault and when: which roles?

Fault Recovery

Secondary copies (slaves) must identify the fault of the master by observing its activity

by using application messages coming from the master and by keeping the timing into account

Even ad-hoc management messages can be used and exchanged

The organization can use

one slave for the control protocol (**if single fault**)

a hierarchy of slaves and more complex protocols (**for multiple faults**)

The entire **resource**, from an external perspective, can tolerate a different number of errors depending on internal strategies and can still provide correct services in case of errors (**fault transparency**)

ACTIVE REPLICATION MODEL

Active copies - all copies are active and consistent in executing all operations

An activity executes the operation for any private data copy

Client external requests to the server can have an either **explicit** or **implicit approach** related to **replication**

*If the client has an FT **explicit** ⇒ **no abstraction***

This organization lacks abstraction because all clients has too much visibility of internal FT details of servers

*If the client has an FT **implicit** ⇒ **FT transparency***

Need of a support that is capable of getting the request and distributing them to server copies and vice versa for results

ACTIVE COPIES REPLICATION

Usually the FT is an **implicit private strategy of resources**

either there exists **one manager only** (**static organization**)

centralized **farm** that received the request and commands the operations, collects the answer and gives it back to the client

or there exist **several managers** (**dynamic**)

Any operation gets a different manager in charge of it, with no central role and also balancing of requests

Policy for choosing the manager: decision

Static or

Dynamic by **locality** / by **rotation**

If several operations are alive at the same time, we need to avoid any interference among the different concurrent managers

ACTIVE COPIES COORDINATION

Active models can decide different coordination models

perfect synchrony (full consistency)

all copies should agree and produce a **completely synchronized view**, with the same internal copy scheduling for all copies (difficult for nested actions or external actions)

different approaches to the synchrony (less consistency)

Even if some minimal threshold can be considered, actions can complete before **all copies agrees on the final outcome**, and the final agreement can take place later (also it does not apply even eventually)

Less synchronous strategies **costs less in time**, mainly client service time, and makes protocols easier and more viable but grant less in **operation ordering and release some semantic properties**

Some **modern Cloud systems** decide of abandoning **perfect synchronicity in favor of an eventual synchronicity**

COPIES COORDINATION

Also different actions on active copies can have different requirements and management

reading actions typically actions that can occur easily in parallel and accessing to a limited number of copies

writing actions those intrinsically require more coordination among copies

Any action that can change the state

Implies more coordination to propagate such a change

In case of a clean state partitioning, where any change applies to different partitions, those actions can proceed independently in parallel without any coordination

Eventually, some actions can require a copy reconciliation of actions that could have been interfering

There are also actions with very specific intrinsic semantics

For instance, the actions on a directory can proceed with some more parallelism

Add/delete of a file, read /write of a file, directory listing

Even semantics properties can distinguish operations and can make possible more efficient behavior and greater parallelism

ACTIVE COPIES UPDATING

Any action requires to update the state of any copy

The **update action** must occur **before delivering the answer** to grant a complete consistency but that impacts on response time (more delay in case of failures) (**eager policies vs lazy**)

If the component employs **different managers for any operation**, it is a manager duty to command the internal actions

If the component defines **parallel operations**, all manager must negotiate and conciliate their decisions, causing also some conflict to be solved and some actions in incorrect order to be undone or redone

Strategies for the operation maximum duration

In case of failure during one operation and before its correct completion, there should be the feature of giving an answer anyway, because of the excess of accumulated delay in finishing internal agreement protocols

ACTIVE COPIES AGREEMENT

Copies can reach an agreement before giving the answer

- **All copies should agree** on the specific action (**full agreement**)
- **Majority voting** (not all copies must agree) **with a quorum** (also weighted)
 - correct copies can go on freely
 - other copies must agree on it and then reinserted in the group (recovery)

Failure detection: who is in charge? when?

Reinsertion detection: who is in charge? when?

There is a strict need of monitoring and execution control

Group semantics

In a group, depending on agreed semantics of actions, there may be also less expensive and less coordinated actions on execution orders

The less the coordination, the less is the cost

Dependability 49

WIDESPREAD REPLICATION MODELS

Which is the FT replication model more common and widespread?

the Master-Slave model is simpler and with only one execution point

the Active Copies is more complex and implies more coordination

In any model, the cost is influenced by the group replication degree, i.e., the **number of copies**, *either working or not*

A search on the most common applications and more widespread ones, the **replication degree is typically very limited** (no more than a few copies)

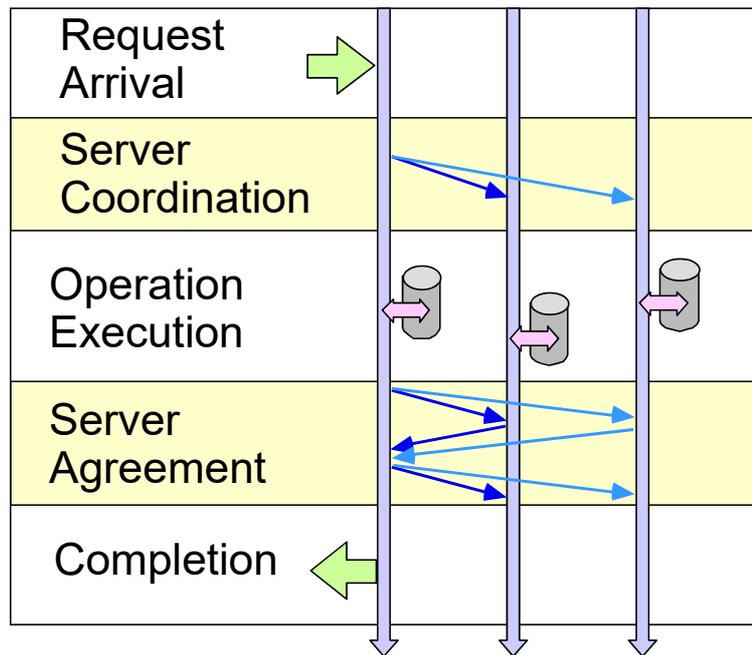
There are also intermediate replication models, non FT oriented, with **a set of resources** that are able to **work independently** on the same kind of operations, and they operate on different service at the same time, and they can share the responsibility of being a back-up of each other (throughput driven and load balancing)

Dependability 50

ACTIVE COPIES OPERATIONS

To make more clear the needs of different steps and one general workflow, we can model the group operation as a sequence of five phases:

- 1) **Request Arrival**
- 2) **Copy coordination**
- 3) **Execution**
- 4) **Copy agreement**
- 5) **Response delivery**



PHASE 1: CLIENT REQUESTS

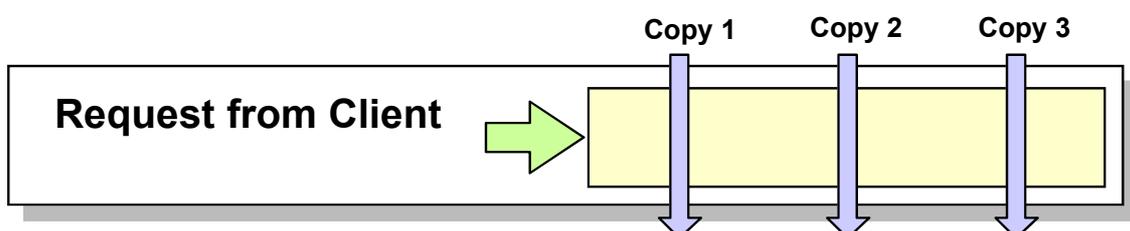
the client can send the request of an operation

- *Only to one of the copies*
- *to all copies*

In case of a delivery to copy only, it is that copy that should propagate the requests to all other ones

The manager is in charge of re-bouncing the first phase

The specific copy can be decided either dynamically or statically



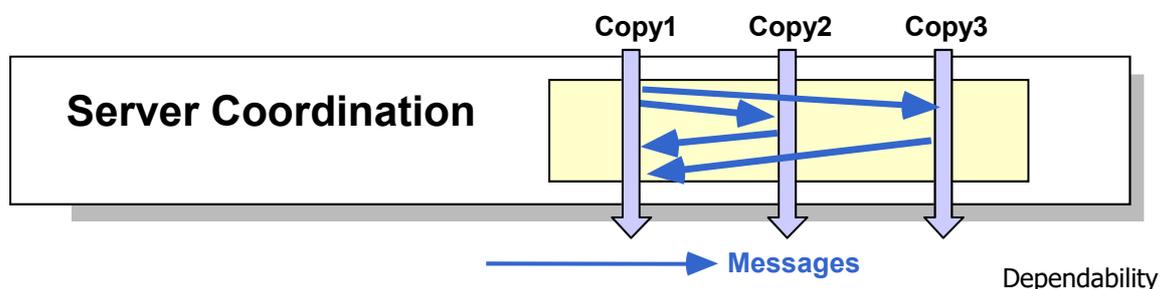
PHASE 2: COPY COORDINATION

The **copies** must coordinate with each other, to define a negotiated policy in scheduling

One **master copy** can become the manager of that operation

- **All copies** must decide how and when to execute the operation to prepare the correct execution
- Different **copies may have different weight** in the group and a different role

First coordination phase



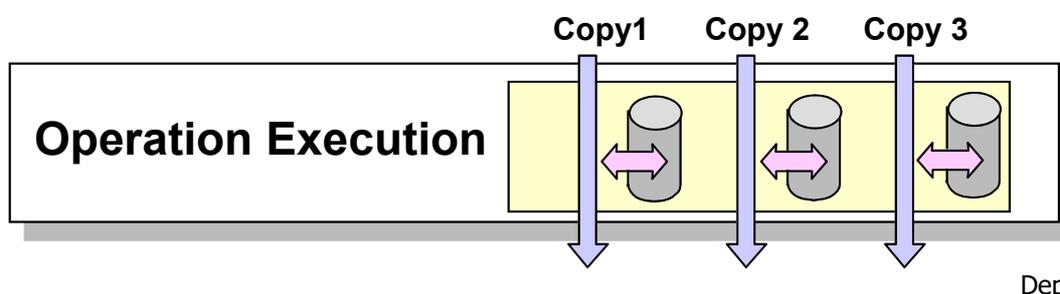
PHASE 3: COPY EXECUTION

The **coordination phase** influences the execution and some **scheduling can be avoided or prevented**

In general, some degree of freedom can be still left to individual decisions

Depending on agreed policy and general scheduling

- All copies execute with proper decision (some copies maybe prevented, up to a master slave case)
- Clashing executions may require coordination o posteriori

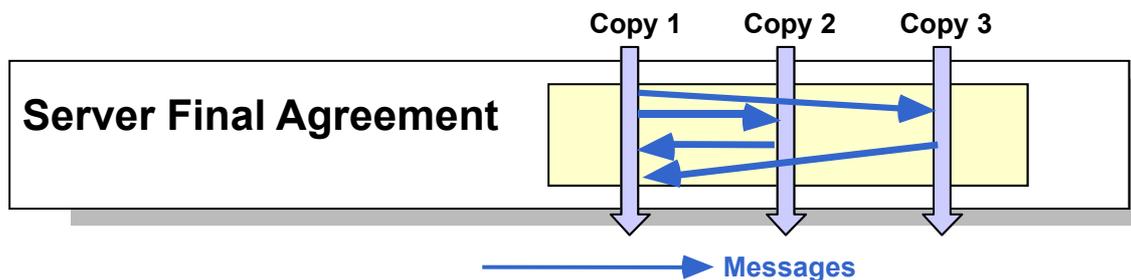


PHASE 4: COPY AGREEMENT

All copies (some are out of the group) must agree on the result to be given back: some results are not conformant to the group whole decision

The group must decide either the commit or also some undo on some actions and the exclusion of related divergent copies from the group for incorrectness

Second coordination phase



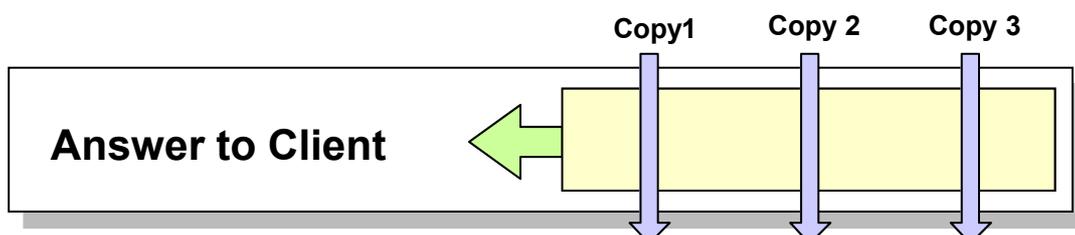
Dependability 55

PHASE 5: RESULT DELIVERY

This phase has the goal of delivering the correct result to the waiting client

the client gets **the operation result**

- One unified **answer** from the copy he has sent the request
- **Answers from all copies separately** (overhead of handling all responses)



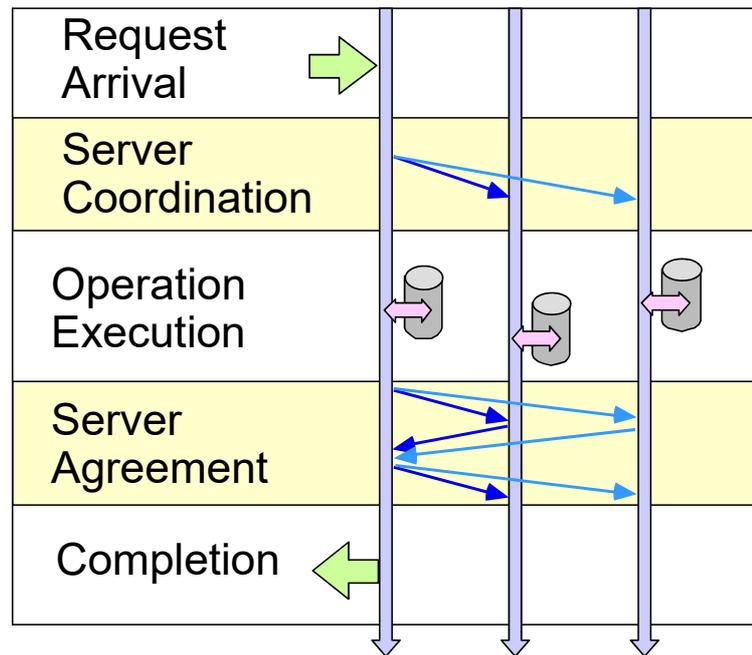
Dependability 56

ACTIVE COPIES OPERATIONS

The sequence of the five phases gives a first idea of the complexity of an active copy replication

The coordination among copies tends to induce a high overhead to be limited

So the **replication degree must be kept low** and **replication policies are to be kept simple**



Dependability 57

UPDATING POLICIES

To classify some FT **resources** replication, we can use two significant directions

- **who decides the updating**
only the **primary** copy or **all copies**
- **promptness in propagation of the updates**
eager (immediate and before the answer) **pessimistic** or **lazy** (delayed after the propagation) **optimistic**

(we can reverse the terms for the client perspective)

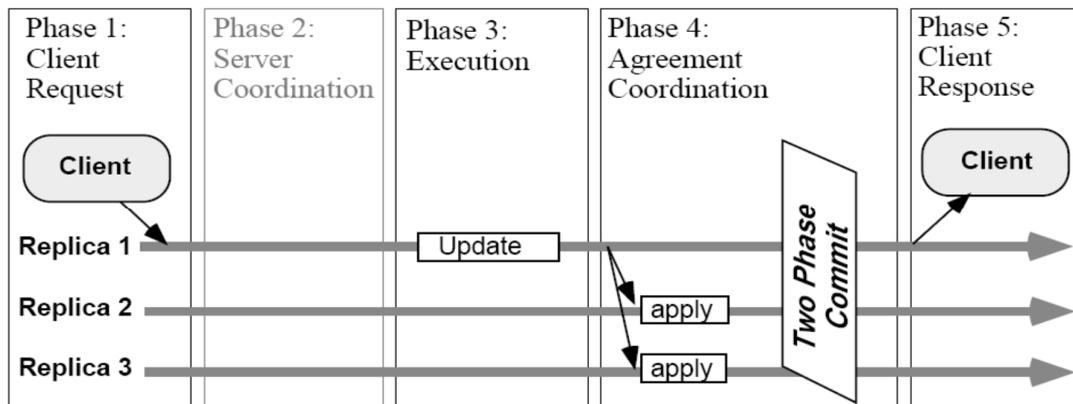
For the updating we can distinguish:

- ***Eager primary copy vs Lazy primary copy***
- ***Eager updating for all copies vs Lazy updating for all copies***

Dependability 58

EAGER PRIMARY COPY

Sticking to one primary, that copy executes and **gives back** the answer only after **having updated the state** of all copies in a pessimistic approach (*one operation at a time with faults*)

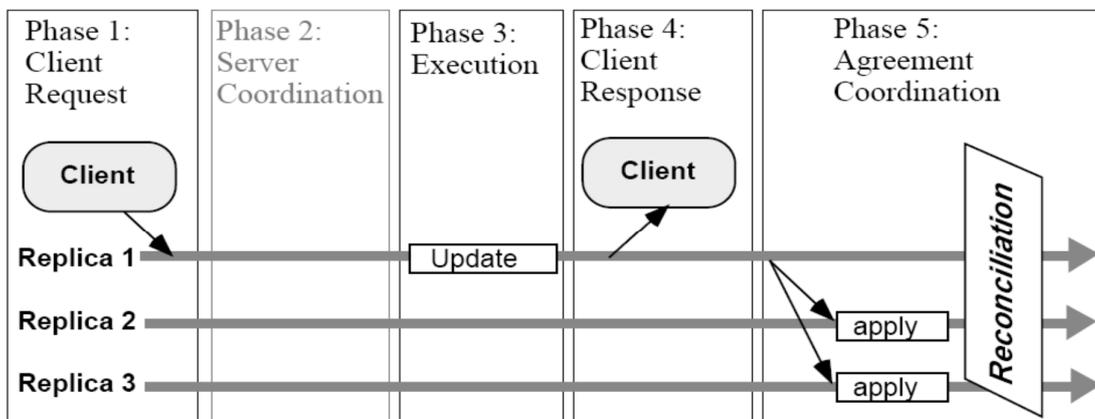


In that case, the manager is in charge of the whole coordination, but the client receives a deferred answer (for correctness sake)

If more operations are active over the replicated object that does not change

LAZY PRIMARY COPY

On the opposite, the manager can first answer to the client and afterwards it updates the copies with an optimistic approach (*also several operations can go on at the same time*)



In this case, the manager must also be able to control the possible reconciliation of the state of the copies ... and some problems may occur if there is a manager crash

UPDATE OF ACTIVE COPIES

Eager policies favor **consistency** and **correctness** of the operations, instead of the promptness of the answer to the client

The goal is **not very fast precocious answers**, because that can lead to undo actions, that are not easy to be done, and, in some cases, impossible to backtrack

Copy coordination are **two phases toward consistency granting** (specially in case of **concurrent actions**)

Let us stress that those two phases are not always necessary but they can obtain the necessary coordination among copies and operations

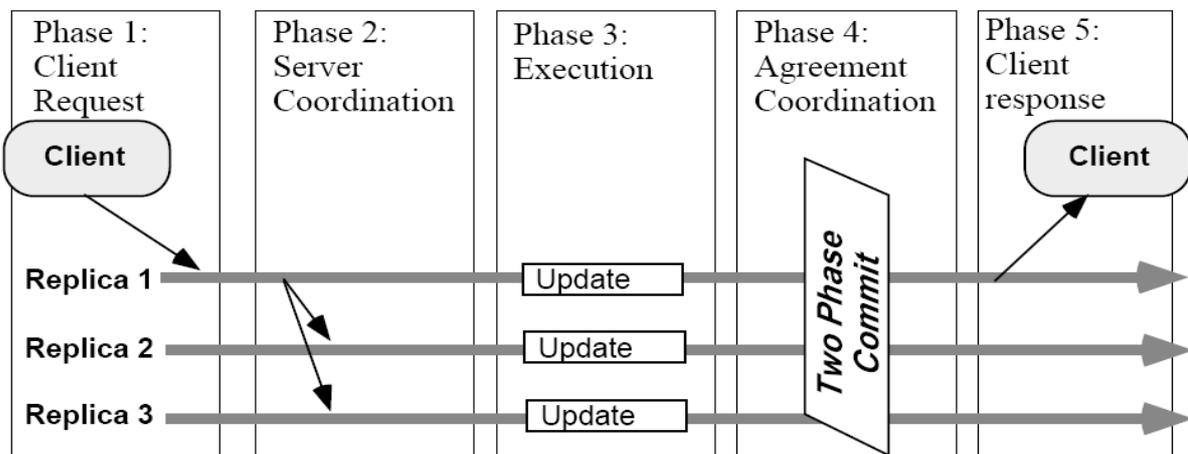
A posteriori coordination to **verify consistency**. If it is not verifies, some **undo** must be considered (two phase protocol and roll back)

A priori coordination can ensure that all **correct copies receive all correct messages** and the **right schedule** is automatically enforced (e.g., atomic multicast)

Dependability 61

OPTIMISTIC EAGER UPDATE

All copies are updated with some enforcing policies in a **optimistic approach (two-phase commit)**, only afterwards the answer is provided to the client

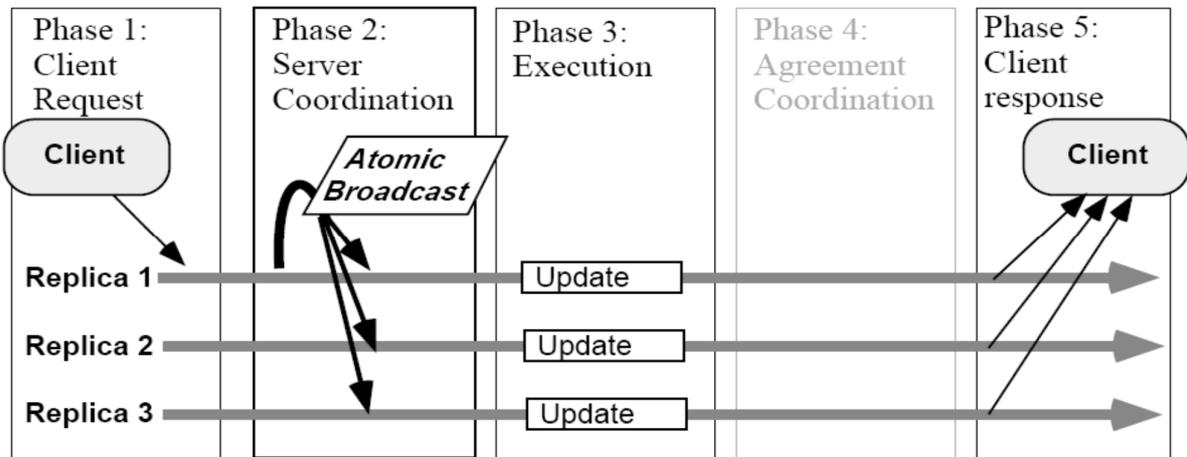


After copy independent executions, the final coordination ensure an agreement, otherwise some backtracking is commanded (*possible **undo***)

Dependability 62

PESSIMISTIC EAGER UPDATE

A different approach for eager update implies coordination but tends to save the final phase of it. The agreement of results is granted via a delivery protocol in a **pessimistic approach**



An **atomic multicast can ensure** that any message is correctly sent to all copies in the same order, so that there is no need for a final check (*no undo*)

Dependability 63

OPTIMISTIC LAZY POLICIES

We use **lazy update**, when one copy can answer with a little (no) coordination with other copies in an **optimistic policy that can deliver the answer very fast ...** as in the case of **Amazon S3 (Amazon Simple Storage Service)**

Amazon memory & persistence support renounces to any strict consistency and provide both **consistent** and **eventually consistent** operations **ones**

Strong consistency has the eager update but slow answer

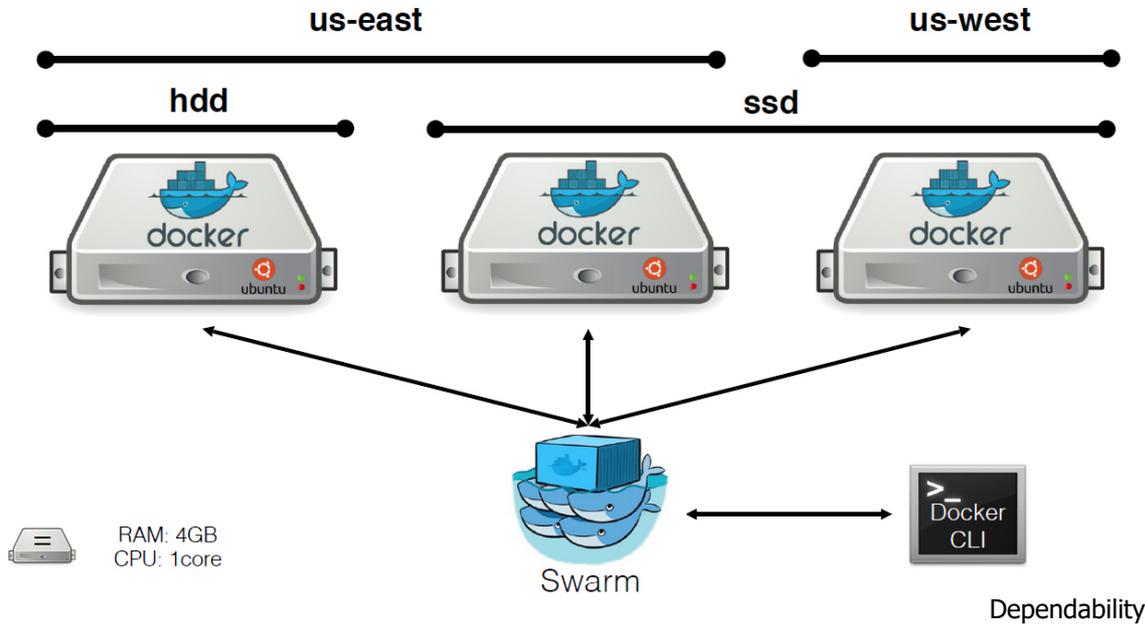
Eventual consistency (called final or tending to infinity) is a lazy update in the direction of **released consistency**: updates are commanded but not awaited for. So concurrent operations over other copies can see different values. On a long term, copy values are reconciliated and a consistent view is achieved.

The **inconsistency window** may depend on many factors: communication delays, workload of the system, copy replication degree ... (**We are happy if it is as small as possible**)

Dependability 64

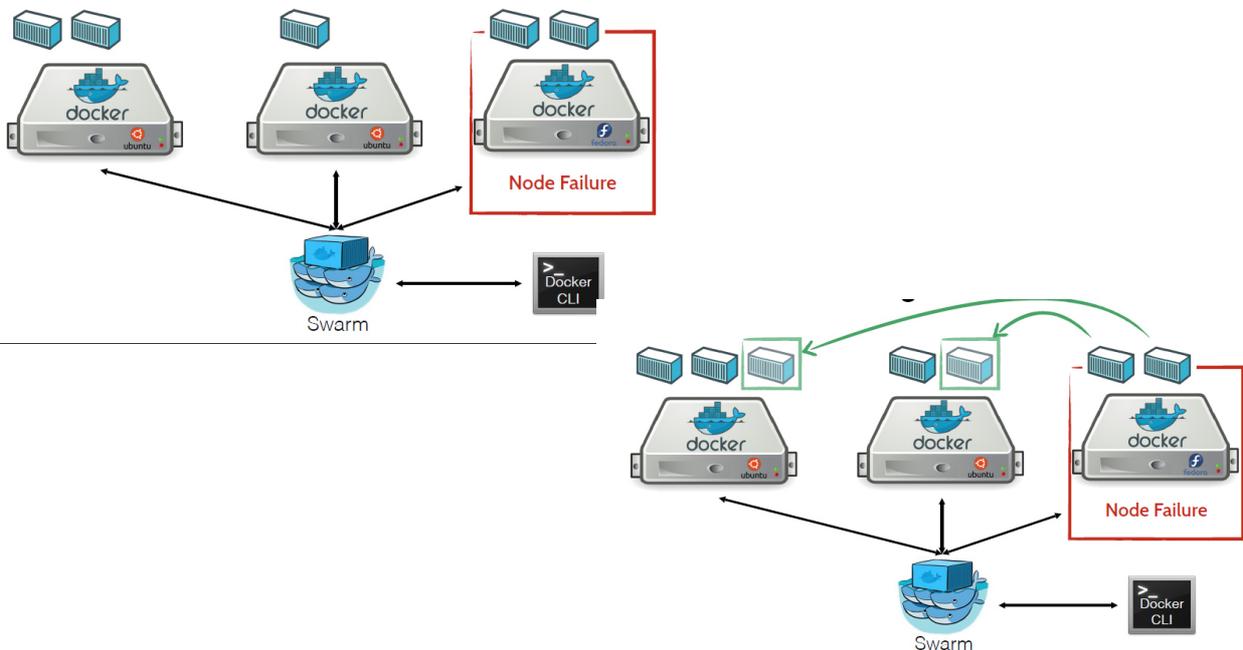
DOCKER SWARM

Docker Swarm proposes the feature of **loading a distributed system**: the example is with **three nodes with a manager invoked via a central console for a portable dynamic loading**



DOCKER SWARM

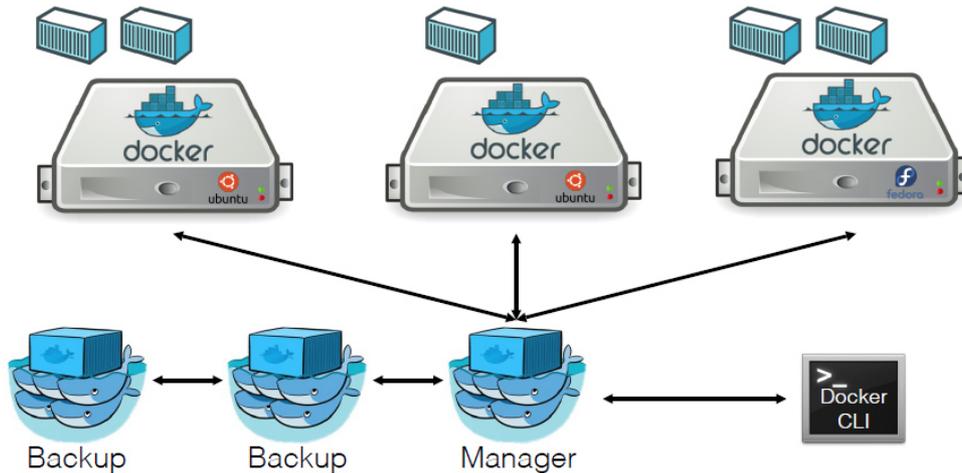
Docker Swarm can take automatically care of the case of failure of a node, and **can transfer some components to the new containers for a 'degraded' execution**



HA DOCKER SWARM

Docker Swarm can also allow **high availability** and can replicate also the manager for the distribution to overcome the single point of failure of the manager

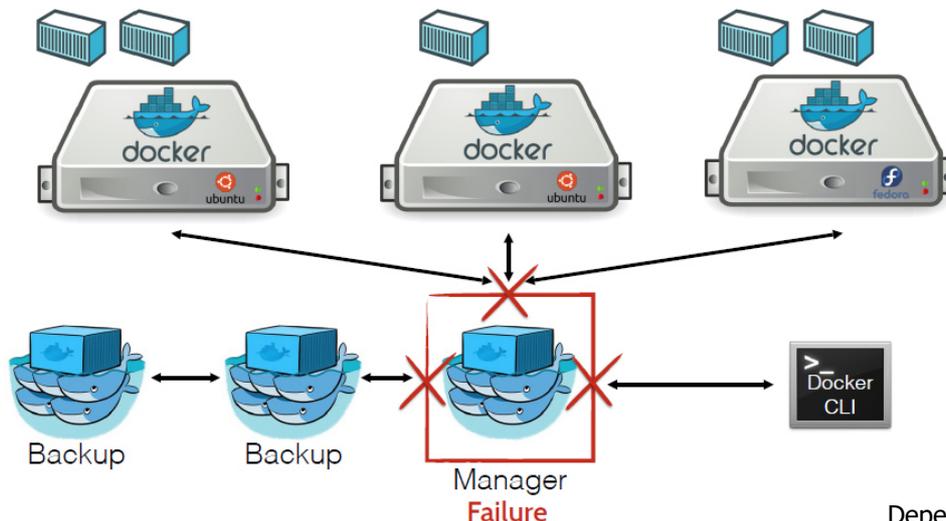
In case of failure of any node, it **can still operate and without interruption**



HA DOCKER SWARM

Docker Swarm can also allow **high availability** and can replicate also the manager for the distribution to overcome the single point of failure of the manager

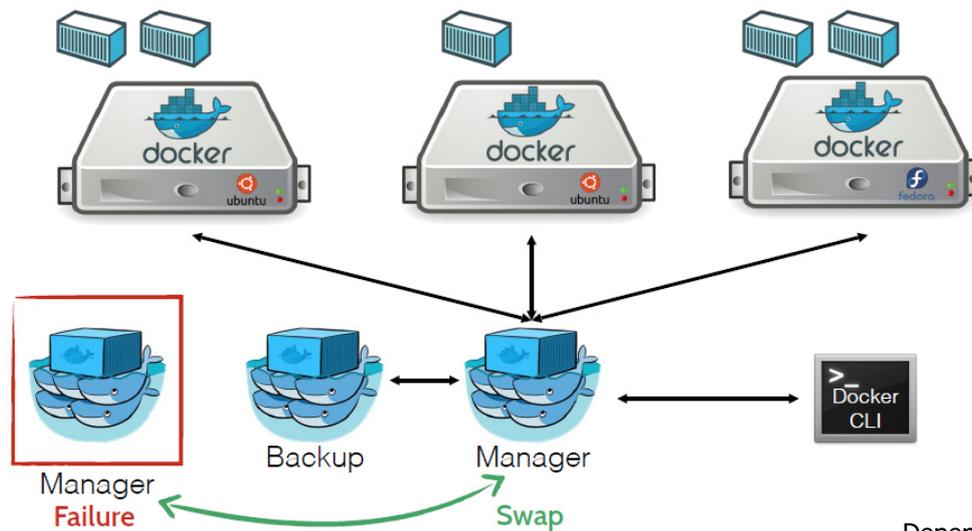
In case of failure of any node, it **can still operate and without interruption**



HA DOCKER SWARM

Docker Swarm can also allow **high availability** and can replicate also the manager for the distribution to overcome the single point of failure of the manager

In case of failure of any node, it **can still operate and without interruption**



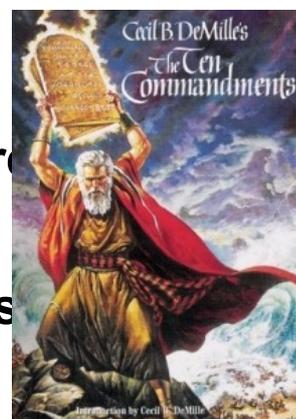
Dependability 69

MORE 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



Dependability 70

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

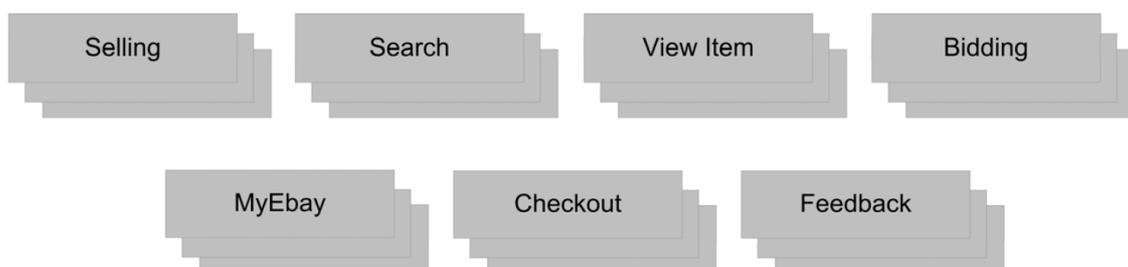
>48 Billion SQL executions/day!



1 - Partition Everything

Pattern: **Functional Segmentation**

- Segment processing into pools, services, and stages
- Segment data along **usage boundaries**

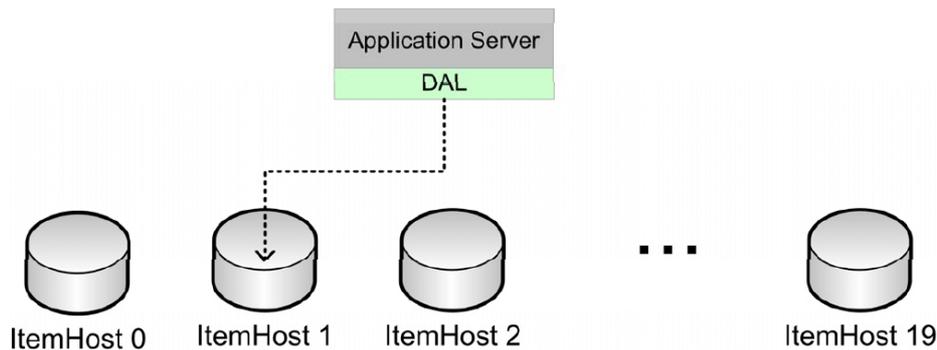


You should split anything you can in **separated localities**
No **large components** (to be kept consistent)

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 Everywhere

Prefer Asynchronous Processing

- Move as much processing as possible to asynchronous flows
- Where possible, integrate disparate components asynchronously

Requirements

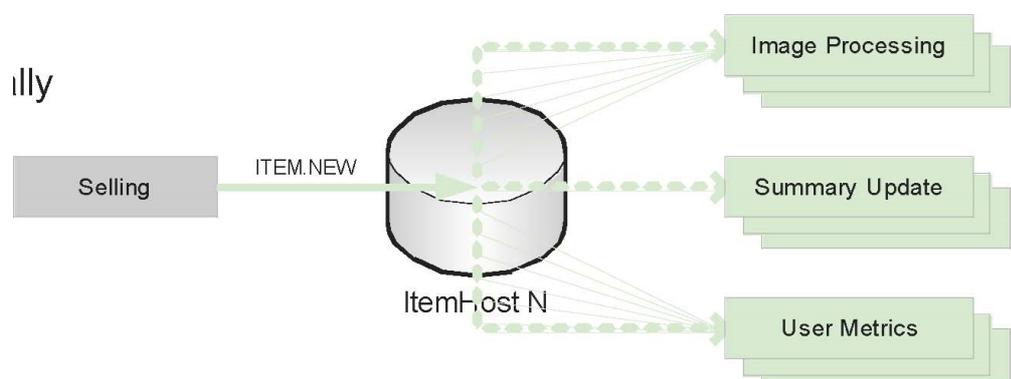
- **Scalability**: can scale components independently
 - **Availability**: can decouple availability state and retry operations
 - **Latency**: can significantly improve user experience latency at cost of data/execution latency
 - **Cost**: can spread peak load over time
- Asynchronous Patterns, Message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

Dependability 75

2 - Asynchrony Everywhere

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

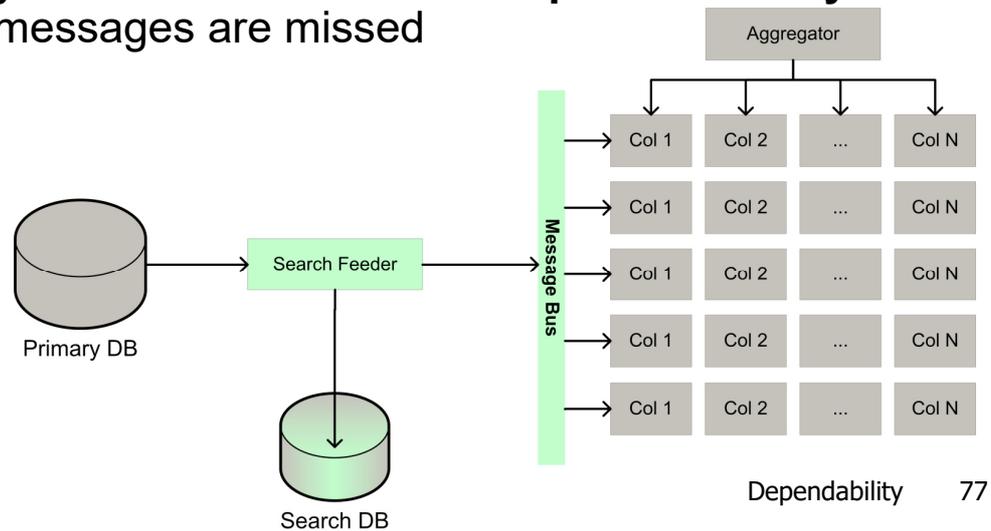


76

2 - Asynchrony Everywhere

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



Dependability 77

2 - Asynchrony Everywhere

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

Dependability 78

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**

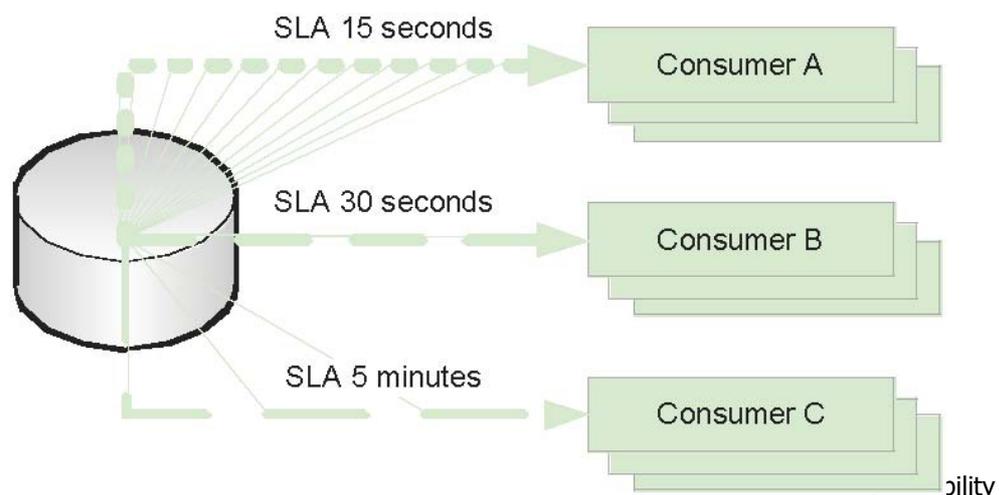
Easy to consider more factors in decisions and explore solution space more thoroughly and quickly

- Automation Patterns
- Adaptive Configuration
- Machine Learning

3 - Automate Everything

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



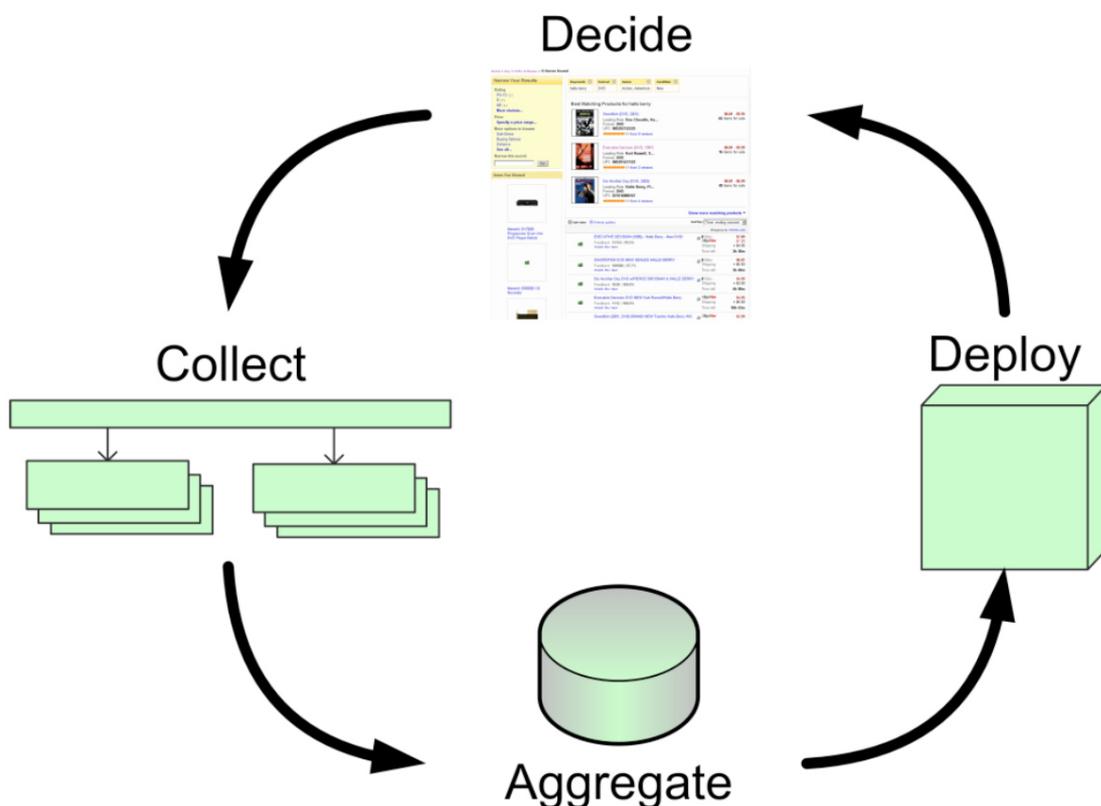
3 - Automate Everything

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**

Dependability 81

3 - Automate Everything



4 - Everything Fails

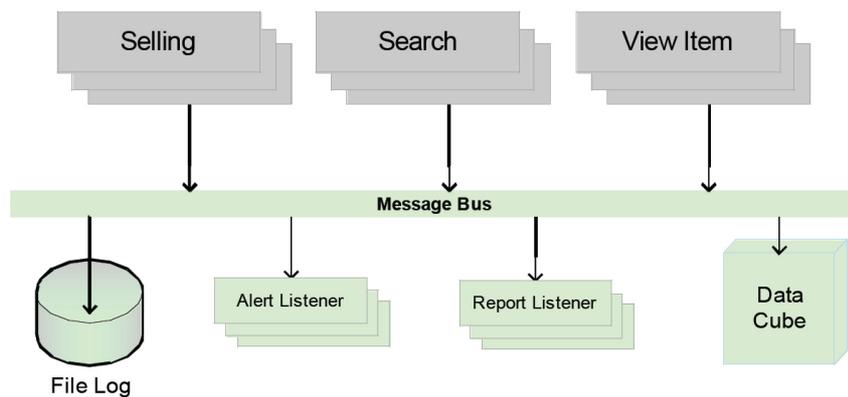
Pattern: **Failure Detection**

- 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**



endability 83

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**

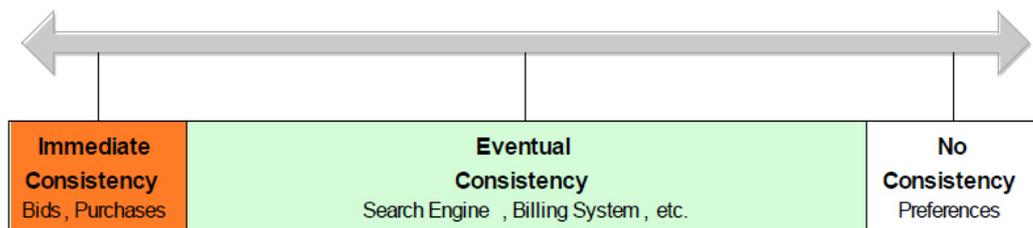
EBAY PRINCIPLES

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



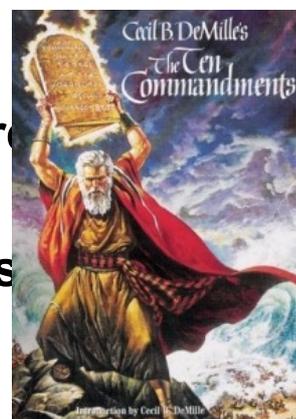
Dependability 87

MORE 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



Dependability 88

CLOUD DATA CENTERS

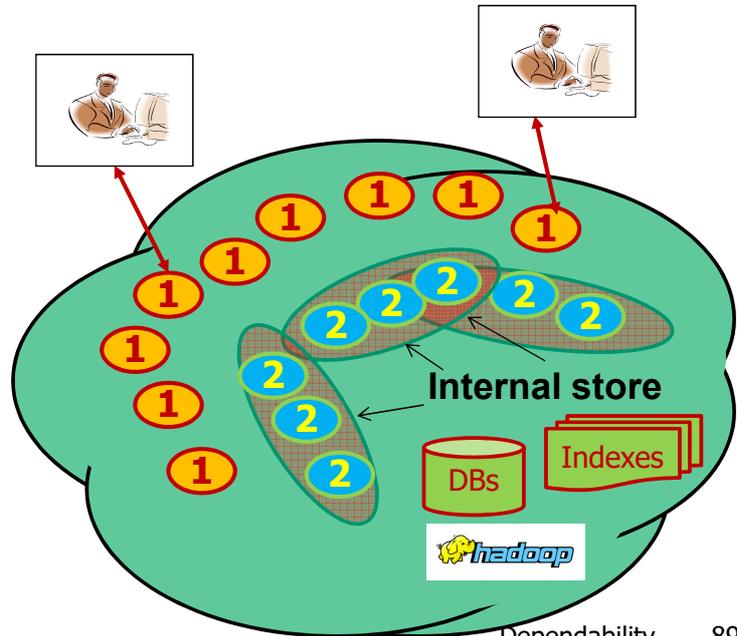
Let us have a look at the **cloud internal organization**...

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**

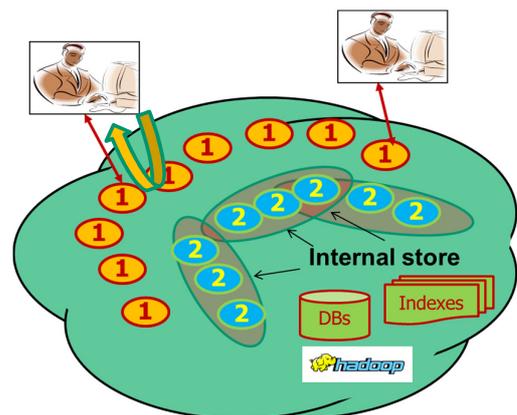
The **second deep level** is the one in charge of **deep data** and their **correct and persistent values**



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 the **CLOUD edge** and must give very prompt answers to **many possible client requests, even concurrent one with user reciprocal interaction**

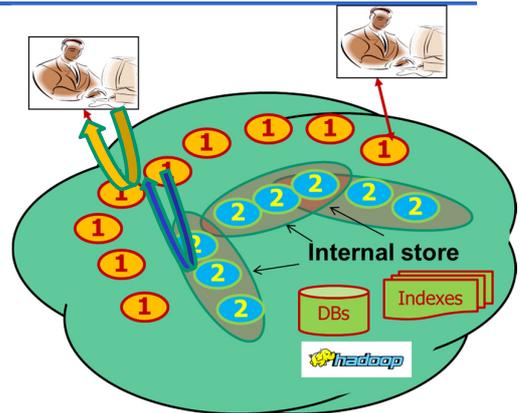


- The edge level has the first requirement of **velocity and should return fast answers**: for **read** operations, no problem; for **write** operations some **problems may arise and updates are tricky**
- Easy **guessing model**: try to **forecast the update outcome** and respond fast, but operate in background with the **level 2**

CLOUD: INTERNAL LEVEL 2

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



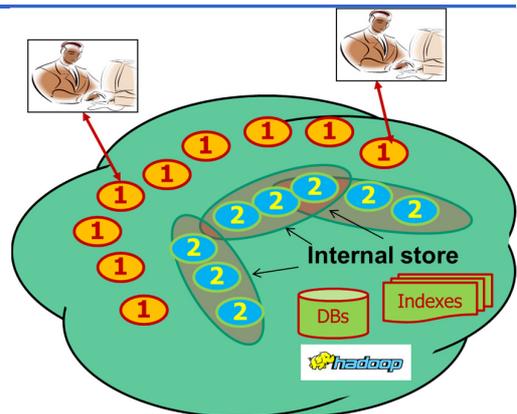
- **Level 2** is in charge of **replicating data and keeping caches** to favor user answers. Replication of course can provide several copies to provide fault tolerance and to spread loads
- **Replication policies** do not require **replication of everything**, but only some **significant parts** are replicated (called **shard** or 'important' pieces of information **dynamically decided**)

Dependability 91

CLOUD TWO LEVELS

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

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, ...)

Dependability 92

CLLOUD 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** (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

Dependability 93

CLLOUD 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 split 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

Dependability 94

CLOUD SHARDS

The definition of **smaller contents for replication or '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 more and more copies of it are available 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)*

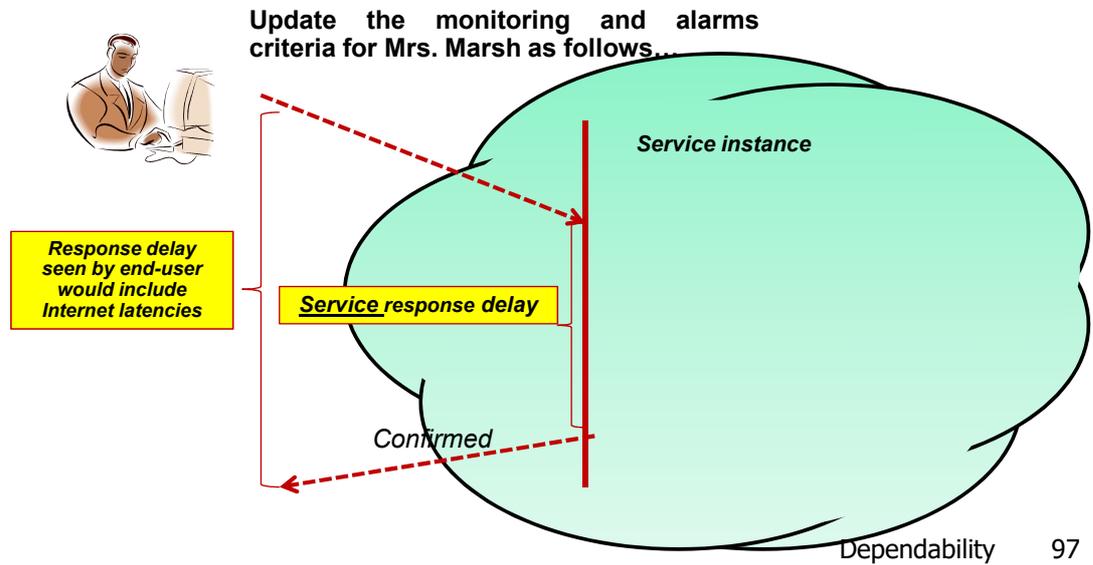
TECHNOLOGY TOOLS

Service	Guaranteed properties and introduced requirements
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

CLOUD SERVICE TIMES

Users expects a very fast answer and some operations accordingly

The system must give fast answer but operate with dependability (reliability and availability)

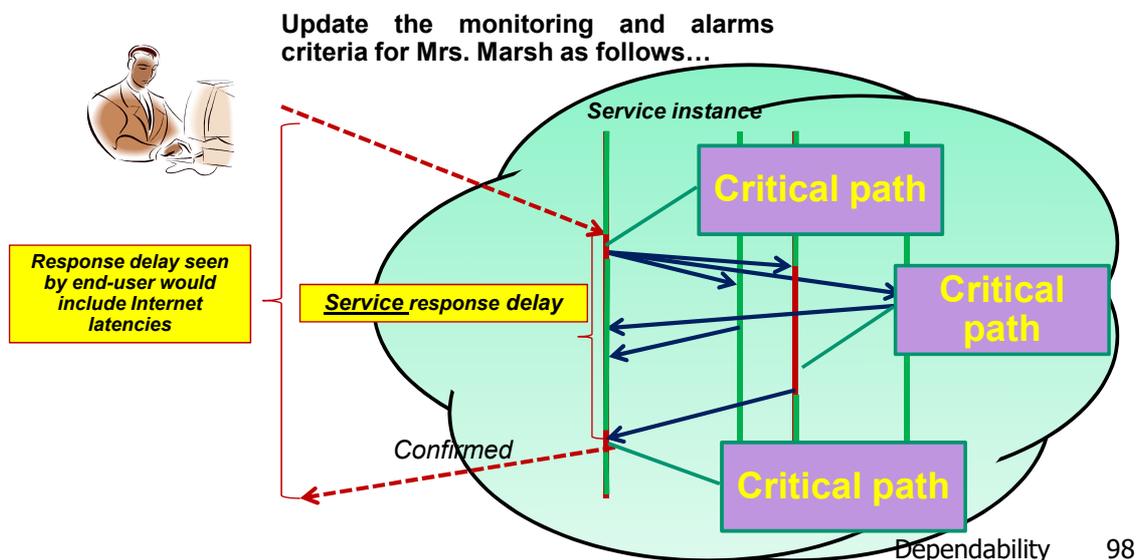


CLOUD CRITICAL PATHS

The fast answers are difficult 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

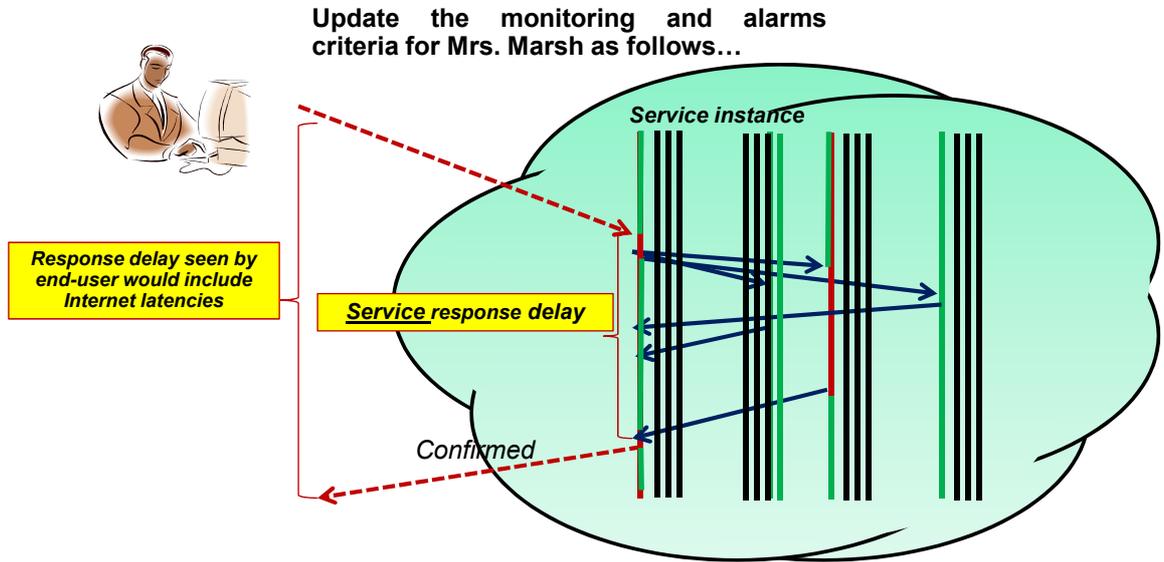


CLOUD REPLICAS AND PARALLELISM

Fast answers can stem from replicas and parallelism

With replication you can favor parallel execution for read operations

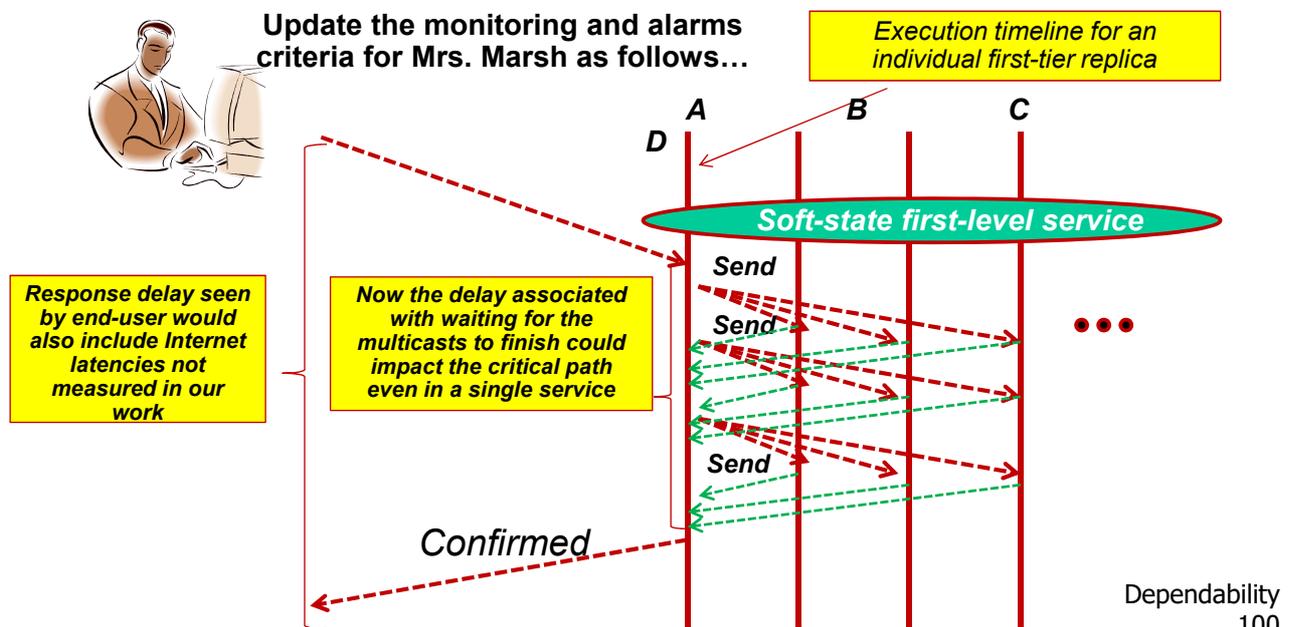
That can speed up the answer



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



CLOUD ASYNCHRONOUS EFFECTS

If we use extensively internal replicas and parallelism

to answer user expected delay, we need to go **less synchronous**, and adopt extensively an **asynchronous strategy**

Answers 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 in a different schedule, their final state can be different and not consistent

If some replica fails, the results cannot be granted (specially if the leader fails) and the given back answer is incorrect

Some agreement 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

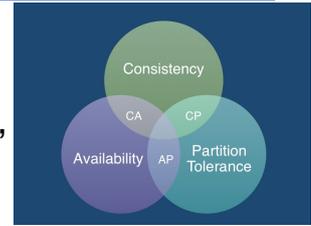
The customer can really feel the difference with small variation

So, there many cases in which you do not need a **real correct answer**, but **some approximation 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 some replica of the data, even in the presence of failures

Partition-tolerance: the system properties still hold even when the system is partitioned

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)**