DSC516: Cloud Computing

Part II: Cloud Building Blocks

Module 3: Cloud Infrastructure

University of Cyprus
Department of Computer Science

Lecture 12b

Cloud Infrastructure Software, Workloads, and Metrics

University of Cyprus
Department of Computer Science

Readings

- **Chapters 2, 7, 8** "The data center as a Computer. An Introduction to the Design of Warehouse-Scale Machines" Barroso, L. A., Holzle, U. & P. Raganathan (2018).
- **"The tail at scale"** J. Dean and L. A. Barroso, Commun. ACM, vol. 56, no. 2, pp. 74–80, Feb. 2013.
- **"Lessons from giant-scale services,"** E. A. Brewer, IEEE Internet Comput., vol. 5, no. 4, pp. 46–55, Jul. 2001.
- **"Harvest, yield, and scalable tolerant systems,"** A. Fox and E. A. Brewer, Proc. Work. Hot Top. Oper. Syst. - HOTOS, pp. 174–178, 1999.
- **"CAP twelve years later: How the 'rules' have changed,"** E. Brewer, Computer (Long. Beach. Calif)., vol. 45, no. 2, pp. 23–29, Jan. 2012.

Learning objectives

- Understand and explain the concepts of **resource management**, **monitoring systems**, **performance debugging**, **blackbox** monitoring, **instrumentation, site reliability engineering.**
- Understand and explain the **software infrastructure building blocks** of WSC offering cloud services.
- Understand and explain the characteristics of **typical workloads running on WSC**.
- Review, understand and explain common techniques for **improving the performance and availability of WSC**.
- Be familiar and explain concepts like **cloud native, load balancing**, **sharding** (partitioning), **replication**, **integrity-checking**, **eventual consistency**, **redundant execution**, **tail-tolerance** in the context of cloud infrastructures.
- Understand, explain and apply the concepts of **Availability**, **Mean Time Between Failure** (MTBF) and **Mean Time to Repair**
- Understand and explain the **CAP theorem** and the concept of **tail-latency**.
- Understand and explain the concept of "**Cloud native**" software.

University of Cyprus
Department of Computer Science

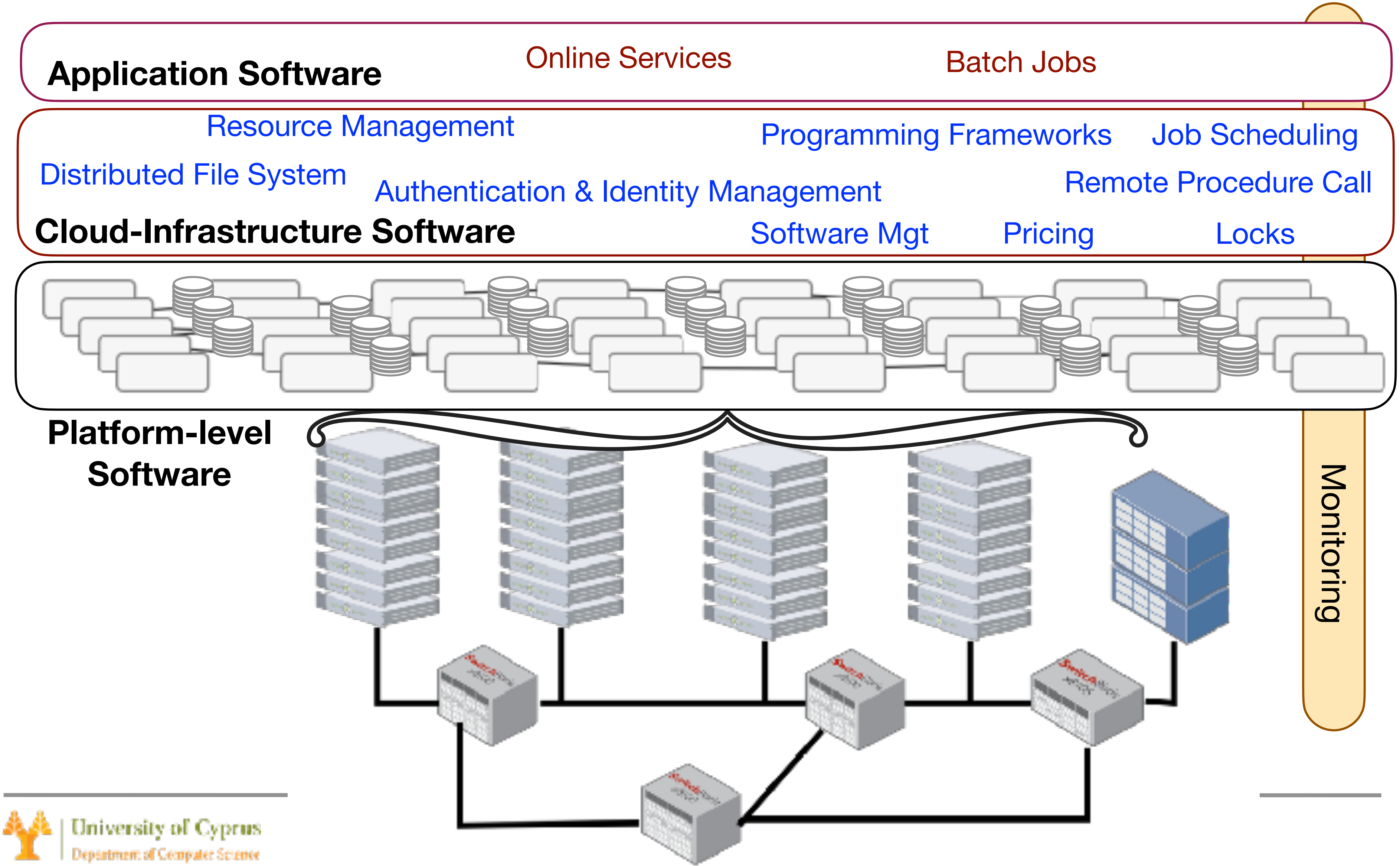
Cloud Infrastructure Software, Workloads, and Metrics

Cloud Infrastructure Software

WSC System Stack - Terminology

- **Platform-level software**:
  - present in all individual servers to abstract the hardware of a single machine
  - provides basic machine abstraction layer
- **Cluster-level infrastructure**: operating system for a data center.
  - distributed file systems, schedulers and remote procedure call (RPC) libraries
  - programming models that simplify the usage of resources at the scale of data centers.
- **Application-level software**:
  - online services and offline computations
- **Monitoring and development software**:
  - keeps track of system health and availability by monitoring application performance, identifying system bottlenecks, and measuring cluster health.

WSC System Stack



Application Software

Online Services

Batch Jobs

Resource Management

Programming Frameworks

Job Scheduling

Distributed File System

Authentication & Identity Management

Remote Procedure Call

Cloud-Infrastructure Software

Software Mgt

Pricing

Locks

Platform-level Software

Monitoring

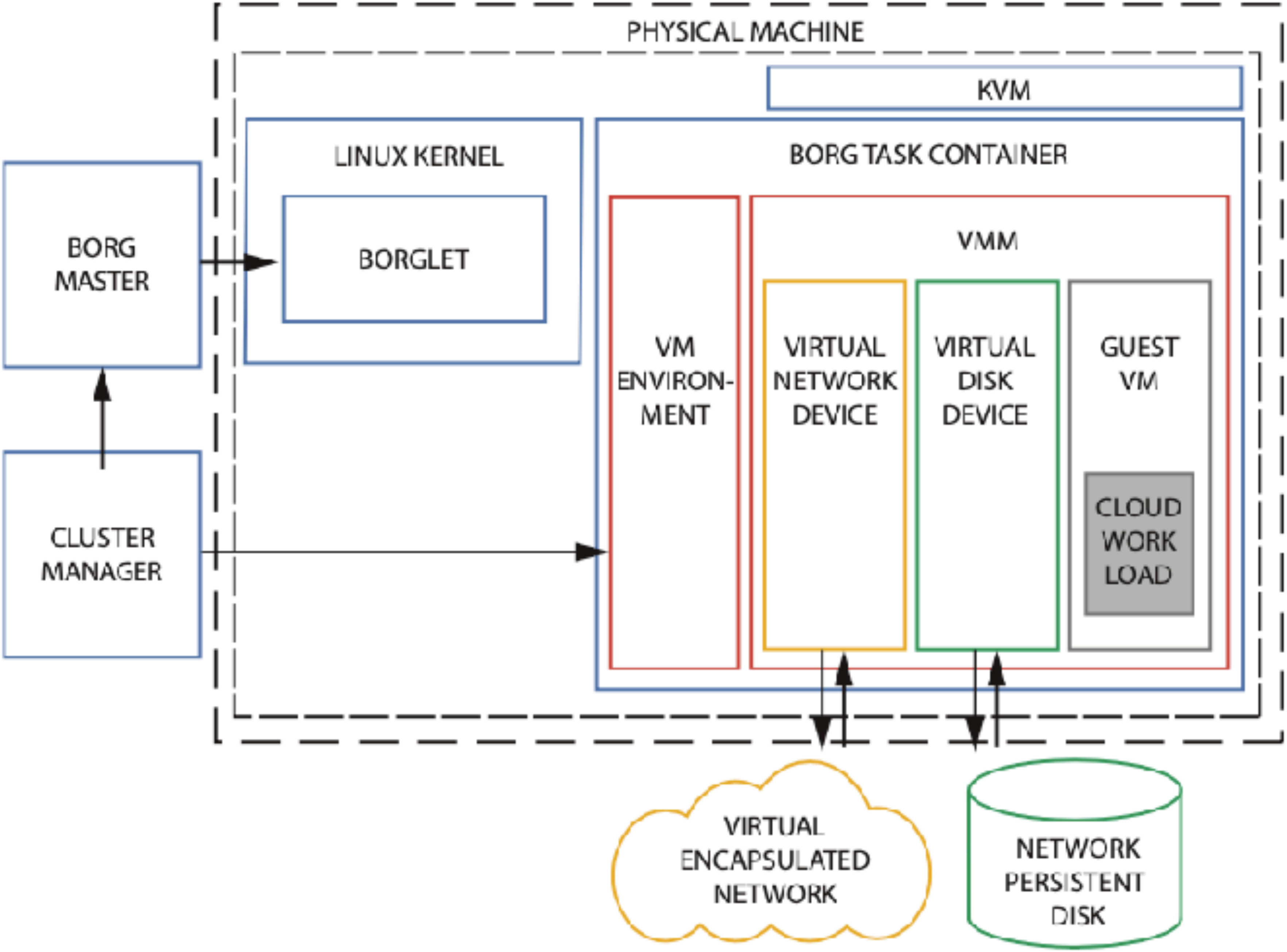


Figure 2.8: Overview of VM-based software stack for Google Cloud Platform workloads.

Cloud Infrastructure Software

Platform-level Software

Features

- Firmware, device drivers, operating system modules, configuration parameters.
- Streamlining of development/testing/optimizations/configuration for increased performance, possible thanks to:
  - **Homogeneity** across devices
  - Mostly **local networking connections** within the same building
    - Lower packet losses, better tuning of transport or messaging parameters for higher communication efficiency
- Virtualization popular in WSCs, especially for IaaS offerings.
  - **VMs**:
    - provide concise and portable interface to manage **security** and **performance isolation** of a customer's application
    - allow multiple guest OS to co-exist with limited additional complexity

From VMs to Containers

- VMs' **downside:** performance, particularly for **I/O-intensive workloads**.
  - In many cases today, those overheads are improving and benefits of VM outweigh their costs.
- The simplicity of VM encapsulation makes it easier to **implement live migration**.
- **Containers:** an alternate popular abstraction that allow for **isolation across multiple workloads** on a **single OS instance**.
  - more lightweight compared to VMs, smaller in size and much faster to start

Cloud Infrastructure Software

Cluster-level Infrastructure Software

Cluster-level Infrastructure Software

- Aim: provide OS-like functionality at a WSC-level scale.
- **Resource Management**: managing user tasks (mapping, scheduling, etc)
- **Cluster Infrastructure:** offer basic functionalities necessary for the infrastructure to operate and be managed properly
- **Application Frameworks:** offer abstractions to facilitate application development

Resource Management

- Controls the **mapping** of user tasks to hardware resources, enforces **priorities** & **quotas**, and provides basic **task management**.
- Simple approach: manual and static allocation of groups of machines to a given user or job
- More useful: present a **higher level of abstraction**, **automate allocation** of resources, allow resource sharing at **finer granularity**.
  - Users specify job requirements at a relatively high level and have the scheduler translate those requirements into an appropriate allocation of resources.
  - Consider also: **power limitations, energy usage optimization**, **maximization** of **provisioned DC power budget usage**, failure domains and fault tolerance, and dealing with **emergencies**.
- **Kubernetes**: a popular open-source program provides such functions for container-based workloads.

kubernetes

Cluster Infrastructure

- Provides a core set of functionalities correctly and with high performance and availability:
  - reliable distributed storage, RPCs, message passing, and cluster-level synchronization.
- Avoid re-implementing tricky code for each application and instead create modules or services that can be reused.
- **Reliable storage** and **locks**: Colossus (successor to GFS), Dynamo, and Chubby @ Google
- Software **image distribution** and **configuration management.**
- **Performance monitoring, debugging** and **optimization.**
- **Health Management:** Automated diagnostics, automated repairs workflow, triaging alarms for operators in emergency situations.

Application Frameworks

- Cluster-level infrastructure software does not fundamentally hide the inherent complexity of a large scale system as a target for the average programmer.
- Challenge: A programmer needs to develop for a cluster w/:
  - a deep and complex memory/storage hierarchy,
  - heterogeneous components,
  - failure-prone components,
  - varying adversarial load from other programs in the same system, and
  - resource scarcity (such as DRAM and data center-level networking bandwidth).
- Some types of higher-level operations or subsets of problems are common enough in large-scale services
  - Pays off to build targeted programming frameworks & simplify development of new products
  - Such frameworks handle data partitioning, distribution, and fault tolerance
  - E.g. Flume, MapReduce, Spanner, BigTable, Dynamo, Google Kubernetes Engine (GKE), CloudSQL, AppEngine

Cloud Infrastructure Software

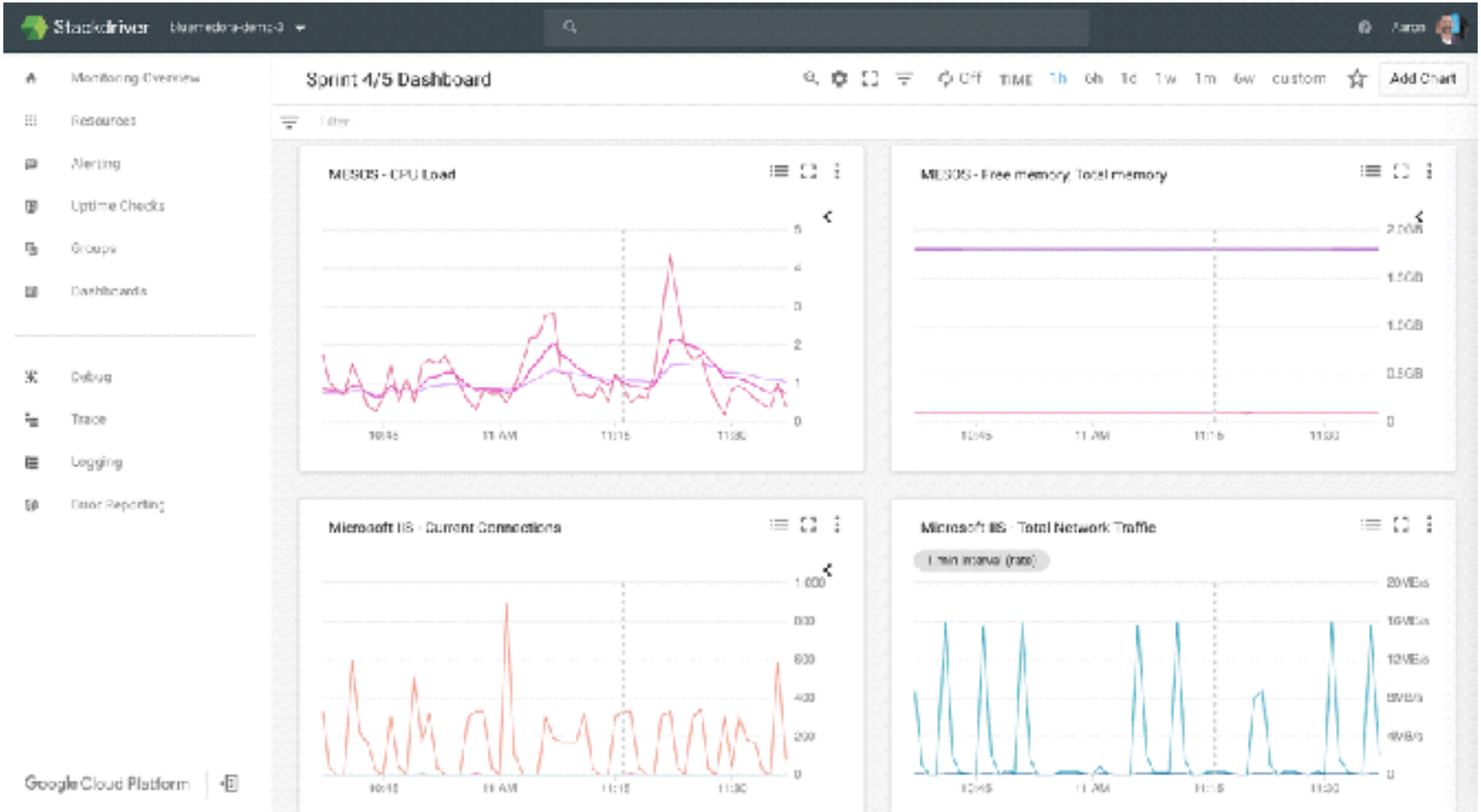
Monitoring Infrastructure

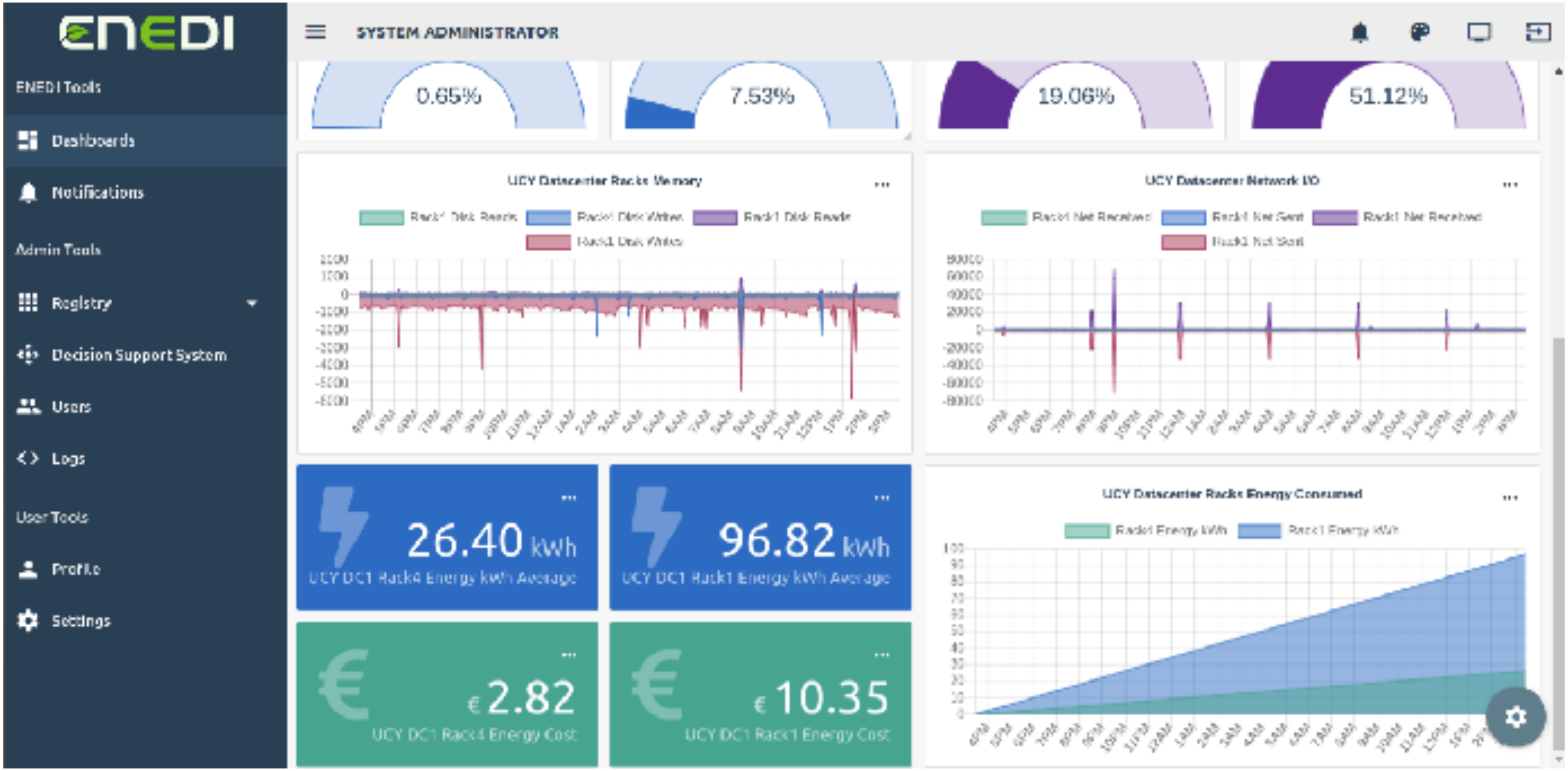
Monitoring Systems

- Concerned with various forms of system **introspection**.
- Can simply be a:
  - **script that polls all front-end servers** every few seconds for
  - *just a few appropriate signals*, such as **latency** and **throughput** statistics for user requests.
- Support a simple language that lets operators create derived parameters based on baseline signals being monitored.
- Generate automatic alerts to on-call operators depending on monitored values and thresholds.

Service-Level Dashboards

- Collect and present key **Service Level Indicators** and displays them to operators in a dashboard:
  - Google Cloud's operations suite (formerly Stackdriver)
- Large-scale services often need more sophisticated and scalable monitoring support
  - more signals to characterize the health of the service; e.g. collect **signal derivatives** over time
  - monitor other **business-specific parameters**


Stackdriver
bluemedora-demo-3
Aaron
Monitoring Overview
Resources
Alerting
Uptime Checks
Groups
Dashboards
Debug
Trace
Logging
Error Reporting
Google Cloud Platform
Sprint 4/5 Dashboard
Off
TIME
1h
6h
1d
1w
1m
6w
custom
Add Chart
Filter
MESOS - CPU Load
5
4
3
2
1
0
10:45
11 AM
11:15
11:30
MESOS - Free memory, Total memory
2.0GB
1.5GB
1.0GB
0.5GB
0
10:45
11 AM
11:15
11:30
Microsoft IIS - Current Connections
1,000
800
600
400
200
0
10:45
11 AM
11:15
11:30
Microsoft IIS - Total Network Traffic
1 min interval (rate)
20MB/s
16MB/s
12MB/s
8MB/s
4MB/s
0
10:45
11 AM
11:15
11:30



0.65%

7.53%

19.06%

51.12%

UCY Datacenter Racks Memory

UCY Datacenter Network I/O

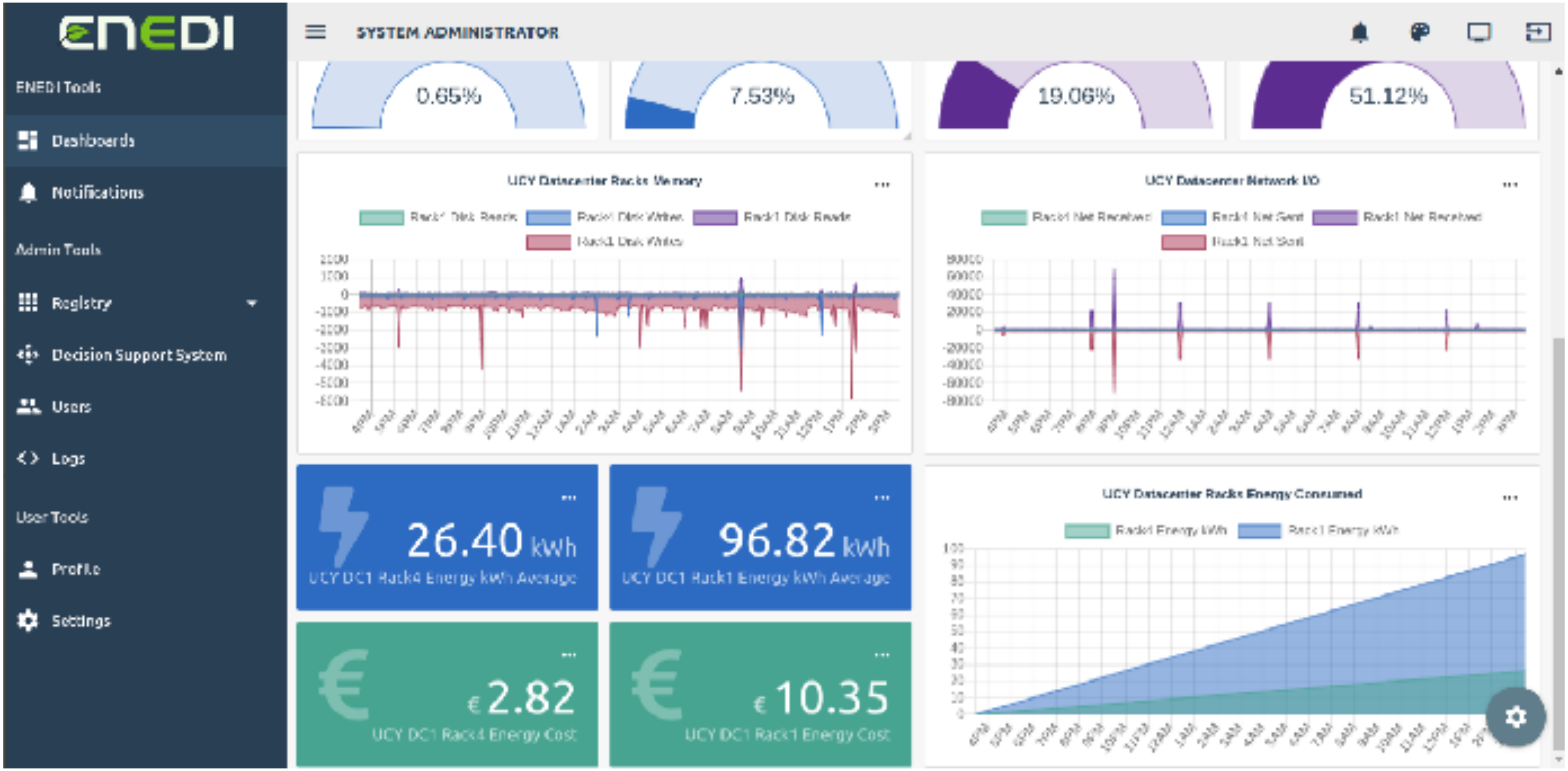
26.40 kWh
UCY DC1 Rack4 Energy kWh Average

96.82 kWh
UCY DC1 Rack1 Energy kWh Average

€ 2.82
UCY DC1 Rack4 Energy Cost

€ 10.35
UCY DC1 Rack1 Energy Cost

UCY Datacenter Racks Energy Consumed



ENEDI
ENEDI Tools
Dashboards
Notifications
Admin Tools
Registry
Decision Support System
Users
Logs
User Tools
Profile
Settings
SYSTEM ADMINISTRATOR
0.65%
7.53%
19.06%
51.12%
UCY Datacenter Racks Memory
Rack4 Disk Reads
Rack4 Disk Writes
Rack1 Disk Reads
Rack1 Disk Writes
UCY Datacenter Network I/O
Rack4 Net Received
Rack4 Net Sent
Rack1 Net Received
Rack1 Net Sent
26.40 kWh
UCY DC1 Rack4 Energy kWh Average
96.82 kWh
UCY DC1 Rack1 Energy kWh Average
€ 2.82
UCY DC1 Rack4 Energy Cost
€ 10.35
UCY DC1 Rack1 Energy Cost
UCY Datacenter Racks Energy Consumed
Rack4 Energy kWh
Rack1 Energy kWh

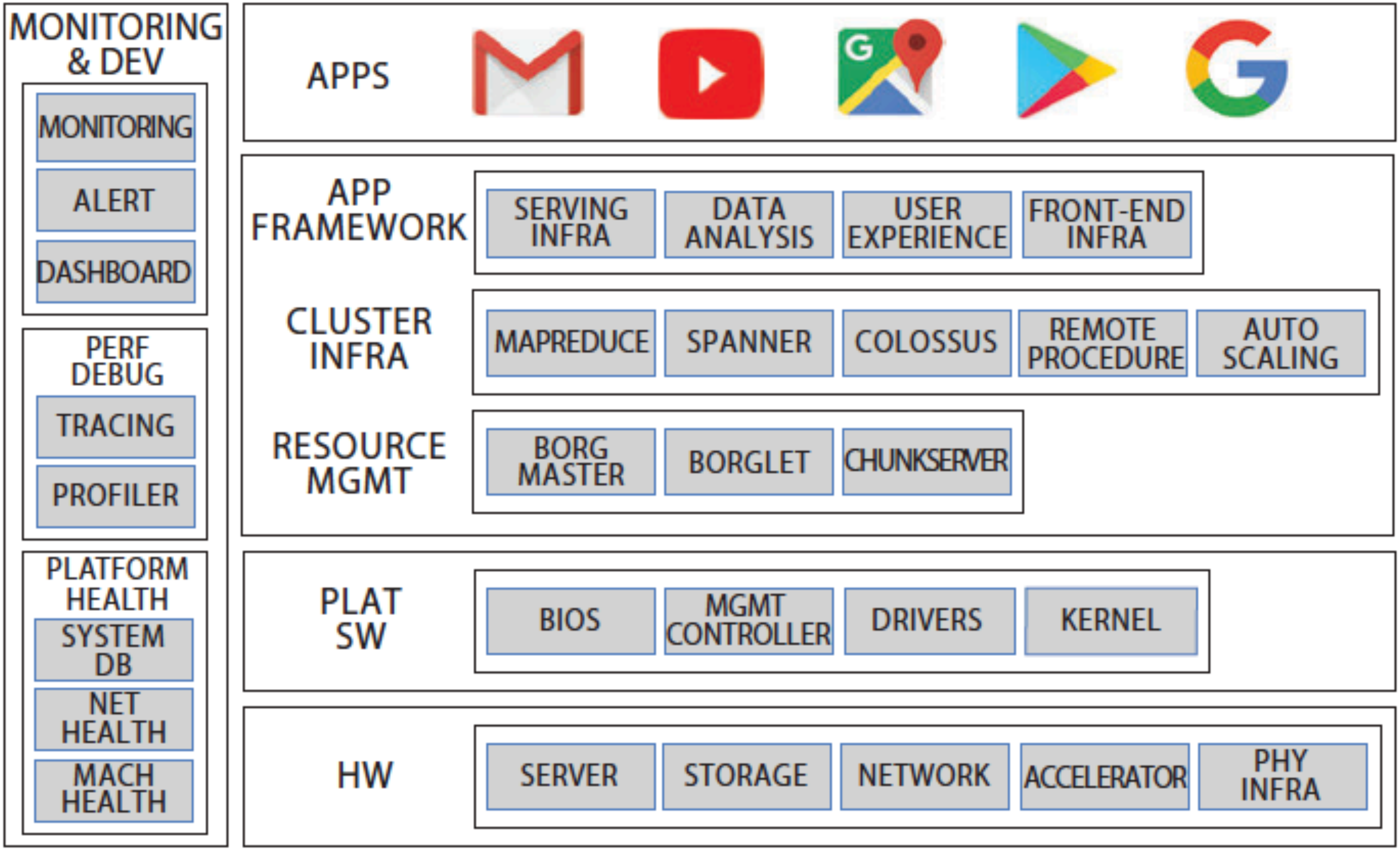
Performance Debugging

- Help operators and service designers understand the complex interactions between many programs, possibly running on hundreds of servers.
- Seek to determine the **root cause** of **performance anomalies** and identify **bottlenecks**.
- **No** need for **real-time operation**.
- Distributed system **tracing tools:**
  - **Black-box monitoring** systems: observe networking traffic among system components and infer **causal relationships** through **statistical inference**: WAP5, Sherlock
  - Application/middleware **instrumentation** systems: explicitly modify applications or middleware libraries for passing **tracing information** across machines and across module boundaries within machines. Log tracing information to local disks for subsequent collection by an **external performance analysis program:** Pip, Magpie, X-Trace, Dapper, profiler GWP

Platform-level Health Monitoring

- Tools that continuously and directly monitor the health of the computing platform - to understand and analyze hardware and system software failures.
- **Site Reliability Engineering**
  - most WSC deployments support "site reliability engineering," which is different from traditional system administration
  - SRE software engineers design **monitoring** and **infrastructure software** to **adjust** to **load variability** and **common faults automatically** so that humans are not in the loop and frequent incidents are self-healing.

Google Software Stack



MONITORING & DEV

- MONITORING
- ALERT
- DASHBOARD

PERF DEBUG

- TRACING
- PROFILER

PLATFORM HEALTH

- SYSTEM DB
- NET HEALTH
- MACH HEALTH

APPS

APP FRAMEWORK

- SERVING INFRA
- DATA ANALYSIS
- USER EXPERIENCE
- FRONT-END INFRA

CLUSTER INFRA

- MAPREDUCE
- SPANNER
- COLOSSUS
- REMOTE PROCEDURE
- AUTO SCALING

RESOURCE MGMT

- BORG MASTER
- BORGLET
- CHUNKSERVER

PLAT SW

- BIOS
- MGMT CONTROLLER
- DRIVERS
- KERNEL

HW

- SERVER
- STORAGE
- NETWORK
- ACCELERATOR
- PHY INFRA

Cloud Infrastructure Software

Basic Programming Concepts

Key Techniques

- Applied to achieve:
  - High Performance
  - High Availability
- Commonly applied in the design and implementation of **both infrastructure** and **application** level software systems

Performance and Availability Toolbox

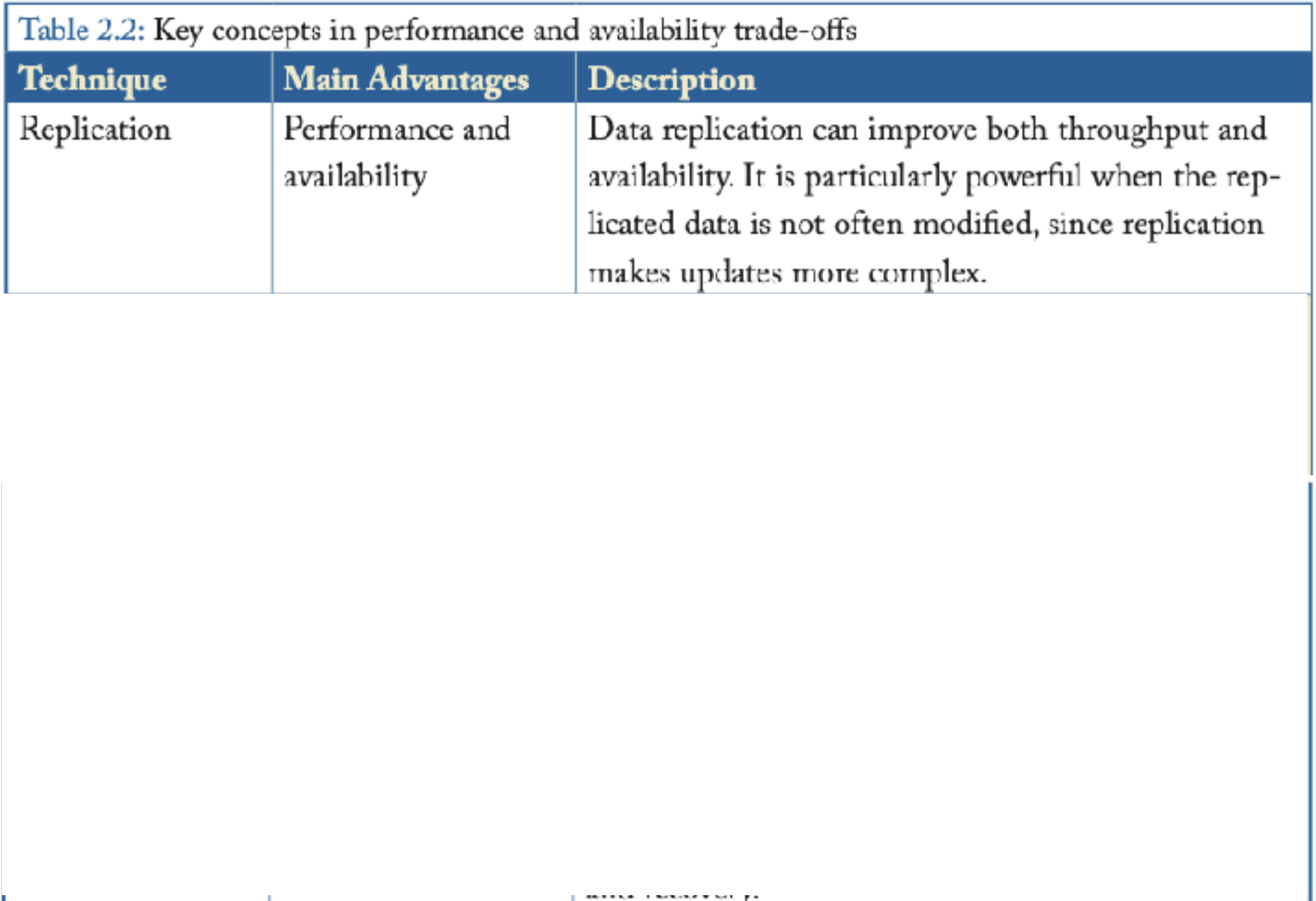


Table 2.2: Key concepts in performance and availability trade-offs

Technique	Main Advantages	Description
Replication	Performance and availability	Data replication can improve both throughput and availability. It is particularly powerful when the replicated data is not often modified, since replication makes updates more complex.

Performance and Availability Toolbox

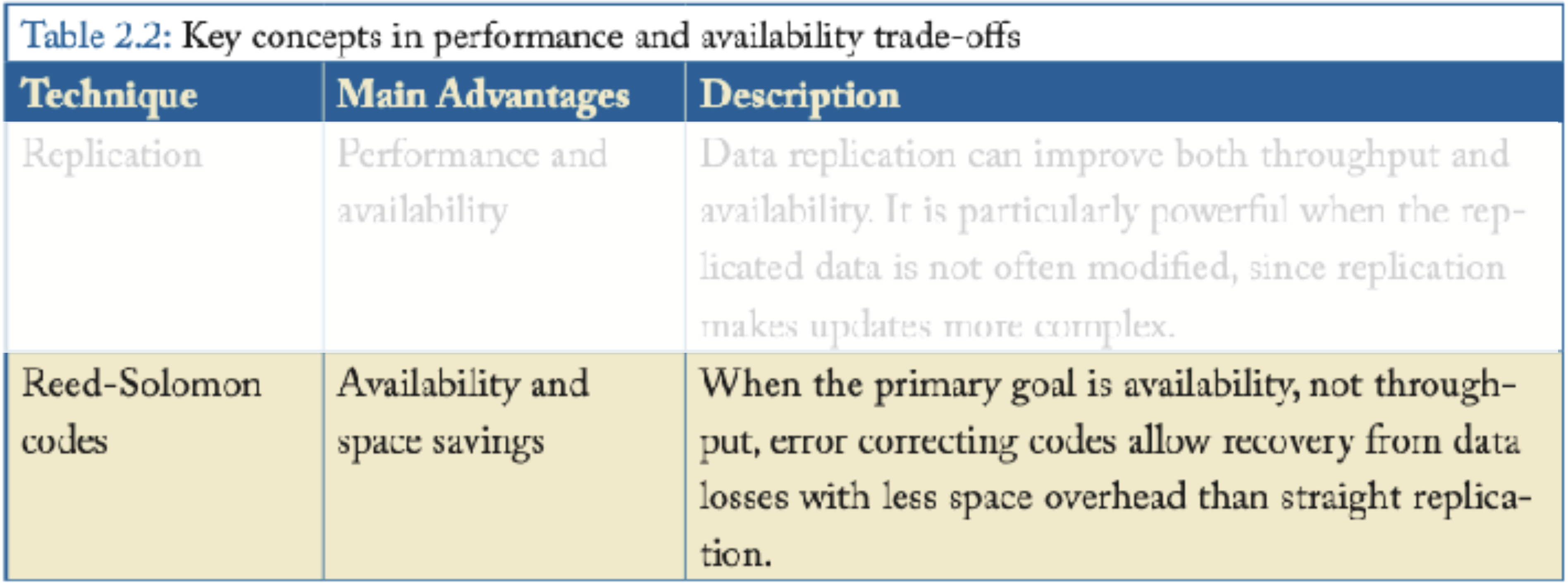


Table 2.2: Key concepts in performance and availability trade-offs

Technique	Main Advantages	Description
Replication	Performance and availability	Data replication can improve both throughput and availability. It is particularly powerful when the replicated data is not often modified, since replication makes updates more complex.
Reed-Solomon codes	Availability and space savings	When the primary goal is availability, not throughput, error correcting codes allow recovery from data losses with less space overhead than straight replication.

Performance and Availability Toolbox

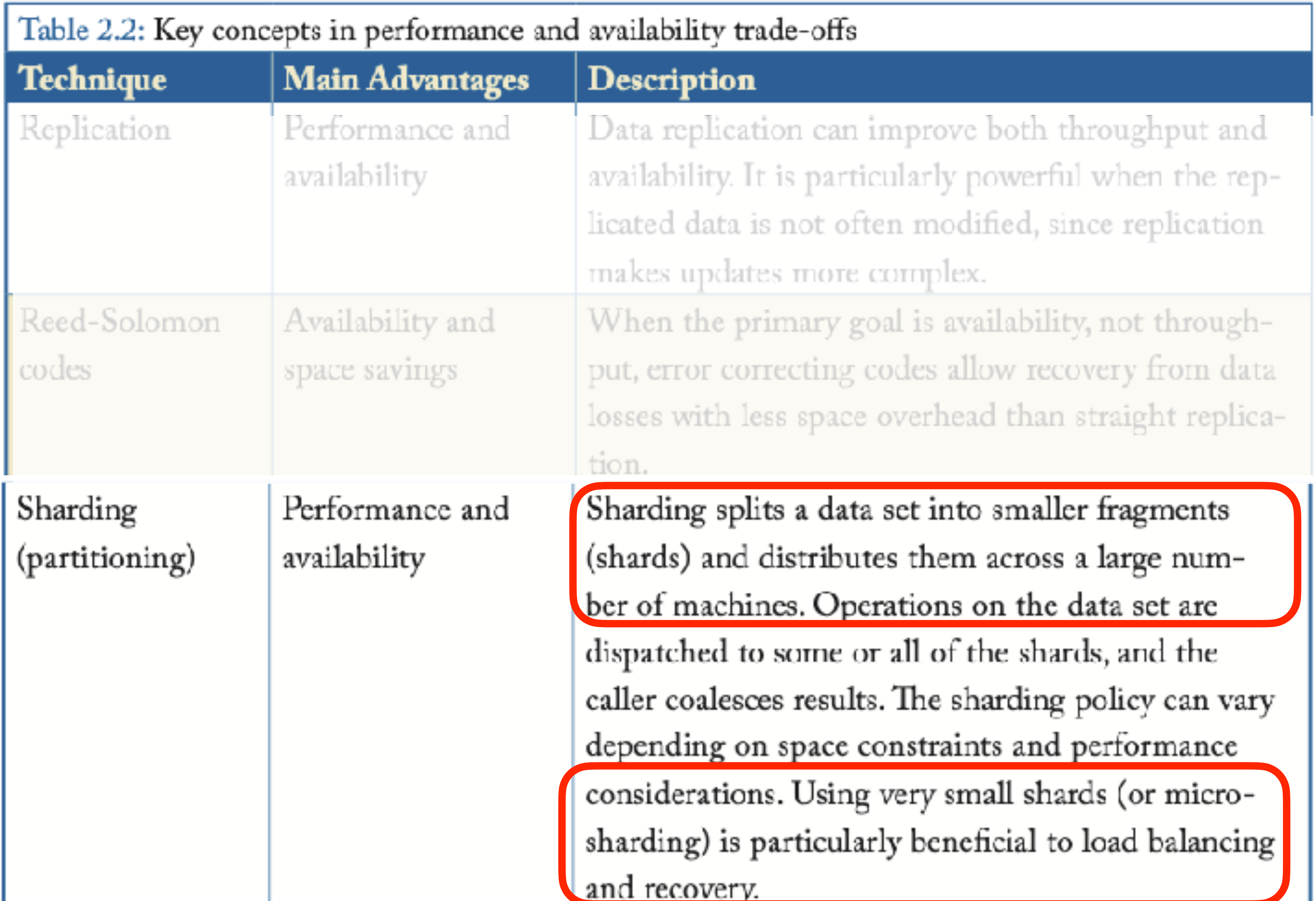
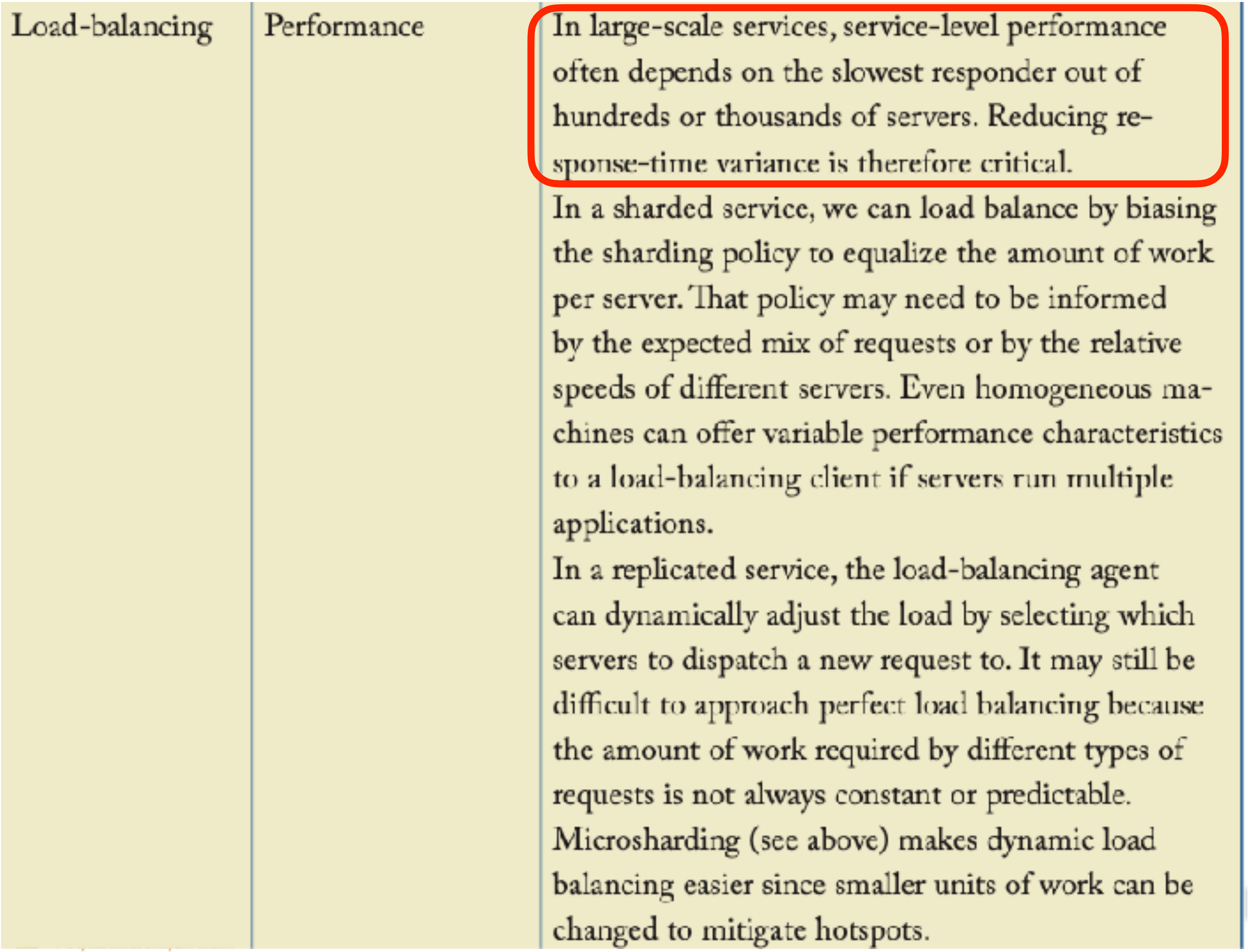
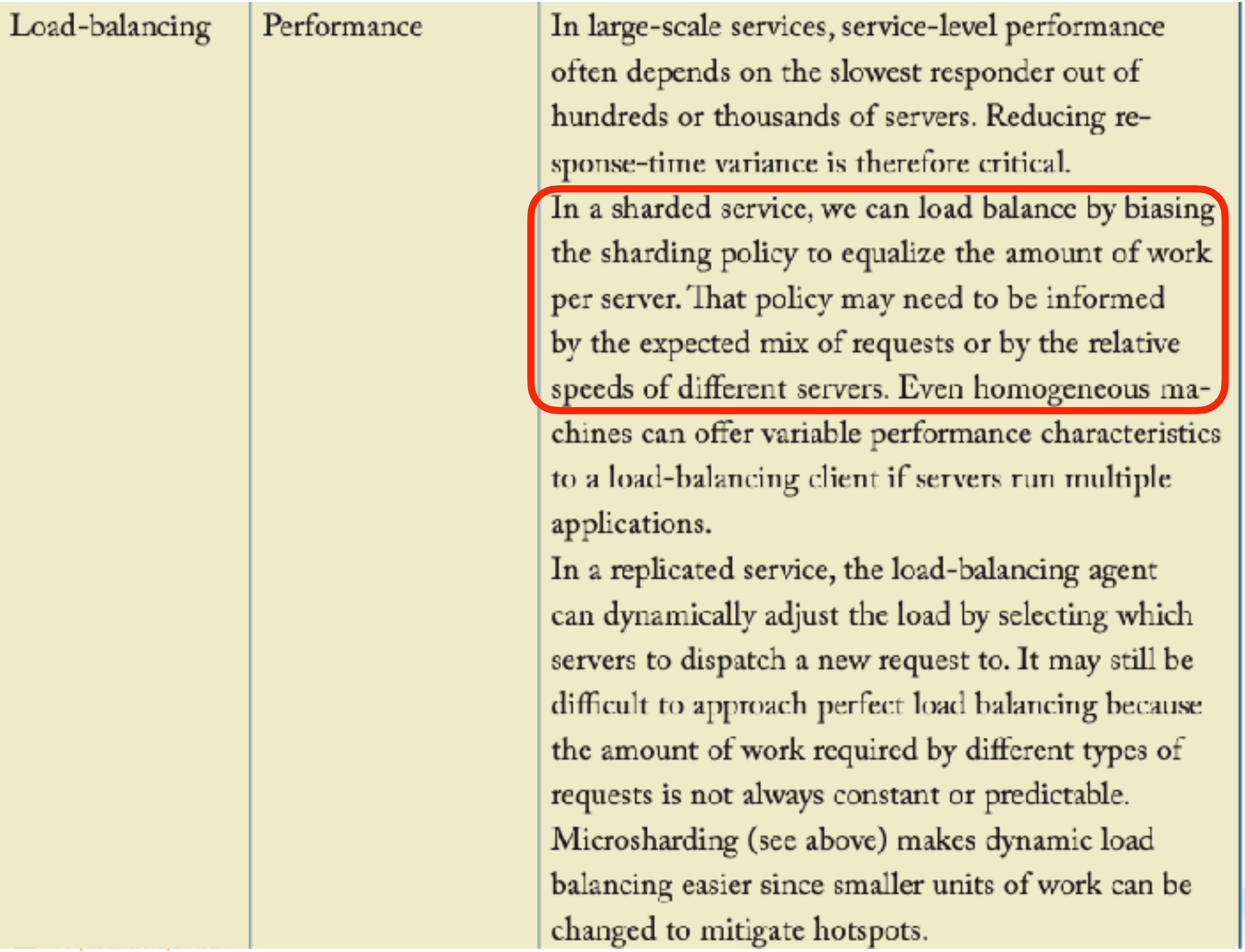


Table 2.2: Key concepts in performance and availability trade-offs

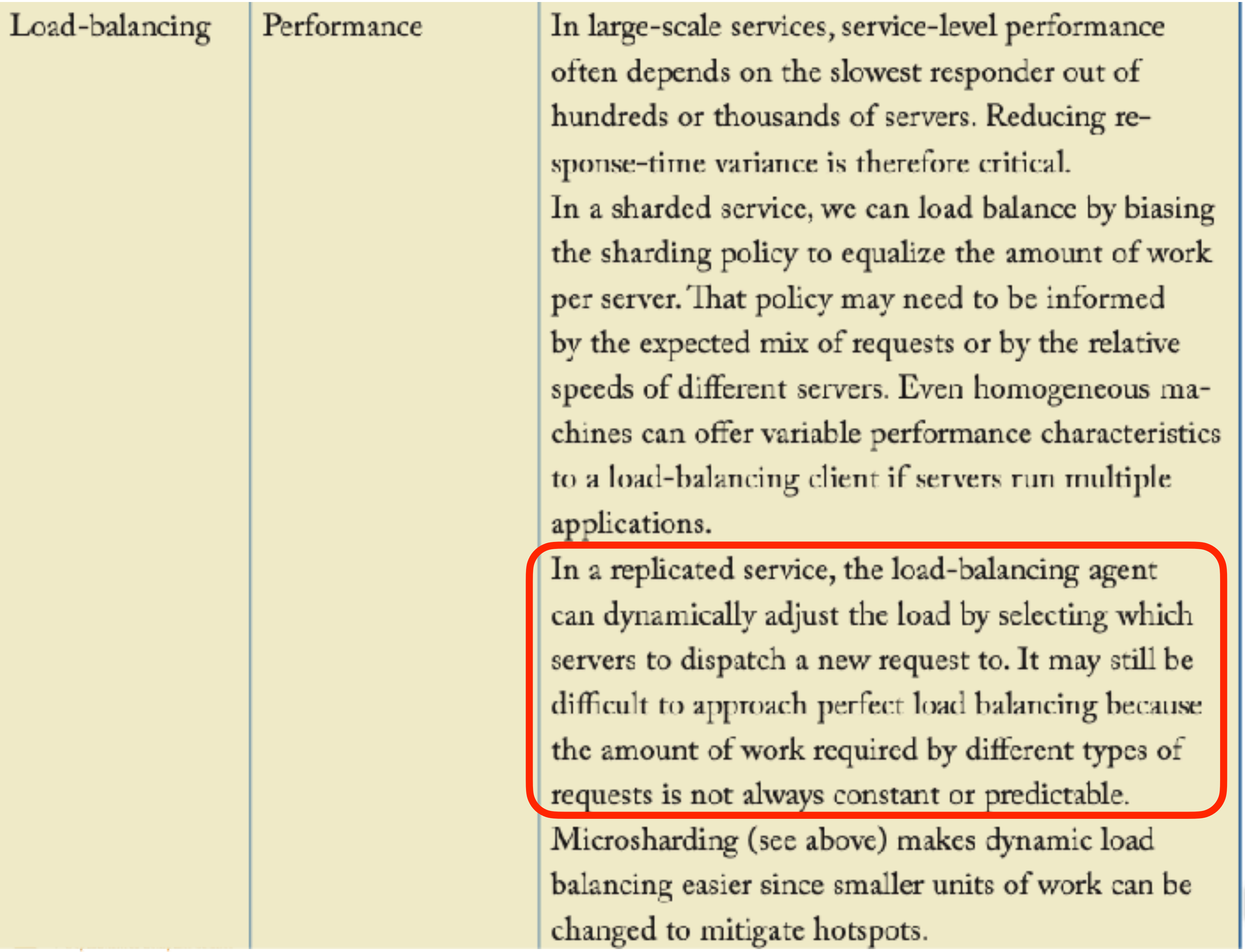
Technique	Main Advantages	Description
Replication	Performance and availability	Data replication can improve both throughput and availability. It is particularly powerful when the replicated data is not often modified, since replication makes updates more complex.
Reed-Solomon codes	Availability and space savings	When the primary goal is availability, not throughput, error correcting codes allow recovery from data losses with less space overhead than straight replication.
Sharding (partitioning)	Performance and availability	Sharding splits a data set into smaller fragments (shards) and distributes them across a large number of machines. Operations on the data set are dispatched to some or all of the shards, and the caller coalesces results. The sharding policy can vary depending on space constraints and performance considerations. Using very small shards (or microsharding) is particularly beneficial to load balancing and recovery.



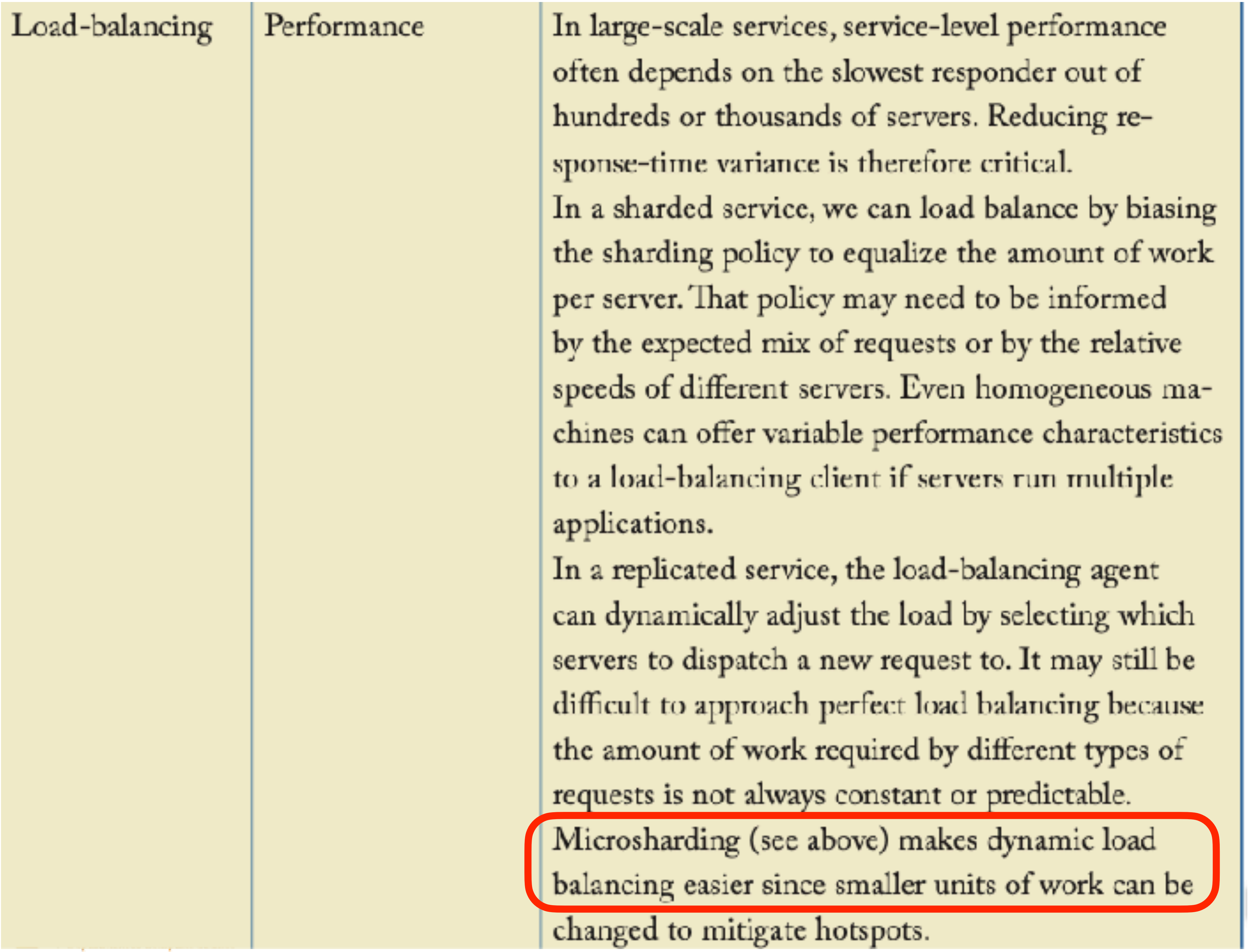
Load-balancing	Performance	In large-scale services, service-level performance often depends on the slowest responder out of hundreds or thousands of servers. Reducing response-time variance is therefore critical. In a sharded service, we can load balance by biasing the sharding policy to equalize the amount of work per server. That policy may need to be informed by the expected mix of requests or by the relative speeds of different servers. Even homogeneous machines can offer variable performance characteristics to a load-balancing client if servers run multiple applications. In a replicated service, the load-balancing agent can dynamically adjust the load by selecting which servers to dispatch a new request to. It may still be difficult to approach perfect load balancing because the amount of work required by different types of requests is not always constant or predictable. Microsharding (see above) makes dynamic load balancing easier since smaller units of work can be changed to mitigate hotspots.



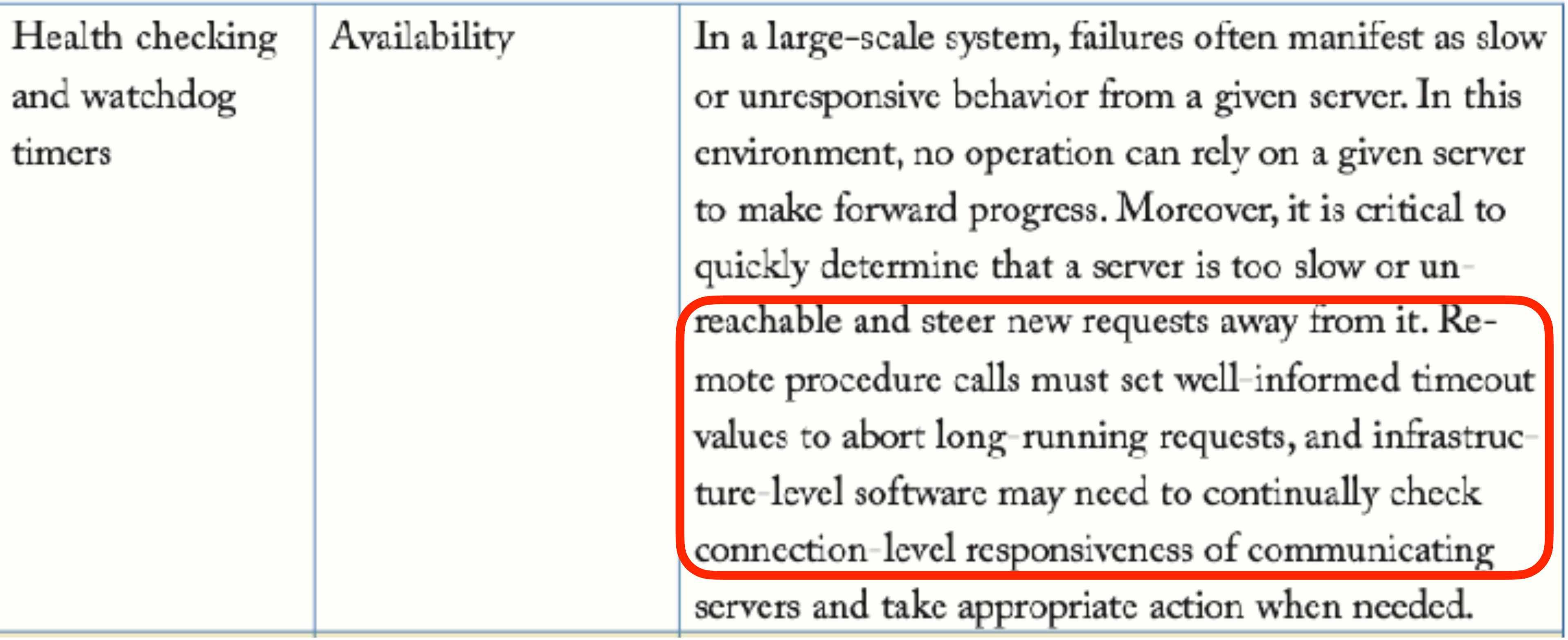
Load-balancing	Performance	In large-scale services, service-level performance often depends on the slowest responder out of hundreds or thousands of servers. Reducing response-time variance is therefore critical. In a sharded service, we can load balance by biasing the sharding policy to equalize the amount of work per server. That policy may need to be informed by the expected mix of requests or by the relative speeds of different servers. Even homogeneous machines can offer variable performance characteristics to a load-balancing client if servers run multiple applications. In a replicated service, the load-balancing agent can dynamically adjust the load by selecting which servers to dispatch a new request to. It may still be difficult to approach perfect load balancing because the amount of work required by different types of requests is not always constant or predictable. Microsharding (see above) makes dynamic load balancing easier since smaller units of work can be changed to mitigate hotspots.



Load-balancing	Performance	In large-scale services, service-level performance often depends on the slowest responder out of hundreds or thousands of servers. Reducing response-time variance is therefore critical. In a sharded service, we can load balance by biasing the sharding policy to equalize the amount of work per server. That policy may need to be informed by the expected mix of requests or by the relative speeds of different servers. Even homogeneous machines can offer variable performance characteristics to a load-balancing client if servers run multiple applications. In a replicated service, the load-balancing agent can dynamically adjust the load by selecting which servers to dispatch a new request to. It may still be difficult to approach perfect load balancing because the amount of work required by different types of requests is not always constant or predictable. Microsharding (see above) makes dynamic load balancing easier since smaller units of work can be changed to mitigate hotspots.

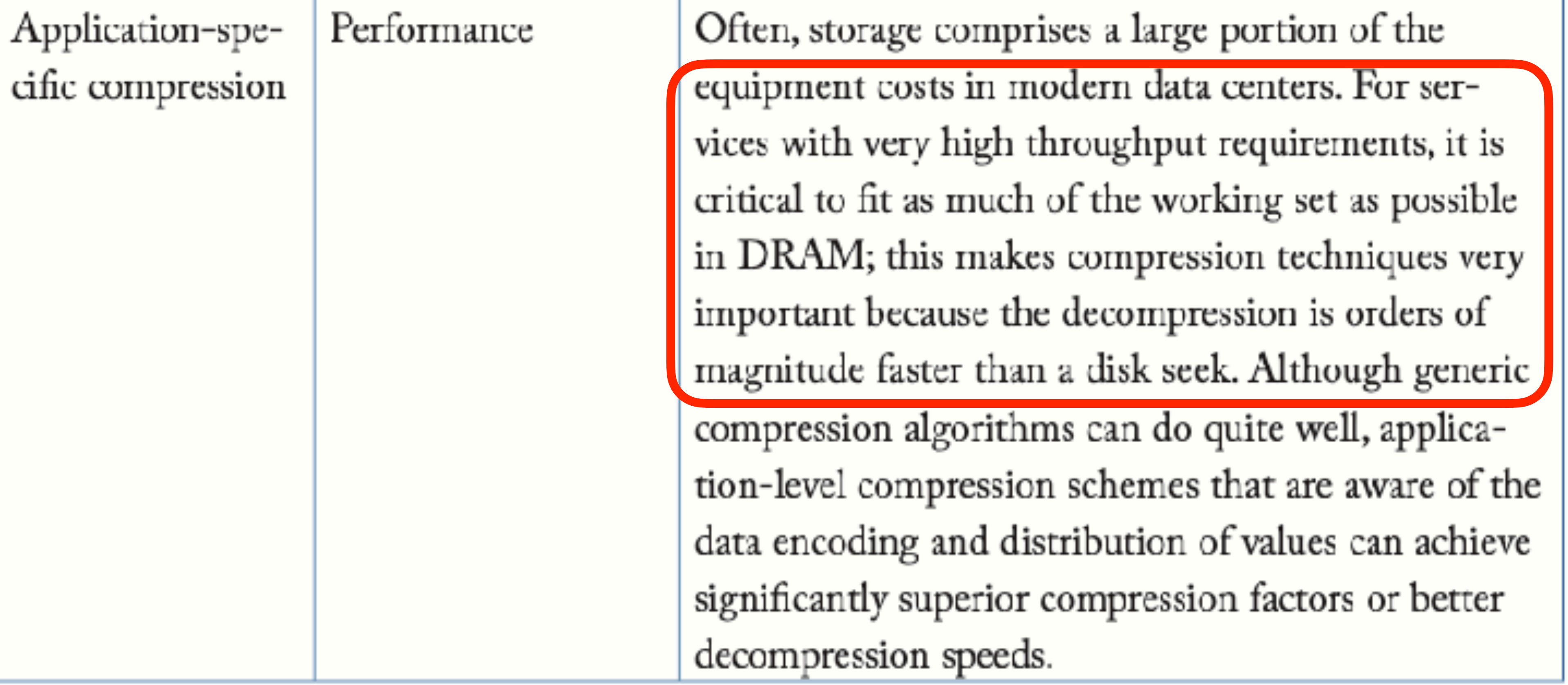


Load-balancing	Performance	In large-scale services, service-level performance often depends on the slowest responder out of hundreds or thousands of servers. Reducing response-time variance is therefore critical. In a sharded service, we can load balance by biasing the sharding policy to equalize the amount of work per server. That policy may need to be informed by the expected mix of requests or by the relative speeds of different servers. Even homogeneous machines can offer variable performance characteristics to a load-balancing client if servers run multiple applications. In a replicated service, the load-balancing agent can dynamically adjust the load by selecting which servers to dispatch a new request to. It may still be difficult to approach perfect load balancing because the amount of work required by different types of requests is not always constant or predictable. Microsharding (see above) makes dynamic load balancing easier since smaller units of work can be changed to mitigate hotspots.



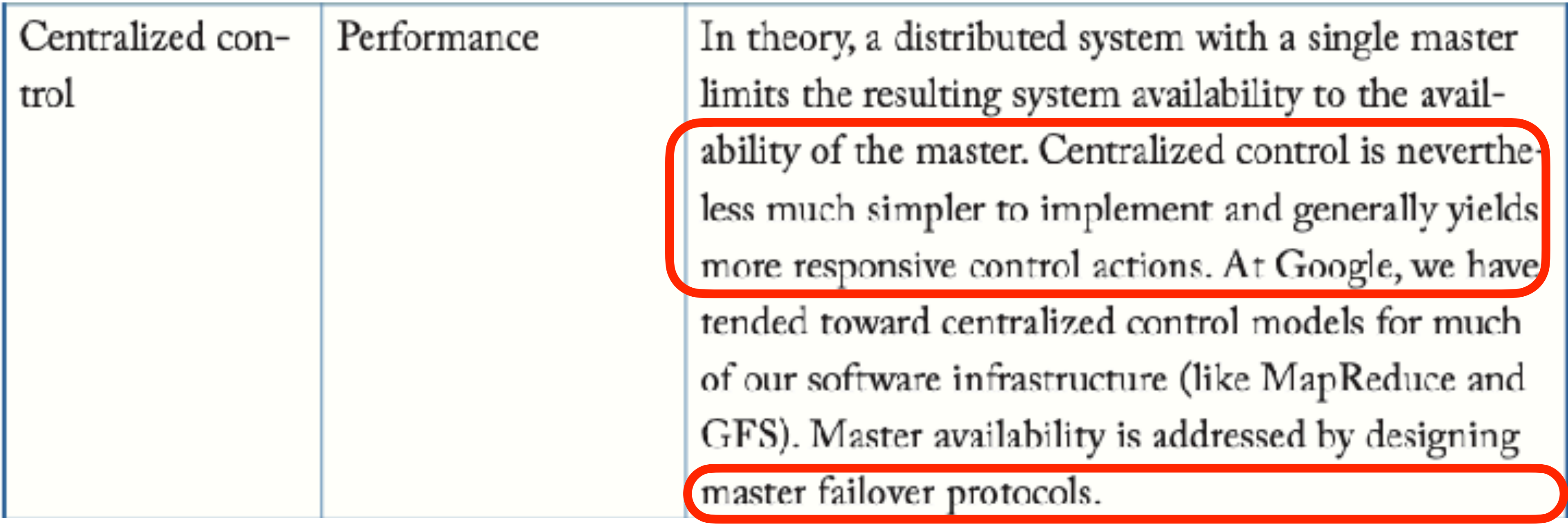
Health checking and watchdog timers	Availability	In a large-scale system, failures often manifest as slow or unresponsive behavior from a given server. In this environment, no operation can rely on a given server to make forward progress. Moreover, it is critical to quickly determine that a server is too slow or unreachable and steer new requests away from it. Remote procedure calls must set well-informed timeout values to abort long-running requests, and infrastructure-level software may need to continually check connection-level responsiveness of communicating servers and take appropriate action when needed.

Health checking and watchdog timers	Availability	In a large-scale system, failures often manifest as slow or unresponsive behavior from a given server. In this environment, no operation can rely on a given server to make forward progress. Moreover, it is critical to quickly determine that a server is too slow or unreachable and steer new requests away from it. Remote procedure calls must set well-informed timeout values to abort long-running requests, and infrastructure-level software may need to continually check connection-level responsiveness of communicating servers and take appropriate action when needed.
Integrity checks	Availability	In some cases, besides unresponsiveness, faults manifest as data corruption. Although those may be rare, they do occur, often in ways that underlying hardware or software checks do not catch (for example, there are known issues with the error coverage of some networking CRC checks). Extra software checks can mitigate these problems by changing the underlying encoding or adding more powerful redundant integrity checks.

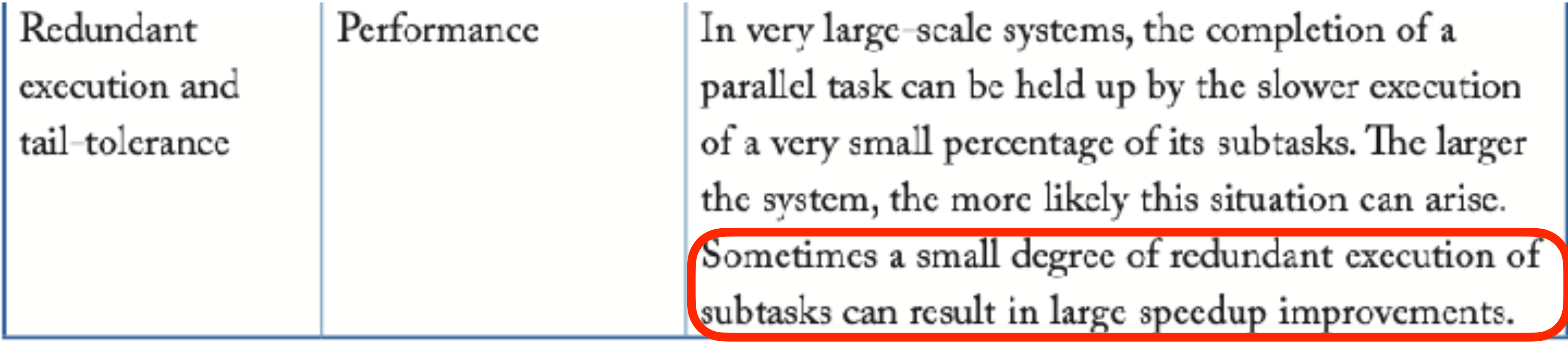


Application-specific compression	Performance	Often, storage comprises a large portion of the equipment costs in modern data centers. For services with very high throughput requirements, it is critical to fit as much of the working set as possible in DRAM; this makes compression techniques very important because the decompression is orders of magnitude faster than a disk seek. Although generic compression algorithms can do quite well, application-level compression schemes that are aware of the data encoding and distribution of values can achieve significantly superior compression factors or better decompression speeds.

Application-specific compression	Performance	Often, storage comprises a large portion of the equipment costs in modern data centers. For services with very high throughput requirements, it is critical to fit as much of the working set as possible in DRAM; this makes compression techniques very important because the decompression is orders of magnitude faster than a disk seek. Although generic compression algorithms can do quite well, application-level compression schemes that are aware of the data encoding and distribution of values can achieve significantly superior compression factors or better decompression speeds.
Eventual consistency	Performance and availability	Often, keeping multiple replicas up-to-date using the traditional guarantees offered by a database management system significantly increases complexity, hurts performance, and reduces availability of distributed applications [Vog08]. Fortunately, large classes of applications have more relaxed requirements and can tolerate inconsistent views for limited periods, provided that the system eventually returns to a stable consistent state.



Centralized control	Performance	In theory, a distributed system with a single master limits the resulting system availability to the availability of the master. Centralized control is nevertheless much simpler to implement and generally yields more responsive control actions. At Google, we have tended toward centralized control models for much of our software infrastructure (like MapReduce and GFS). Master availability is addressed by designing master failover protocols.



Redundant execution and tail-tolerance	Performance	In very large-scale systems, the completion of a parallel task can be held up by the slower execution of a very small percentage of its subtasks. The larger the system, the more likely this situation can arise. Sometimes a small degree of redundant execution of subtasks can result in large speedup improvements.

Cloud Infrastructure Software

Cloud Software

Cloud Computing in Data Centers

- Cloud computing provides:
  - efficiency
  - flexibility
  - cost savings.
- The **cost efficiency** achieved through co-locating multiple VMs on the same physical hosts to increase utilization.
- At a high level, a VM is similar to other online web services, built on top of cluster-level software to leverage the entire warehouse data center stack.
- A VM-based workload model simplifies the migration of on-premise computing to WSCs.
- However, on WSC there are **additional challenges**:
  - I/O **virtualization overheads**
  - **availability model**, and
  - **resource isolation**.

Cloud challenges: I/O Virtualization

- A VM does not have direct access to hardware resources like local hard drives or networking.
  - All I/O requests go through an abstraction layer, such as *virtio*, in the guest operating system.
  - The **hypervisor** or **virtual machine monitor** (VMM) translates the I/O requests into the appropriate operations:
    - storage requests are redirected to the network persistent disk or local SSD drives
    - networking requests are sent through virtualized network for encapsulation.
- I/O virtualization often incurs some performance overhead, but:
  - improvements in *virtualization techniques* and *hardware support* for virtualization have steadily reduced these overheads

Cloud challenges: Availability Model

- Large-scale distributed services achieve high availability by:
  - running multiple instances of a program within a data center,
  - at the same time maintaining N + 1 redundancy at the data center level to minimize the impact of scheduled maintenance events.
- However, for many enterprise applications horizontal scaling is often not possible (e.g. due to older relational DBs).
- Using **live migration** technology can help ensure high availability:
  - moving running VMs out of the way of planned maintenance events, including system updates and configurations changes.

Cloud challenges: Resource Isolation

- Variability in latency of individual components is amplified at scale at the service level due to **interference effects.**
  - In cloud computing, malicious VMs can exploit multi-tenant features to cause severe contention on shared resources, conducting Denial of Service (DoS) and side-channel attacks.
  - This makes it particularly important to balance the tradeoff between security guarantees and resource sharing.

Cloud Infrastructure Software, Workloads, and Metrics

Cloud Application Workloads

Cloud Native

- Cloud-native architecture and technologies are an approach to **designing**, **constructing**, and **operating workloads** that are **built in the cloud** and **take full advantage of the cloud computing model**.
- Cloud-native technologies empower organizations to:
  - Build loosely coupled systems that are resilient, manageable, and observable
  - Run scalable applications in dynamic public, private, and hybrid clouds
  - Combine development with robust automation, which allows high-impact changes frequently and predictably with minimal toil
  - Accelerate business transformation to achieve high velocity and growth.

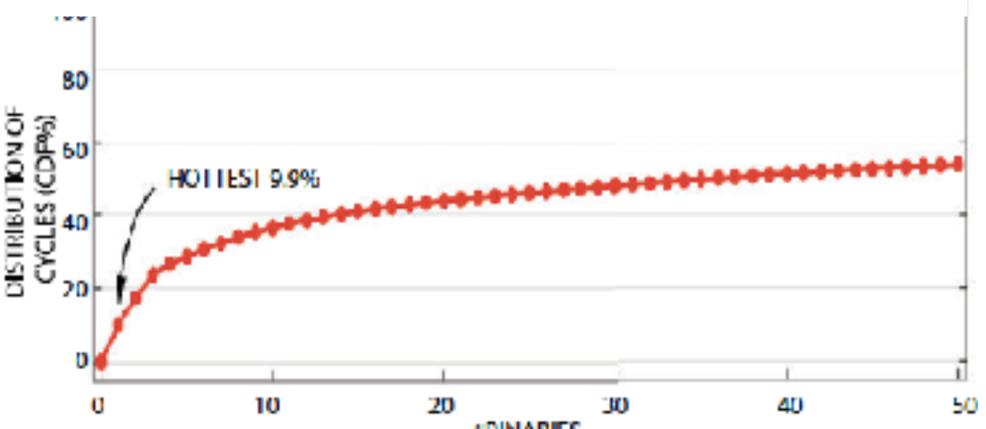
Cloud Native Software

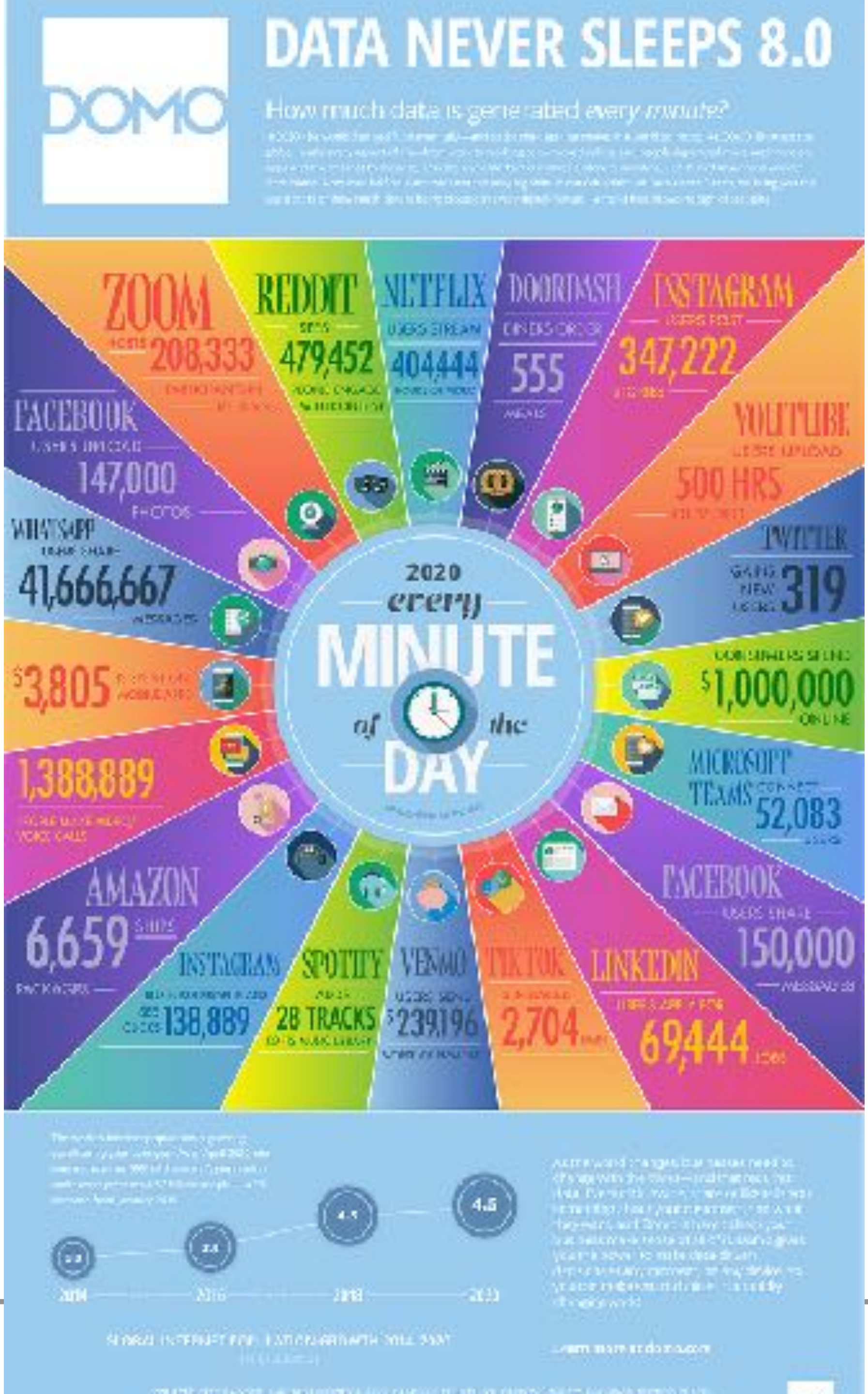
- **Cloud "Native"** ethos emphasizes speed and agility properties:
  - highly dynamic environment
  - API-driven self-service operation
  - instantaneous, on-demand resource allocation.
- Realization of these properties thanks to **containers** and **orchestrators**, allow developers to build software that emphasizes **scalability** and **automation**, and **minimizes operational complexity** and **toil**.
- Other technologies are frequently adopted at the same time:
  - **Microservices**: decomposition of larger, often monolithic, applications into smaller, limited-purpose applications that cooperate via strongly defined APIs, but can be managed, versioned, tested, and scaled independently.
  - **Service meshes**: allow application operators to decouple management of the application itself from management of the networking that surrounds it. Service discovery systems allow applications and microservices to find each other in volatile environments, in real time.

Workload Diversity

- Breadth of services offered results in **diversity** in application-level requirements. E.g.
  - Google Search: non need for high-performance atomic updates; inherently forgiving of hardware failures
  - AdSense: clicks on ads are small financial transactions, which need many of the guarantees expected from a transactional database management system.
- Many workloads exist with **spread-out popularity**:
  - **Top 50** workloads at Google account for only about **60% of the total WSC cycles**, with a **long tail** accounting for the rest of the cycles
- Speed of workload **churn**: product requirements evolve rapidly.

Designing DCs as special-purpose hardware **not an option.**




DOMO
DATA NEVER SLEEPS 8.0
How much data is generated every minute?
ZOOM
HOSTS 208,333
PARTICIPANTS IN MEETINGS
REDDIT
SEES 479,452
PEOPLE ENGAGE WITH CONTENT
NETFLIX
USERS STREAM
404,444
HOURS OF VIDEO
DOORDASH
DINERS ORDER
555
MEALS
INSTAGRAM
USERS POST
347,222
STORIES
YOUTUBE
USERS UPLOAD
500 HRS
OF VIDEO
FACEBOOK
USERS UPLOAD
147,000
PHOTOS
WHATSAPP
USERS SHARE
41,666,667
MESSAGES
TWITTER
GAINS NEW USERS 319
CONSUMERS SPEND
$1,000,000
ONLINE
$3,805 IS SPENT ON MOBILE APPS
1,388,889
PEOPLE MAKE VIDEO/VOICE CALLS
MICROSOFT TEAMS CONNECT
52,083
USERS
AMAZON
6,659 SHIPS
PACKAGES
FACEBOOK
USERS SHARE
150,000
MESSAGES
INSTAGRAM
138,889
CLICKS
SPOTIFY
ADDS
28 TRACKS
TO ITS MUSIC LIBRARY
VENMO
USERS SEND
$239,196
WORTH OF PAYMENTS
TIKTOK
IS INSTALLED
2,704 TIMES
LINKEDIN
USERS APPLY FOR
69,444 JOBS
2020
every
MINUTE
of the
DAY
4.5
2014
2016
2018
2020
GLOBAL INTERNET POPULATION GROWTH 2014-2020
Learn more at domo.com

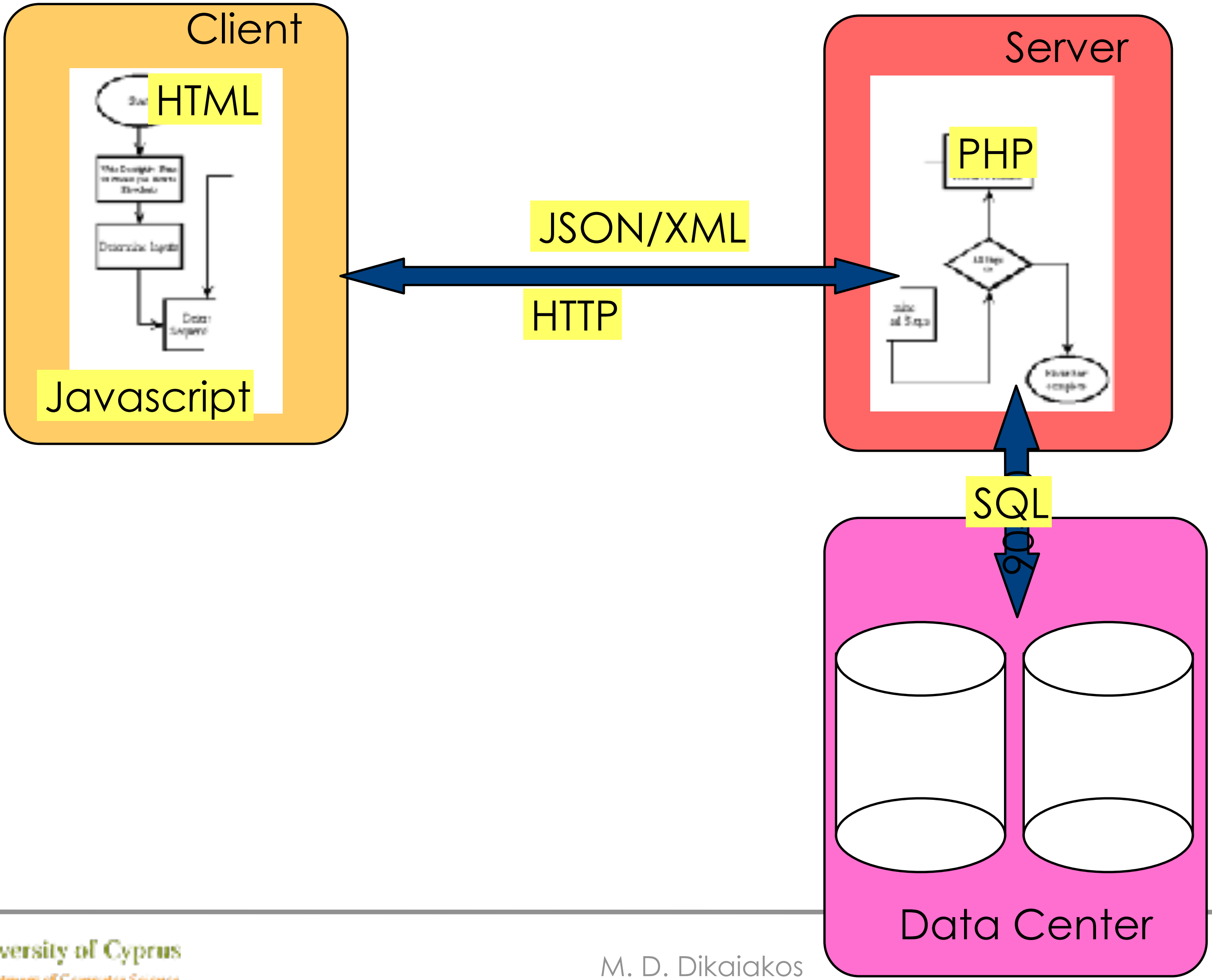
Cloud Application Case Studies

- Web Applications
- Web Search Engine
- Video streaming
- Machine Learning

Cloud Application Workloads

Giant-Scale Web Services

Web Application



Key assumptions for giant-scale services

- Service provider has limited control over the clients and the IP network
- Queries drive the service [e.g. HTTP get]
- Read-only queries greatly outnumber updates (queries that affect the persistent data store)

Advantages

- Access anywhere, anytime. A ubiquitous infrastructure facilitates access from home, work, airport, and so on.
- Availability via multiple devices. Infrastructure handles most of the processing => users can access services from "thin clients", which can offer far more functionality for a given cost and battery life.
- Groupware support. Centralizing data from many users allows service providers to offer group-based applications (calendars, teleconferencing systems, group-management systems).
- Lower overall cost. Infrastructure services have a fundamental cost advantage over designs based on stand-alone devices:
  - can be multiplexed across active users;
  - end-user devices have very low utilization (less than 4 percent), while infrastructure resources often reach 80 percent utilisation (moving anything from the device to the infrastructure effectively improves efficiency by a factor of 20);
  - centralizing the administrative burden and simplifying end devices also reduce overall cost.
- Simplified service updates. Most powerful long-term advantage is the ability to upgrade existing services or offer new services without the physical distribution required by traditional applications and devices.

Architecture

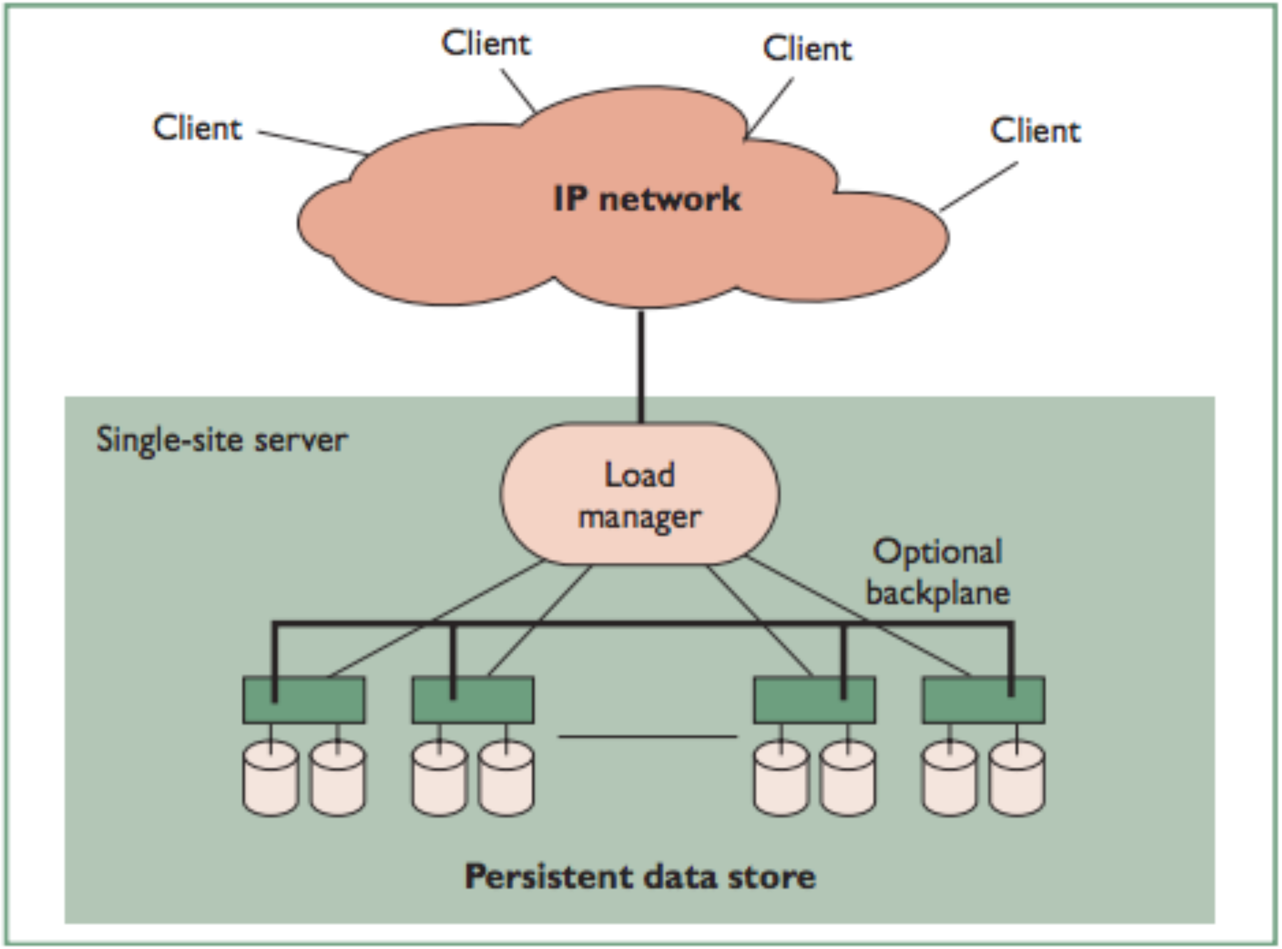


Figure 1. The basic model for giant-scale services. Clients connect via the Internet and then go through a load manager that hides down nodes and balances traffic.

"Lessons from giant-scale services," E. A. Brewer, IEEE Internet Comput., vol. 5, no. 4, pp. 46–55, Jul. 2001.

Basic Components of the Model

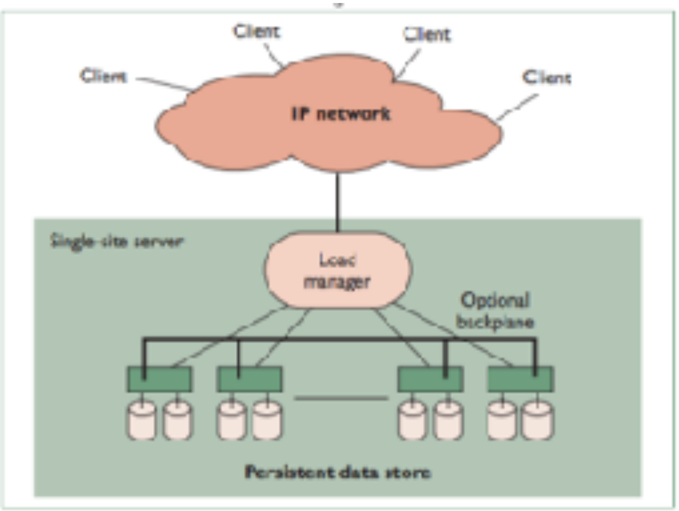
- **Clients** (πελάτες), such as Web browsers, standalone email readers, agents initiating queries to the services.
- The best-effort **IP network:** provides access to the service.
- **Load manager** (εξισορροπητής φορτίου):
  - provides a level of indirection between the service's external name and the servers' physical names (IP addresses) to preserve the external name's availability in the presence of server faults
  - balances load among active servers.

  Traffic might flow through proxies or firewalls before the load manager.
- **Servers** (εξυπηρετητές/διακο□ιστές/διαθέτες): the system's workers, combining CPU, memory, and disks into an easy-to-replicate unit.
- The **persistent data store** (βάση δεδο□ένων): a **replicated** or **partitioned** "database" that is spread across the servers' disks. It might also include network attached storage such as external DBMSs or systems that use RAID storage.
- Optional **backplane:** a system-area-network handling inter server traffic:
  - redirecting client queries to the correct server
  - coherence traffic for the persistent data store.
- **Auxiliary Systems** (not shown): Nearly all services have several other service-specific pieces that we can largely ignore in the basic model. Examples include user-profile databases, ad servers, site management tools, and support for logging and log analysis. Many of these subsystems can be viewed as additional sets of nodes with their own persistent data store.

Key Challenges

Main challenges to deploying giant-scale services:

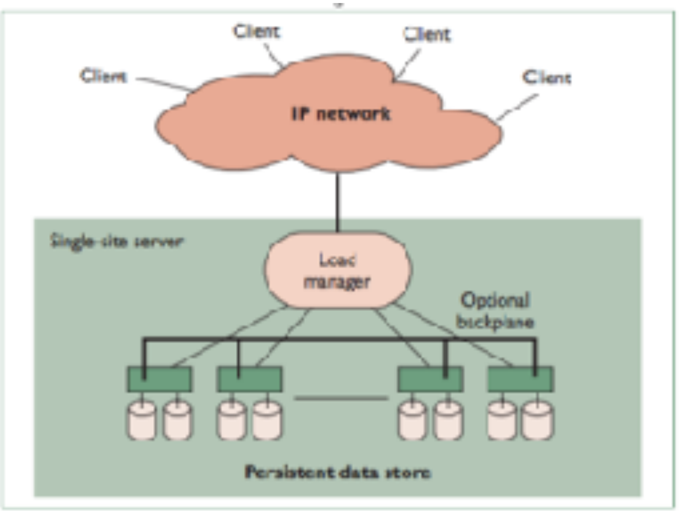
- Load Management
- High Availability
- Evolution and Growth



Key Challenges

Main challenges to deploying giant-scale services:

- **Load Management**
- High Availability
- Evolution and Growth



Load balancing (εξισορρόπηση φορτίου)

- **Goal:** balanced distribution of incoming load to available servers.
- Load Balancer Functionality:
- Provide the External Name: can be a domain name or a set of IP addresses depending on the approach.
  - Challenge: make the external name **highly available despite failure** of some of the nodes and the corresponding loss of their internal names.
- Load Balance the Traffic: This can be done with or without feedback from the nodes.
  - Goals: **higher overall utilization** and **better average response time**.
- Isolate Faults From Clients: detect faults and dynamically change the routing of traffic to avoid down nodes.
  - Key metric: **reaction time**, as some clients lose service until the detection and failover occurs

Load balancing

- **Key issue:** does the manager understands the distribution of data across the nodes.
- There are three choices:
  - **Symmetric**: all nodes are equal in functional capability (but perhaps not query capacity). Symmetric nodes greatly simplify load management because any query can go to any node.
  - **Asymmetric**: In this case, nodes vary in functionality and the load manager must correctly map each query to a node that can handle it. The common case is a partitioned database, in which the load manager must understand the partitioning to route the queries correctly.
  - **Symmetric with Affinity**: This is the symmetric case with an optimization for locality. Due to caching effects, it is very useful to try to partition the queries even in the symmetric case so that a given node tends to get the same queries repeated.
    - Although the manager understands the partitioning, it is not required for correctness and any node can still handle any query.

Round-robin DNS

- Have DNS distribute different IP addresses for a single domain name among clients in a rotating fashion ("round-robin DNS"):
- Roughly balancing the load at the time of DNS lookup, but providing little support for availability when a node fails:
  - Node failures reduce system capacity, but not data availability
- **Persistent data store** is implemented by simple replication of all content to all nodes: which works well when the total amount of content is small.
- No need for a **backplane**, since all servers can handle all queries and there is no coherence traffic.

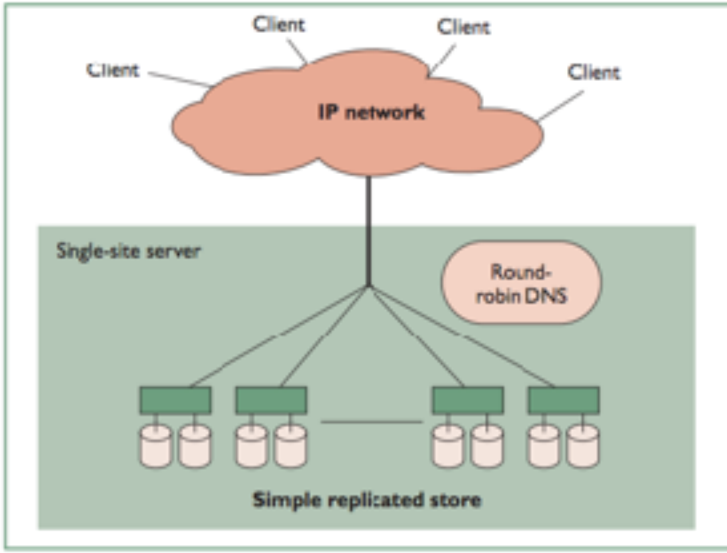
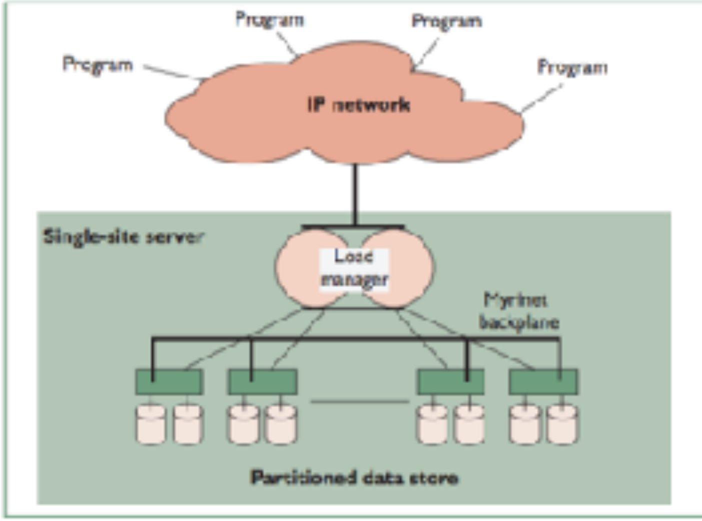


Figure 2. A simple Web farm. Round-robin DNS assigns different servers to different clients to achieve simple load balancing. Persistent data is fully replicated and thus all nodes are identical and can handle all queries.

Load Balancer Switch

- Load management implemented in a "**layer 4" switch**:
  - Rewrites TCP connections from external IP addresses to one of the internal node names.
  - Balances load based on outstanding connections and can respond quickly to failed nodes by avoiding them for new connections
- "**Layer-7**" (application layer) switches can be used instead, parsing HTTP requests and URLs at wire speed, and routing queries to the proper nodes.
- Load balancer is in the path of the traffic and therefore has to be fault tolerant. Eg.
  - Pair of "level 4" switches that automatically **failover** to each other.



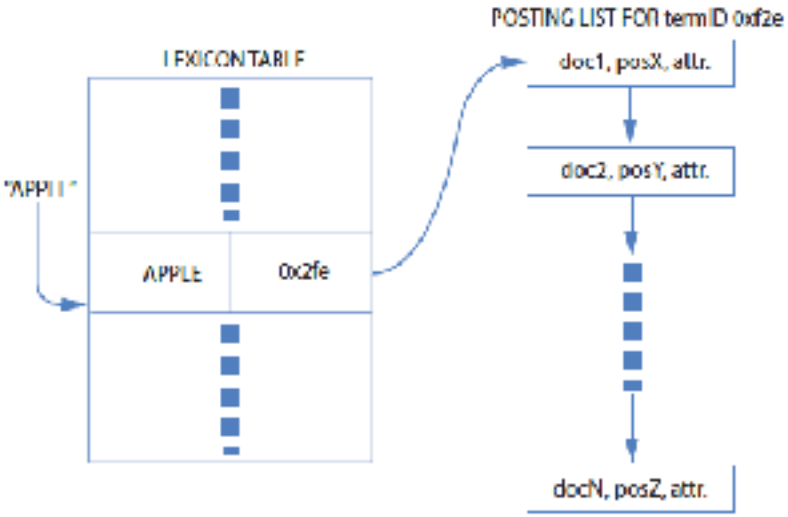
- **Persistent store** partitioned across the nodes, possibly without any replication:
  - **Node failures** reduce effective store size and its overall capacity.
  - **Nodes no longer identical** - some queries may need to go to specific nodes.
  - **Backplane** enables queries to get to the right node or nodes:
    - Sometimes the load manager can pick the right node directly, but this requires service-specific and query-specific knowledge in the load manager.
    - If service uses caching of data from other nodes, the backplane is used for the cache coherence traffic.

Cloud Application Workloads

Web Search

Workload example: Web search

- Searching for needles in a haystack
- If we assume the web to contain **100 billion documents**, with an average document size of **4 KB** (after compression), the haystack is about **400 TB**.
- The database for web search is an **index** built from that repository by inverting that set of documents to create a repository in the logical format



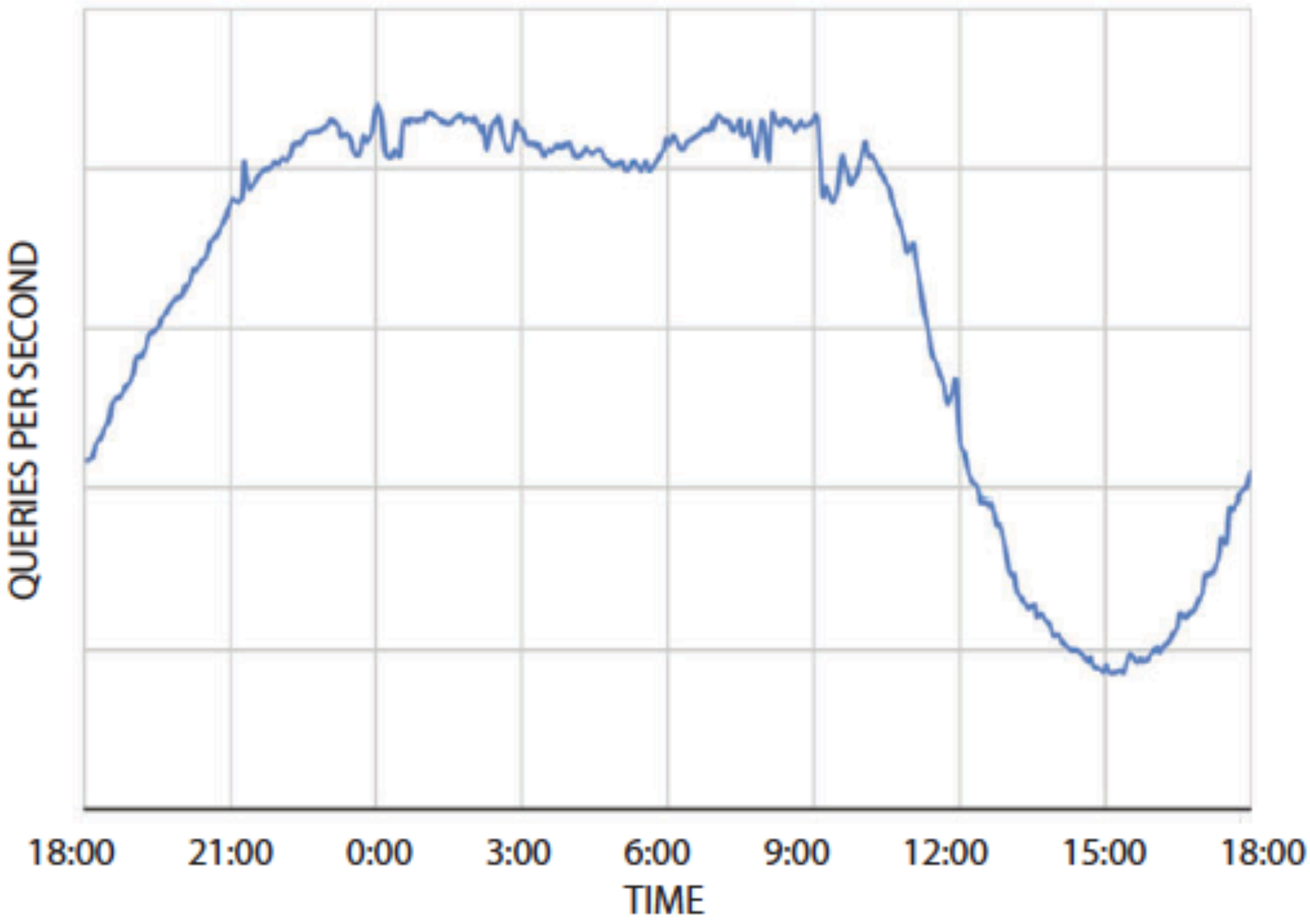
- The search algorithm must traverse the posting lists for each query term until it finds all documents contained in all three posting lists.
- Then, it ranks the documents found using a variety of parameters and returns the highest ranked documents to the user.

Web search Algorithm

- Split huge inverted index (**sharding**) into **load-balanced subfiles** and **distribute them across thousands of machines**.
  - For throughput or fault tolerance, multiple copies of index subfiles can be placed in different machines
- A user query is received by a **front-end web server** and distributed to the machines in the index cluster.
- Index-serving machines compute local results, pre-rank them, and send their best results to the front-end system (or some intermediate server), which selects the best results.
- At this point, only the list of doc_IDs corresponding to the resulting web page hits is known.
- To compute the actual title, URLs, and a query-specific document snippet that gives the user some context, the list of doc_IDs is sent to a set of machines containing copies of the documents themselves.
- A repository this size needs to be partitioned (sharded) and placed in a large number of servers.

Web search Performance

- Total user-perceived latency for operations described above needs to be a **fraction of a second**: **heavy emphasis on latency reduction**.
- High **throughput** is also a key performance metric because a popular service may need to support **many thousands of queries per second**.
- Index is updated frequently, but can be considered a **read-only** structure.
- No need for index lookups in different machines to communicate with each other except for the final merge step: computation **is very efficiently parallelized**.
- No logical interactions across different web search queries: room for **further parallelism**.
- If index sharded by doc_ID, workload has **relatively small networking requirements** in terms of **average bandwidth**: data exchanged between machines is typically the size of the queries themselves
- Some **bursty behavior:** servers at the front-end act as traffic amplifiers as they distribute a single query to a very large number of servers ==> burst of traffic not only in the request path but possibly also on the response path.
- Diurnal fluctuation of requests: traffic at peak usage hours can be more than twice as high as off-peak periods


QUERIES PER SECOND
18:00
21:00
0:00
3:00
6:00
9:00
12:00
15:00
18:00
TIME

Cloud Application Case Studies

- Web Applications
- Web Search Engine
- Video streaming
- Machine Learning

Cloud Application Workloads

Video Streaming

Workload example: video

- IP video traffic represented 73% of the global internet in 2016, and is expected to grow to **83% by 2022** [circa 2017]
  - Live video will grow **15-fold** between 2016 and 2021 [Cisco].
  - Consumer Internet video traffic will grow 4.3-fold from 2017 to 2022, a compound annual growth rate of 34%.
  - In July 2015, users were uploading 400 hr of video per minute to YouTube, and in February 2017 users were watching 1 billion hours of YouTube video per day.
- **Video transcoding** is a crucial part of any video sharing infrastructure.
- **Video serving** cost components:
  - **Computing costs** for video transcoding,
  - **Storage costs** for the video catalog (both originals and transcoded versions)
  - **Network egress** costs for sending transcoded videos to end users.

Workload example: video

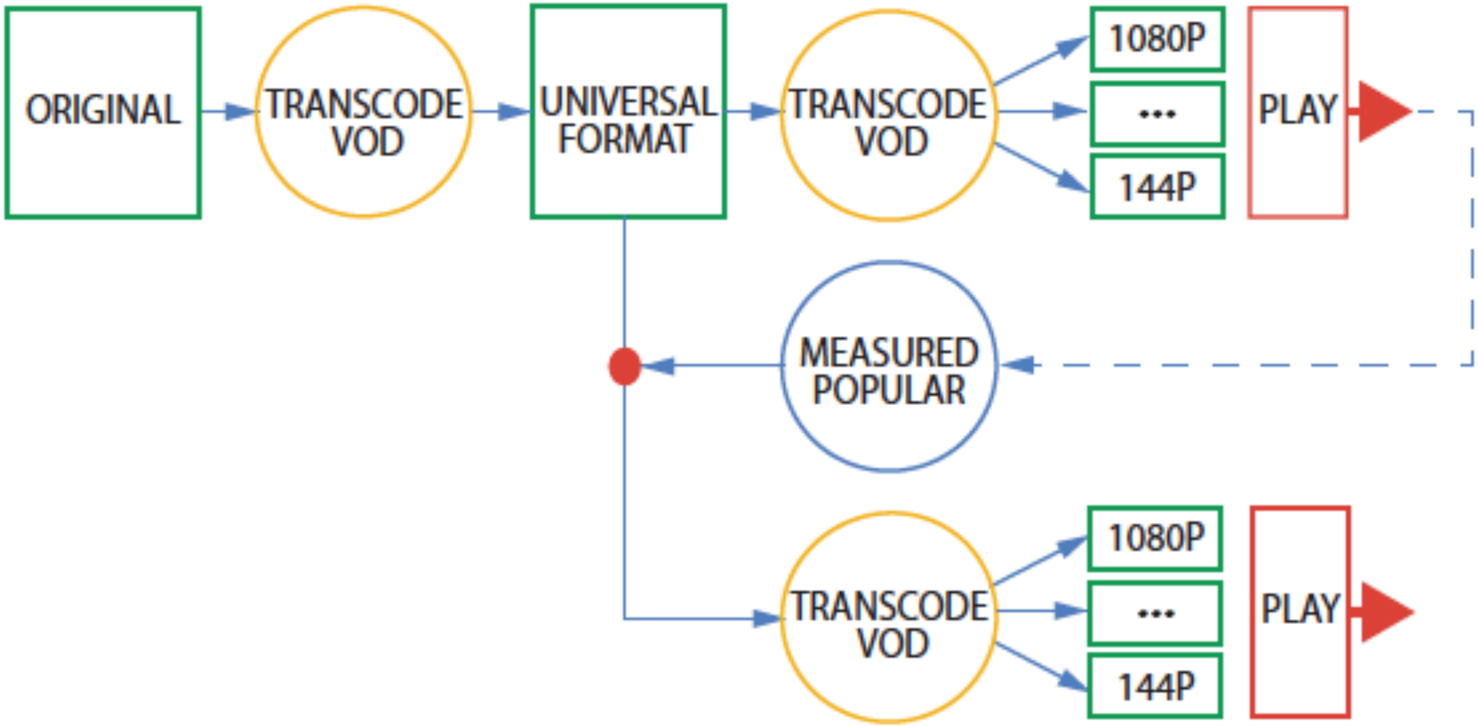


Figure 2.5: The YouTube video processing pipeline. Videos are transcoded multiple times depending on their popularity. VOD = video on demand.

Once video chunks are transcoded to their playback ready formats in the data center, they are distributed through the Edge network, which caches the most recently watched videos to minimize latency and amplify egress bandwidth.

Cloud Application Case Studies

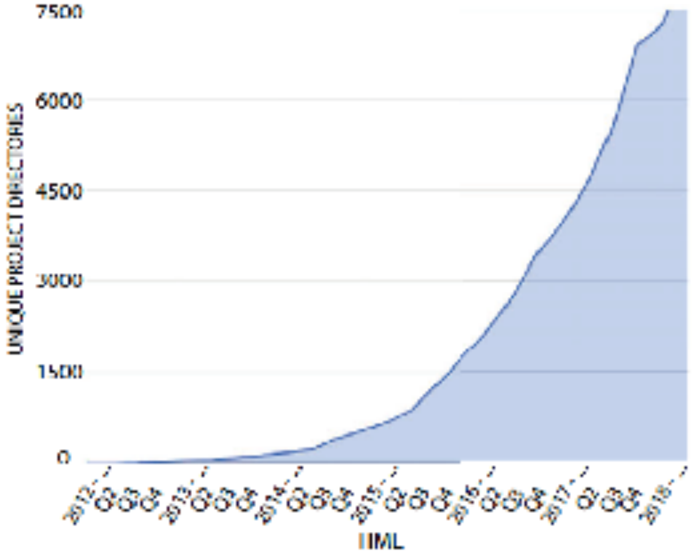
- Web Applications
- Web Search Engine
- Video streaming
- Machine Learning

Cloud Application Workloads

Machine Learning

Workload example: ML

- Impressive growth of ML use
- Deep Neural Networks (DNNs) applied to a wide range of applications including speech, vision, language, translation, search ranking, and many more.
- DNN application phases: **training** (or learning) and **inference** (or prediction),
- DNN workloads further classified: convolutional, sequence, embedding-based, multilayer perceptron, and reinforcement learning.



: 2.6: Growth of machine learning at Google.

Training workload

- Determines the weights or parameters of a DNN, adjusting them repeatedly (multiple *epochs*) until the DNN produces the desired results:
  - **iterative**, **Floating point** arithmetic
- Multiple learners process subsets of input training set and reconcile the parameters across the learners either through *parameter servers* or *reduction across learners*:
  - **parallelism**.
- Training can be done **asynchronously**, or **synchronously**.
- Synchronous provides **better model quality**;
  - but, the training performance is limited by the slowest learner.

Inference workload

- Inference uses DNN model developed during the training phase to make predictions on data:
  - **user facing**
  - with **strict latency constraints**
  - **floating point** (single precision, half precision) or
  - **quantized** (8-bit, 16-bit) computation
- **Lower precision** inference enables **lower latency** and **improved power efficiency**.

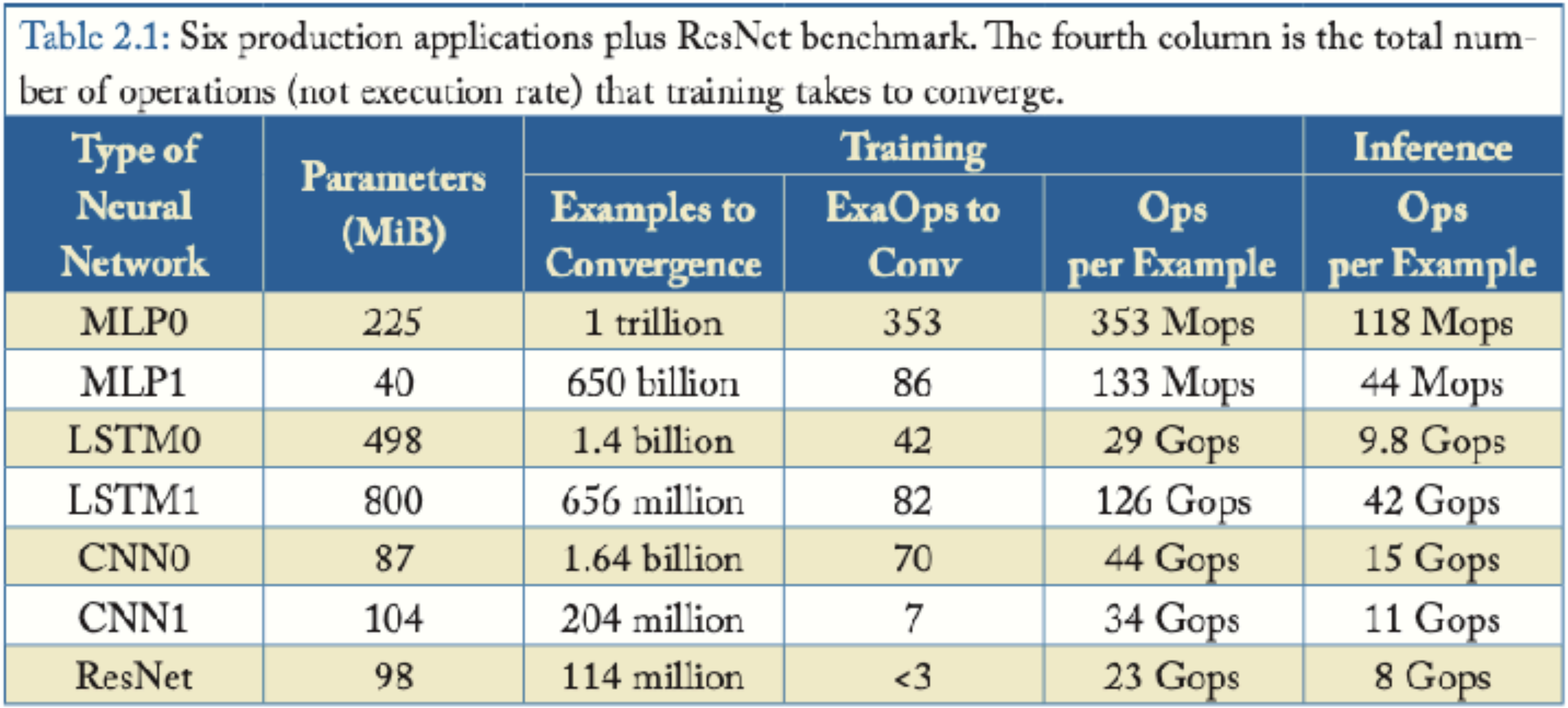


Table 2.1: Six production applications plus ResNet benchmark. The fourth column is the total number of operations (not execution rate) that training takes to converge.

Type of Neural Network	Parameters (MiB)	Training			Inference
		Examples to Convergence	ExaOps to Conv	Ops per Example	Ops per Example
MLP0	225	1 trillion	353	353 Mops	118 Mops
MLP1	40	650 billion	86	133 Mops	44 Mops
LSTM0	498	1.4 billion	42	29 Gops	9.8 Gops
LSTM1	800	656 million	82	126 Gops	42 Gops
CNN0	87	1.64 billion	70	44 Gops	15 Gops
CNN1	104	204 million	7	34 Gops	11 Gops
ResNet	98	114 million	<3	23 Gops	8 Gops

MiB means 2^{20};

Previous Class

- Discussed the elements of the software stacks typically running on top of WSC hardware:
  - Platform-level software: OS, VMMs
  - Cluster-level software infrastructure: resource managers, distributed file systems, schedulers and remote procedure call (RPC) libraries; programming models
  - Application-level software
  - Monitoring and development software
- Presented some common techniques and concepts applied to achieve high performance and availability in cloud infrastructures: load-balancing, replication, redundancy, encoding, etc.
- Reviewed the algorithmic patterns and characteristics of typical large-scale workloads running on cloud infrastructures: Giant-scale Internet Services, Web Search, Video streaming, ML

University of Cyprus
Department of Computer Science

Cloud Infrastructure Software, Workloads, and Metrics

Availability and Performance

Availability and Performance

Faults and Failures

Why care about failures?

Faults and Failures

- The sheer scale of WSCs requires that internet services software tolerates relatively high component fault rates.
  - Disk drives can exhibit annualized failure rates > 4%.
- Different deployments have reported between **1.2 and 16** average server-level restarts per year.
- With such high component failure rates, an application running across thousands of machines may need to react to failure conditions **on an hourly basis**.

Faults and Failures

- **Hourly Cost of Downtime** now exceeds $300,000 for 91% of SME and large enterprises.
- Overall, 44% of mid-sized and large enterprise survey respondents reported that **a single hour of downtime**, can potentially cost their businesses over one million ($1 million).
- Catastrophic outage that interrupts a major business transaction or occurs during peak business hours can exceed millions of dollars per minute.

ITIC Annual Hourly Cost of Downtime survey, 2022

Availability builds Trust

- Users **trust** that the services they increasingly **rely on** will be **always available**.
- This expectation translates into a **high-reliability** requirement for building-sized computers.
- Determining the appropriate level of reliability is fundamentally a **tradeoff** between the **cost of failures** (including repairs) and the **cost of preventing** them.

Hardware vs. Software

- In WSCs, *hardware reliability* alone cannot deliver *sufficient availability.*
- **Why?**
- How do we measure server reliability?
  - **Mean Time Between Failure (MTBF)**
  - **Mean Time to Repair (MTTR)**

MTBF in WSC with ideal servers

- Assume ultra-reliable servers with **MTBF = 30 years** =10000 days (unrealistic)
- Assume a **10000-server** WSC.
- *How many server failures expected on average*, per day?
  - **1**
- If an application running on the WSC depends on the availability of the entire cluster, what would be its MTBF?
  - **Less than a day**

Realistic prospects

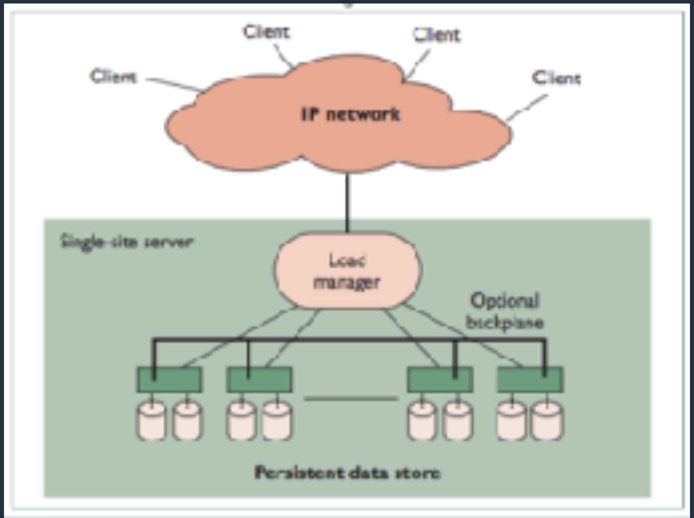
- Server-MTBF much less than 30 years.
- Real-life WSC cluster MTBF in the range of a few hours between failures.
- Software infrastructure and application software quite complex, not bug-free, lead to failures too.
- WSC applications must work around failed servers in software, either with:
  - code in the application itself or
  - via functionality provided by middleware.

Availability, Unavailability & Failure

- A system's **Availability** is the fraction of time during which it is available for use;
- **Unavailability** is the fraction of time during which the system is not available for some reason.
- **Failures** are one cause of unavailability, but are often much less common than other causes such as **planned maintenance** for hardware or **software upgrades**. Thus:
  - a system with zero failures may still have availability of less than 100%, and
  - a system with a high failure rate may have better availability than one with low failures, if other sources of unavailability dominate.

Availability and Performance

Availability in Giant-scale Services



High Availability (υψηλή διαθεσι□ότητα)

- Major driving requirement behind giant-scale system design, in the presence of component failures, natural disasters, and also constantly evolving features and unpredictable growth.

- **Availability Metrics** (□ετρικές):
  - Uptime
  - Yield
  - Harvest

Uptime (λειτουργικός χρόνος)

uptime = (MTBF – MTTR)/MTBF

- Fraction of time a site is handling traffic
- Typically measured in nines - traditional infrastructure systems aim for 4 to 5 nines (0.9999 to 0.99999)

downtime = MTTR/MTBF

Yield (απόδοση)

yield = queries completed/queries offered

- Fraction of queries that are completed
  - Some queries are dropped because the system does not have enough capacity to serve them
- Typically, yield is numerically close to Uptime.
  - Why?

WHICH ONE IS MORE USEFUL IN PRACTICE, AS A METRIC: YIELD OR UPTIME?

- **Yield** maps directly to **user experience**
- Yield correctly reflects that **not all seconds have equal value**:
  - being down for a second when there are no queries has no impact on users or yield, but reduces uptime
  - being down for one second during peak and off-peak times has equal impact on uptime but vastly different yields

How to Improve Uptime?

uptime = (MTBF – MTTR)/MTBF

- Reduce frequency of errors
  - increase MTBF
- Reduce time fixing errors
  - decrease MTTR
- Which approach is preferable?
  - Giant-scale systems should focus on improving MTTR and simply apply best effort to MTBF
  - Why?

Harvest (συγκο□ιδή)

- In systems based on queries, we can also measure **query completeness:**
  - How much of the database is reflected in the answer
  - Or how many features supported by a service are operational

harvest = data available/complete data

DQ (data per query) Principle

DQ = *total amount of data that has to be moved per second on average*

Data per query x **queries per second** **~ constant**

- DQ bounded by the underlying **physical limitation of the hardware**

- The DQ value is measurable and tuneable

- At the **high utilization level** typical of giant-scale systems, the DQ value approaches this limit

Utility of DQ

- Principle rather than a literal truth:
  - the system's overall capacity tends to have a particular physical bottleneck (στενωπός), such as total I/O bandwidth or total seeks per second
- Absolute value of DQ not that important: relative value under various changes provides a useful guide:
  - **Best possible result** under multiple faults is a **linear reduction in DQ**.
  - DQ often **scales linearly with the number of nodes:**
    - Early tests on single nodes tend to have predictive power for overall cluster performance.
  - All proposed hardware/software changes can be evaluated by their **DQ impact**.
  - We can translate future traffic and feature predictions into **future DQ requirements** and thus into hardware and software targets.

Measuring and Tuning DQ

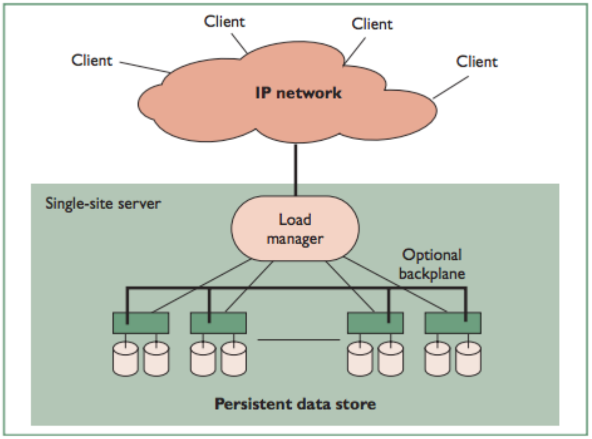


Figure 1. The basic model for giant-scale services. Clients connect via the Internet and then go through a load manager that hides down nodes and balances traffic.

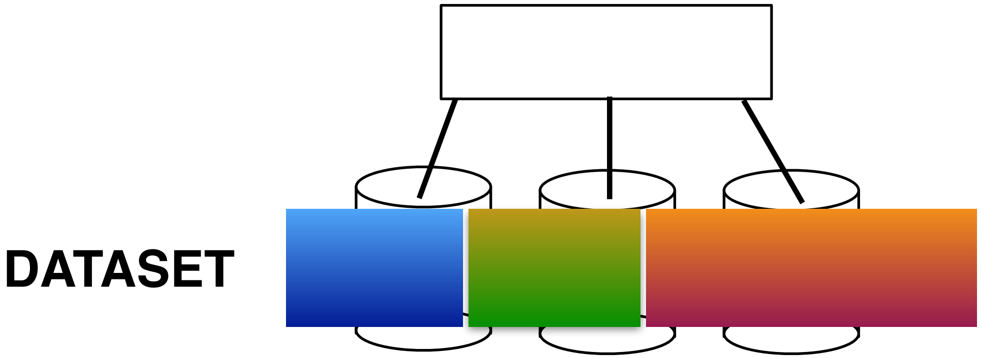
http://www.seleniumhq.org/

How do we measure the DQ of an infrastructure?

- Define target workload (φορτίο)
- Use a load generator to measure a given combination of hardware, software and db size against this workload
- Given the metric and the load generator, it is easy to measure relative impact of faults

- How do we improve DQ?
  - DQ scales linearly with the number of nodes
  - We can translate future traffic predictions into future DQ requirements and this into hardware and software target - convert traffic predictions into capacity planning decisions

Improving Availability



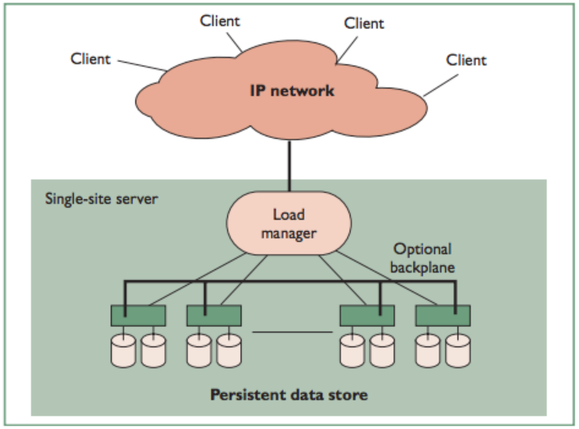
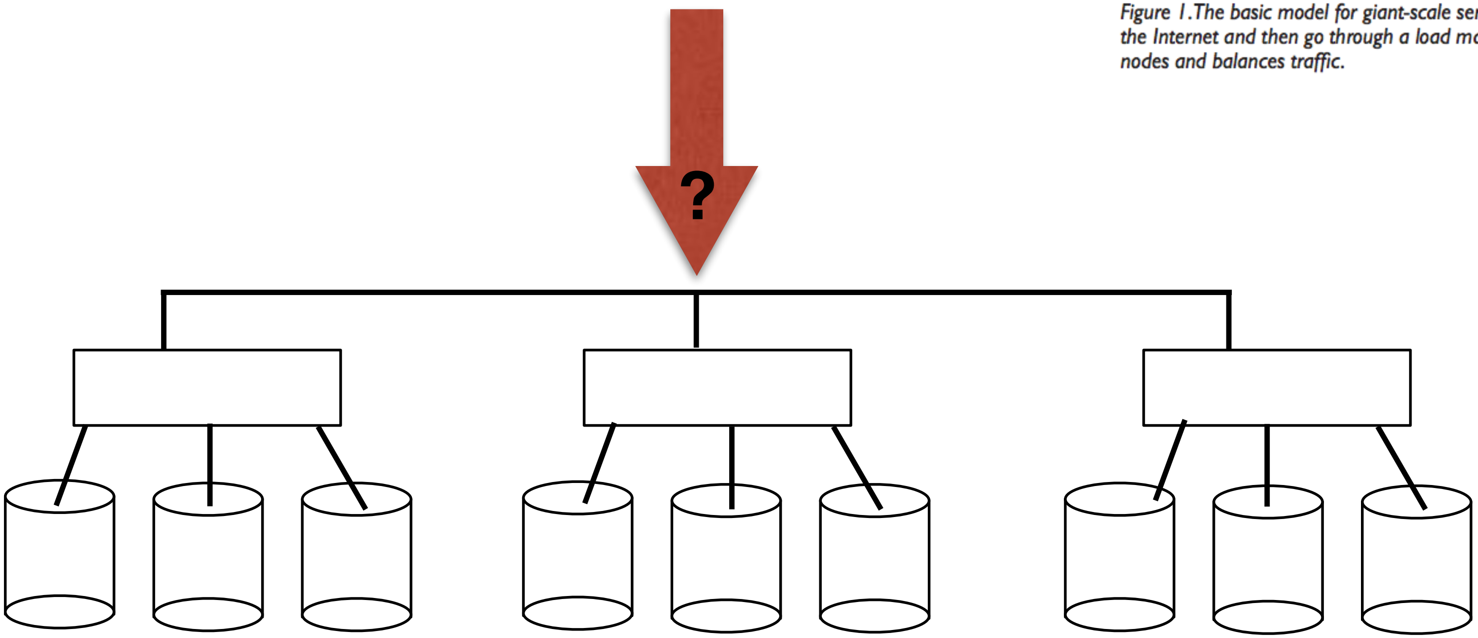
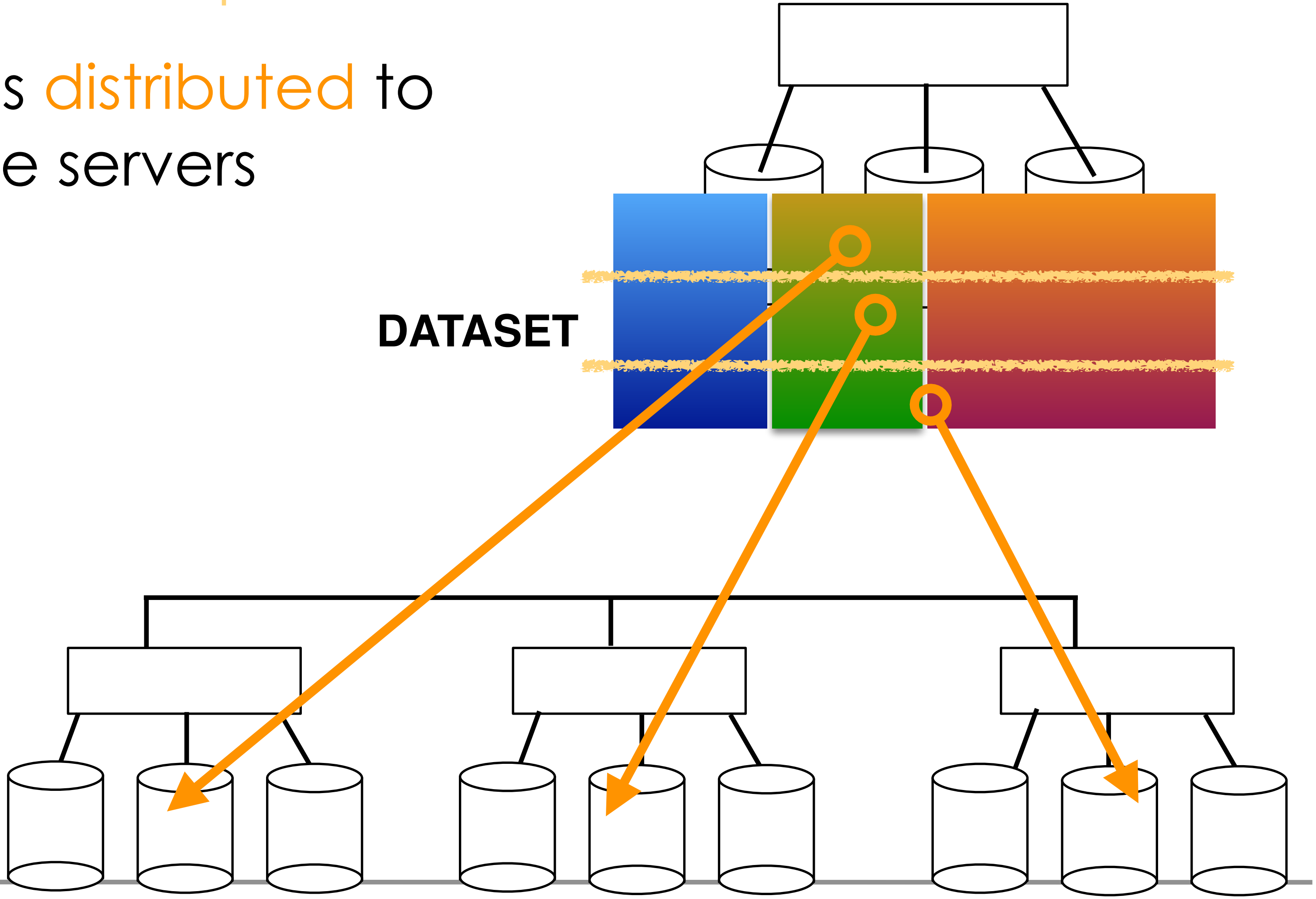


Figure 1. The basic model for giant-scale services. Clients connect via the Internet and then go through a load manager that hides down nodes and balances traffic.



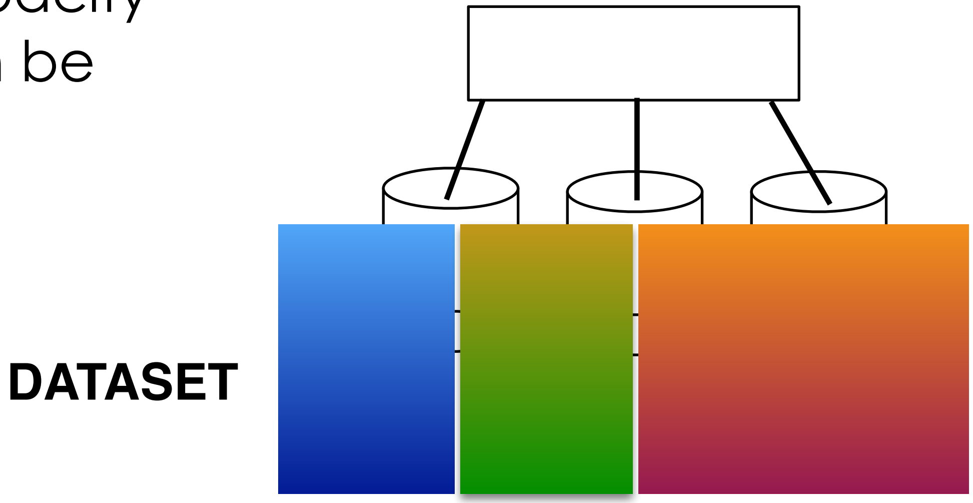
Partitioning (κατάτ□ηση-δια□ελισ□ός)

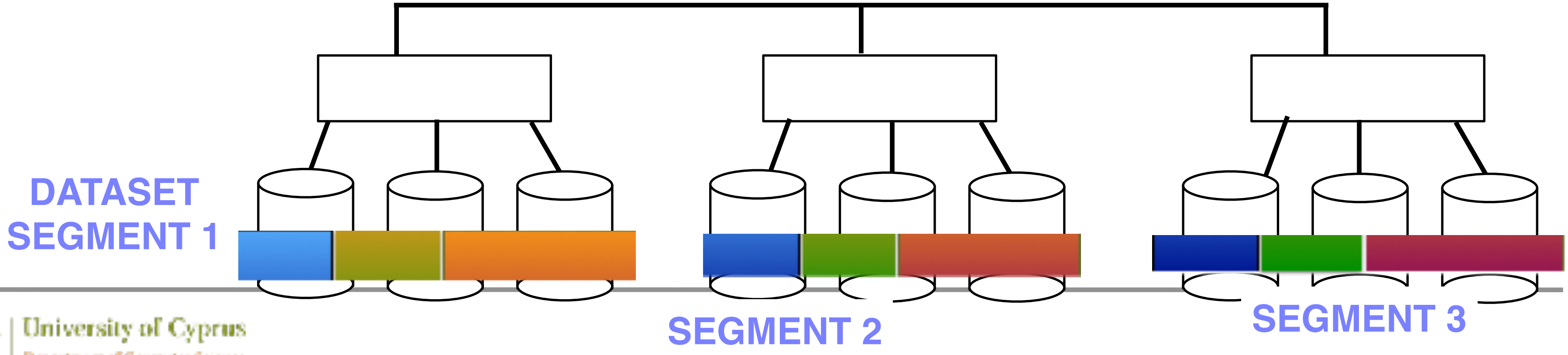
- Persistent data partitioned
- Partitions distributed to available servers



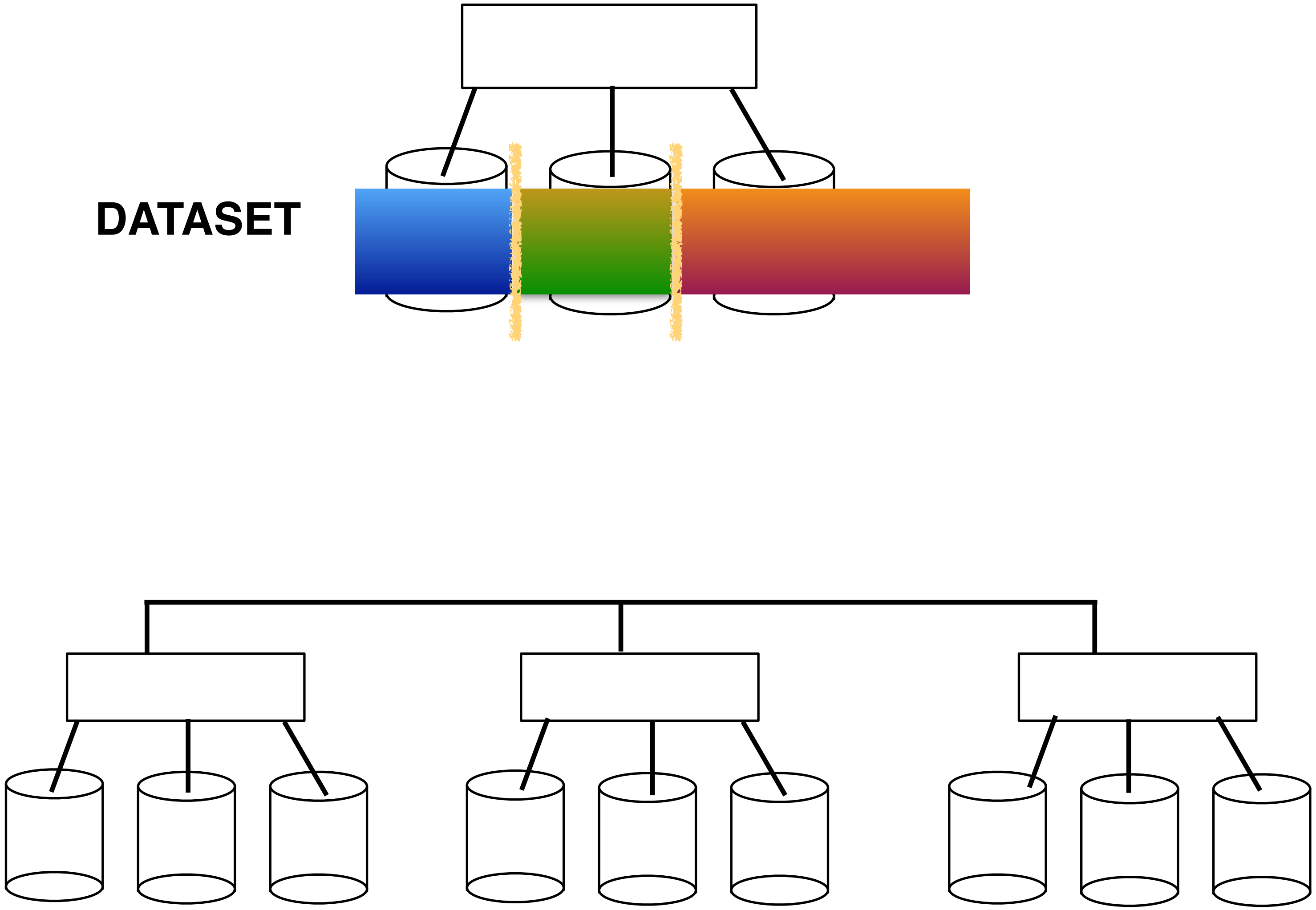
Partitioning

- Outcome: aggregate capacity increase (queries that can be executed per second)

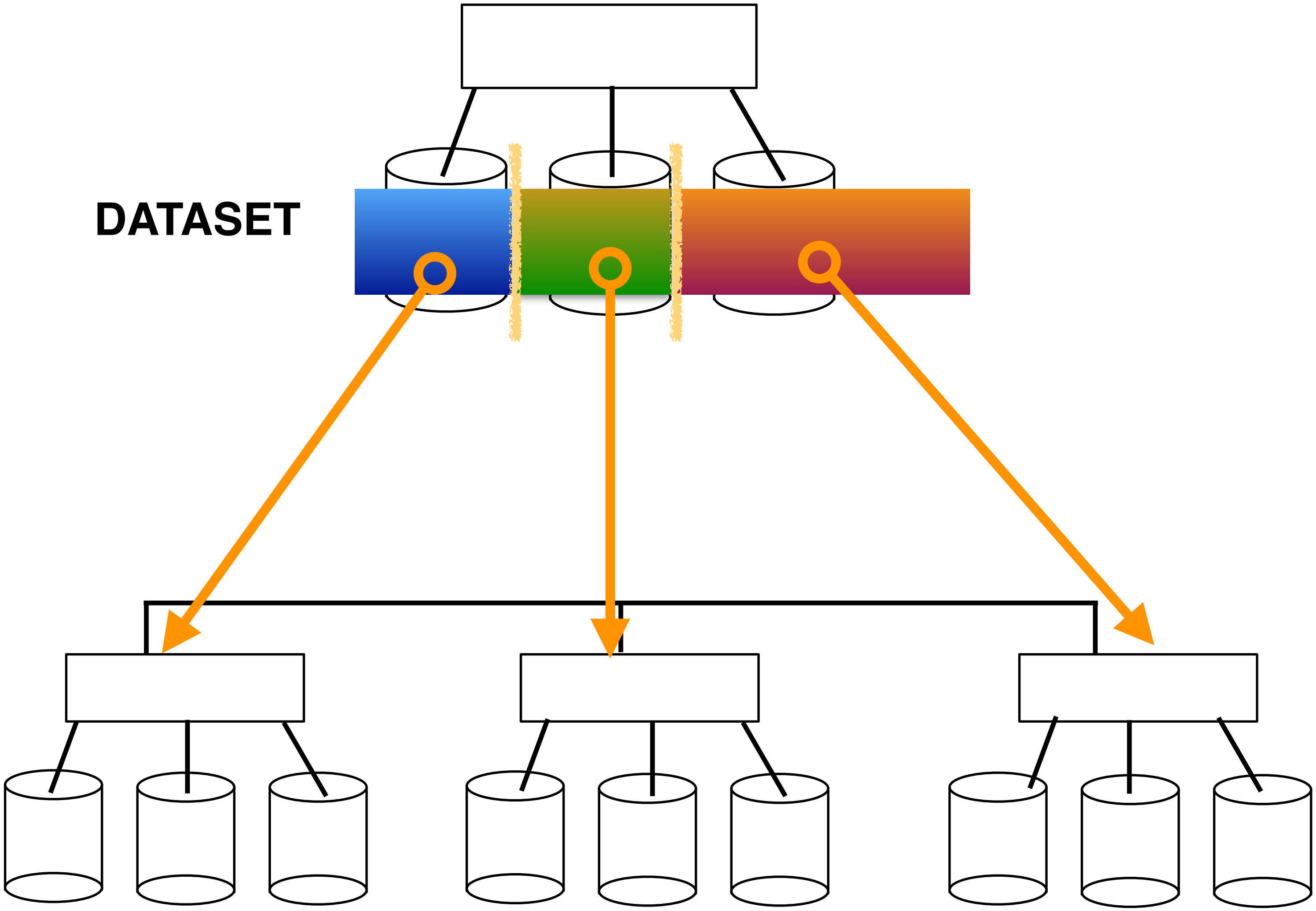




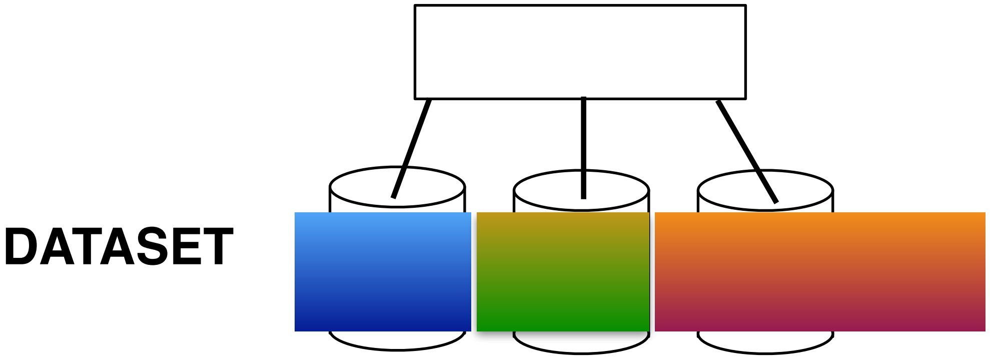
Partitioning - by functionality

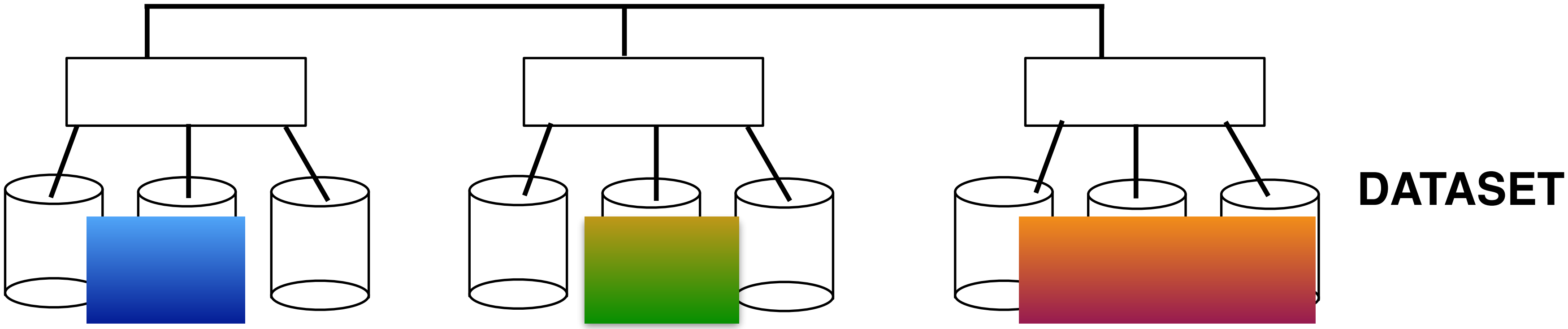


Partitioning (κατάτ□ηση-δια□ελισ□ός)



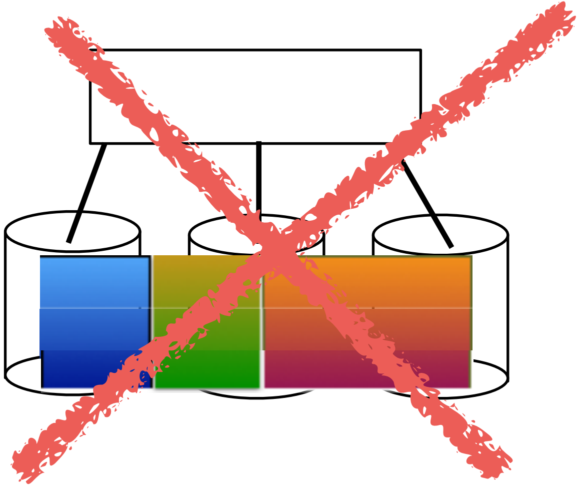
Partitioning (κατάτ□ηση-δια□ελισ□ός)

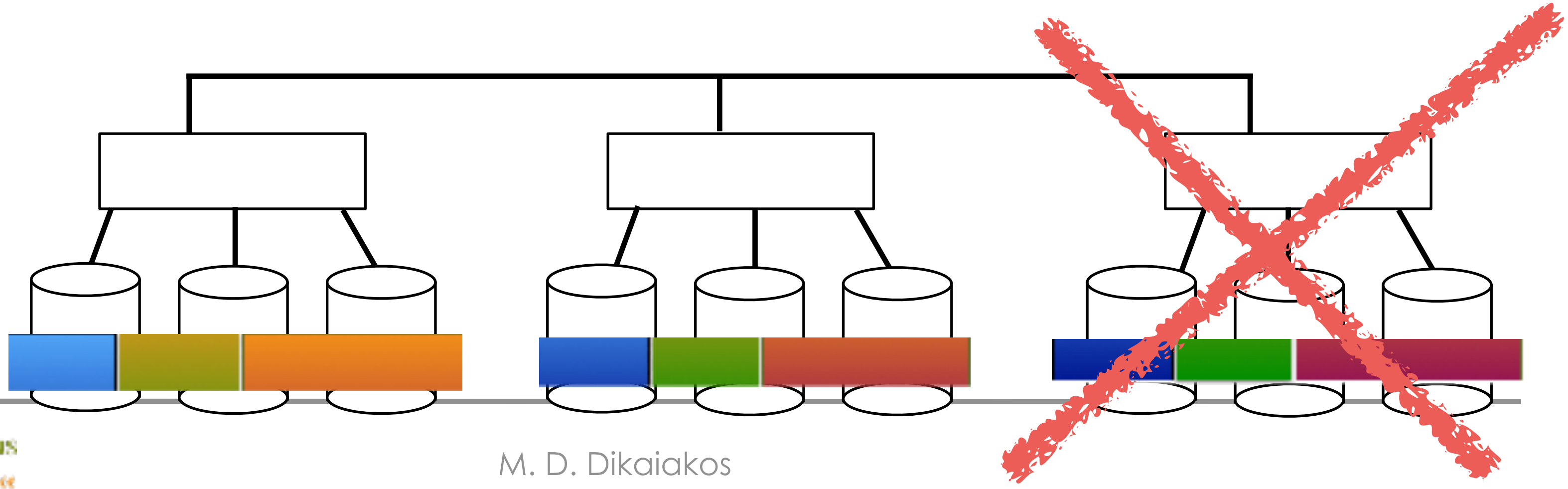




Partitioning and Faults

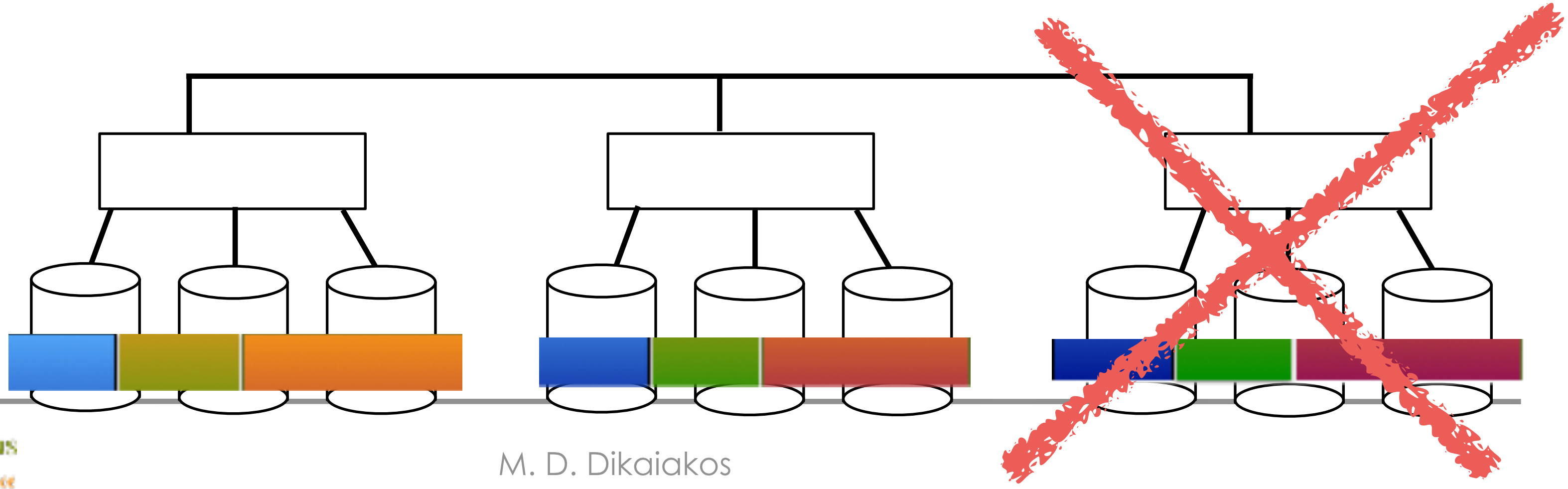
- What is the effect of failure on:
  - **Yield?** (απόδοση)
  - **Harvest?** (συγκο□ιδή)



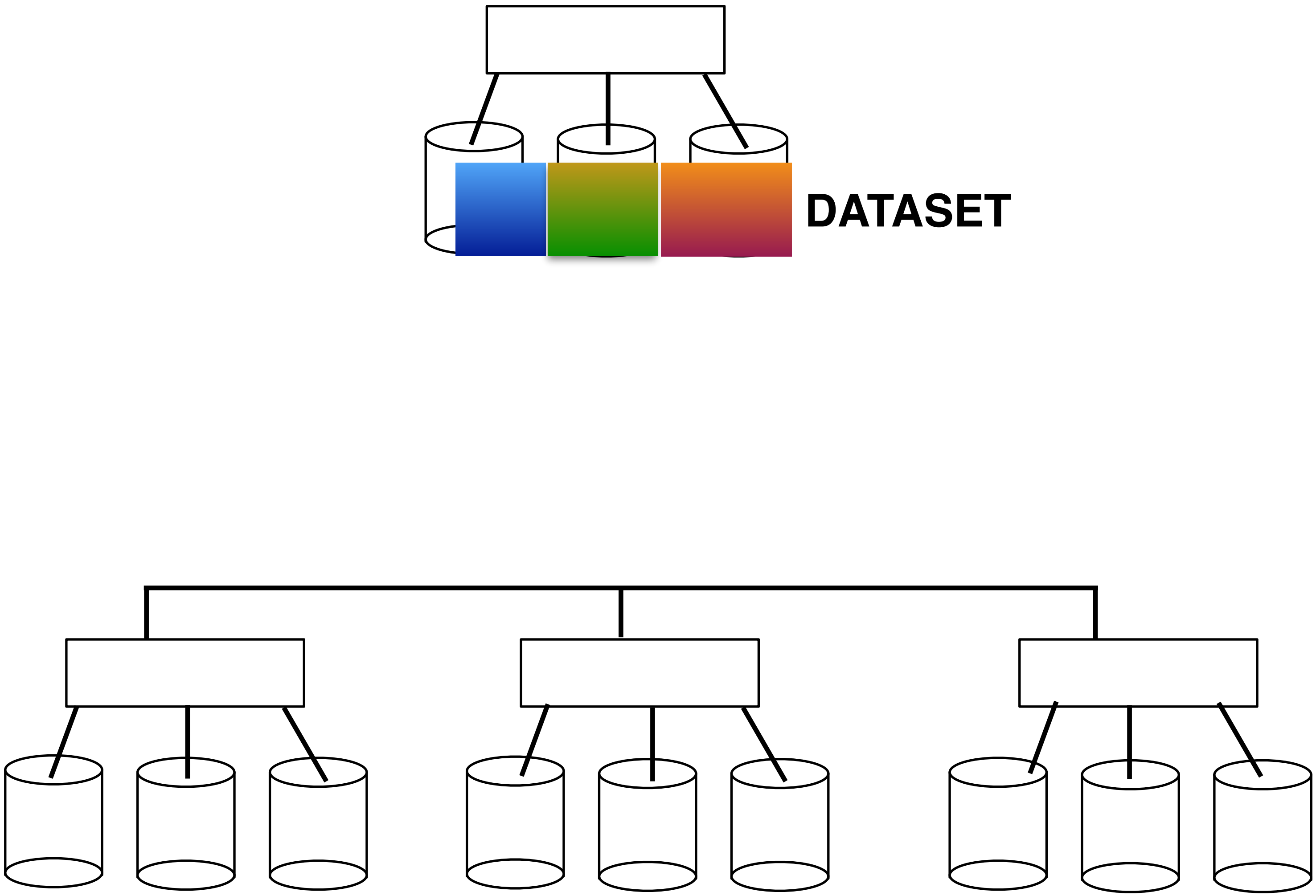


Partitioning and Faults

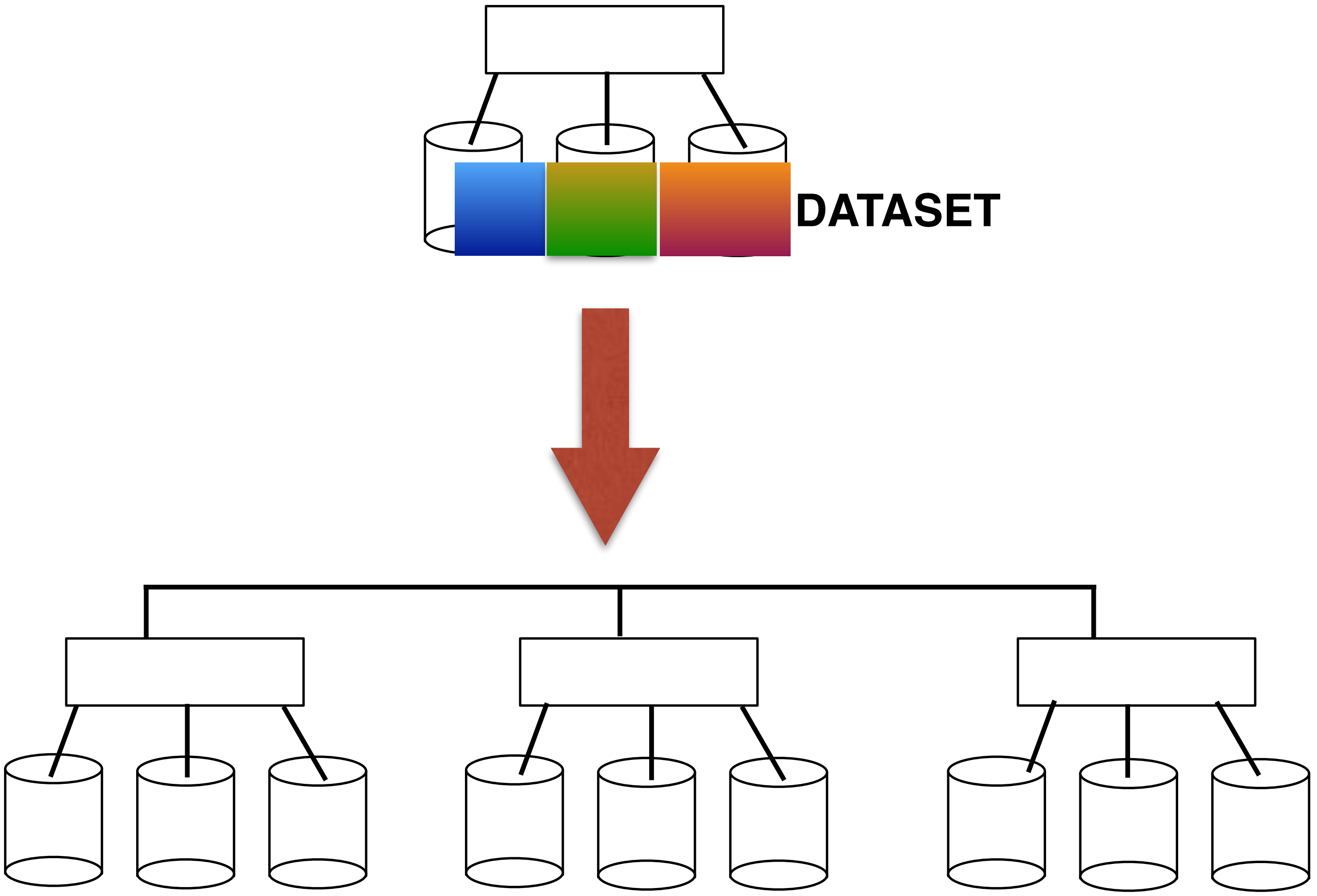
- Some queries do not have answers since some data is missing after the failure, so:
  - Harvest drops.
  - Yield (queries completed/queries offered) remains the same



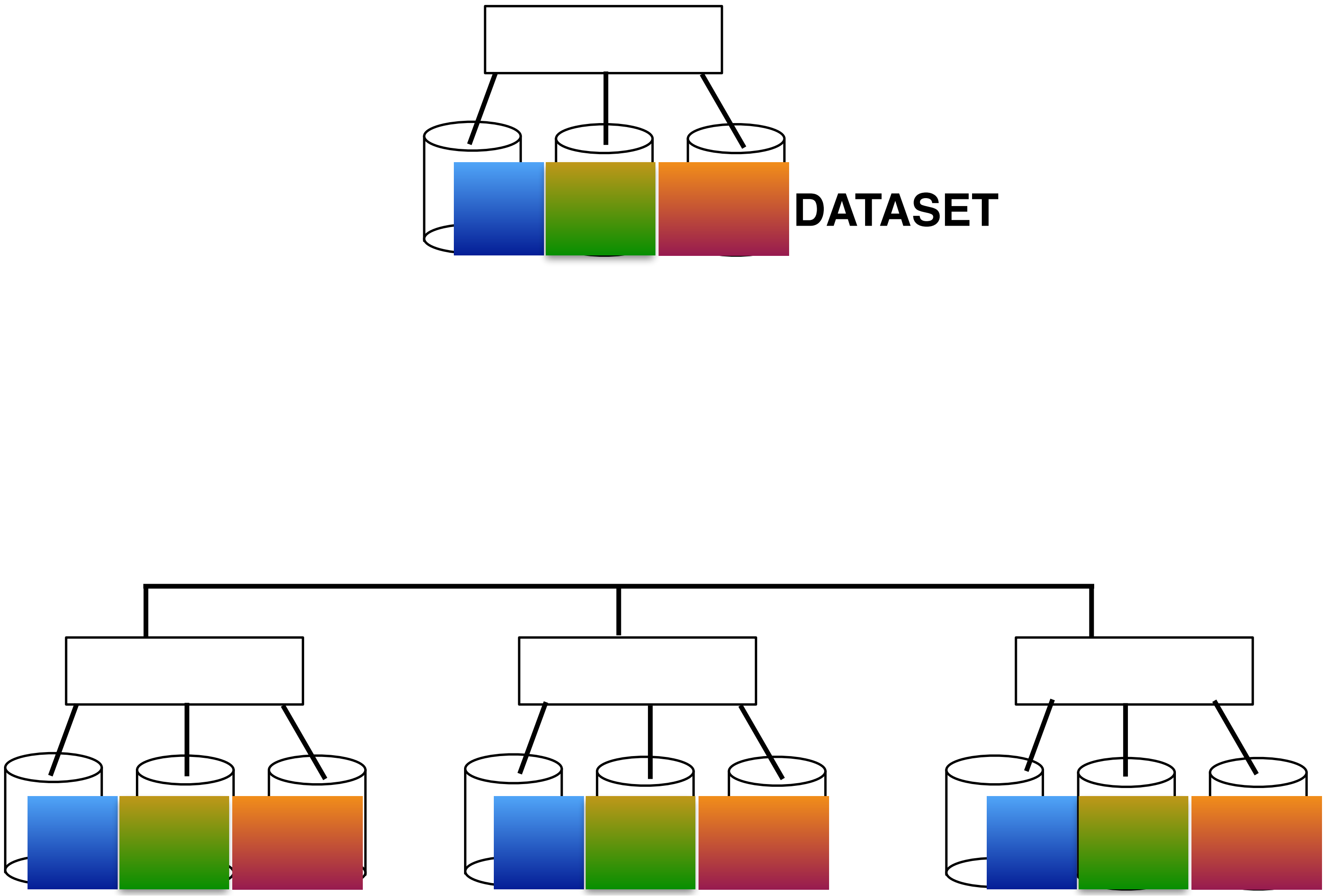
Replication (αντιγραφή-αναπαραγωγή)



Replication (αντιγραφή-αναπαραγωγή)



Replication (αντιγραφή-αναπαραγωγή)

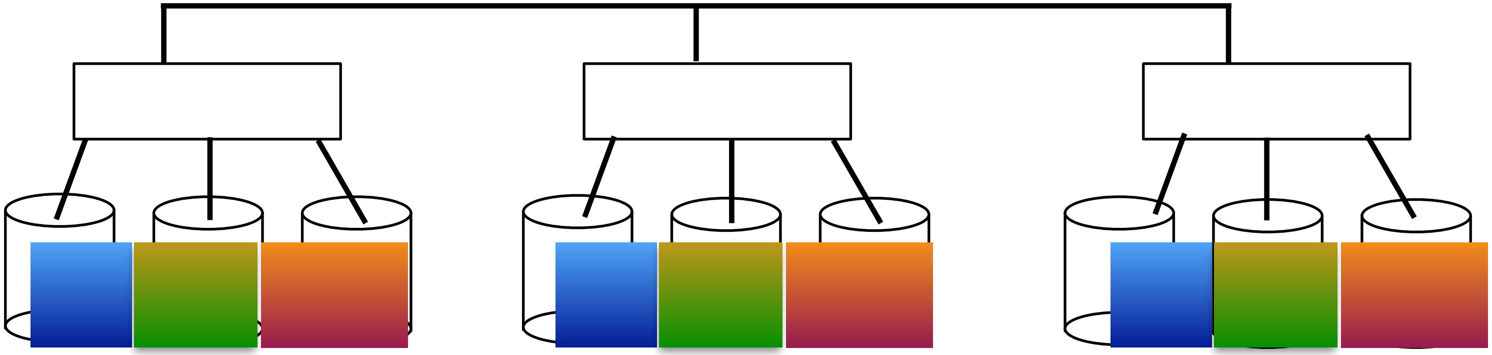


DATASET

DATASET Replica 1 Replica 2 Replica 3

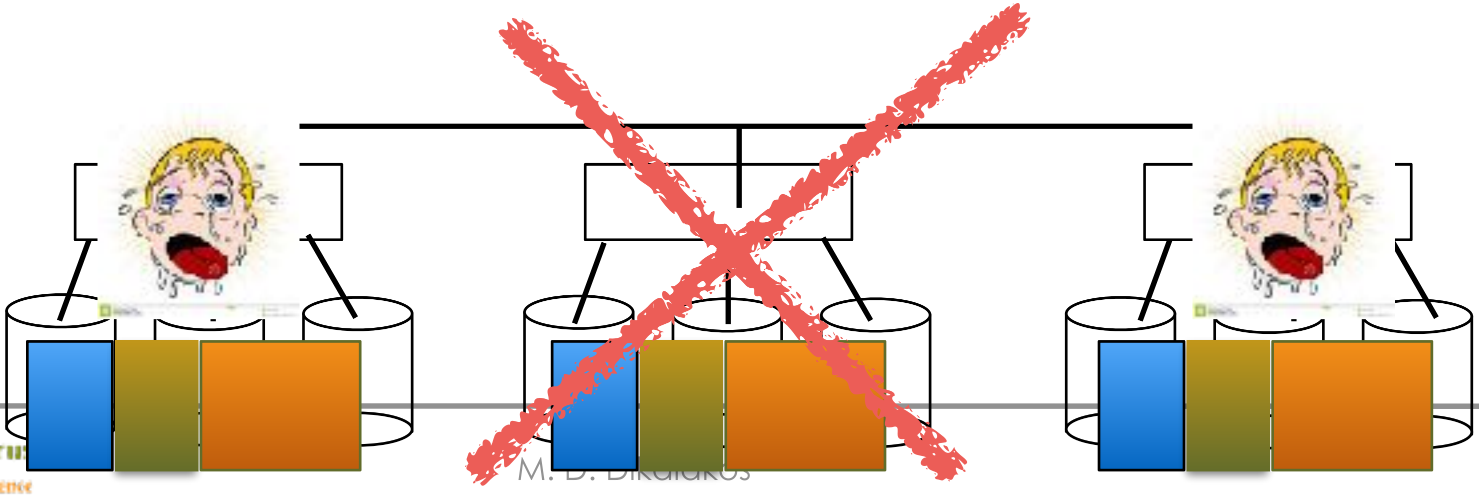
Replication (αντιγραφή-αναπαραγωγή)

- Provides multiple consistent copies of data in processes running in different computers.
- Seeks to improve:
  - performance
  - availability
  - fault tolerance
- The traditional view of replication silently assumes that there is enough excess capacity to **prevent faults from affecting yield**.



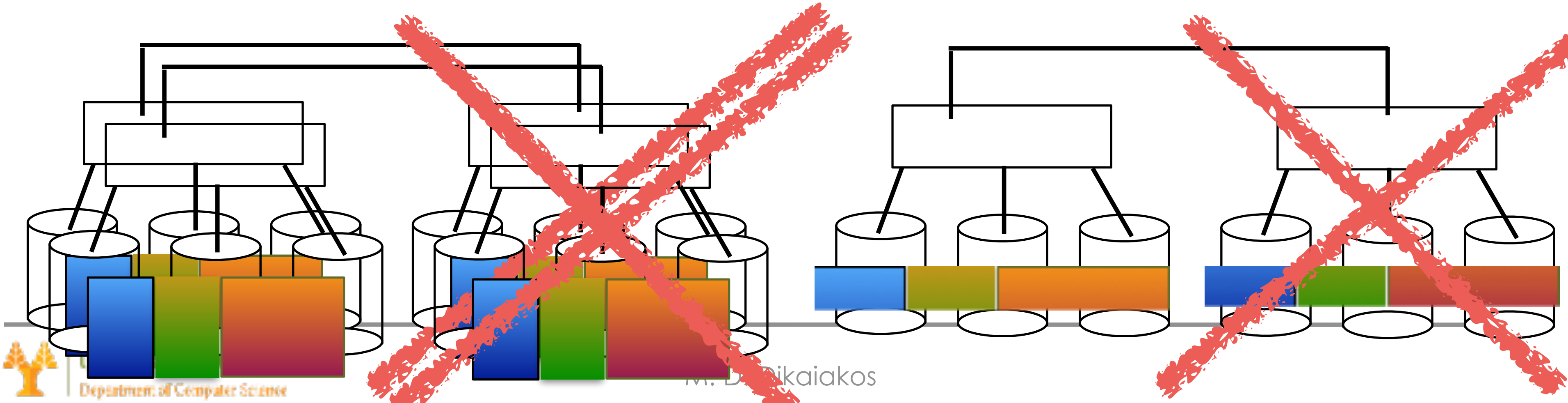
Replication and faults

- What is the effect of failure on:
  - Yield? (απόδοση)
  - Harvest? (συγκο□ιδή)
- Load redirection problem: under faults, the remaining replicas have to handle the queries formerly handled by the failed nodes.
- Under high utilization, this is unrealistic.



Replication vs Partitioning

- Replication is a traditional technique **for increasing availability**
- Consider a *two-node cluster* that faces a fault *in one node*:
  - The replicated version maintains 100 percent harvest but drops to 50 percent yield
  - The partitioned version drops to 50 percent harvest but remains at 100 percent yield
  - Both versions have the same initial DQ value and lose 50 percent of it under one fault:
    - Replicas maintain D (data per query) and reduce Q (queries per sec - yield)
    - Partitions keep Q constant and reduce D (and thus harvest)



Replication vs Partitioning

- We can influence whether faults impact yield, harvest, or both:
- **Replicated systems** tend to map faults to reduced capacity (and to reduced yield at high utilization)
- **Partitioned systems** tend to map faults to reduced harvest, as parts of the database temporarily disappear, but the capacity in queries per second remains the same

Key insights

- Replication on disk is cheap (disks are cheap, replication is easy)
- Accessing replicated data requires “DQ points”:
  - for **true** replication you need not only another copy of the data but also twice the DQ value.
- Partitioning has no real savings over replication:
  - You need *less disk space* than in replication (no storage of copies)
  - the real bottleneck is not storage space but the DQ constant

Key insights

- The DQ constant is independent of whether the database is replicated or partitioned. WHY?
  - Exception: replication requires more DQ points than partitioning for heavy write traffic, which was rare in giant-scale systems (not anymore - see Facebook). WHY?
- Easier to grow systems via replication than by repartitioning onto more nodes
- Can vary the replication according to the data's importance and control which data is lost in the presence of a fault.
- Can exploit randomisation to make lost harvest a random subset of the data.

Avoiding saturation

- Avoiding saturation at a reasonable cost simply by good design is unrealistic:
  - Peak-to-average ratio for giant-scale systems seems to be in the range of **1.6:1** to **6:1** (circa 2001!), which can make it expensive to build out capacity well above the normal peak.
  - Single-event bursts can generate far above-average traffic.
  - Some faults (power failures, natural disasters, cyberattacks) are not independent - overall DQ drops substantially in these cases and remaining nodes become saturated.
- What can we do?

Graceful degradation

- Graceful degradation under excess load is critical for delivering high availability
- The DQ principle suggests:
  - limit Q (capacity) to maintain D - the focus will be on maintaining harvest, using for example, Admission Control (AC) - έλεγχος πρόσβασης
  - reduce D and increase Q by *dynamic database reduction*
  - combination of both techniques
- Graceful degradation is the explicit process for managing the effect of saturation on availability - explicitly decide how saturation should affect uptime, harvest and quality of services. Some approaches:
  - cost-based AC (estimate the cost of each query, measured in DQ, and deny expensive queries)
  - priority or value-based AC (e.g. stock trade requests vs rest)
  - reduced data freshness - reduces work per query ==> increased yield, decreased harvest

Online Evolution and Growth

- Traditional tenet of highly available systems: **minimal change.**
- But, this conflicts with:
  - growth rates of Internet services
  - "Internet time", namely the practice of frequent product releases
- Must plan for: continuous growth and frequent functionality updates
- Must cope with software that is never perfect

Availability and Performance

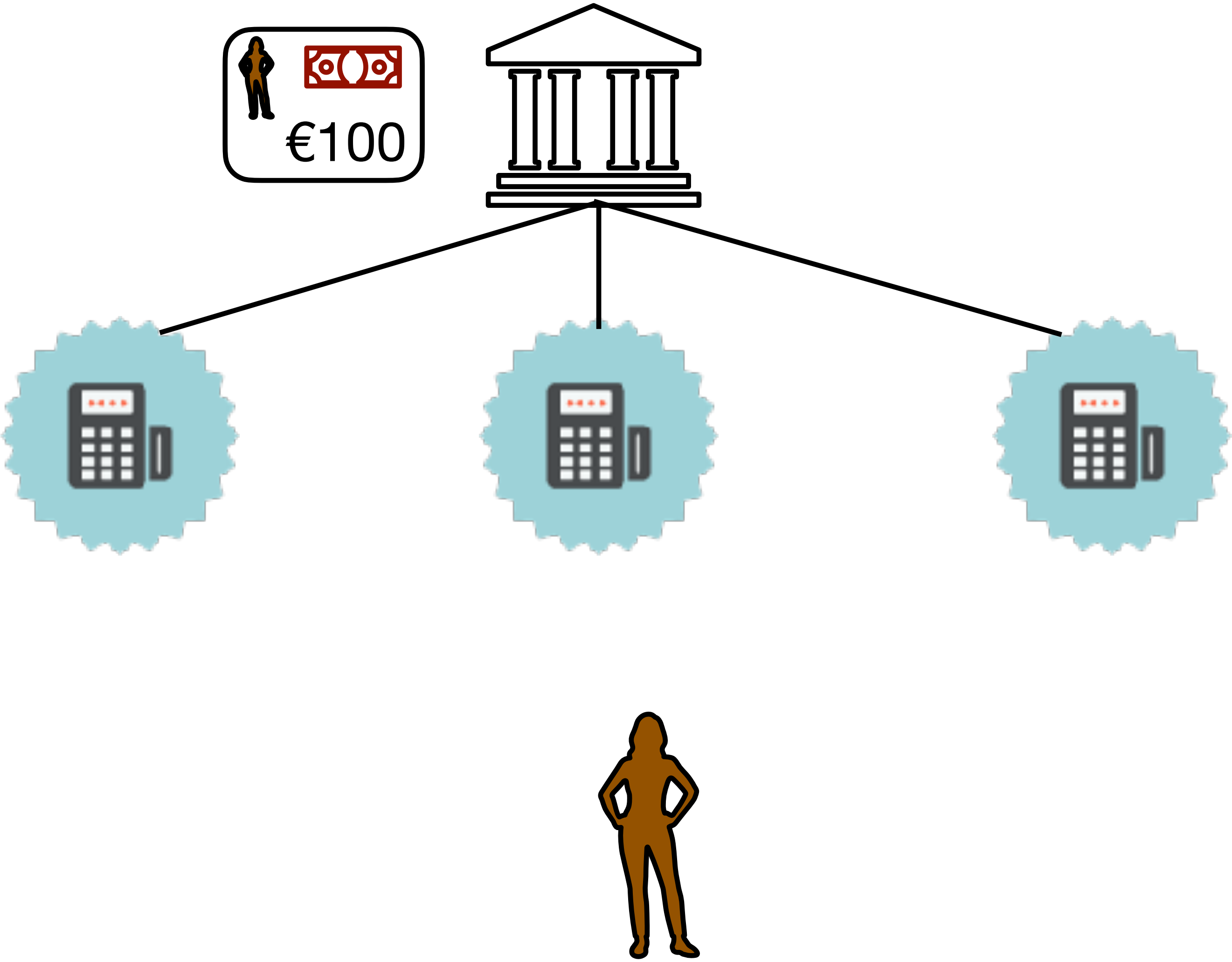
CAP Theorem

CAP Theorem

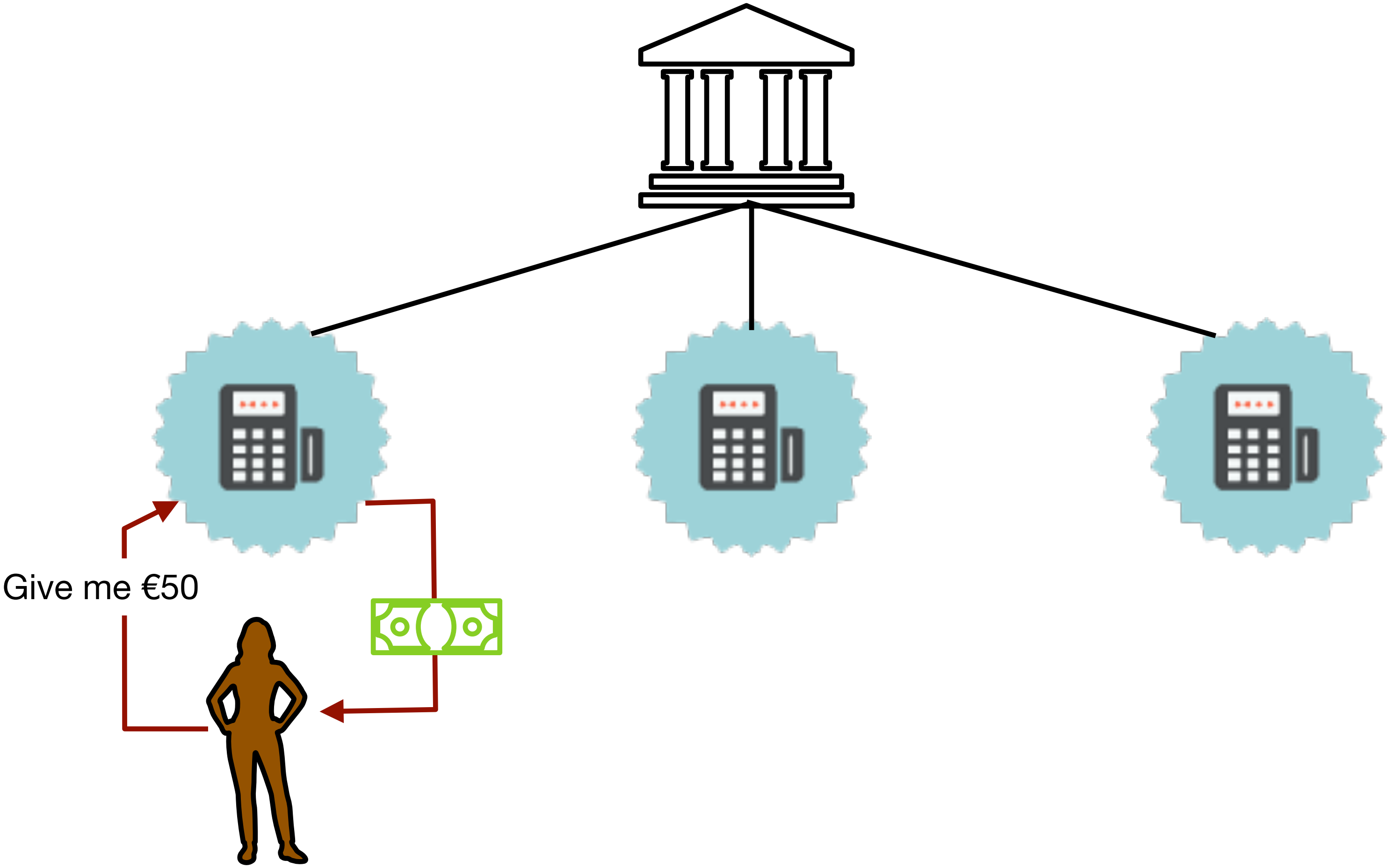
Any networked shared-data system **can have at most two of three desirable properties**:

- **Consistency (C)** equivalent to having a single up-to-date copy of the data:
  - Every read receives the most recent write or an error.
- **High availability** (A) of that data
  - Every request receives a (non-error) response, without the guarantee that it contains the most recent write.
- Tolerance to network **partitions** (P)
  - The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.

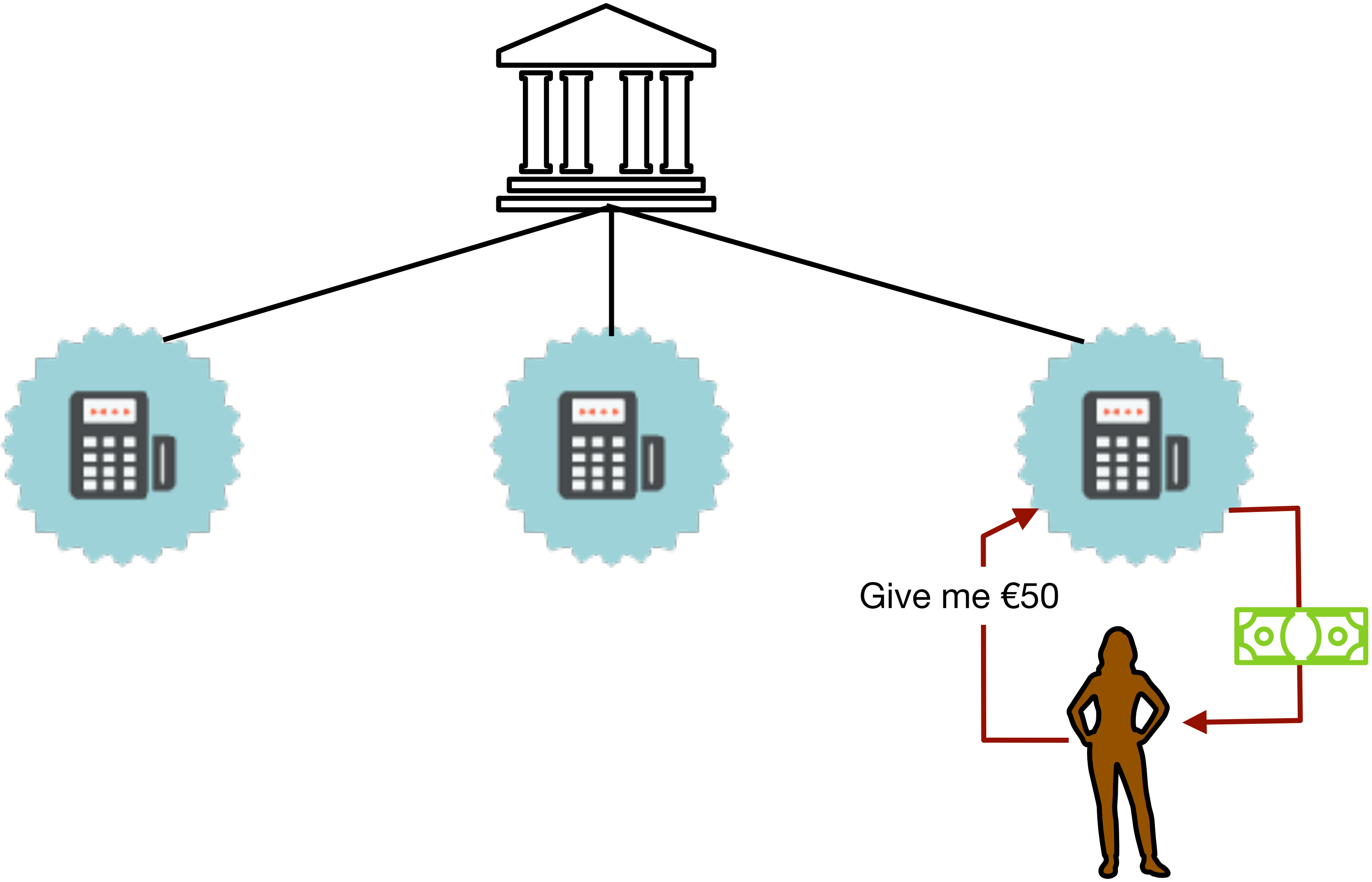
Availability



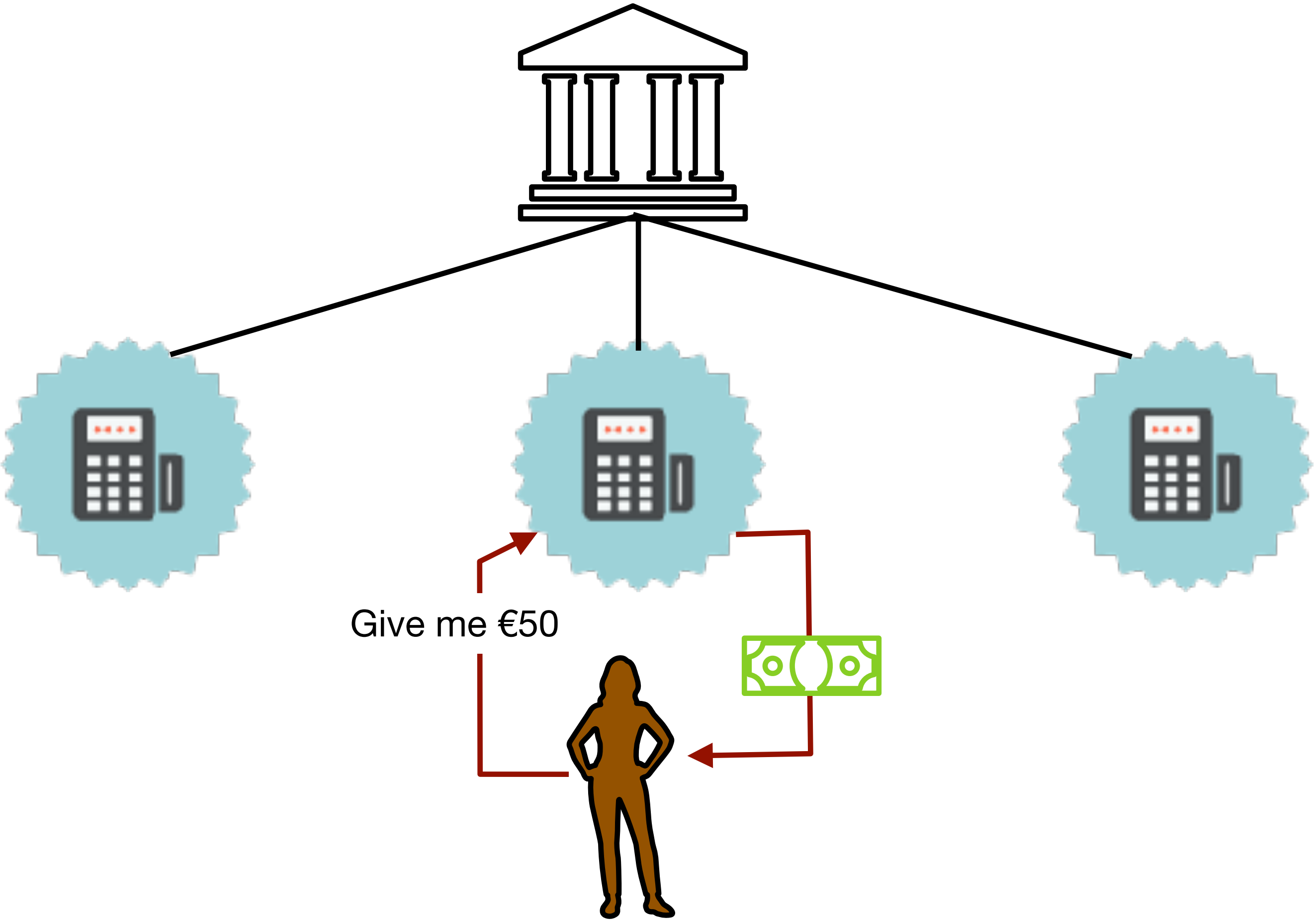
Availability



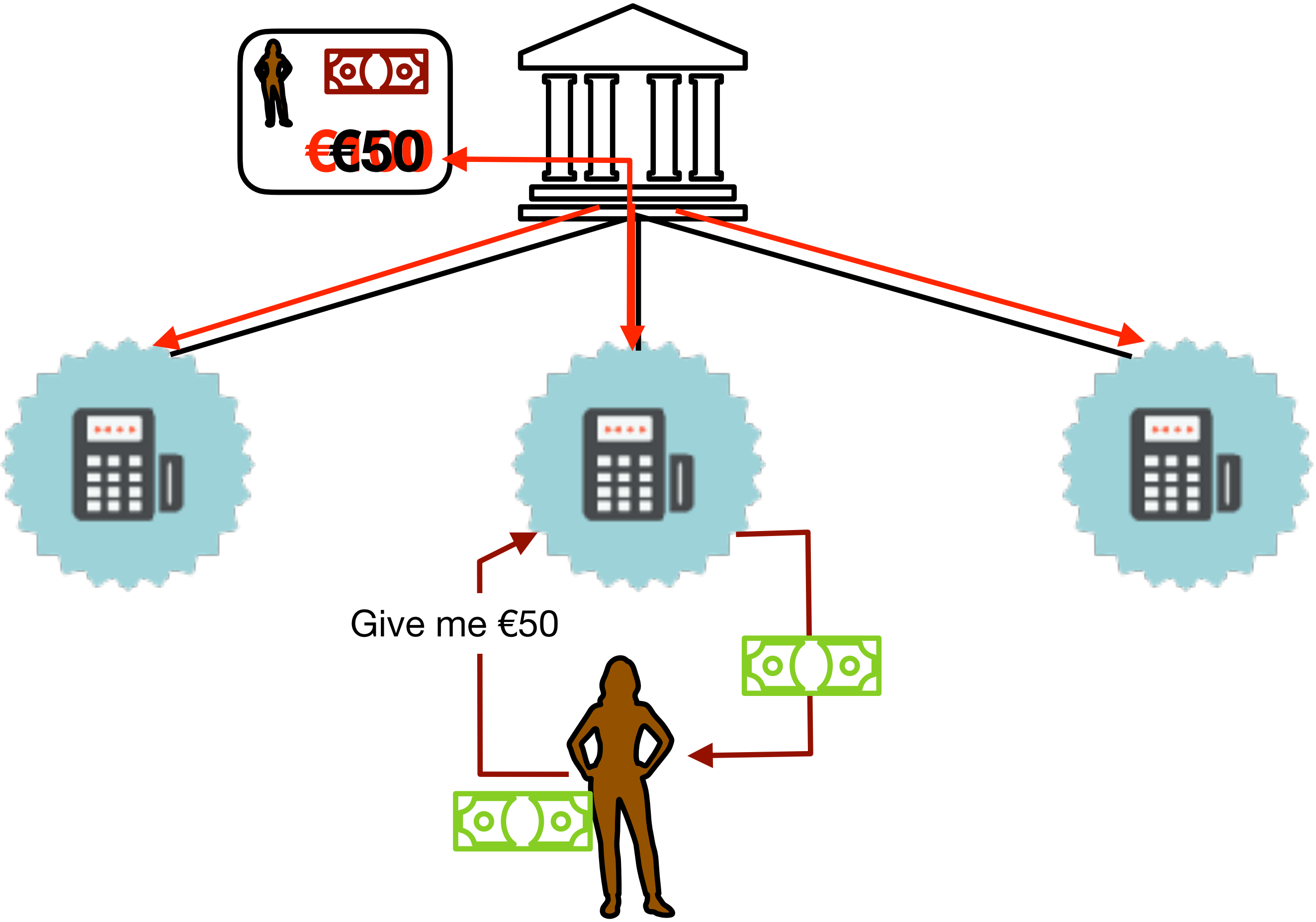
Availability



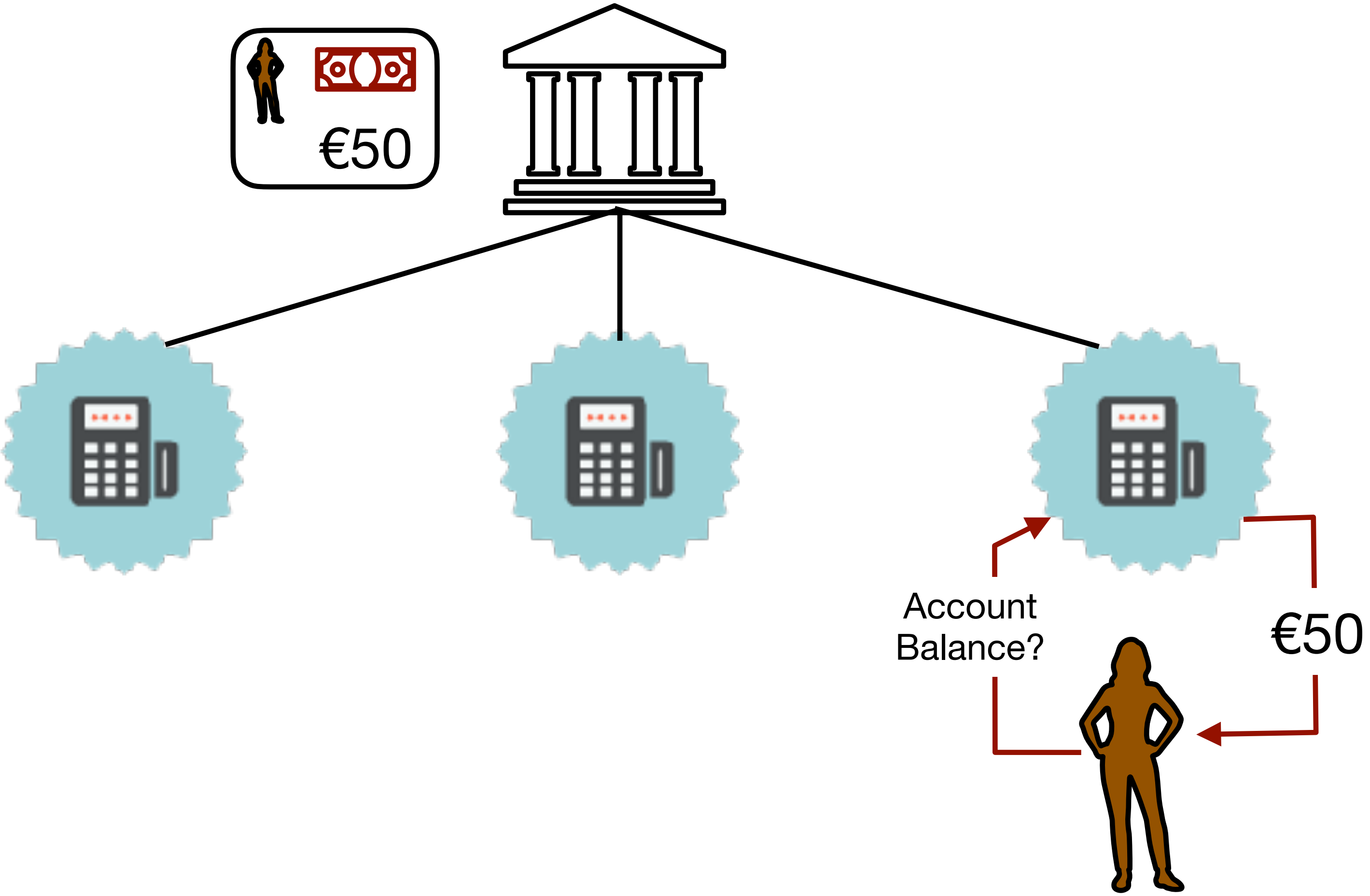
Availability



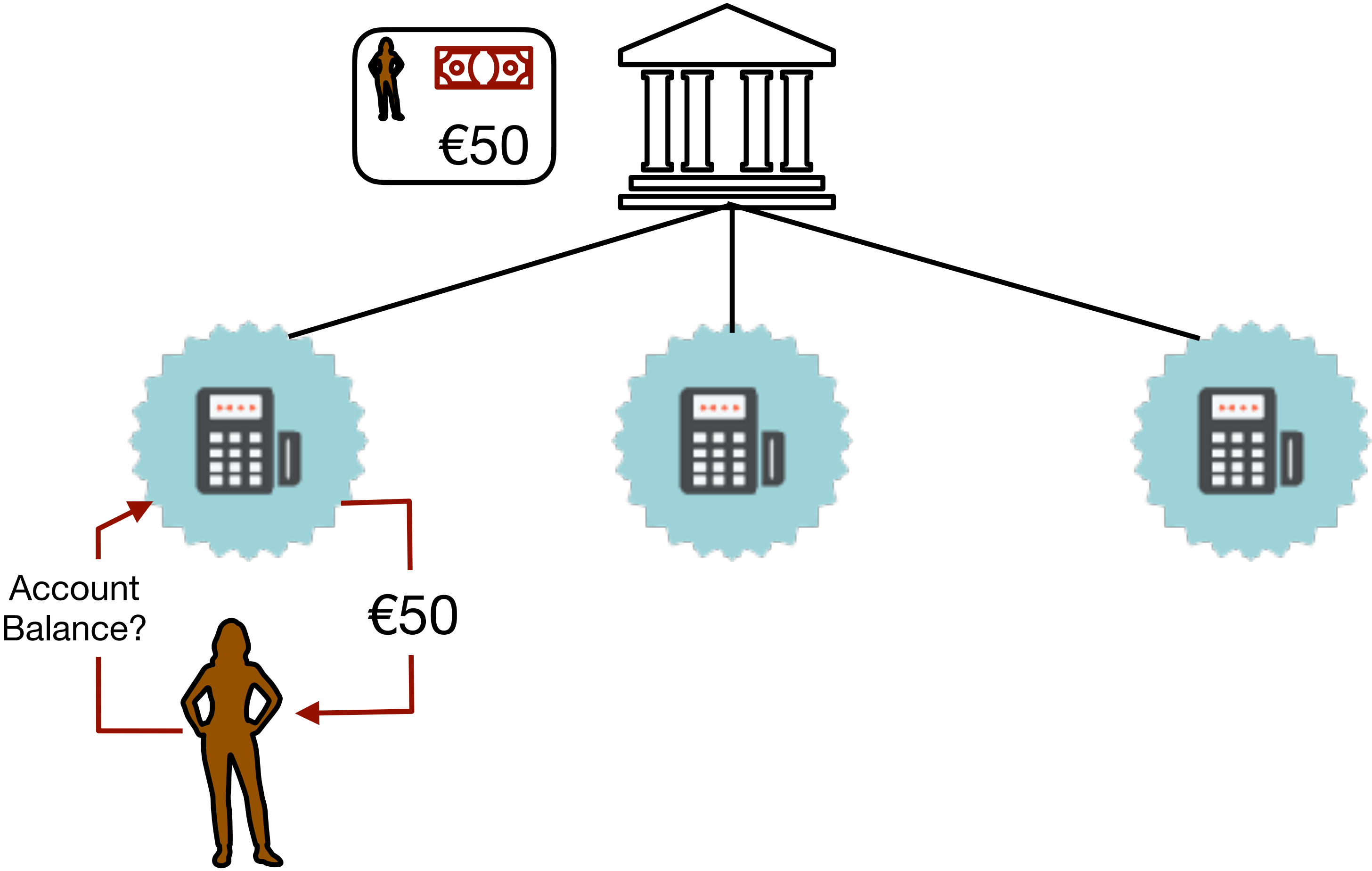
Consistency



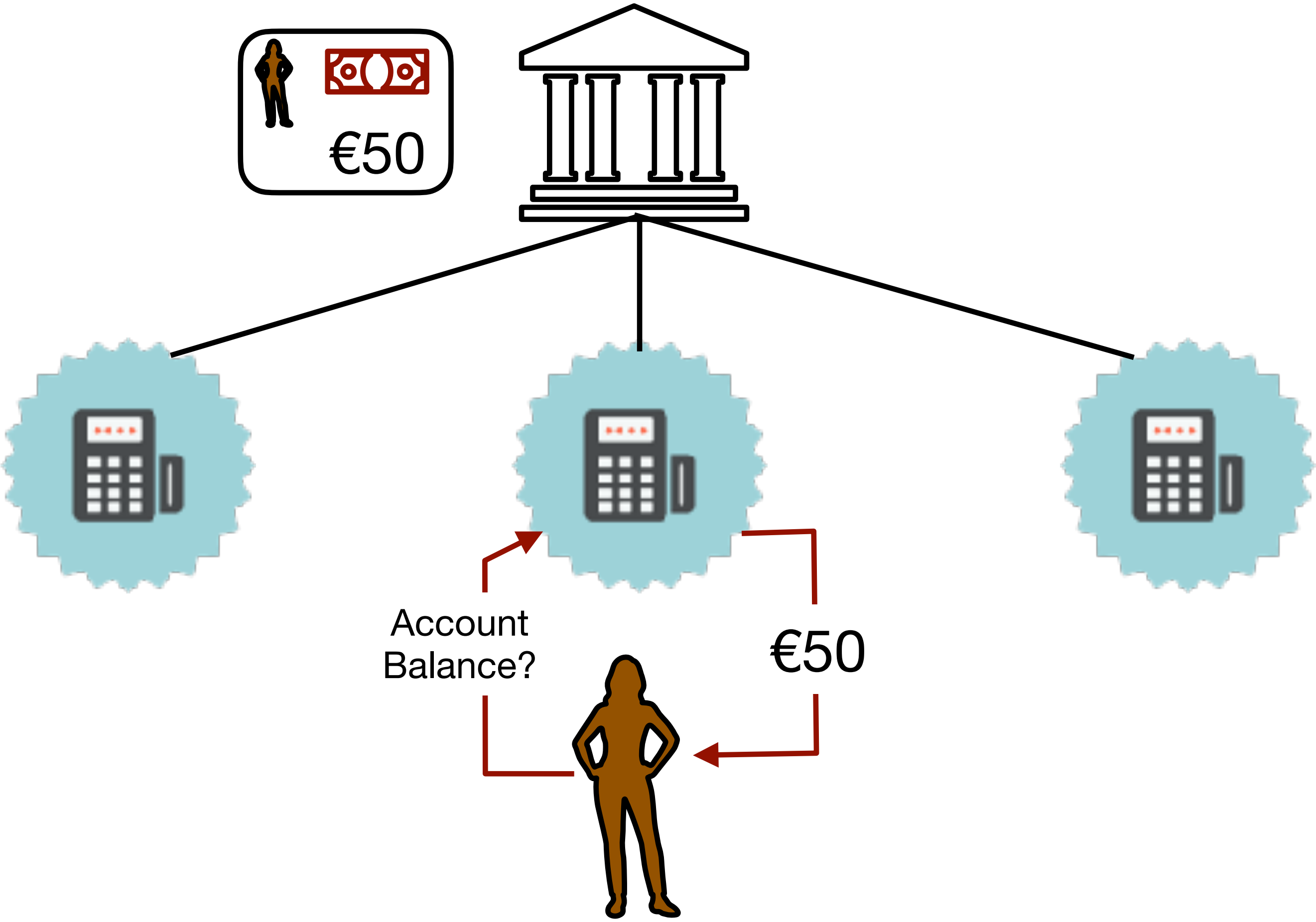
Consistency



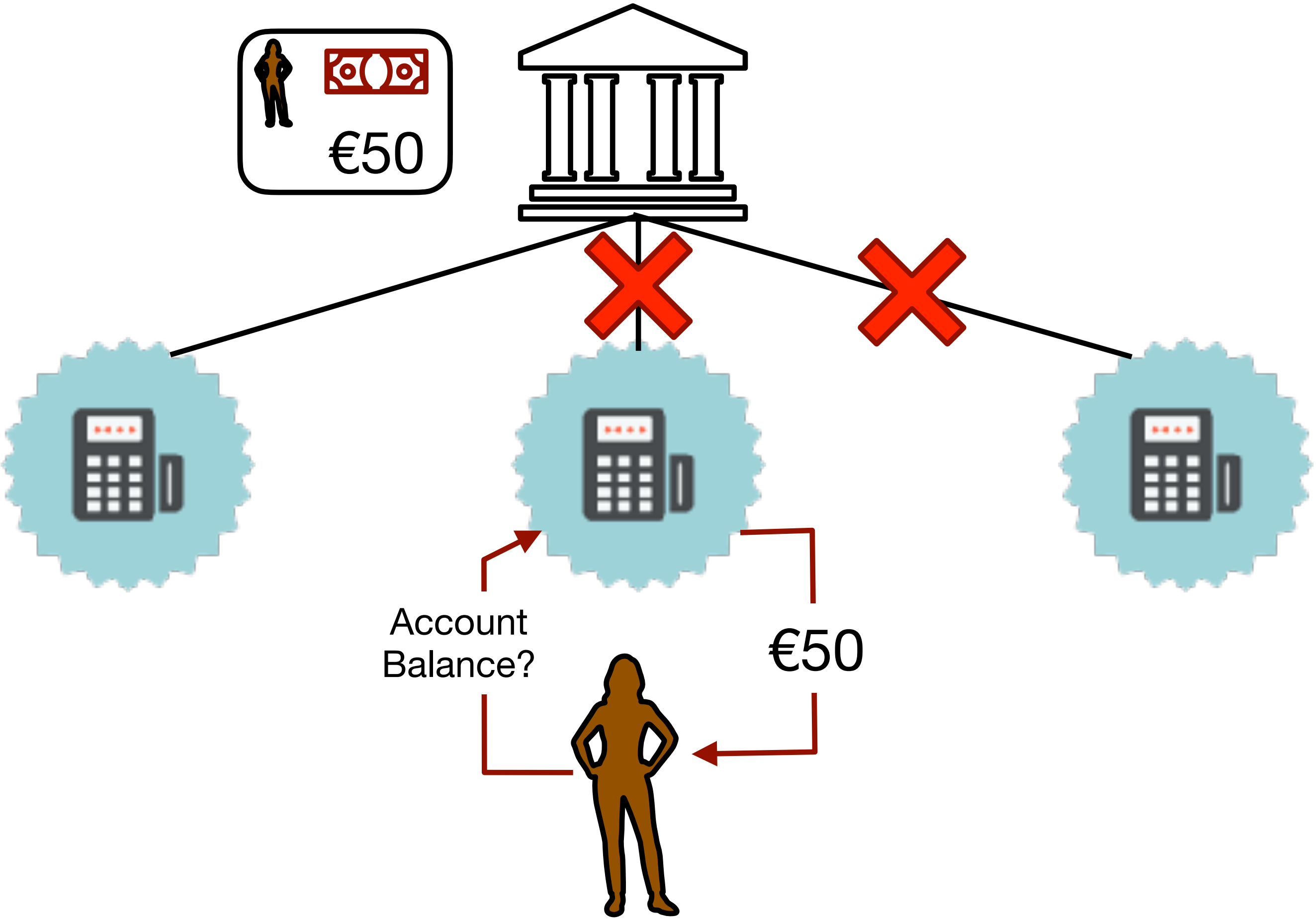
Consistency



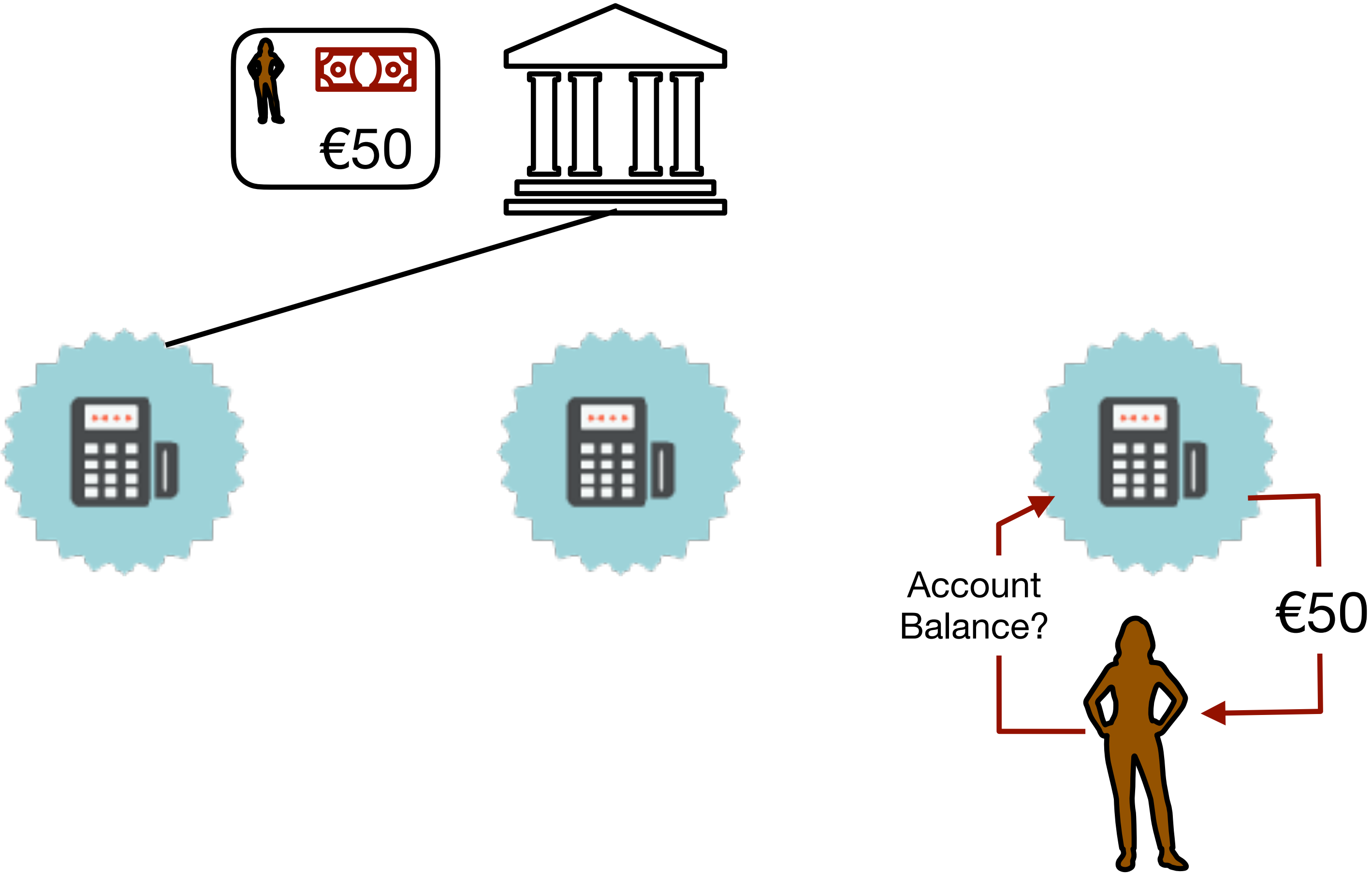
Consistency



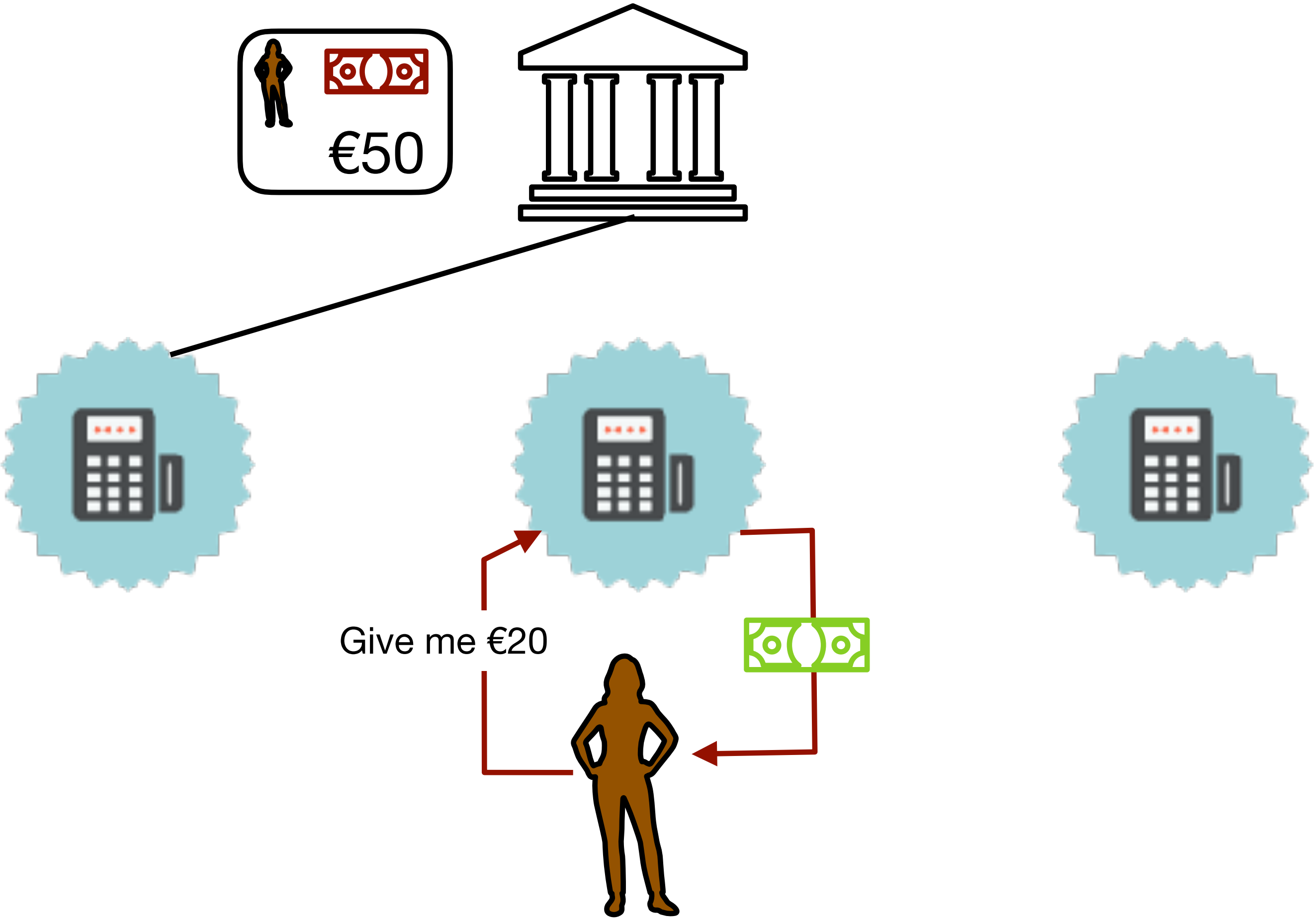
Partitioning & Availability



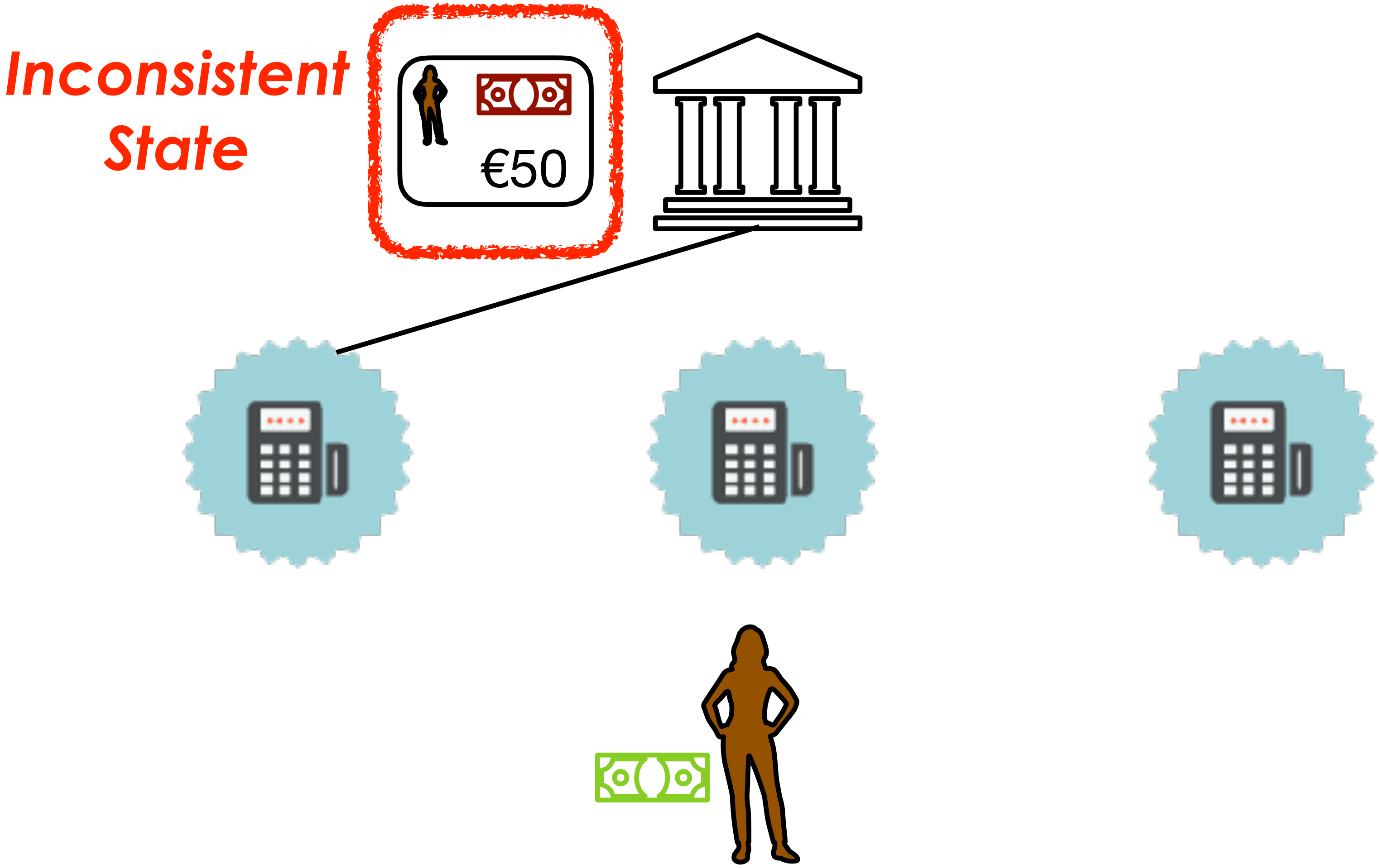
Partitioning & Availability



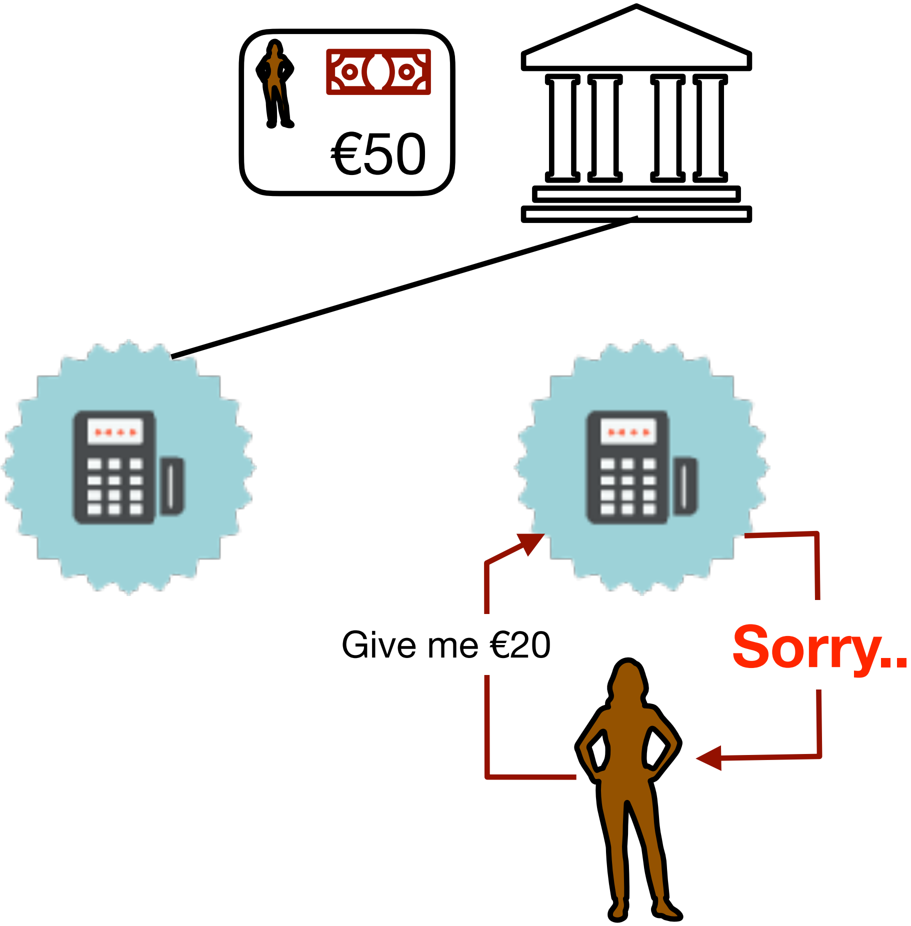
Partitioning => Inconsistency



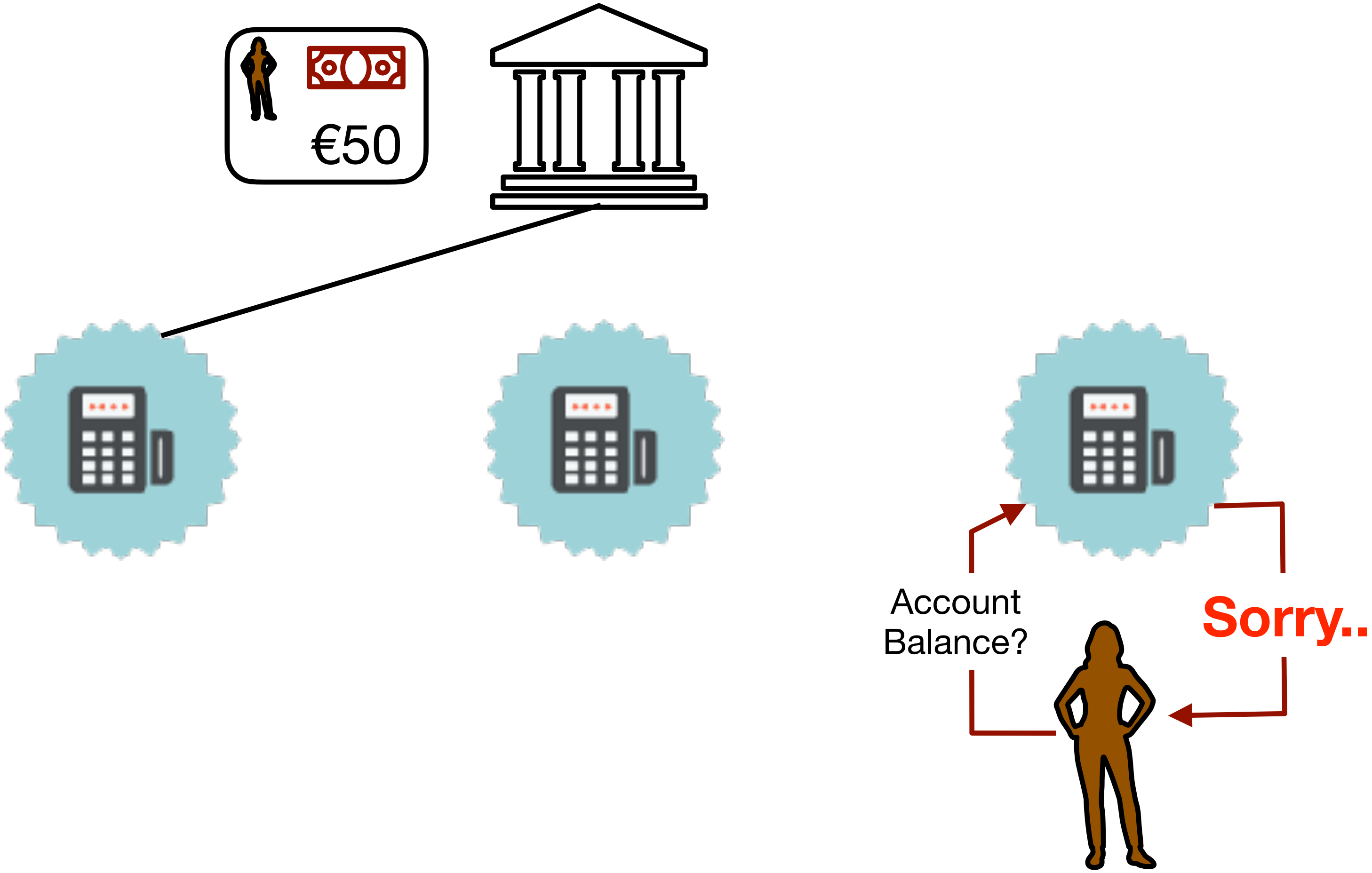
Partitioning & Availability



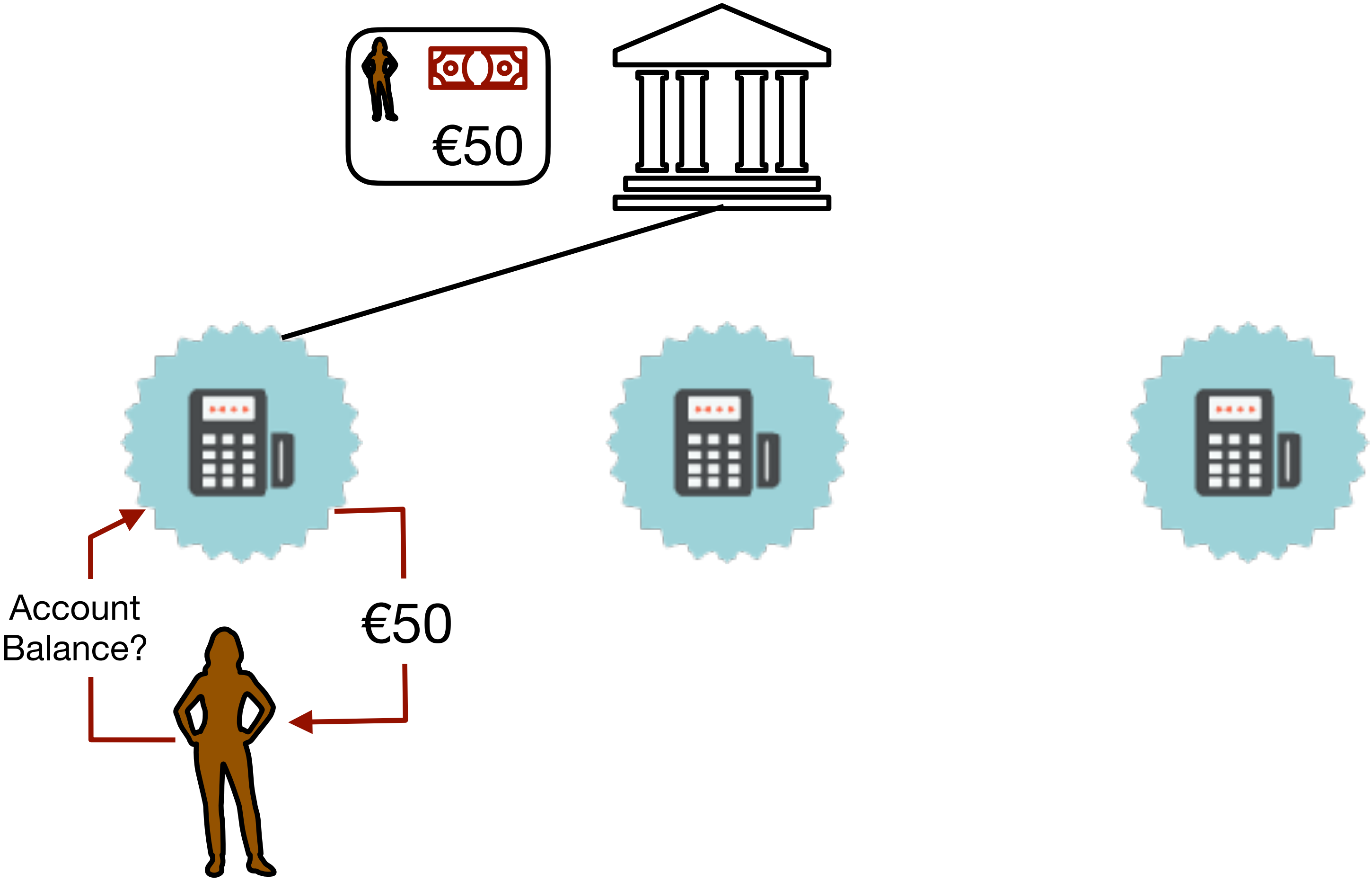
Partitioning & Consistency



Partitioning & Consistency



Partitioning & Consistency



CAP Concepts

- Strong **Consistency** means single-copy ACID consistency
  - A strongly-consistent system provides the ability to perform updates
- High **Availability** is assumed to be provided through redundancy, e.g. data replication;
  - Data is considered highly available if a given consumer of the data can always reach some replica
- **Partition**-resilience means that the system as whole can survive a partition between data replicas.

ACID properties

University of Cyprus
Department of Computer Science

- Atomicity: the entire transaction succeeds or fails
- Consistency: the entire collection is never left in an invalid or conflicting state
- Isolation: concurrent transactions cannot interfere with each other
- Durability: once a transaction completes, system failures cannot invalidate the result

CAP Theorem Purpose

- Makes **explicit the trade-offs** in designing distributed infrastructure applications.
- Guides the design of Internet-scale systems services that:
  - **Tolerate partial failures** by emphasizing simple composition mechanisms that promote fault containment.
  - Incorporate engineering mechanisms that translate **partial failures** into smoothly-degrading functionality rather than a lack of availability of the service as a whole.

Strong CAP Principle

Strong **Consistency**, High **Availability**, **Partition**-resilience: Pick **at most 2.**

- CA without P: Databases that provide distributed transactional semantics can *only do so in the absence of a network partition* separating server peers.
- CP without A: In the event of a partition, further transactions to an ACID database may be blocked until the partition heals, to avoid the risk of introducing merge conflicts (and thus inconsistency).
- AP without C: Web caching provides client-server partition resilience by replicating documents, but *a client-server partition prevents verification of the freshness of an expired replica*.
- In general, any distributed database problem can be solved with:
    - Expiration-based caching to get AP, or
    - Replicas and majority voting to get PC (the minority is unavailable).

Weak CAP Principle

- In practice, many applications are best described in terms of **reduced consistency** or **availability**.
- So, more often we see systems to follow the Weak CAP Principle:

The stronger the guarantees made about any two of strong consistency, high availability, or resilience to partitions, the weaker the guarantees that can be made about the third.

Metrics

- How do we argue about the CAP properties of a system?
- We need metrics that represent and measure a system from the CAP perspective.

Yield and Harvest

- At least two metrics for correct behavior:
- **Yield** (απόδοση παραγωγής) = queries completed/queries submitted
  - the probability of completing a request
  - the common metric and is typically measured in “nines”: “four-nines availability” means a completion probability of 0.9999
  - in practice, good High-Avail. systems aim for **four** or **five nines**
- **Harvest** (βαθ□ός συγκο□ιδής) = data available/complete data
  - measures the fraction of the data reflected in the response, i.e. the completeness of the answer to the query.
- In systems based on queries, we can measure query completeness—how much of the database is reflected in the answer
  - this can be extended to features supported by a service

Tradeoff (queries)

- In the presence of faults there is typically a tradeoff between providing:
  - **no answer** (reducing yield)
  - an **imperfect answer** (maintaining yield, but reducing harvest).
- Some applications **do not tolerate harvest degradation** because any deviation from the single well-defined correct behavior renders the **result useless**.
  - E.g., a sensor application that must provide a binary sensor reading (presence/absence) does not tolerate degradation of the output
- Some applications **tolerate graceful degradation of harvest**
  - E.g., online aggregation allows a user to explicitly trade running time for precision and confidence in performing arithmetic aggregation queries over a large dataset: useful when approximate answers are ok; helps avoiding work that looks unlikely to be worthwhile based on preliminary results.

Tradeoff (updates)

- Same tradeoff applies to “single-location” updates
- Those changes that are localized to a single node (or technically a single partition):
  - Updates that affect reachable nodes occur correctly but have limited visibility => reduced harvest
  - Updates that require unreachable nodes fail => reduced yield

Dealing with CAP's effects

Strategy 1: Trading Harvest for Yield— Probabilistic Availability

- Nearly all systems are probabilistic wrt faults:
- Availability requires probabilistic approaches
- Address probabilistic systems directly, so that we can understand and limit the impact of faults. Need to explore and decide:
  - the expected nature of faults
  - what needs to be available
- Example - search engine index decomposition:
  - By **randomly** placing index shards on nodes, we can ensure that 1 fault in 100-node cluster results in 1% random loss of results & linear harvest degradation for more faults => Average-case and worst- case fault behavior the same.
  - By **replicating** a **high-priority subset of data**, we reduce the probability of losing that data. This gives us more precise control of harvest, both increasing it and reducing the practical impact of missing data

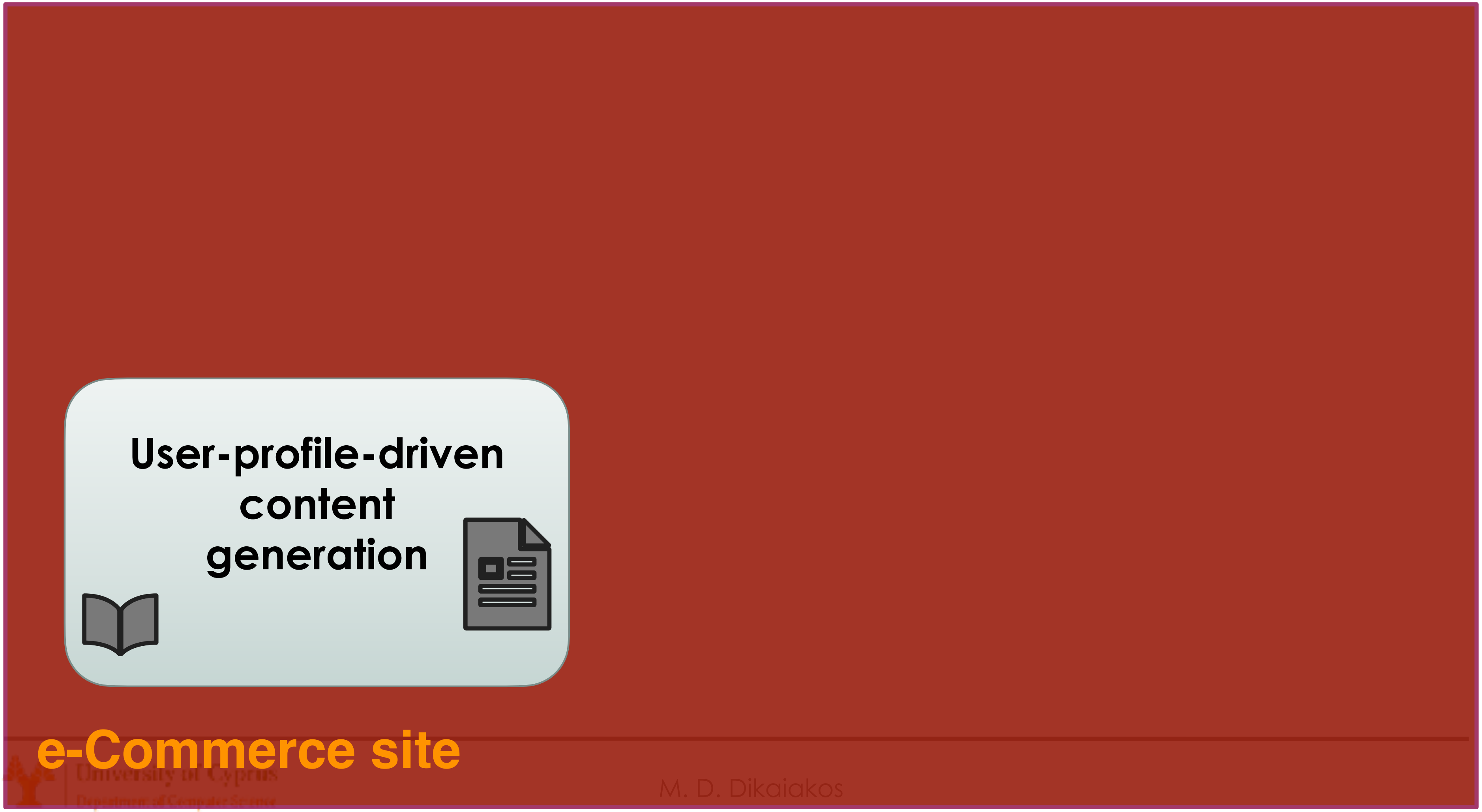
Dealing with CAP's effects

Strategy 2: **Application Decomposition** and **Orthogonal Mechanisms**

- Some large applications can be decomposed into subsystems:
  1. That are **independently intolerant to harvest degradation** (i.e. they fail by reducing yield).
  2. Whose **independent failure allows the overall application to continue functioning with reduced utility**.
- The application as a whole is then tolerant of harvest degradation.

Example

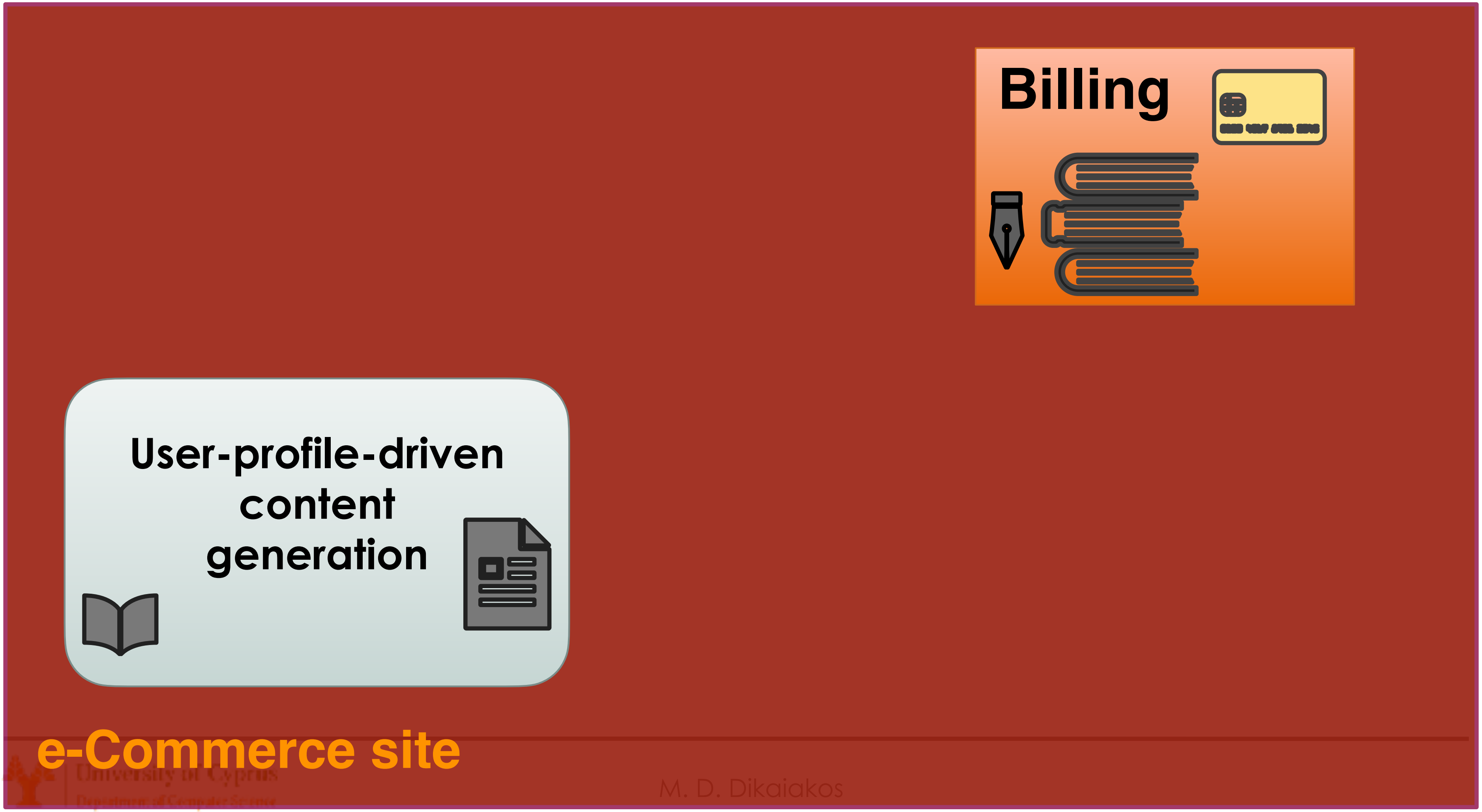
User-profile-driven content generation from a static corpus: read-only subsystem



Example

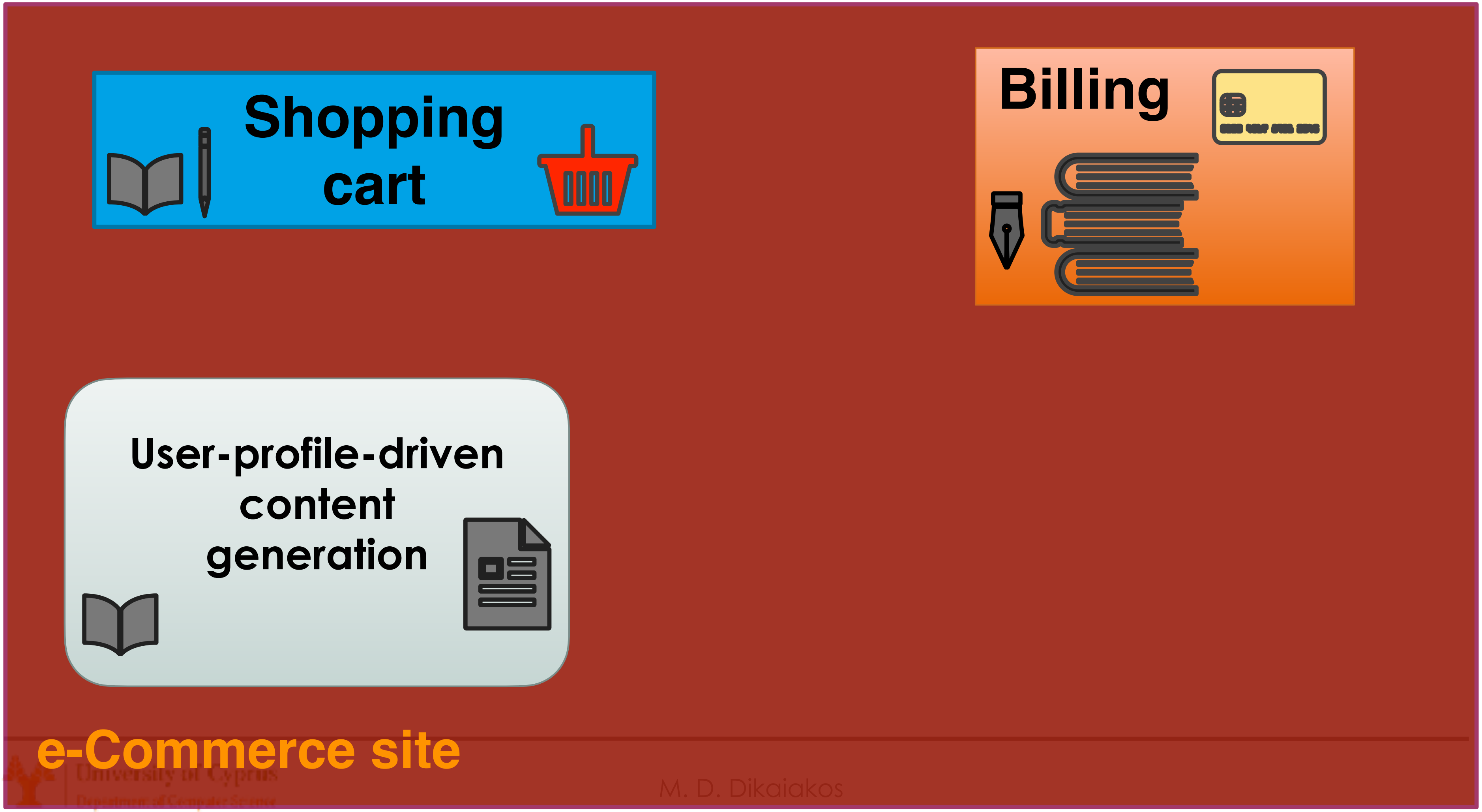
Billing: fully transactional subsystem (read / write / state heavy)

If it fails, the whole system probably needs to stop



Example

Shopping cart: manages state that must be persistent over the course of a session but not thereafter



Shopping cart

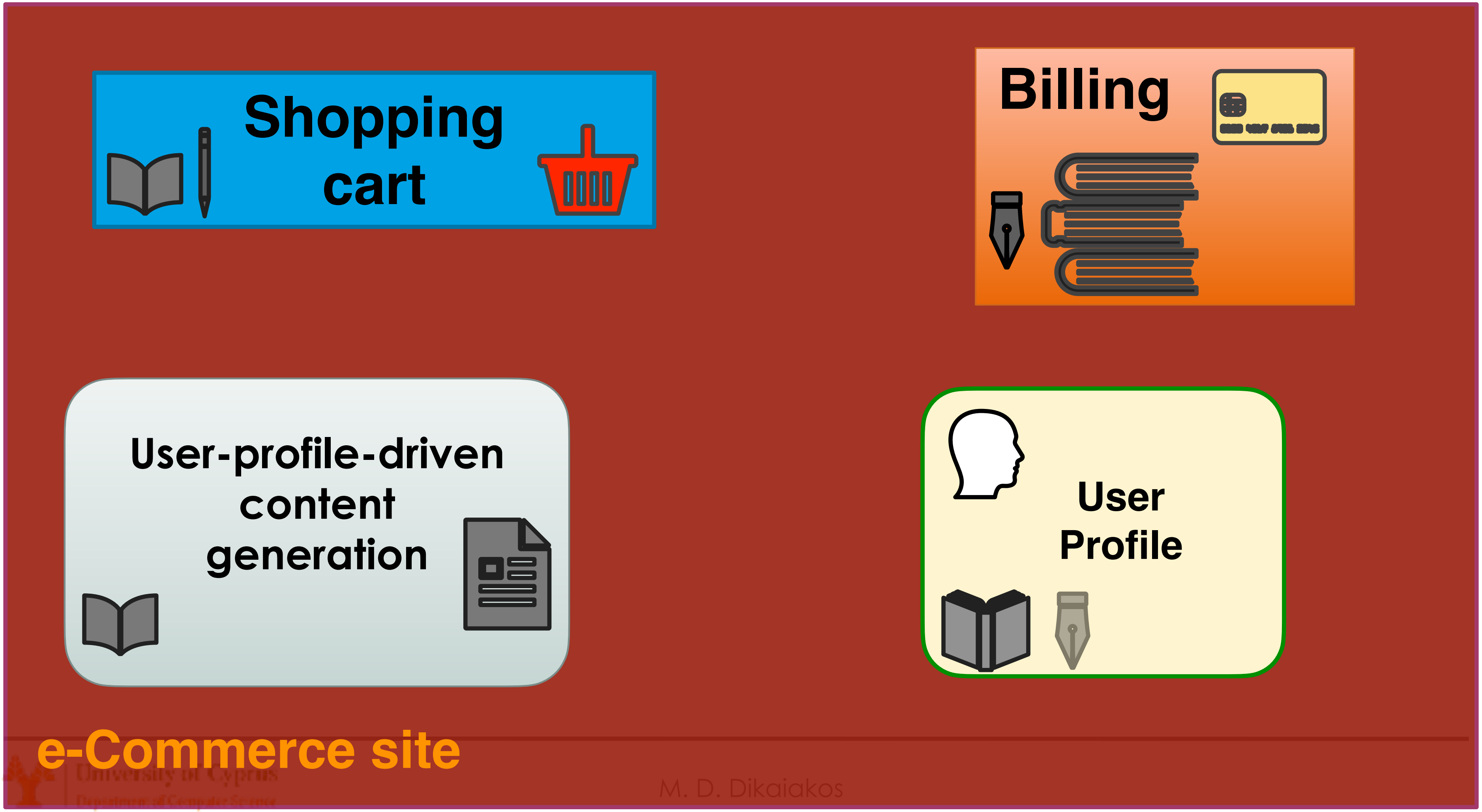
Billing

User-profile-driven content generation

e-Commerce site

Example

User personalization profile sub-system: manages truly persistent but read-mostly/write-rarely state.



Shopping cart

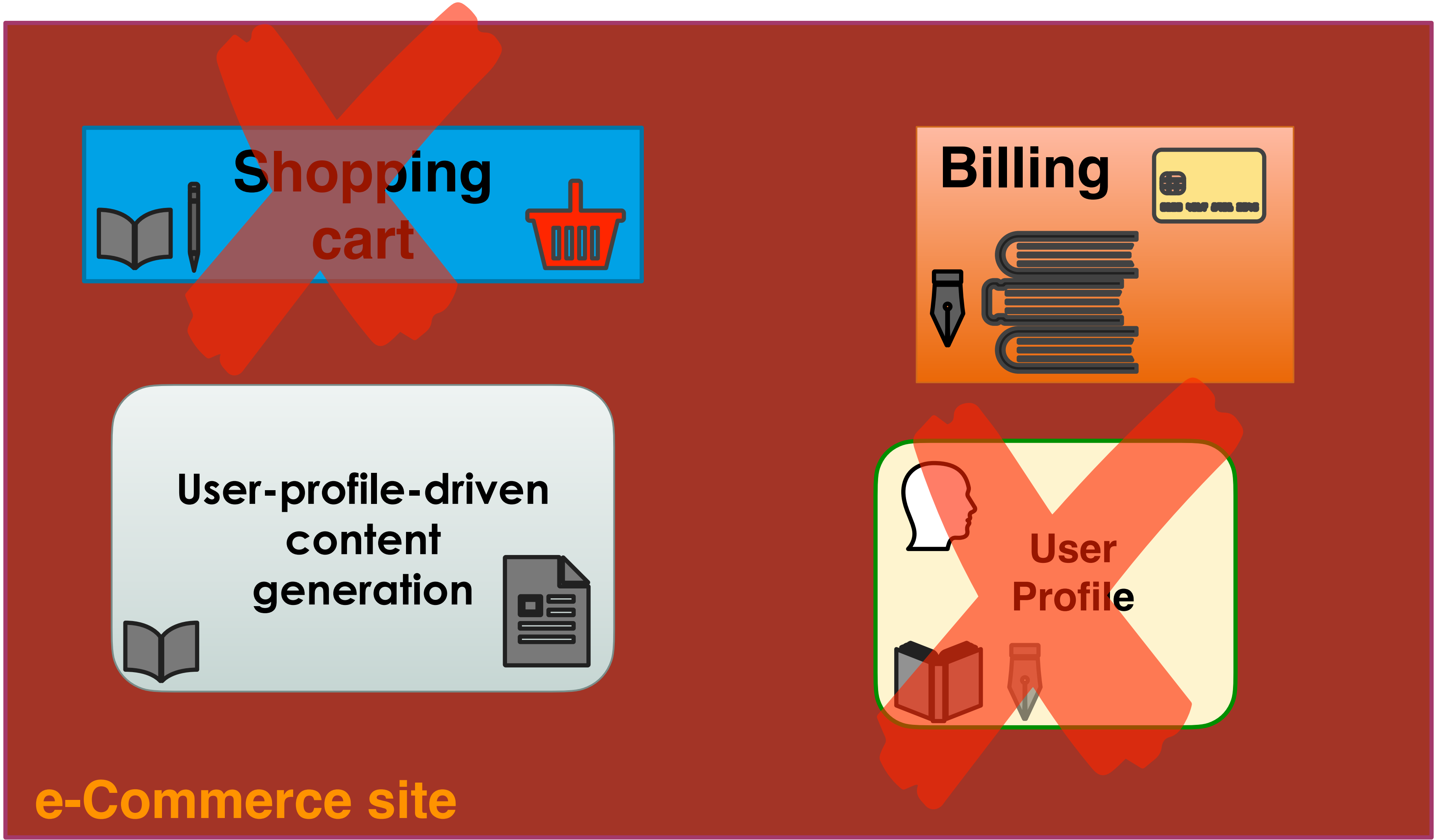
Billing

User-profile-driven content generation

User Profile

e-Commerce site

Example



Shopping cart

Billing

User-profile-driven content generation

User Profile

e-Commerce site

Orthogonal Decomposition

- Traditionally, the boundary between subsystems with differing state management requirements and data semantics has been characterized via **narrow interface layers**
- In some cases it is possible to do even better, if we can identify orthogonal mechanisms:
  - An orthogonal mechanism is **independent** of other mechanisms, and has essentially **no runtime interface** to the other mechanisms(except possibly a configuration interface).
  - Orthogonal approaches are particularly useful in adding operational features such as security or robustness to legacy applications, without requiring special changes to the core application code.

Availability and Performance

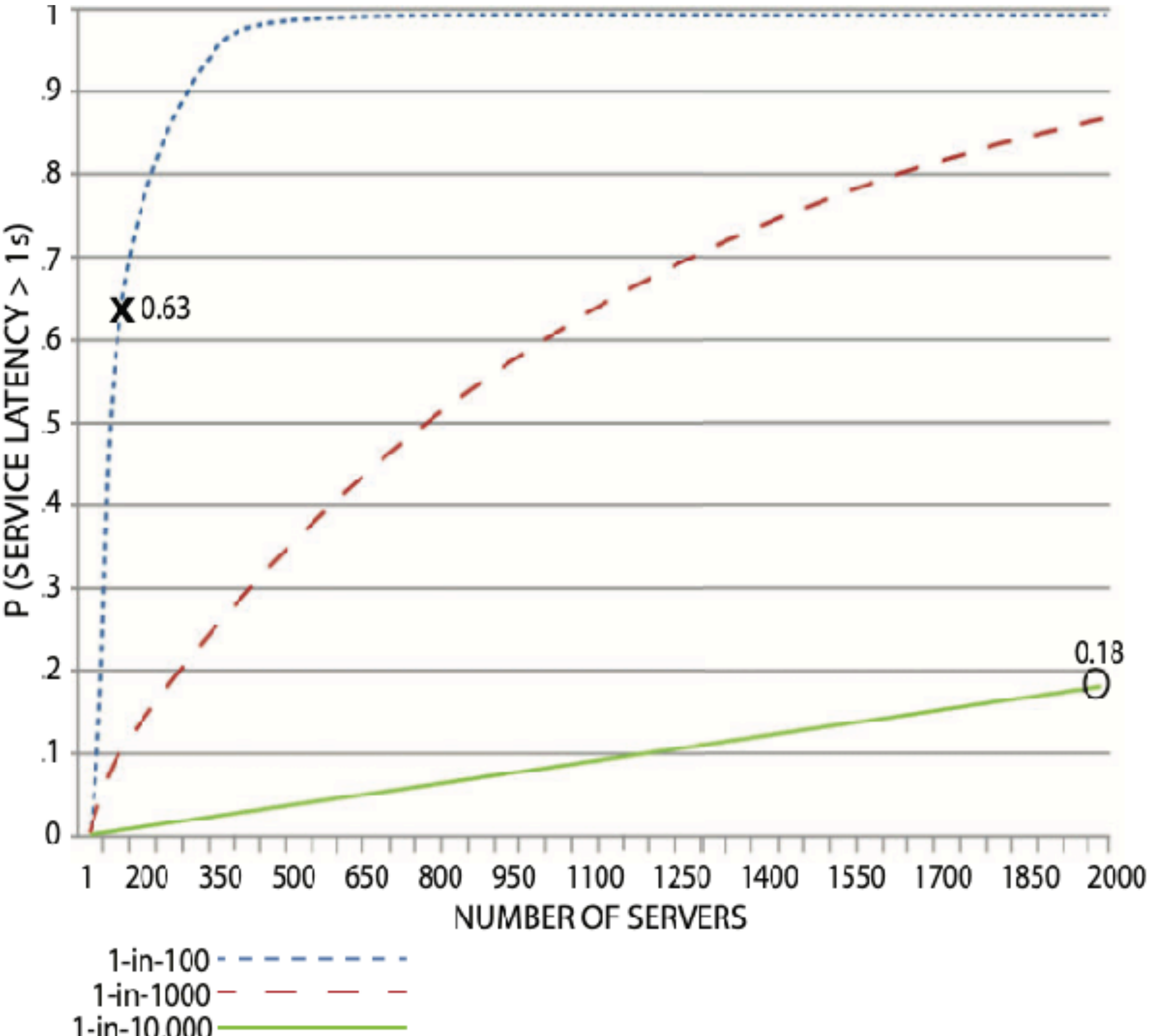
Tail Latency Concerns

Tail-tolerance

- Techniques for high performance and availability are great, but… not enough!
- As systems scale, **eliminating all possible sources of performance variability** in individual system components to deliver service-wide responsiveness with **acceptable tail latency levels** is impractical.
- **Tail latency** refers to the latency of the slowest requests, that is, the tail of the latency distribution.

Tail-tolerance

- Consider a hypothetical system where each server typically responds in **10ms** but with a 99th percentile latency of **1sec**.
  - If a user request is handled on just one such server, **1 user request in 100** will be slow
- If a user request must collect responses from 100 such servers in parallel, then **63%** of user requests will take more than 1 s (marked as an "x" in the figure).


P (SERVICE LATENCY > 1s)
1
.9
.8
.7
.6
.5
.4
.3
.2
.1
0
X 0.63
0.18
O
1
200
350
500
650
800
950
1100
1250
1400
1550
1700
1850
2000
NUMBER OF SERVERS
1-in-100
1-in-1000
1-in-10,000

Take Home Practice

Can you prove the 63% claim?

University of Cyprus
Department of Computer Science

Sample Questions

- Describe the main functionalities of Resource Management software in cloud computing infrastructures. Provide some examples of the inputs that a RMS needs and the objectives it is trying to reach.
- Describe some core functionalities offered by Cluster Infrastructure Management softwares.
- Be familiar and explain concepts like load balancing, sharding (partitioning), replication, integrity-checking, eventual consistency, redundant execution, tail-tolerance in the context of cloud infrastructures.
- Explain what cloud native development means and what are the key requirements that it tries to meet.

Sample Questions

- Failures in you data centre happen once a week. It takes 7 minutes to recover. What is your uptime and how can you realistically increase your uptime by an extra 9?
- Explain which concepts the letters of the acronym CAP correspond to and what the CAP theorem says.
- Describe a key difference in the profile of a cloud workload corresponding to Google's Web search and AdSense services. Explain why this difference can be very important for the design, implementation, and configuration of the underlying cloud infrastructure.