VII. Replication and Transparency

Transparency: *Abstraction from ...* I-36 I-37

◁ peculiarities and deficits of underlying hardware

◁ hard to handle characteristics of geographical distribution

▶ methods to ensure portability and standard conformance of systems

▷ methods to ensure reliable message transfer or encryption etc.

... through the use of intermediate software layers that

▶ provide a more comfortable 'system/programming model'

▶ let the user rely on features without worrying about implementation

$\Longrightarrow$ *Lots of arguments are in favor of transparency*

Disadvantages:

◁ Implementing 'good' and portable middleware itself is hard.

◀ Systems may get 'slower' and costlier at runtime due to overhead.

◁ Most middleware systems provide programming models of their own and require more discipline among users and developers.

Two Main Reasons for Replication

- **Reliability:** *Failures and malfunctioning components of a DS are not visible to the user but will be compensated internally.*
  - Replication of components is indispensable for any robust system.
  - Heterogenous Replication allows for diversification for hardware VII-3
  - Cross-Region Replication reduces network partitioning impacts - VII-5

  $\Longrightarrow$ *Abstraction* from detailed hardware combined with migration & relocation transparency allows for exchanging components.

- **Performance** and **Scalability**: *Distribution does not impede the perceived performance for users. A distributed system should scale for high loads as well as for load variations.*
  - **Replication** is also the key technique for performance:
    - ∗ Data replication and *Caching* for fast access
    - ∗ Service replication based on current system load
    - ∗ 'Service Pools' using on demand activation

Example: AWS Replication in three Regions

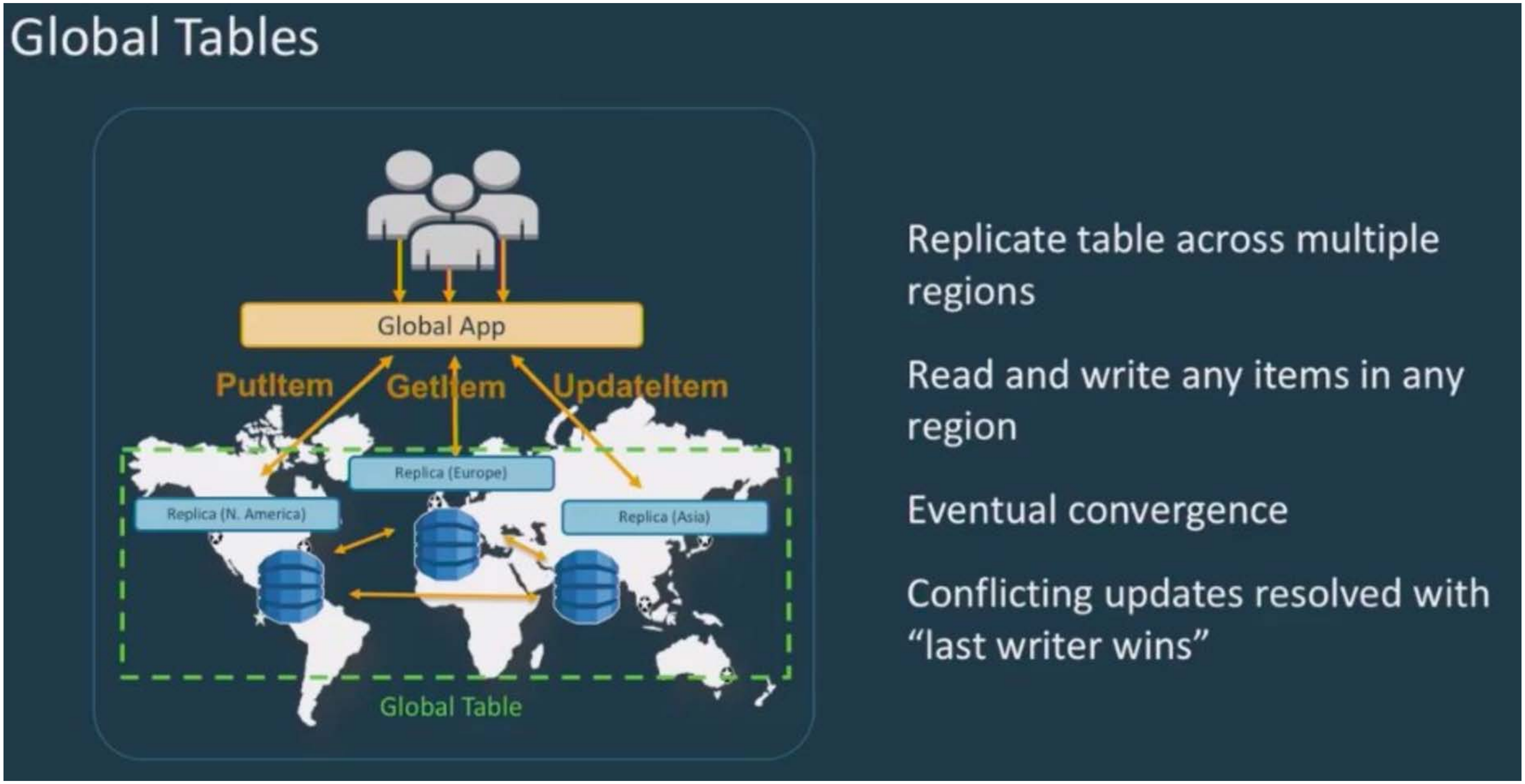


Fig.: pbs.twimg.com/media/DmTn4aVW4AA8dNM.jpg

- Load-Balancing for different regions locally
- Hand-Over in the case of network problems

$\Longrightarrow$ Copying of data and handling of changes

Example: 2 Replicated Service Regions - Running

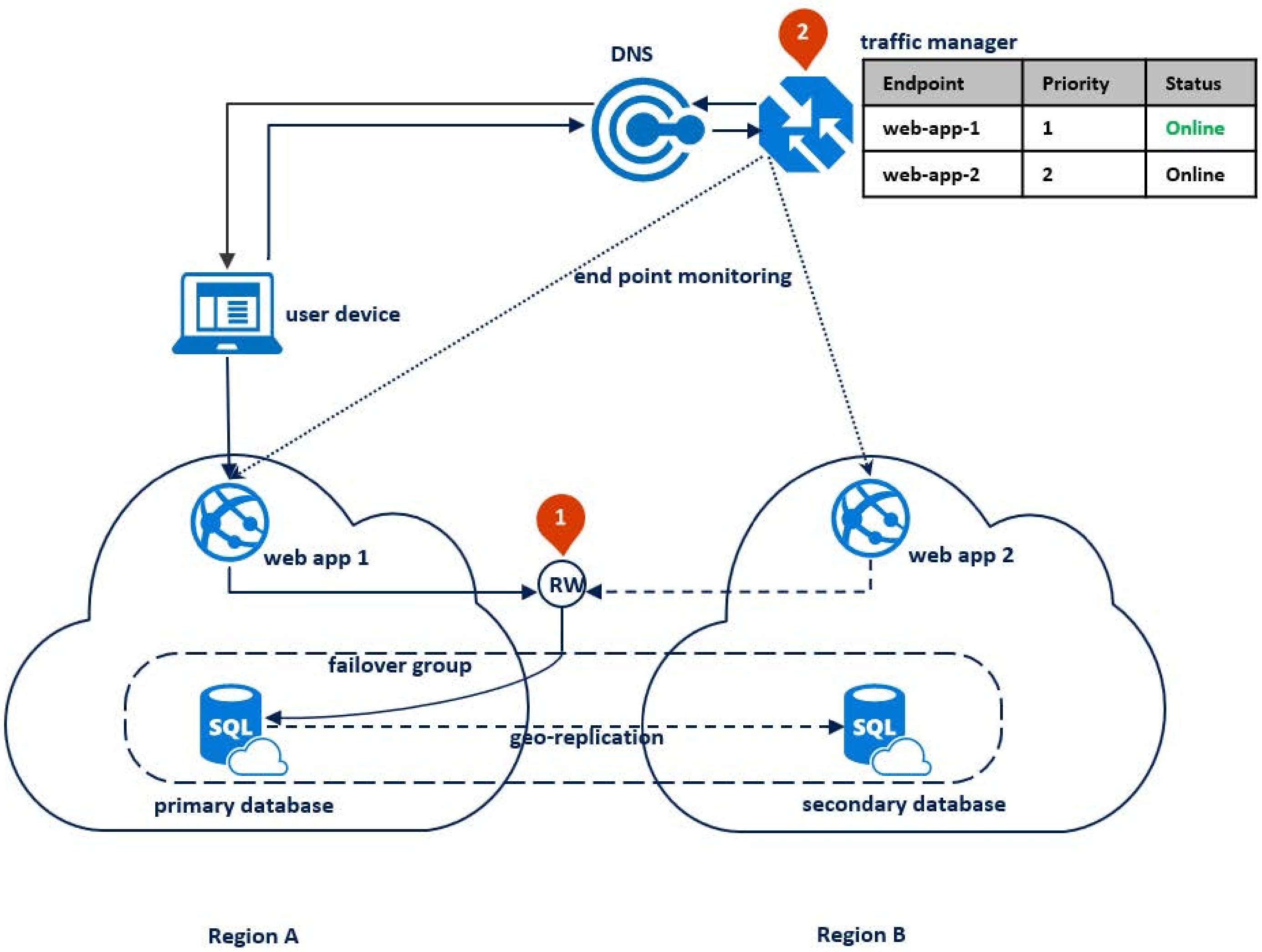
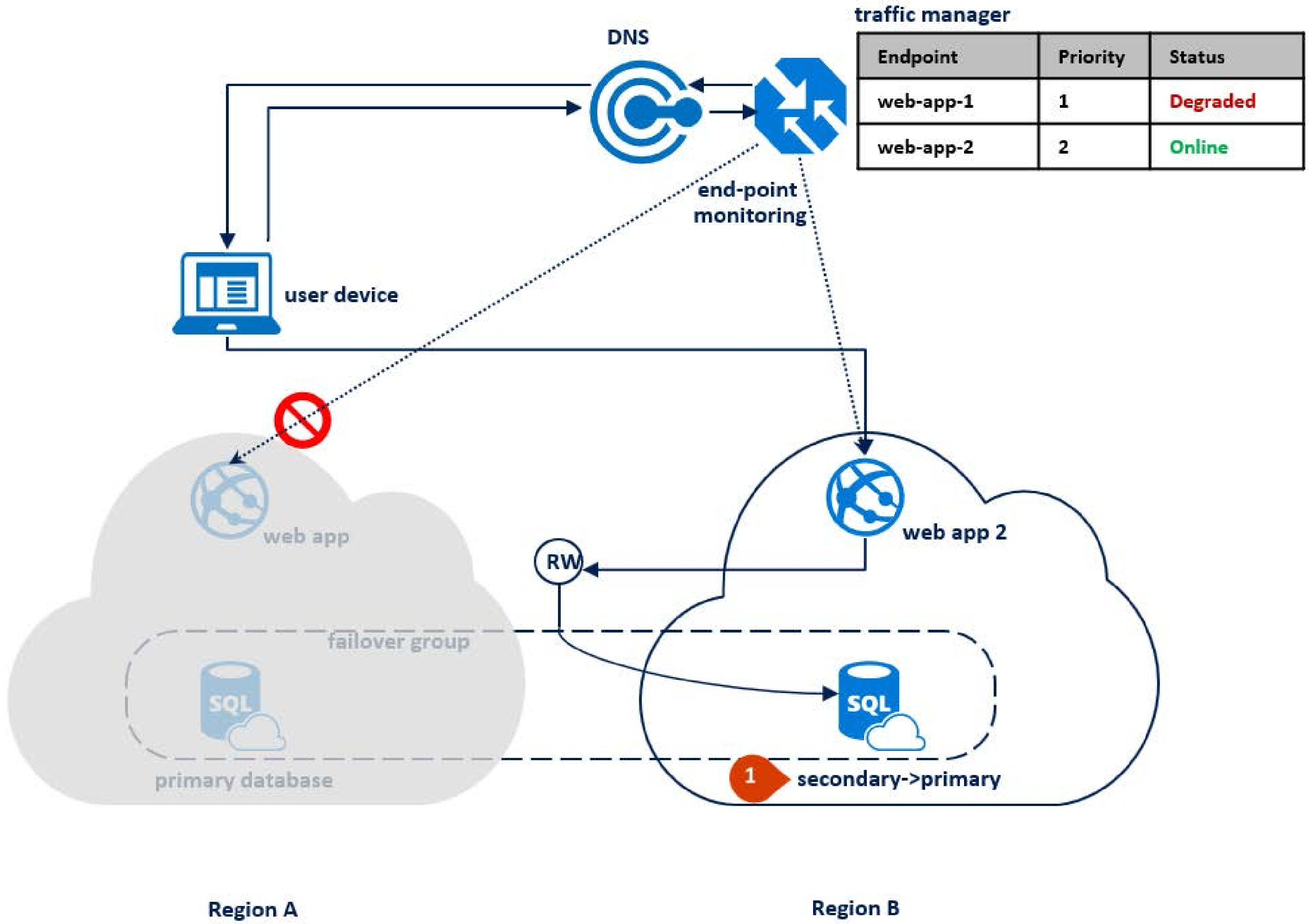


Fig.: docs.microsoft.com/en-us/azure/azure-sql/database/designing-cloud-solutions-for-disaster-recovery

Example: 2 Replicated Service Regions - Handover

Fig.: c.f. pg. VII-4



Endpoint	Priority	Status
web-app-1	1	Degraded
web-app-2	2	Online

Different 'Target Directions' for Replication

1. **Active** components, e.g., compute nodes, communication, server
2. **Passive** components, e.g., 'information', data, database tables

Common characteristics: $Resources$ exhibit typical problems

Availability, competition, bottlenecks

$\Longrightarrow$ Usage discipline needed (Synchronization) **and** Management (deadlocks, fairness etc.)

Common abstract system model: Client–Server V-1/2

- C initiates compute task; S computes; C waits for result
- C initiates read; S transfers data; C waits for result

Different Approaches to make Replication transparent:

VII.1 *Models for Managed Active Servers*

VII.2 *Techniques for Data Replication*

VII.1 Replication of Active Components

- Potential for Replication: All hardware and software layers, e.g., processor, OS components, **services** (daemons), . . ., applications
  $\Longrightarrow$ *Containers and Cloud technologies support replication*
- Different transparency levels w.r.t. **external client view**:
  - **Broker**-Models: communication partner information for clients VII-10
  - Server Addressing: IDs vs. service properties, e.g., yellow pages
  - Server internals: Server state relevant for availability?
    e.g.: Buffer store empty/full; saturated vs. idle server
    $\Longrightarrow$ 'addressing' takes state of requested server into account
  - specialized vs. comparable servers (or worker)
- Server logging w.r.t. Client requests: **stateful** vs. **stateless** VII-15
  Example: RPC semantics w.r.t. failures and repeated requests
- **Load-balancing** essential for performance transparency VII-14

Levels of Replication

- **Iterative Server:** Executes a single job at a time *no replication*
  Example: simple search engine, `batch` compute-server
  - ◀ Client blocks if server is in use
  - ◀ single-point-of-failure

  $\Longrightarrow$ no performance transparency

- **Quasi-concurrent Server:** *virtual replication*
  Many jobs, but only a single job is processed at a time
  Queueing system for jobs or time-sharing via multiplexing
  Example: workstation with single-threaded server OS
  - ▶ low or average load $\Longrightarrow$ performance transparency
  - ◀ high loads $\Longrightarrow$ no performance transparency
  - ◀ potential for deadlocks for 'truly parallel' usage
  - ◀ single-point-of-failure

Levels of Replication cont'd

- **Concurrent Server:** *monolithic replication*

  Example: Multiprocessor workstation; centralized DB server

  ▶ no problems w.r.t. *consistency* on server level

  ◀ single-point-of-failure

- **Distributed, concurrent Servers:** *Always 'true' replication?*

  1. **Co-operative Server:** multiple servers required for a single job
     e.g., distributed file system (NFS) *no replication*

     ▶ Load balancing ensures performance transparency

     ◀ multiple single-points-of-failure (**AND**-Model) I-28

  2. **Replicated Servers:** > 1 autonomous server **true replication**
     i.e. each **single** server is able to process complete jobs
     Example: redundant name services, mail server, DB systems

     ▶ Performance as well as failure transparency (**OR**-Model) I-28

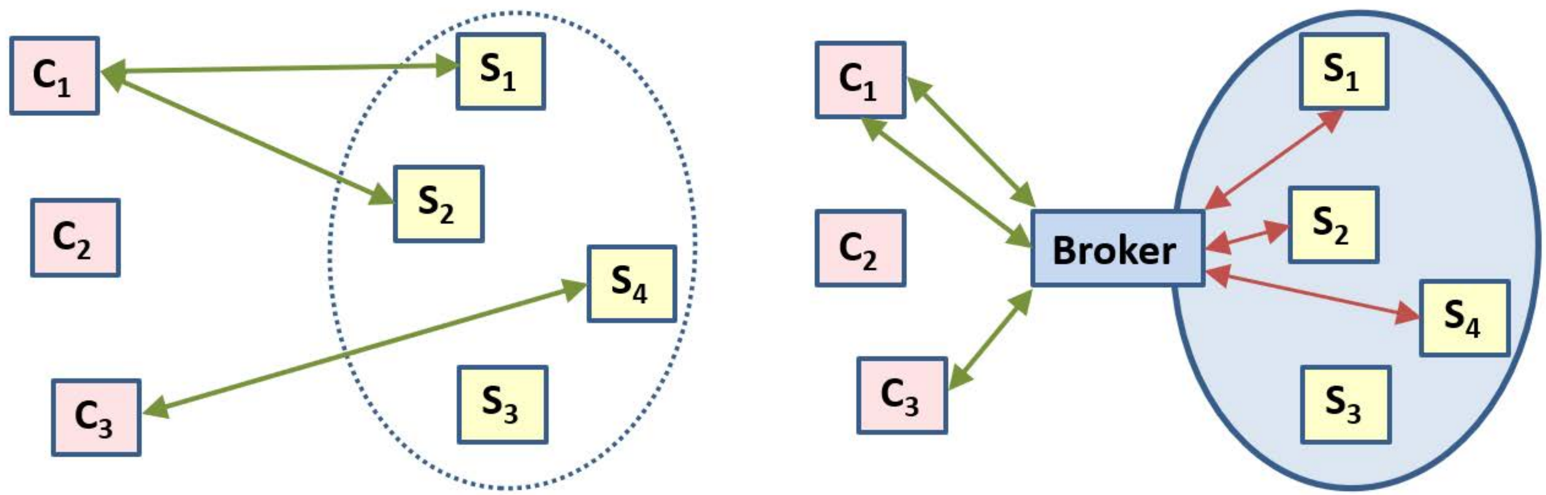
     ◀ costly w.r.t. hardware and software, esp. Consistency

Broker Models: Architectures for True Replication

◀ **Without Broker:** Client has to know about all servers (needed) lhs

◁ transparency missing, distributed load-balancing impossible

▷ no overhead due to broker interposition



▶ **Using a Broker:** abstracts from concrete naming schemes rhs

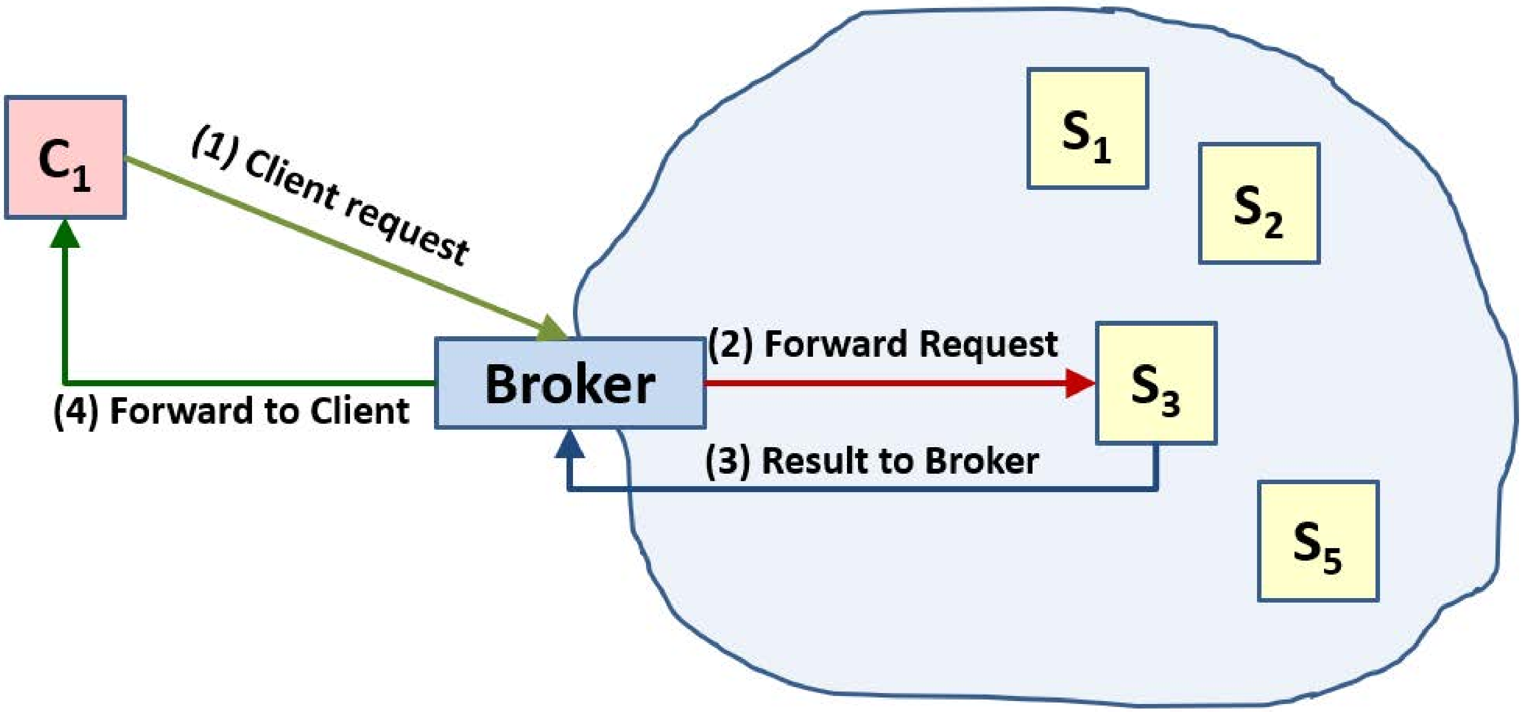
▷ **Naming transparency** (except Broker or **broadcast** search)

▷ Broker controls load $\Longrightarrow$ load-balancing much easier

◁ Overhead due to broker interposition

Broker – Server Organization

- **Forward** Design: *internal delegation*
  only broker visible/known to client; broker handles request/result
  Problem: additional overhead for copying and communication

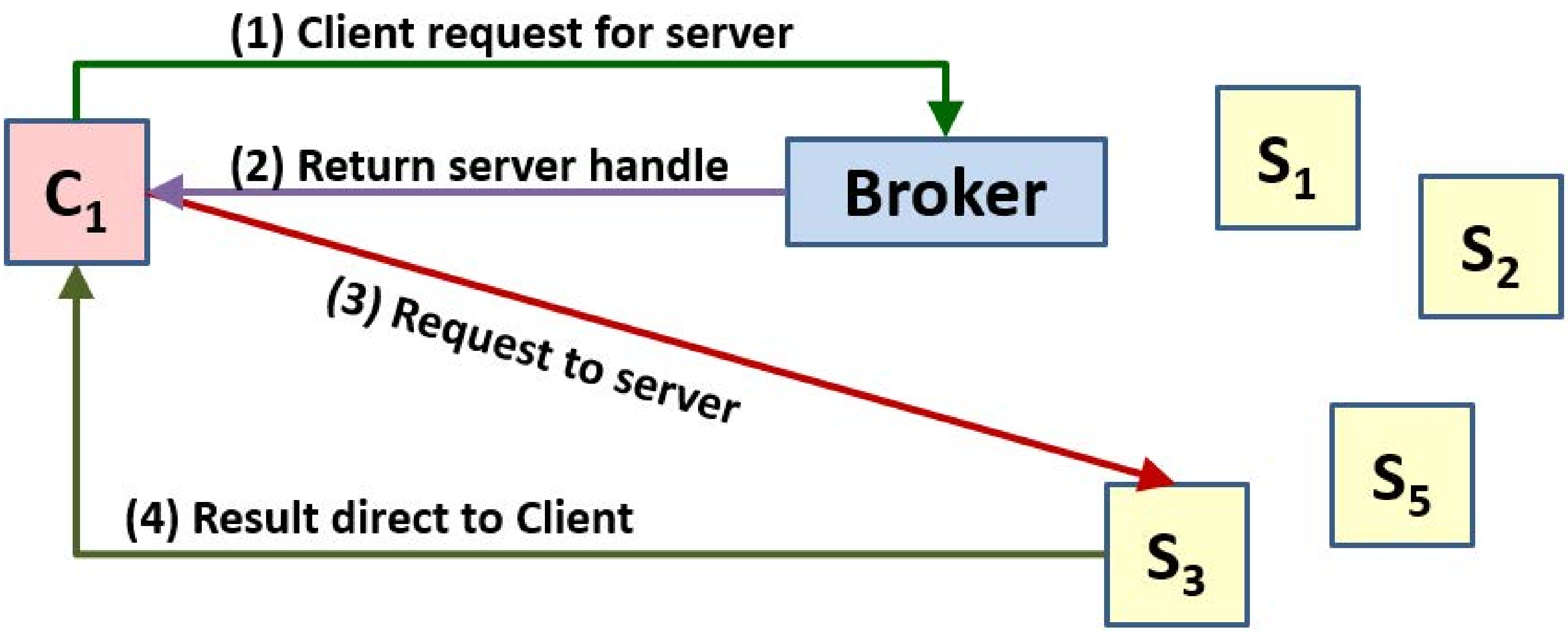


$\Longrightarrow$ Client cannot re-use Server without Broker

Broker – Server Organization cont'd

- **Handle-driven** Design: *externally visible delegation*
  Client sends inquiry to broker; broker sends *server handle* to client
  direct server–client interaction for request/reply (Security?)
  Problem: Client side caching of addresses annuls broker decisions

cf. Web Services

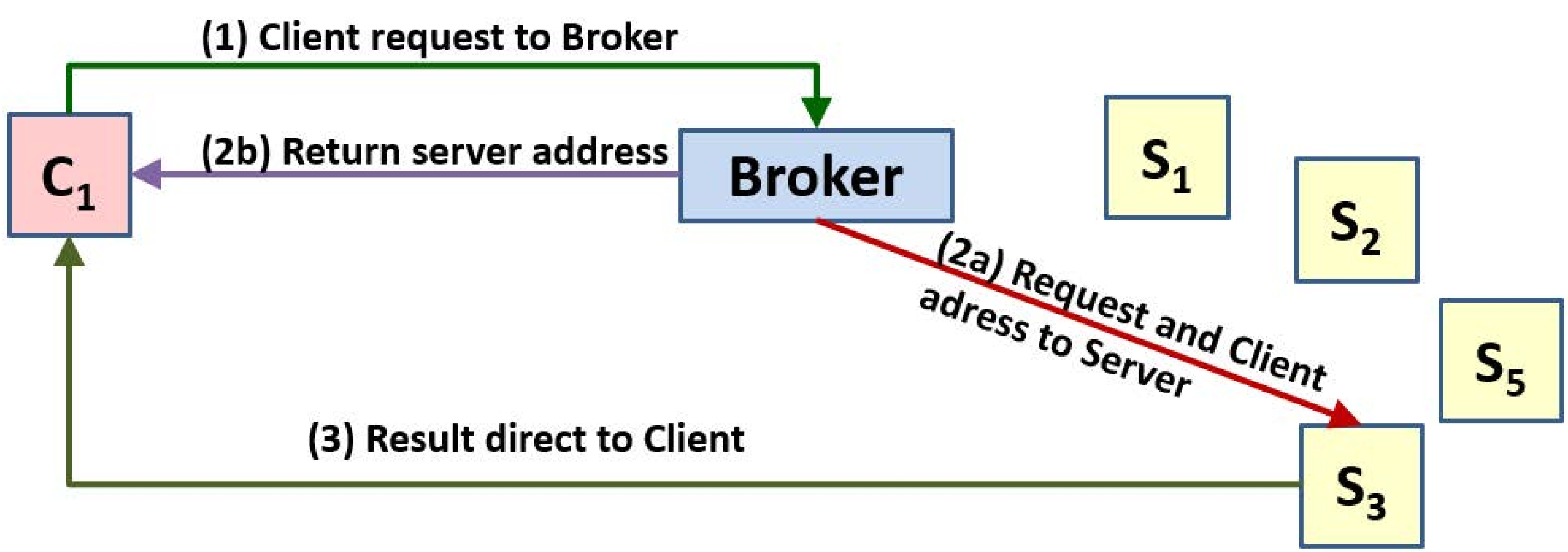


$\Longrightarrow$ Avoids Broker Bottleneck to some extent

Broker – Server Organization cont'd

- **Hybrid** design: *combination of other models*
  Client sends (full) request to broker;
  Broker hands over client address plus request to suitable server
  and server address back to client
  direct server–client interaction used for result only
  Problem: costly requests are sent even if no server is available

V-13 call back



Trade-Off: Broker bottleneck vs. dissemination of system knowledge

Outlook: Load-Balancing required in any DS/PS

- **Models:** How to assign requests/jobs to servers?
  - **centralized:** Broker decides how to distribute load
    Basis: Information via *forwarding/handle-driven*
    Adaptive Pool: start/terminate servers based on current load
    e.g., avoid *cold-start* problem in container/cloud environments
  - **de-centralized:** Avoids broker bottleneck problem
    - One role for *Server/Broker*: If a server has high loads, it acts as broker and *forwards* job to other server.
    - **Bidding** and **Forwarding**: broadcast search for server first (or 'best' w.r.t. QoS etc.) answer wins

- **Problem:** Complex jobs that require a couple of (different) servers
  $\Longrightarrow$ **Dependencies** complicate load distribution and introduce additional boundary conditions
  $\Longrightarrow$ Hierarchical designs are easier: Broker, Server, Sub-Server

Stateful vs. Stateless Servers

- **Stateless service**: No state information about jobs processed
  e.g., Time-Service, Name-Service need no client information
- **Atomicity-of-requests**: server stores jobs under processing
  e.g., transaction models; Assignment: Job $\longleftrightarrow$ Server
  Avoiding duplicate processing for at-most-once RPC semantics
  ▷ fault tolerance due to repeated execution of jobs
- **Stateful service**: processed jobs result in server state changes
  e.g., DB data storages; File-Server
  ▷ information re-use avoids communication and allows for efficiency
  ◀ assumptions about 'global' system state among servers/clients?

Example Trade-off: performance vs. fault tolerance in a file system

- classical: open file pointers; hot stand-by replication
- stateless: each new request triggers 'full' overhead, e.g., `http 1.0`.
- **optimal:** external stateless – internal caching with replication

Perspective: Service and Cloud Eco Systems

c.f. DSG-SOA-M

- **Service Descriptions:** capture non-technical information, too !
  - **Interfaces** define operations and parameter types
  - external **protocols** use services via RPC, msg passing, ...
  - internal **state information**: specifies which ops are available
  - QoS attributes define robustness, security, trustworthiness ...
- **Infrastructure: Offer, Search, Find, Bid/Negotiate, Choose**
  - Compute, Storage and Communication Resources via Clouds
  - **Broker acts as a 'Trader'**: teams up Requester and Provider
  - free/with costs directory services ... electronic markets
  - Search/Match based on functionality, state and attributes
  - Compensation for unavailable services, replicated execution etc.
- **Pricing, Negotiating and Billing of services**

Agent platforms

$\Longrightarrow$ **Long-term Goal:** Service Ecosystems with **on-the-fly** service replication and composition at run time.

Example: Car Data Eco Systems

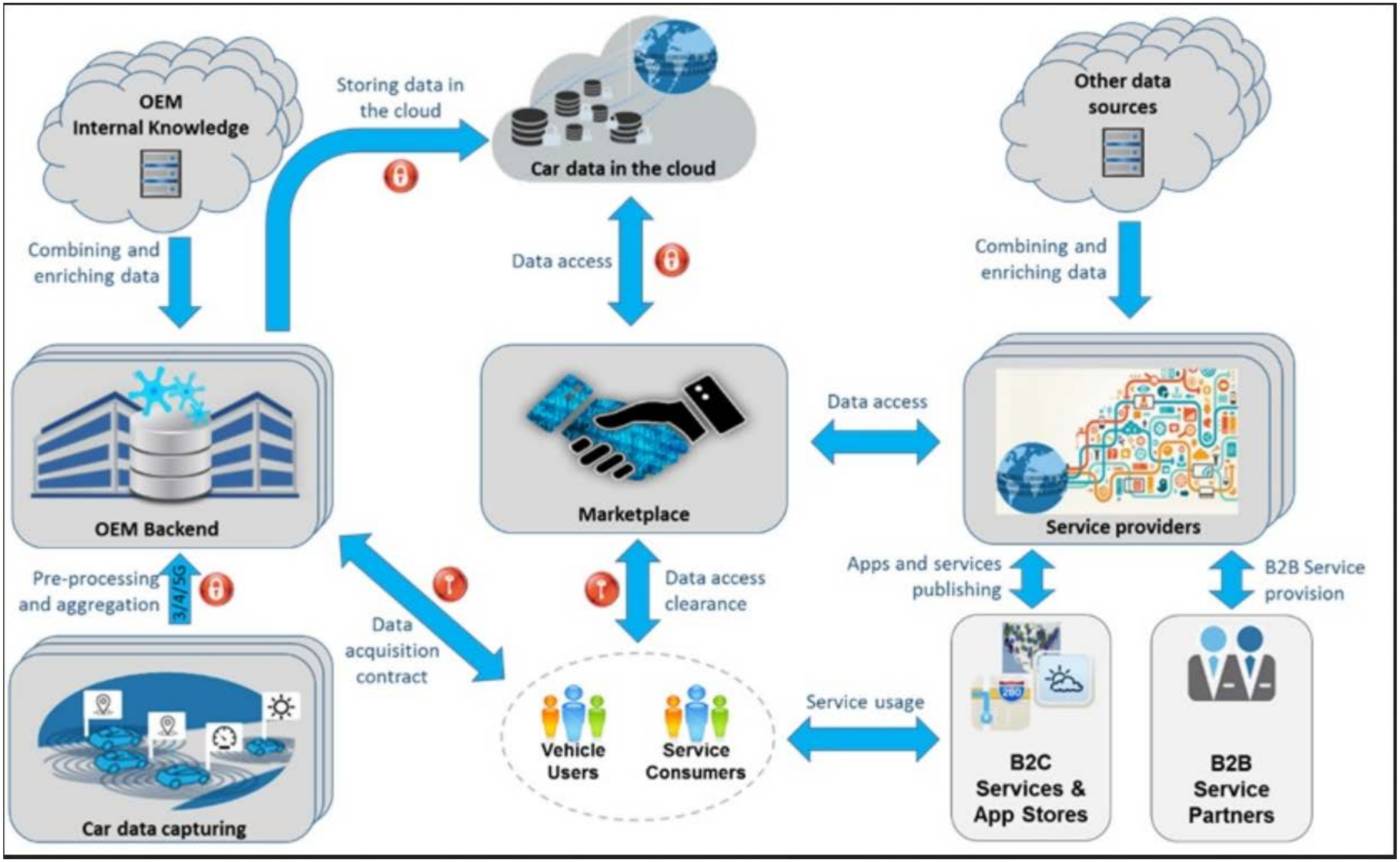


Fig.: cordis.europa.eu/docs/results/h2020/644/644657_PS/figure-1-automat-eco system-modules-and-actors.jpg

VII.2 Replication of Passive Components

passive $\approx$ Data: Text, Tables, Objects, DB entries,
Variable, Segment/Page/Word

Different Aims require different Measures:

- **Failure Transparency:**
  - 'Duplicating' data mandatory for robustness
  - 'Healing' via exchange protocols to recover data from copies
- **Performance Transparency:**
  - local vs. remote access and *locality* principle $\Longrightarrow$ Caching
  - distribute peak loads by means of distributed data access

▷ *Virtual Shared-Memory Programming Model* K. Li 1986
remote data are loaded *on demand* $\approx$ ''*networked MMU*'
$\Longrightarrow$ **Location transparency** on programming model level

Choice of Techniques motivated by Usage Context

Trade-Off: Advantages of copies **vs.** Costs for consistency
$\Longrightarrow$ **No strategy matches 'all' use cases**!

Wide range of applications $\Longrightarrow$ **Different profiles**

- **Example:** WWW client using a local cache:
  - Web presence of a company: Write-Ops are rare
    optimistic $\Longrightarrow$ Client initiates consistency checks (re-load)
    long time periods before expiration date
  - Stock exchange quotation: Write-Ops are frequent
    pessimistic $\Longrightarrow$ expiration date is always set to 'now'

  Client-side advantage: **centralized** Write-Ops
  Remark: many Clients $\Longrightarrow$ **Proxy** used as an efficient cache
- **Example:** Parallel shared-memory programming: many Writes
  pessimistic $\Longrightarrow$ Wait and lock data before Write-Ops

Suitable Techniques for Transparency Aims

- **Performance Transparency**: wide scope of techniques and VSM
  - ▷ HW, system software, middle-ware, programming: **Caching**
  - ▷ Copies on platforms with comparable 'reliability' and performance
  - ▷ Distribute copies dynamically based on usage
  - ▶ *Optimistic* strategies are admissible
  - ▷ **Migration** may also be an alternative for sequential use of data at different locations

- **Failure Transparency:** highly restricted scope of techniques VII-27
  - ◁ **Multiple copies at different locations** are mandatory
  - ◁ Wide range w.r.t. performance: fast . . . persistent storage media
  - ◀ *Pessimistic* strategies are required
  - ◁ Strong synchronization combined with logical coupling is costly

    Trade-Off: high costs vs. 'critical' periods in case of crashes

Remark: Similar Trade-Off as number of snapshots and rollback cf. VI.4

Replication: Original(s) vs. Copies

'Naive' Consistency: Copies are always identical to 'original data'

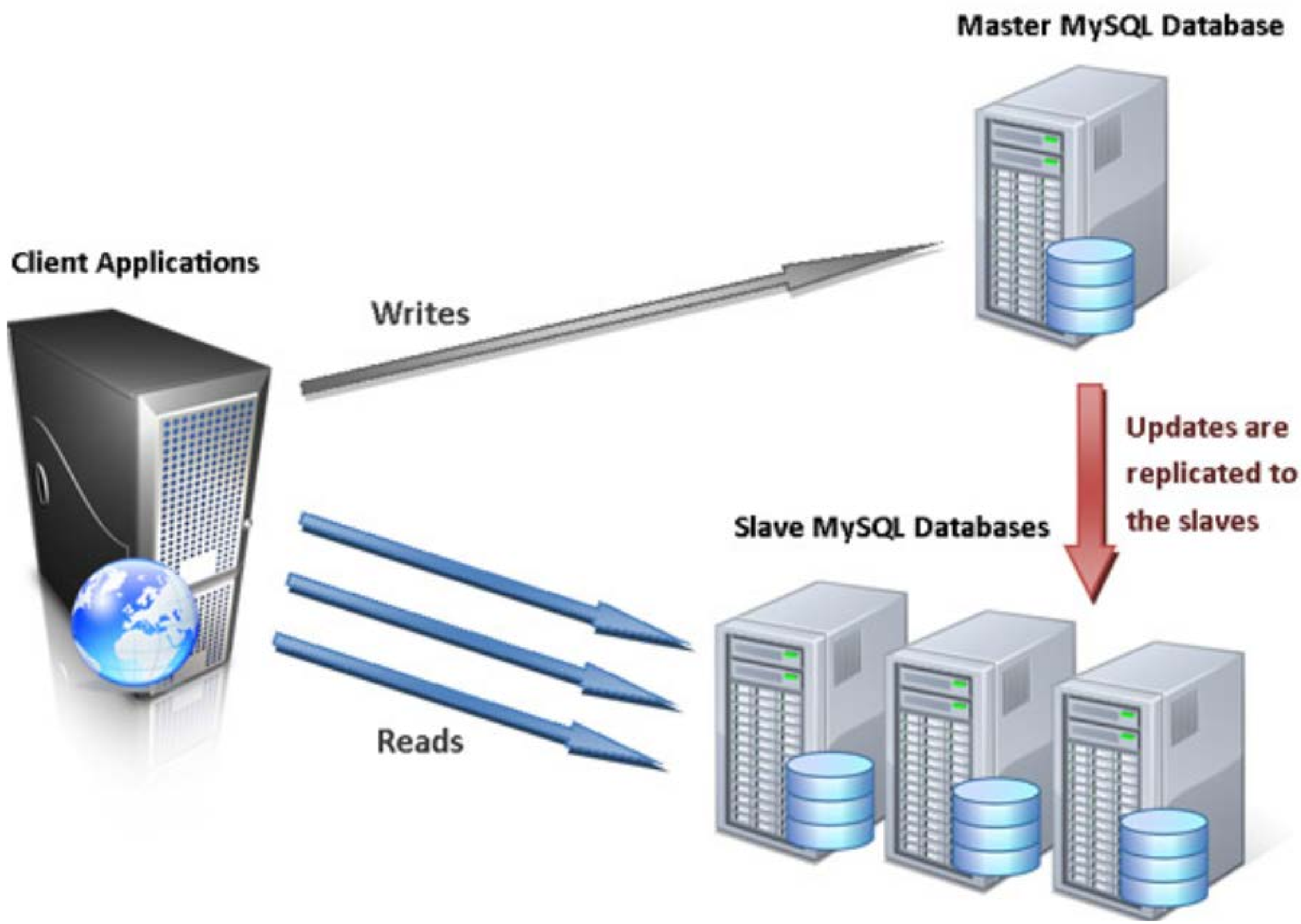
$\Longrightarrow$ *Strict definition not feasible in distributed systems?* see VII.2.1

Three Replication Levels:

0. *Single Read Single Write* $\approx$ No replication
   Migrate original to location of usage (trashing problem)
1. **M***ultiple* **R***ead* **S***ingle* **W***rite* $\approx$ **Read Replication**
   only a few writes but lots of read accesses
   Write is only allowed on a **single original**; Read on n copies VII-23
2. **M***ultiple* **R***ead* **M***ultiple* **W***rite* $\approx$ **Full/Write Replication**
   Writes at different locations $\Longrightarrow$ enhance write performance
   Writes are allowed on copies; **Consistency hard to achieve** VII-24
   Essential decision: rely on a single 'original' vs. majority quorum

Example: Non-Transparent MRSW Configuration

Fig.: www.learncomputer.com/mysql-replication/



MRSW – Roles and Strategies

- **Unique Owner** $\approx$ holds the single 'original' that is written
- **Copy Set** $\approx n$ Processes hold (almost) identical copies
- **Manager** $\approx$ co-ordinates write accesses $\Longrightarrow$
  Invalidation of Copies vs. **Propagation** of 'new' original VII-25

Variants of Implementations: How to implement the **manager role** c.f. Broker

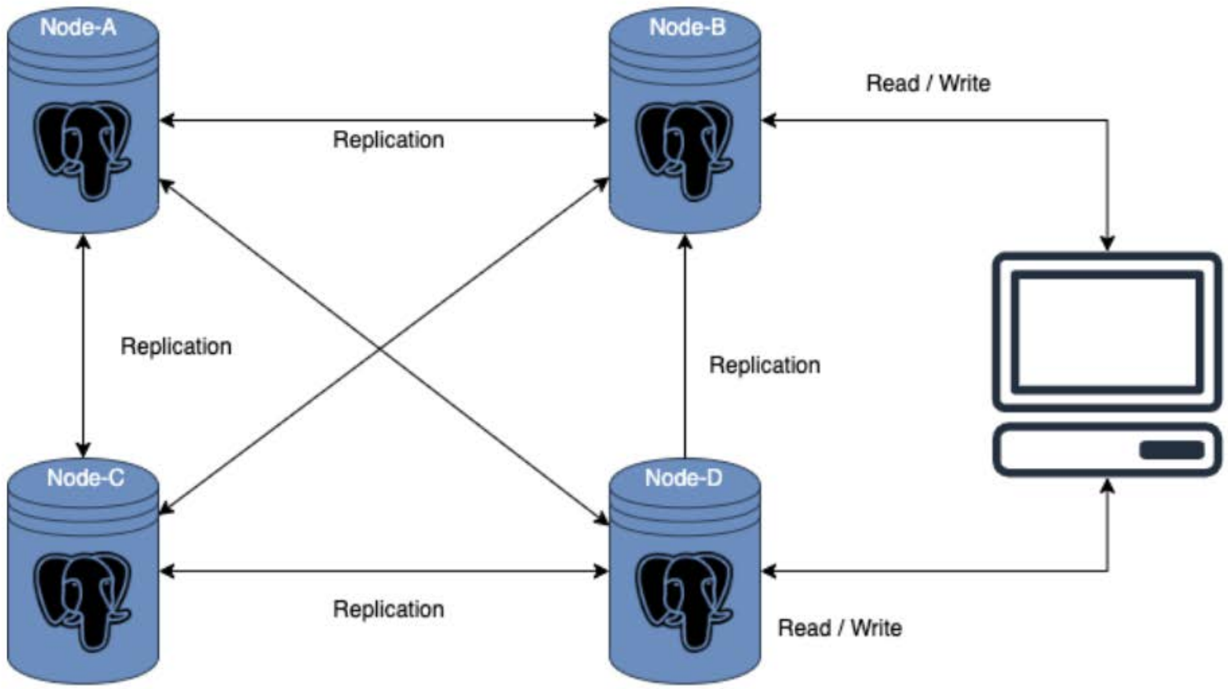
1. centralized: Manager organizes all accesses and delegates to owner
2. *Owner invalidates*: Owner = Manager
3. Distributed Manager:
   (a) fixed distribution of Owner roles to different processes realistic?
   (b) dynamic distribution: Owner role migrates with Write accesses
   Owner information has to be retrievable; outdated $\Longrightarrow$ forward
   (c) *Broadcast and (fixed) distribution*: Requests go to all processes
   $\Longrightarrow$ all processes check all requests, but only Owner responds

Global knowledge assumption? critical for all other than 3.(c)

MRMW – Additional Role and Strategies

Fig.: www.percona.com/blog/2020/06/09/multi-master-replication-solutions-for-postgresql/

▶ **Owner Set** $\approx$ many 'Originals' that can be written



- Before or after Write $\Longrightarrow$ all processes of the Owner Set informed
  - ◁ Manager manages Owner Set as in centralized mutex setting
  - ▷ Agreement Protocol among Owner Set in case of changes
- Manager provides sequence numbers to sequentialize changes

Basics for Handling Changes of Data

How to Propagate Changes (*Coherence Policy*): Original or a single MRMW copy is written:

- **Write-Invalidate**: Invalidate all copies in Copy/Owner Set
  $\Longrightarrow$ avoids communicating complete, possibly huge objects for short read periods by using (ObjID,Flag)-Msgs
- **Write-Update**: Propagate new original to Copy/Owner Set
  $\Longrightarrow$ copies that are valid over long times due to infrequent writes

Note: Leasing of copies helps to reduce number of copies.

When to Propagate Changes: New value is published . . .

- *synchronous:* after acknowledged change from **all** copies
- *asynchronous:* directly and process to update or invalidate is triggered directly afterwards
- *semi-synchronous:* after acknowledged store at a predefined minimum set of nodes, e.g., Owner vs. Copy set

Trade-Offs for Performance Transparency

few vs. many copies $\Longrightarrow$ Costs for invalidate/update

single vs. multiple write $\Longrightarrow$ Costs for consistency mechanisms

`write`-Bottleneck vs. Overhead for sequential consistency

Size of replicated data chunks: (Units)

- Segments, Pages, Records, Words (technical view) c.f. Page-size in OS
- Objects, DB entries, methods, variables (logical view)

Small Units:

▶ fewer conflicts $\Longrightarrow$ less write overhead
update strategies are suitable

◀ lots of Copy Faults $\Longrightarrow$ more read overhead (locality) in application

Big Units:

▶ fewer Copy Faults due to locality

◀ more Write conflicts: **false sharing** due to 'shared units'
lots of copies and more write Overhead $\Longrightarrow$ invalidate suitable

Reduced Options for Failure Transparency

VII-20

Different types of Errors: (flawless communication pre-assumed)

1. **Copy** is lost (no longer accessible)
   - ▶ Read Copy: similar to a read fault $\Longrightarrow$ no problem at all
   - ◀ Write Copy: update current Owner Set
2. **Original** is lost
   - ◀ MRSW or MRMW using a single Owner $\Longrightarrow$ Algorithm fails
   - ▶ If there are copies: **Majority** determines 'new' Original Voting

$\Longrightarrow n > 1$ **distributed 'Originals' have to be kept consistent**

- More than one write copy (Originals)
- If Original is lost $\Longrightarrow$ Election Algorithms for new Original
- Majority vote among Write copies in case of inconsistencies

3. **Network Partitioning:** worst case because not all of the Originals/Copies are accessible!

see VII.2.2

Example: Adaptable Algorithm for both Objectives

IE^3 Concurrency, June 2000

Idea: Boundary Restricted Multiple-Reader Multiple Writer where

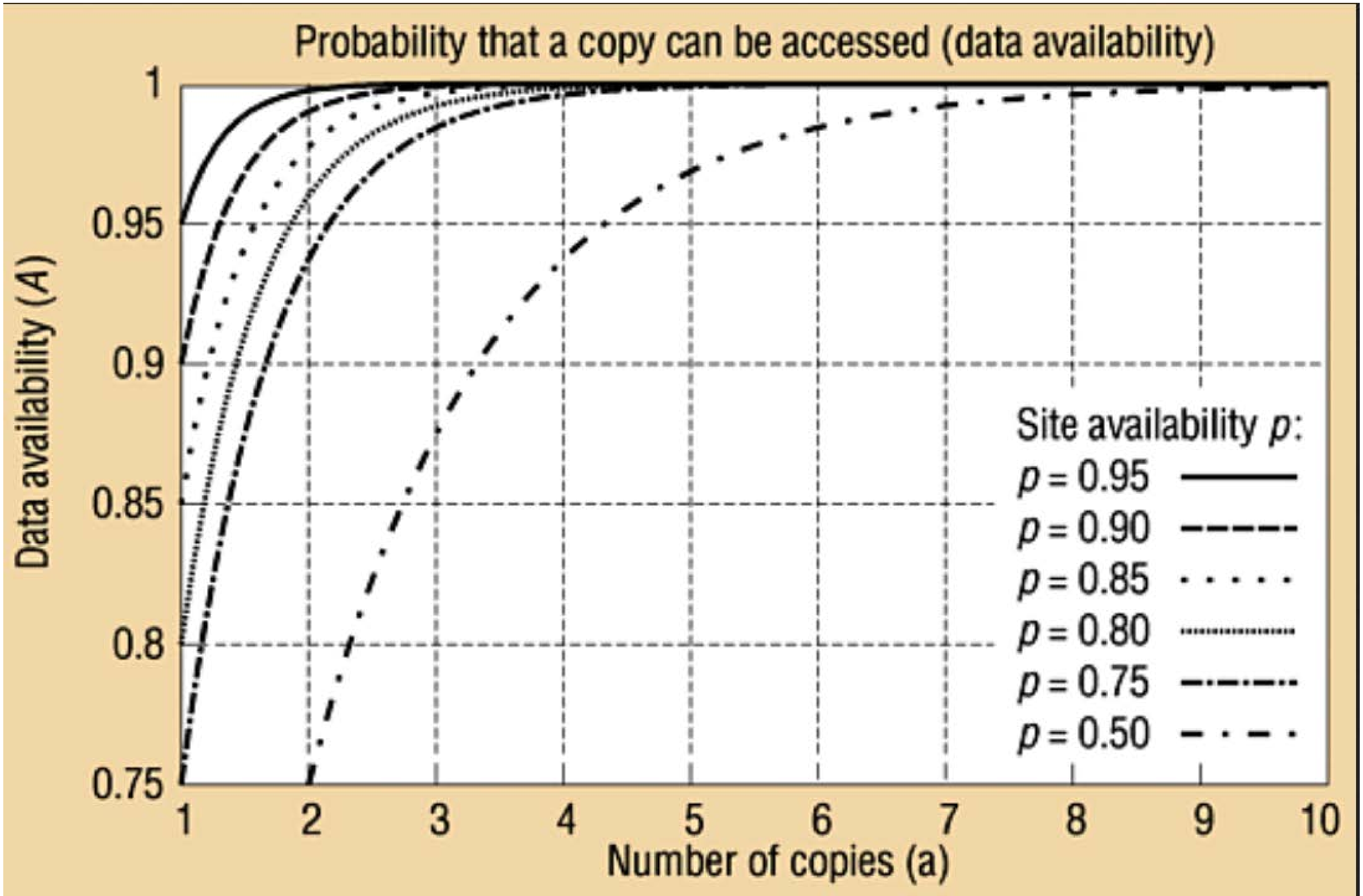
- **Min**imum R/W_{min} numbers of copies guarantee **reliability**
- **Max**imum R/W_{max} numbers reduce **consistency costs**

Algorithm: (*implements 'sequential consistency'*) VII-33

- **3 Levels of Access Rights** for nodes w.r.t. Read/Write:
  - (local read, local write) $\Longrightarrow$ 1 Original
  - (local read, global write) $\Longrightarrow$ Copy handling like MRSW
  - (global read, global write) $\Longrightarrow$ no copy available, i.e., locked
- Alternating **Phases** w.r.t. **global** Reads and Writes

  Read request: assign copy; ensure R_{min} copies are distributed
  If R_{max} copies are distributed reduce (invalidate) copies first.
  Write request: assign copy; ensure W_{min} copies are distributed at least
  Reduce the number when performing the update via invalidate.

Availability: a Copies vs. Node Availability



B. Fleisch, H. Michel et al.: Fault Tolerance and Configurability in DSM Coherence Protocols. IE^3 Concurrency 8(2) June 2000, pg. 10-21

VII.2.1 Consistency Models

Objective: Approximate properties of centralized shared memory in a message-based widely distributed system.

- ▶ MRSW: 1 Owner $\Longrightarrow$ organization of changes **easy**
  Delay: Write in one process vs. propagation via messages
- ◀ MRMW: additional overhead to localize the 'most recent copy' even parallel writes may be allowed
  Delay: Localization plus propagation times

Naive Goal: (*Single-Processor Strict Consistency*)

Any read of a memory cell x results in exactly that value that has been written into x by the **most recent** *write.*

- ◁ writes and reads 'almost' at the same time
- ◁ upper limit for message transfer is 'speed of light'

$\Longrightarrow$ **Strict consistency is no reasonable objective in DS!**

Realistic Approximations for Strict Consistency

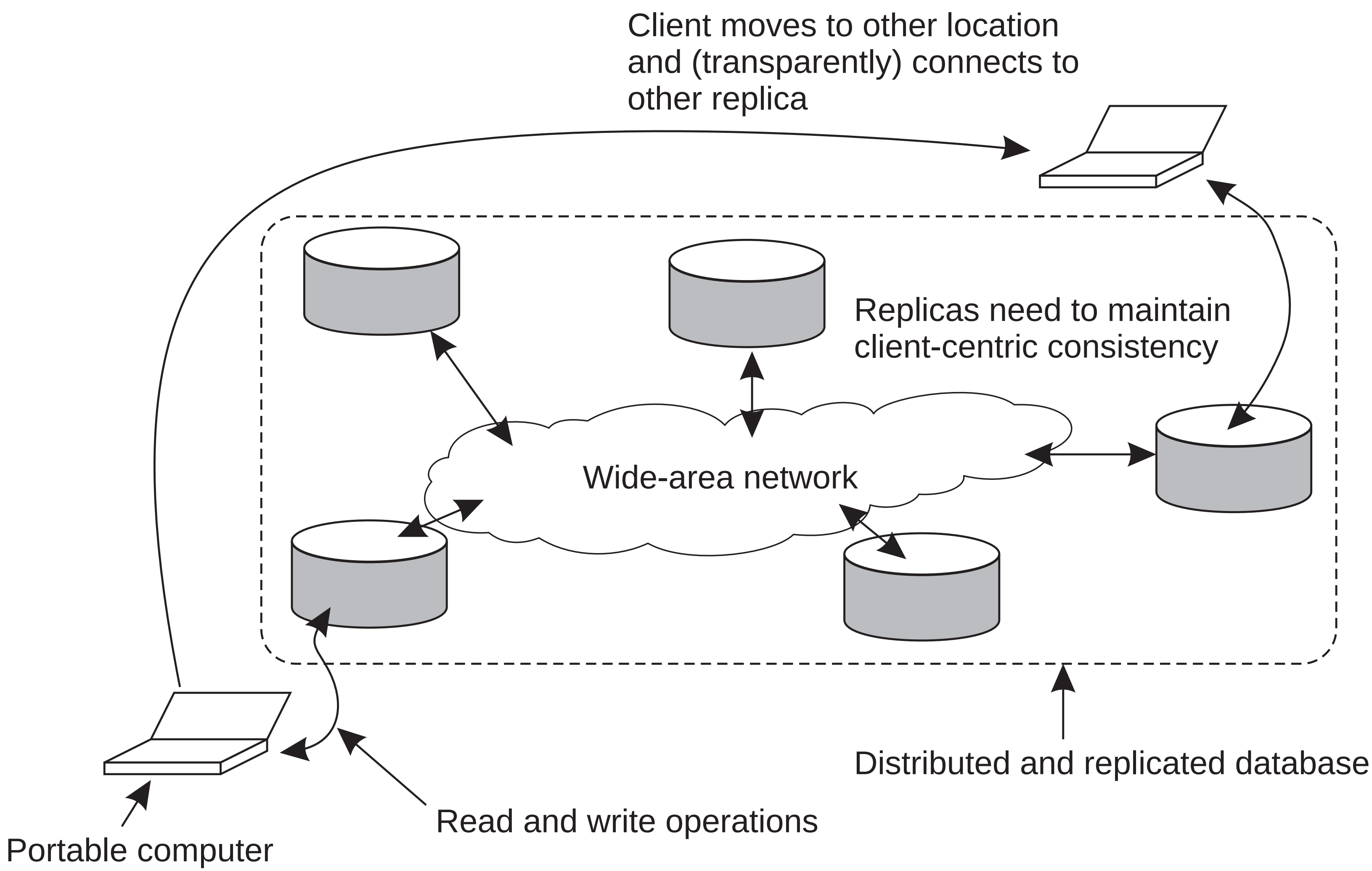
Consistency is costly $\Longrightarrow$ *Suitable model based on applications.*

Note: Transparency allows **implicit** models only (no VSM).

- **Client-centric Consistency:** VII-32
  - ∗ If a client uses the **same** copy, everything seems consistent
  - ∗ **Eventual Consistency**: global inconsistencies are tolerated, but after a long period without `Write` $\Longrightarrow$ Guarantee that after a 'finite time' all copies are up-to-date
  - ∗ If the client accesses a different copy:
    1. monotonous Read: copy is always the same or newer
    2. monotonous Write: `Write` propagated before a new `Write`

- **Data-centric Consistency:** VII-33 ff.
  - ∗ based on a 'global view' on the overall state(s)
  - ∗ compares original(s) and replicated copies of data

Isolated Client View on Consistency



taken from: **Fig. 6.19** in: A. Tanenbaum/M. van Steen: Distributed Systems, pg. 318

Global Consistency – Transparent Models – 1

Sequential Consistency: *The result of any (parallel) execution is the same as if the operations of all the processors were executed in some (arbitrary) sequential order where the operations of each individual processor appear in this sequence in (exactly) the same order as specified by its program.* ♦ c.f. Lamport 1979

- **Weakened Condition:** No true parallelism among processors
  Non-deterministic interleaving/**serialization** instead of parallelism
- Interaction **without** implicit assumptions about execution order works correctly; typical problems with dependencies and non-determinacy otherwise. c.f. III-6/8

Example:

```
a=0; b=0; parbegin { br=b; ar=a; } || { a=a+1; b=b+1; } parend;
```

correct values for (ar,br) are (0,0); (1,1); (1,0)

incorrect: (0,1) because b is incremented and updated before a

▶ **Clear semantics, implementable and transparent** VII-34

Implementing seq. Consistency by Write-Invalidate

1. C wants to write O, but not in local memory $\Longrightarrow$ write-fault
2. Get copy from current Owner (broadcast)
3. Send Invalidate message to all processes in Owner Set (broadcast)
4. C reads/writes O until different Client C′ start similar request (2.)
5. C″ tries to read O $\Longrightarrow$ read-fault because invalidated
6. Get copy from current Owner C ...

c.f. paging VII-23 3.b/c

- Write waits until all copies have been invalidated **and**
- Lock first, write exclusively afterwards

$\Longrightarrow$ Accesses are serialized $\Longrightarrow$ **sequential consistency** guaranteed

Problems: (1) Processes try to start Writes/Updates in parallel
$\Longrightarrow$ global **Sequencer** needed for serializing Write-Requests!
(2) Lost invalidate messages lead to outdated reads

Global Consistency – Transparent Models – 2

Causal Consistency: *Write operations that are* **potentially causally related** *are seen by every node of the system in the same order. Concurrent writes that are not causally related, may be seen in different order by different nodes.* ♦ Hutto et al. 1990

- **Weakened Condition:** execution order matters only iff the operations are causally ordered
- Causality is induced via $Read/Write$ dependencies

◀ **Implementation:** requires a causality analysis of the program
Each variable uses an attached vector clock for propagation

Example: T $\longrightarrow$

```
P1: write(1);                                   write(3);
P2:            read(1);   write(2);
P3:                                           read(2); ?read(1)?
P4:                           read(1);                      read(3);   !read(2)!
```

incorrect: `read(1)` after `read(2)` in P3 because `write(1)` $\xrightarrow{\sqsubset}{}^{*}$ `write(2)`
correct: entire run without `?read(1)?`; e.g., `!read(2)!` in P4 is ok because `write(2)` and `write(3)` are concurrent.

Global Consistency – Transparent Models – 3

PRAM/Processor/FIFO consistency: *All processes see writes from* **one process** *in the order they were executed in this process. Writes from different processes may be seen in a different order on different processes.* ♦ Goodman 1989

- **Weakened Condition: No** causality **among** different nodes
- Easy to implement via **message ordered broadcast** for all updates and buffering on the receiver side for out-of-order messages.

Example:

```
a=0; b=0; parbegin
            { a = 1; if (b == 0) then kill(P2) }      /* P1 */
       //   { b = 1; if (a == 0) then kill(P1) }      /* P2 */
         parend;
```

correct: Test in P1 (P2) before assignment in P2 (P1) $\Longrightarrow$ kill P2 (P1) c.f. seq. consistency

also correct here: both test occur before any update $\Longrightarrow$ both processes are killed

◀ **hard to use practically!**

Global Consistency – Transparent Models – 4

Singla et al. ACM SPAA 1997

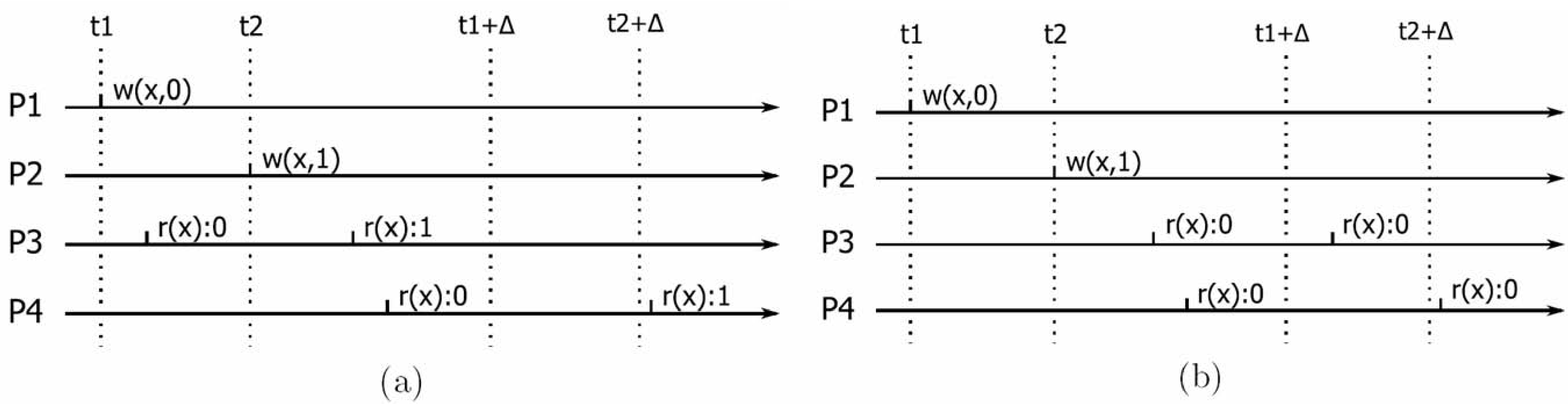
Delta Consistency: *An update will propagate through the system and all replicas will be consistent after a fixed time period, i.e., the result of any read operation is consistent with a read on the original (copy) except for a (short) bounded interval of* Δ **time units** *after a write.* ♦

- **Weakened Condition:** bounded delay of updates is tolerated where 'Delay-Models' differ based on the application at hand:
  - Approximation of global virtual time with $\Delta[t]$ ticks delay
  - 'Distance' among version numbers is kept below Δ
  - Only number Δ of 'important' state changes counts
- Realistically implementable via:
  - Use global, virtual time clocks
  - Check 'freshness' and update outdated copies (pull/push)

▶ **Typically used in** *Read-centered* **applications**, e.g., Caching for web servers ... Content Distribution Networks

P2P also

Example: Δ Consistency and Logical Time

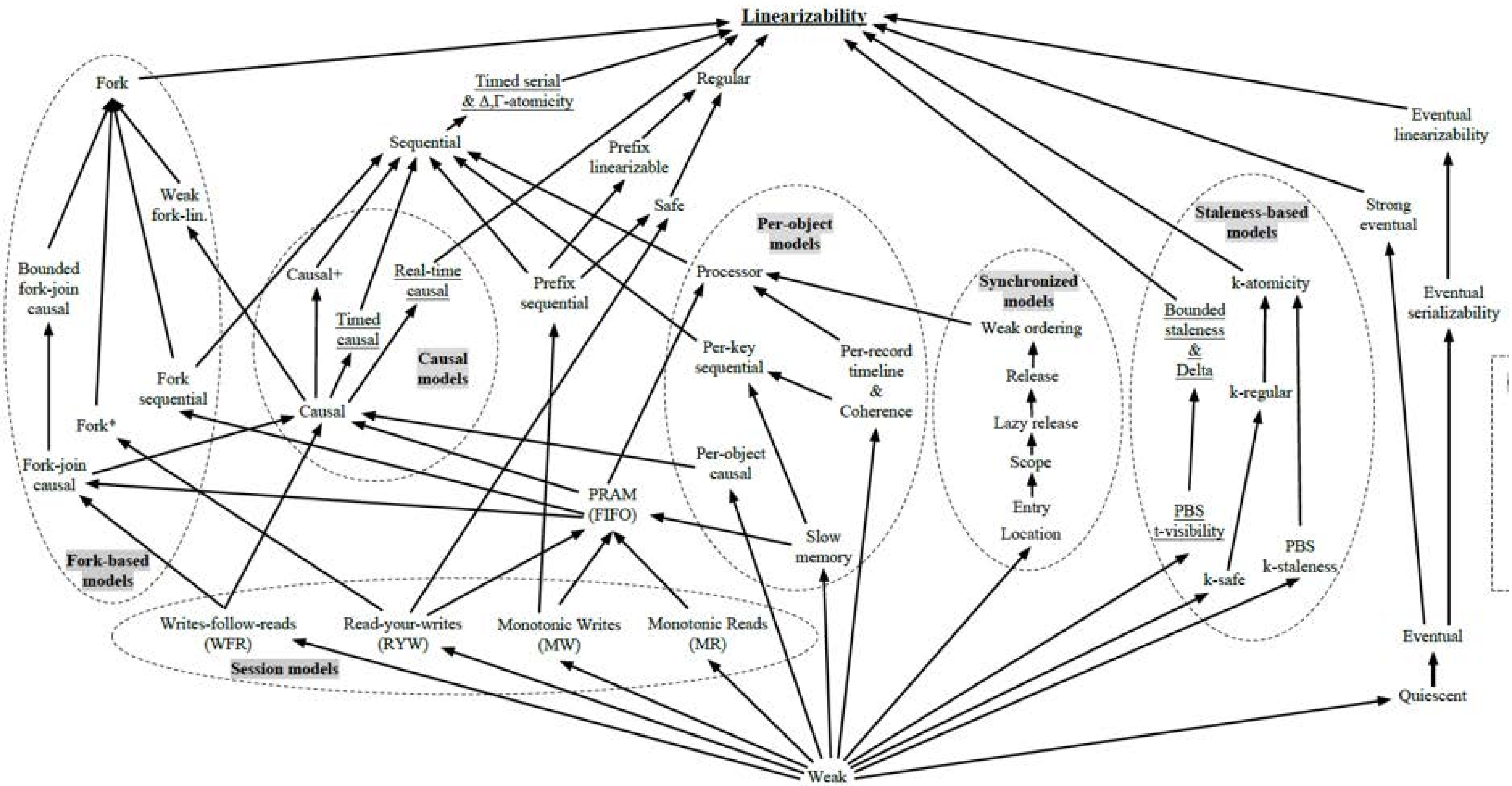


from: C. Simons Context-Aware Applications in Mobile Distribut Systems Diss. 2007, pg. 84

(a) valid run despite outdated reads because delay is still within tolerated range Δ.

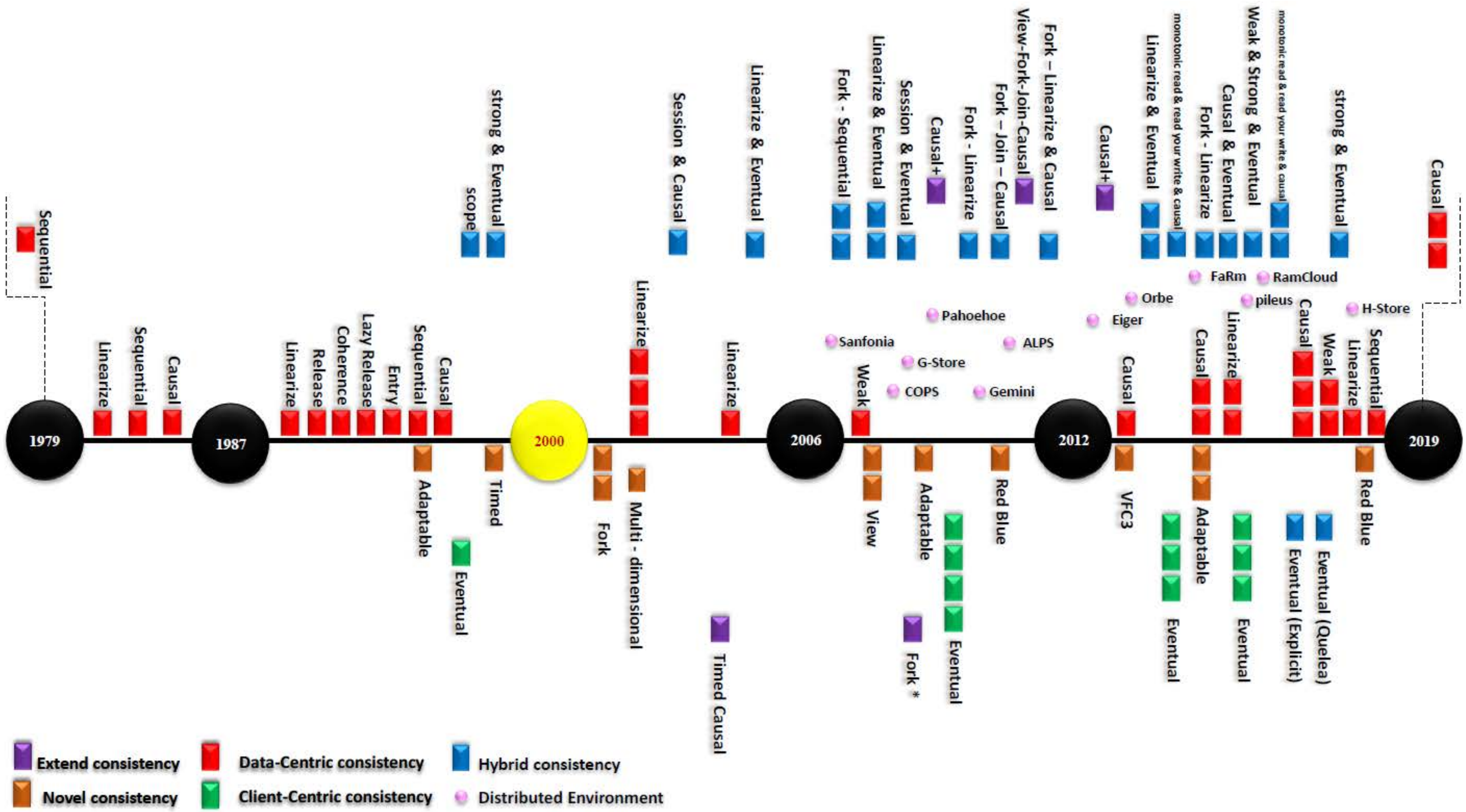
(b) last access in P_4 is invalid because delay is out of range Δ.

Outlook - A whole Bunch of Consistency Models . . .



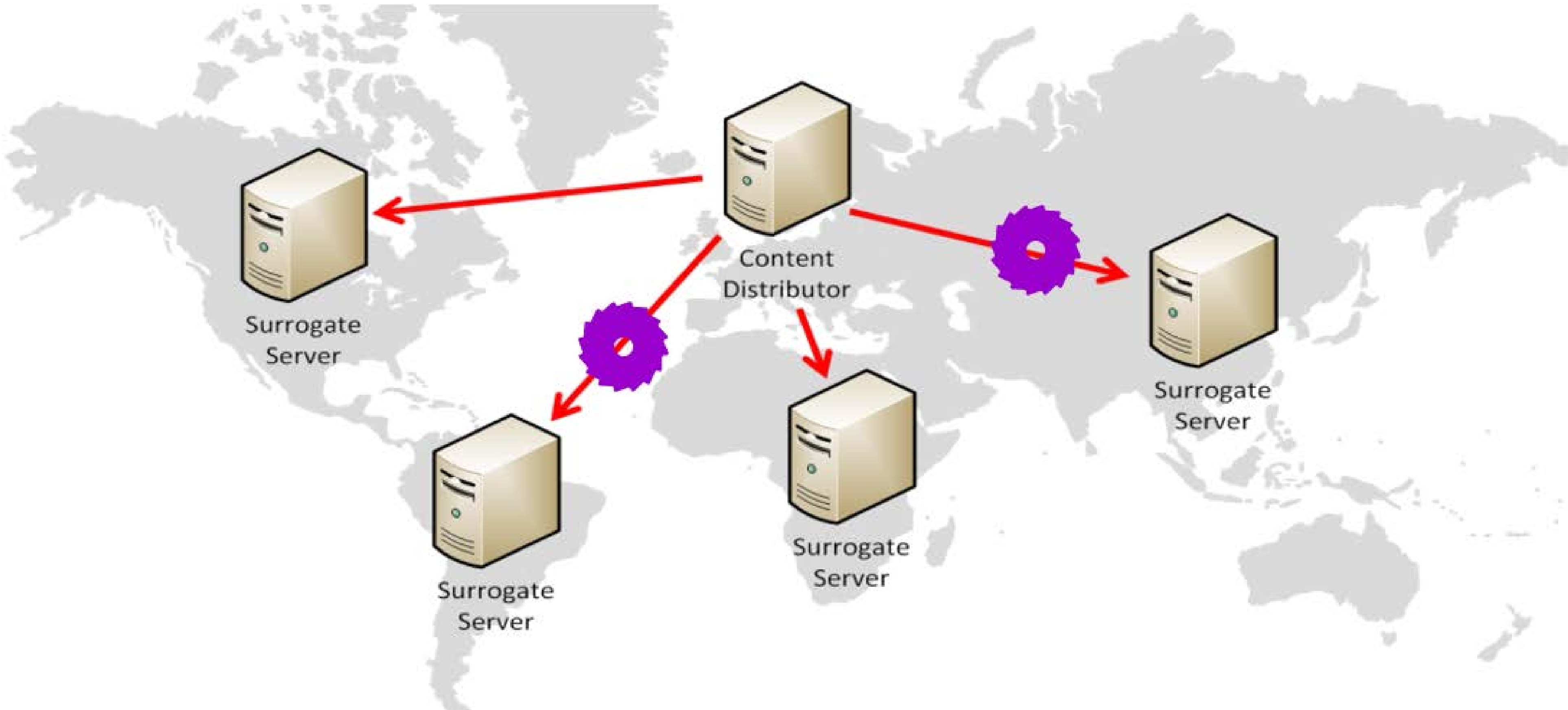
P. Viotti, Paolo, M. Vukolic: Consistency in Non-Transactional Distributed Storage Systems. in: ACM Computing Surveys (49)1; 07/2016

. . . and still evolving . . .



H. Aldin, H. Deldari, M. Moattar, M. Ghods: Consistency models in distributed systems: A survey on definitions, disciplines, challenges and applications. in: deepai.org, 08/2019

VII.2.2 Replication and Network Partitionings



- Worst-Case for Availability Considerations
- Worst-Case for Replication Schemes w.r.t. Consistency

Trade-Off for Network Partitionings - 1

Problem for (all) protocols:

- Read-Fault: attempt to get a copy may **block**
- Write: consistency mechanisms (invalidate/update) **block**

  Reason: Consistency requires 'global' answers for Writes

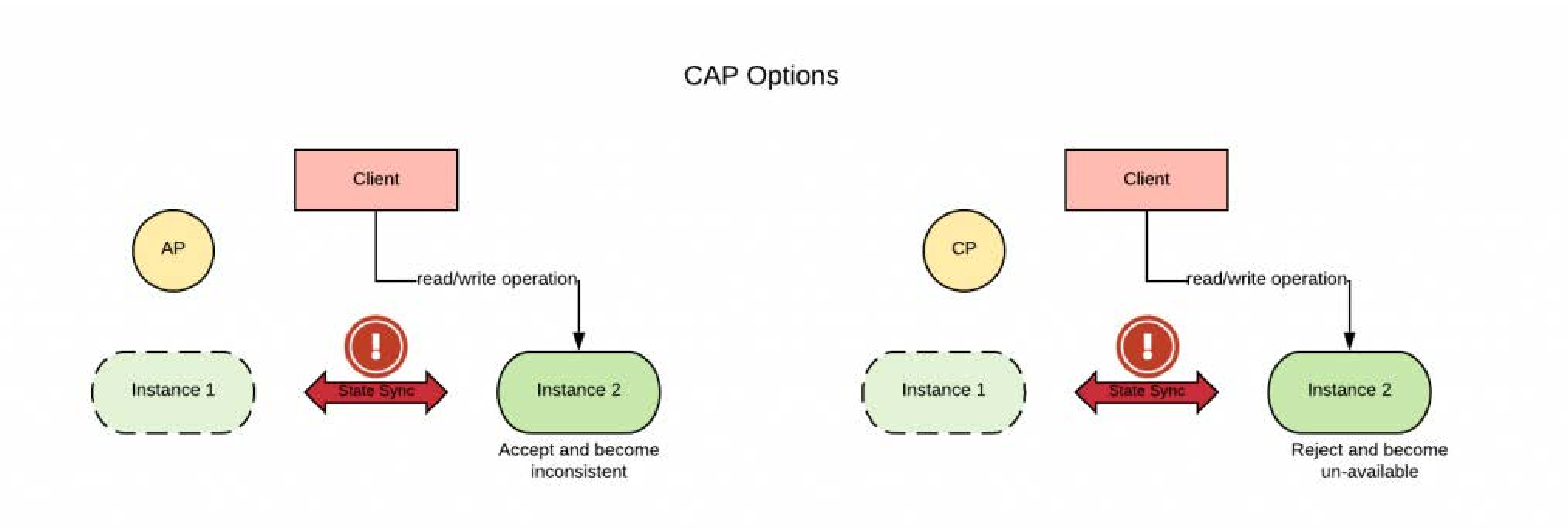


Fig.: www.openshift.com/blog/stateful-workloads-and-the-two-data-center-conundrum

$\Longrightarrow$ **Modified Protocols** trade consistency for availability

Trade-Off for Network Partitionings - 2

Modified Protocols relax consistency to some extent

1. **Primary Copy** $\approx$ MRSW or MRMW using a *lock manager*
   part of system 'with' primary copy resumes work; rest is blocked

2. **Majority Vote** w.r.t. Write or Read copies
   Network part that holds majority of copies goes on; rest is blocked
   Example: $\frac{1}{3}$ required for Read; $\frac{2}{3}$ for Write

3. **Majority** is required for Write only $\Longrightarrow$ all processes may read

4. Accept **deviations** of values for better availability
   - ▷ record and log changes for later recovery
   - ◁ complex rules for combining diverging data:
     Examples (from DB): `latest`, `primary(`n_k`)`, `Program`, `Notify`

questi-
onable
!

The C-A-PT Theorem aka CAP Theorem

CAP-Theorem: ∃ Trade-Off among the following properties:

Consistency – **A**vailability – **P**artitioning **T**olerance (of network)

$\Longrightarrow$ **?You can't fulfill all three at the same time?** $\Longleftarrow$

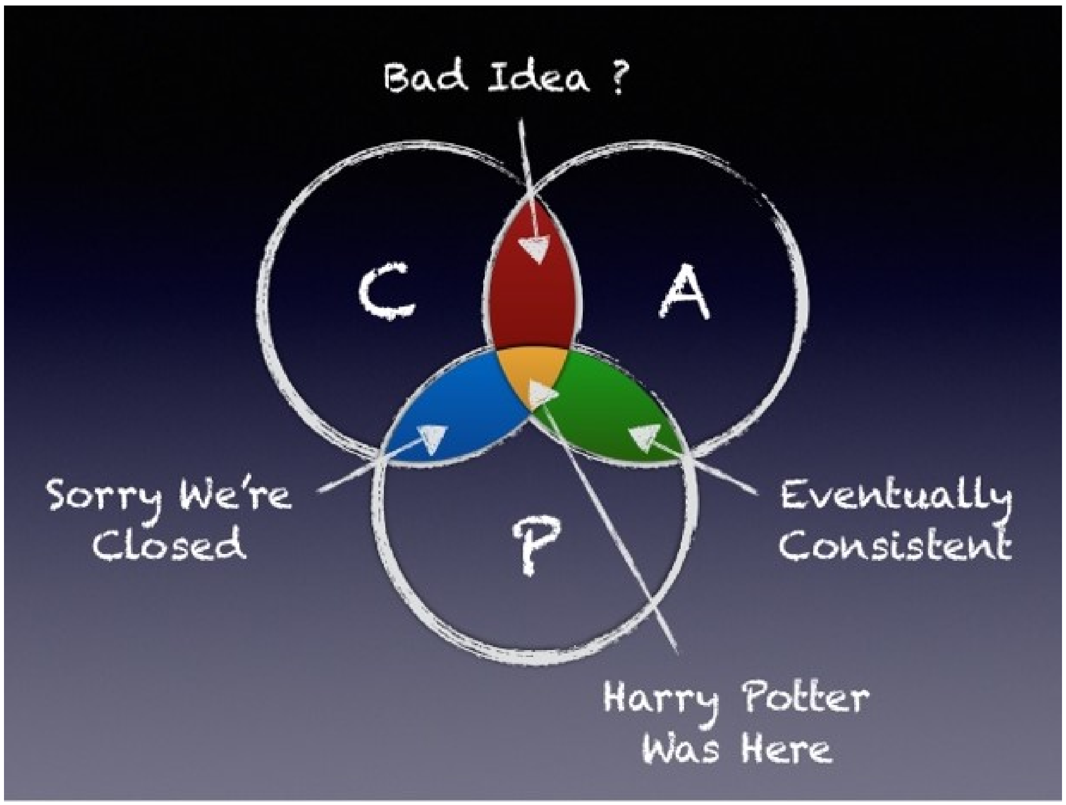


Fig.: www.slideshare.net/christoph evg/revisiting-the-cap-theorem

$\Longrightarrow$ **How frequent are Network Partitions in your application?**

Consistency for Real-life Scalable Applications – 1

- **Theory:** There exists a more general 'Trade-Off' between *Safety*' vs. *Liveness* in an '*unreliable*' system that is suffering from faulty communication. Gilbert Lynch IEEE Computer No. 45(2) 02/12 pg. 30 ff.
  - ∗ **Safety** Property: Holds at all times in a system
  - ∗ **Liveness** Property: Holds 'eventually', i.e. after a finite amount of time, the system reaches a state where the property holds.

- **Real Life:** *What does C-A-P really mean?*
  - ∗ **C**onsistency: All nodes read the most recent value due to the chosen Consistency model
  - ∗ **A**vailability: Each active node reacts within a time frame that is acceptable due to the chosen Service Level
  - ∗ **P**artitioning **T**olerance: Even in the presence of network partitionings, the system works and respects its consistency model

Consistency for Real-life Scalable Applications – 2

- **Practical Considerations:** Availability is not enough
  - ∗ Maximum time of tolerated delays for 'availability' is critical
  - ∗ Effect of '**Latency**' as an additional important aspect is missing

  Connection: Arbitrary high 'Latency' $\Longrightarrow$ no 'Availability' at all!

Extended model that takes "Latency" into account:

"A more complete portrayal of the space of potential consistency tradeoffs for DDBSs can be achieved by rewriting CAP as PACELC: if there is a partition (P) how does the system tradeoff between availability and consistency (A and C); else (E) when the system is running as normal in the absence of partitions, how does the system tradeoff between latency (L) and consistency (C)?"

D. Abadi: Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. IEEE Computer, 45(2):37-42, 2012.

Example choices w.r.t. CAP-Trade-Off

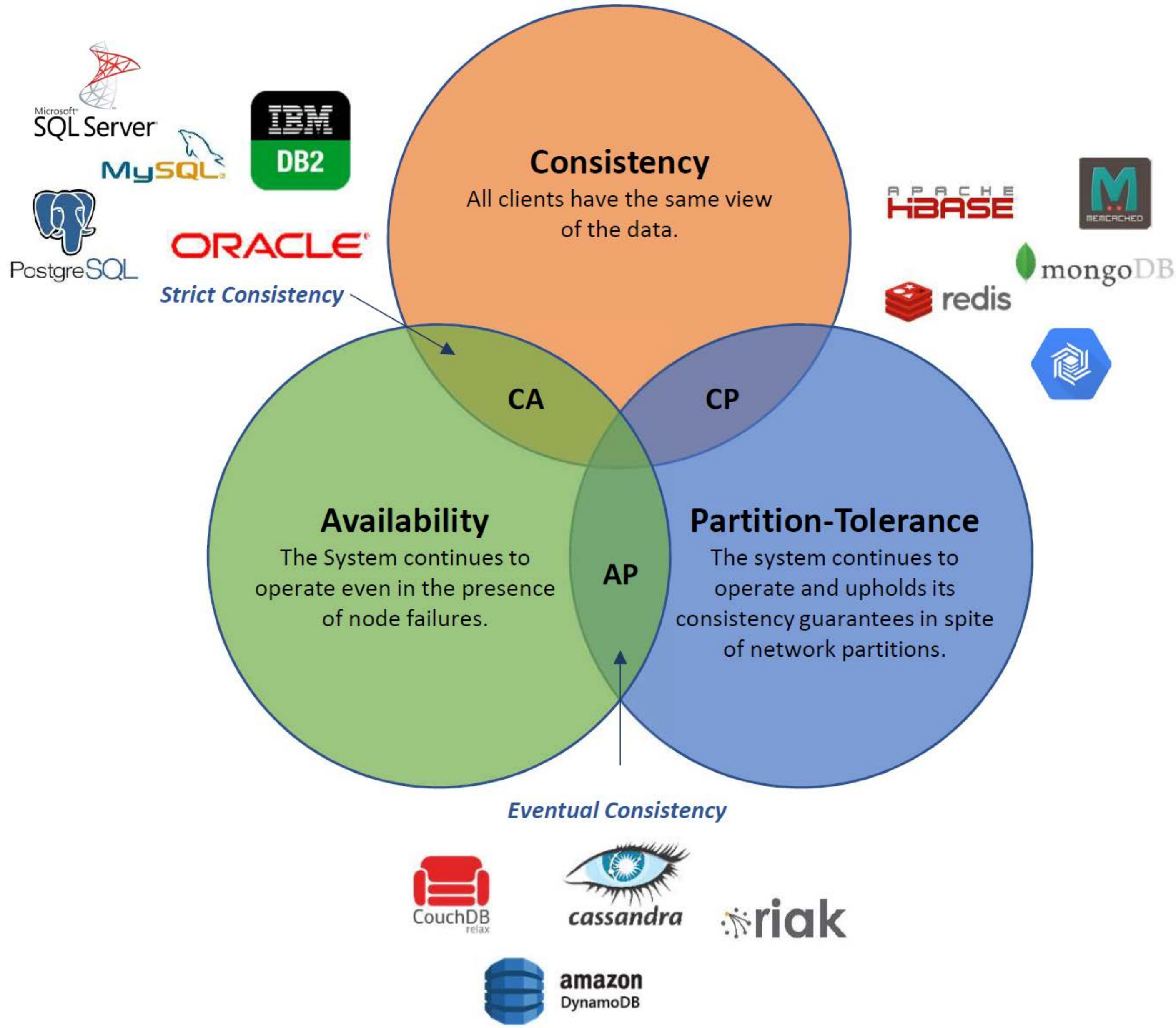


Fig.: *G. D. Samaraweera and M. J. Chang, Security and Privacy Implications on Database Systems in Big Data Era: A Survey, in: IEEE Transactions on Knowledge and Data Engineering, doi: 10.1109/TKDE.2019.2929794.*

Consistency and Availability – Outlook

c.f. Papers in vc

Further Readings: (*required for MSc students!*)

- W. Vogels: Eventual Consistency. in: Comm. of the ACM 52(1):40-44, 2009
- Simon S.Y. Shim: The CAP Theorems Growing Impact. in: IEEE Computer, 45(2):21-22, 2012.
- Eric Brewer: CAP twelve years later: How the "rules" have changed. in: IEEE Computer 45(2):23-29, 2012
- IS. Gilbert and N. Lynch: Perspectives on the CAP Theorem. in: IEEE Computer, 45(2):30-36, 2012.
- D. Abadi: Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. in: IEEE Computer, 45(2):37-42, 2012.
- Eric Brewer: Spanner, True Time & The CAP Theorem.

  in: Google TR45855, February 2017

Summary – Replication and Transparency

- **No Performance/Failure-Transparency in Distributed Systems possible without Replication.**
- Techniques used are too complex to offer them on user level $\Longrightarrow$ *Typical issue for OS and Middleware levels.*
  Example: User → Caching → Replication Management
  → Snapshots and Logging → Backup-Server
- Overhead and additional resource usage is typically high
- High, in parts even contradicting, demands make solutions with specific 'Quality-of-Service' offers tailored to specific classes of applications the most promising proceedings.

Importance: *Real problem in all recent systems in the context of Online-Storage, Data-Clouds ...*

End VII

End of DSG-(I)DistrSys-B/M Lecture Material

[Shingal et al. 1994]:*A distributed system consists of autonomous computers without any shared memory and without a global clock. Computers communicate using message-passing on a communication network with arbitrary delays.* ♦

⟹ Overview w.r.t. characteristics, types and problems of DS
⟹ Basic mechanisms to overcome shortcomings of DS
⟹ First 'ideas' how to architect and program such systems

Outlook: DSG Offerings on master level

▶ DSG-DSAM-M: *Distributed Systems and Middleware* (winter term)

▷ DSG-SOA-M: *Service-Oriented Architectures* (summer term)

▶ DSG-SEM-M: Seminar discussing advanced aspects, e.g.,
  * Advanced Distributed Algorithms
  * Consistency Models and Cloud Applications

▶ DSG-Project-M: winter term topics are tba.