

Methods of Cloud Computing

Programming Cloud Resources 1: Scalable and Fault-Tolerant Applications



Complex and Distributed Systems
Faculty IV
Technische Universität Berlin



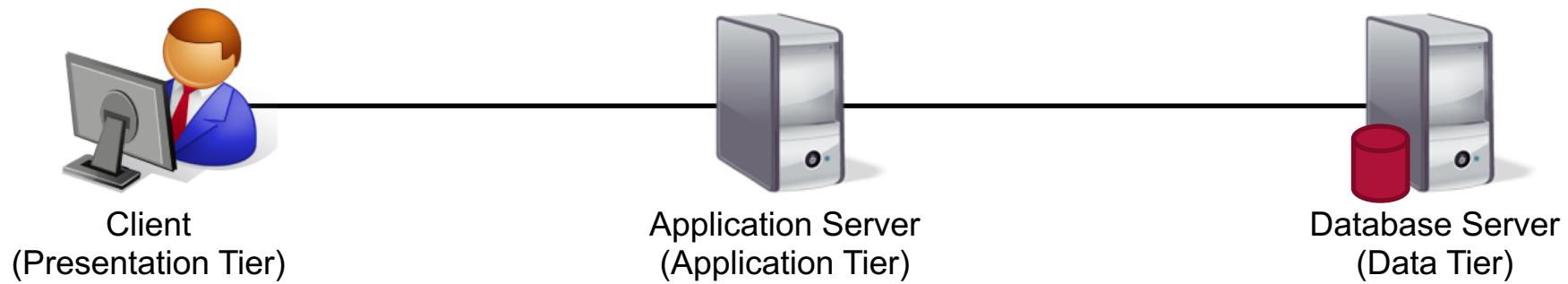
Operating Systems and Middleware
Hasso-Plattner-Institut
Universität Potsdam

Overview

- Intro
- Partitioning
- Replication and Consistency
- CAP Theorem
- Case Studies

How to Achieve Scalability/ Availability?

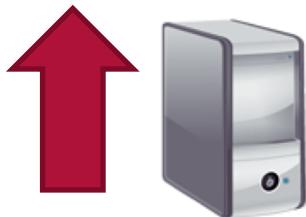
- Let's start with a classic 3-tier architecture:
 - What happens if we increase the number of clients?
 - What happens if one of the components goes down?



- What are techniques to achieve scalability/availability?

How to Achieve Scalability/ Availability?

- Two principal methods to scale



Vertical Scalability (Scaling Up)

Idea	Increase performance of a single node (more CPUs, memory, ...)
Pro:	Good speedup up to a particular point
Con:	Beyond that point, speedup becomes very expensive



Horizontal Scalability (Scaling Out)

Idea	Increase number of nodes
Pro:	Cheap to grow total amount of resources
Con:	More difficult for applications to efficiently utilize the resources

Commodity Servers and the Cloud

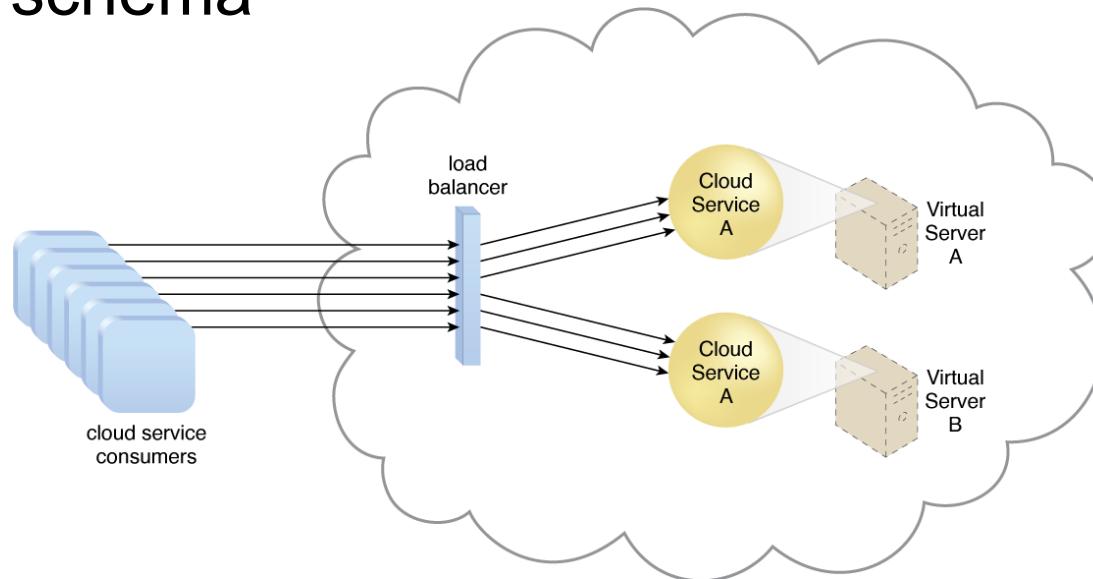
- Hardware trend: CPUs don't become faster as fast as they used to → datacenters house large sets of inexpensive commodity servers
- Scaling applications therefore typically means scaling-out across many nodes
- VMs and containers make it easy to replicate services across many nodes, especially with Infrastructure-as-Code and automation/orchestration tools

Fundamental Cloud Architectures

- Workload distribution
- Resource pooling
- Dynamic scaling / elastic capacity

Workload Distribution Architecture

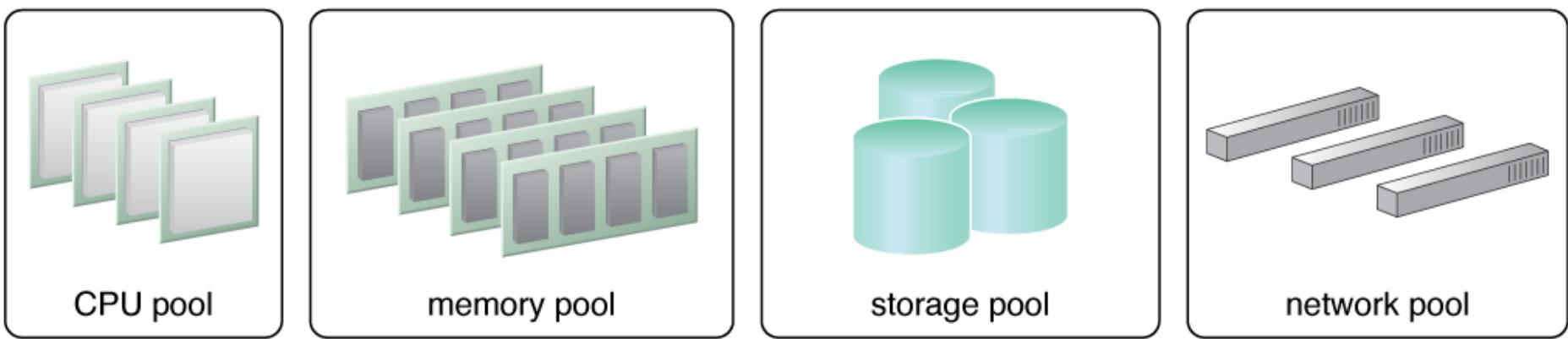
- Idea: Workload is equally shared among several instances of identical services
- Simplest model: monitoring load, distributing requests to the available resources / services according to a pre-selected schema



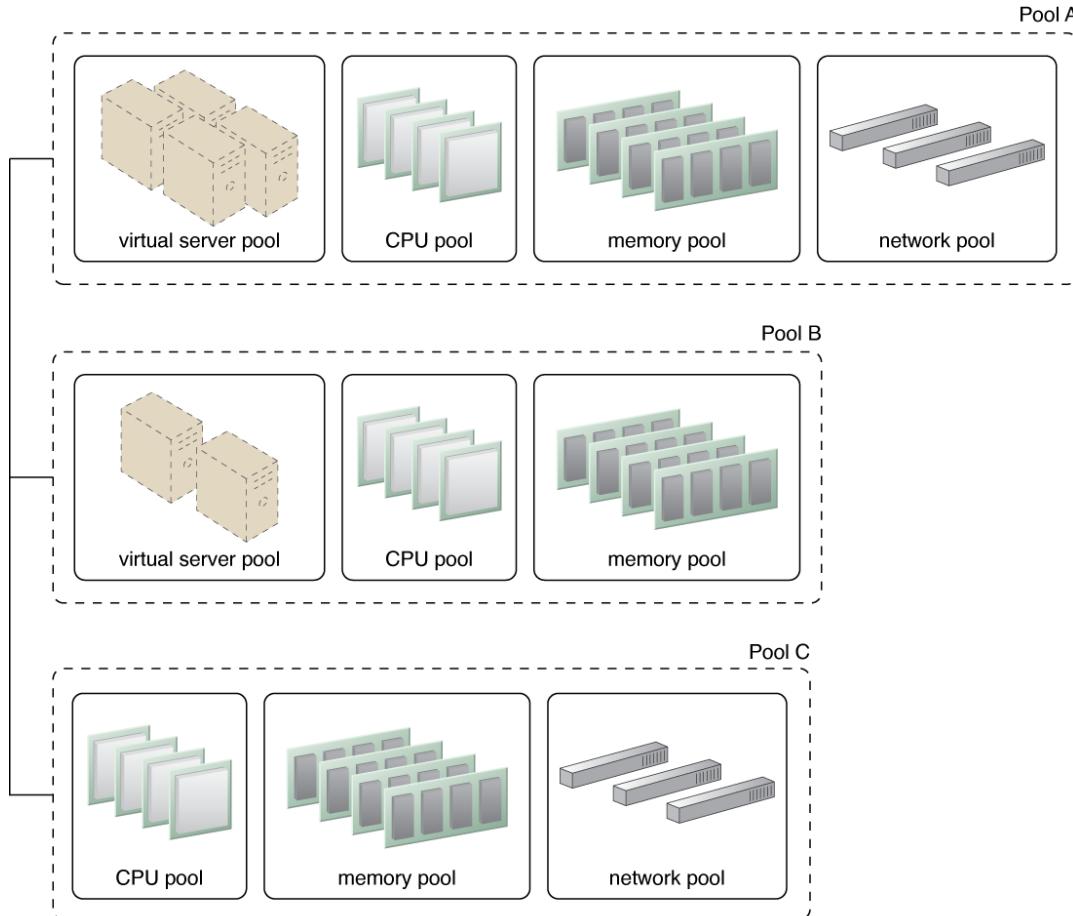
Copyright © Arcitura Education

Resource Pooling Architecture

- Combining existing resources into pools to create the illusion of “infinite” resources
- Examples
 - Storage pools: logical storage space composed out of physical hard drives
 - CPU pools: set of available nodes for processing



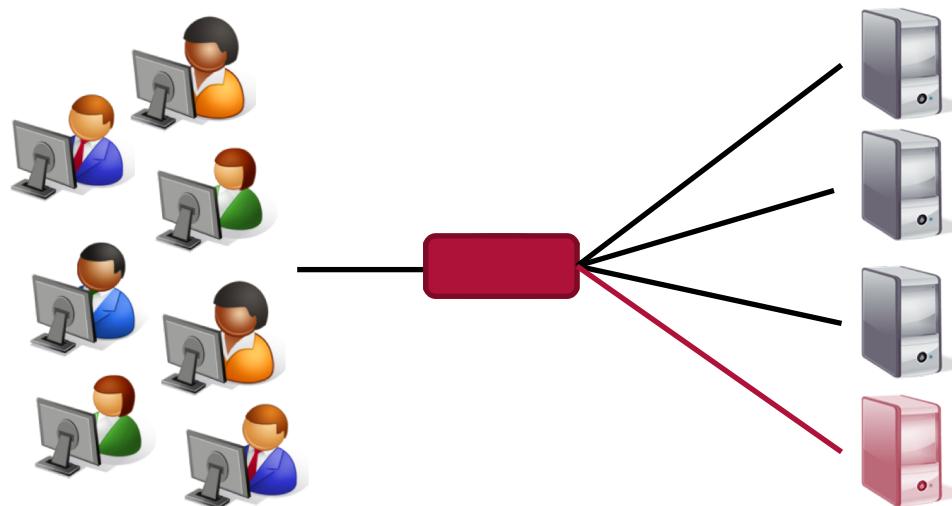
Assignment of Pools to VMs



Copyright © Arcitura Education

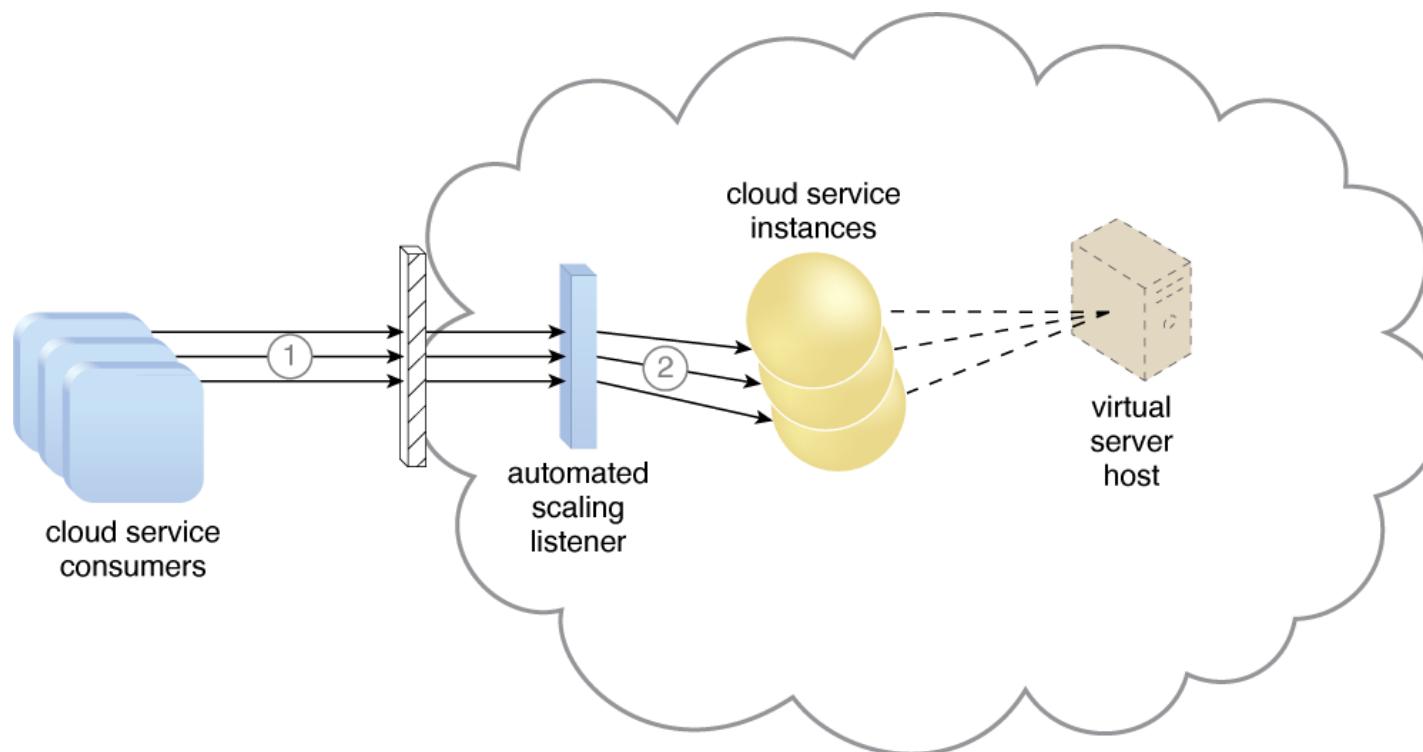
Dynamic Scaling Architectures

- Idea: pre-defined scaling conditions that trigger the dynamic allocation of resources from shared pools



Simple Dynamic Scaling Architecture (1/3)

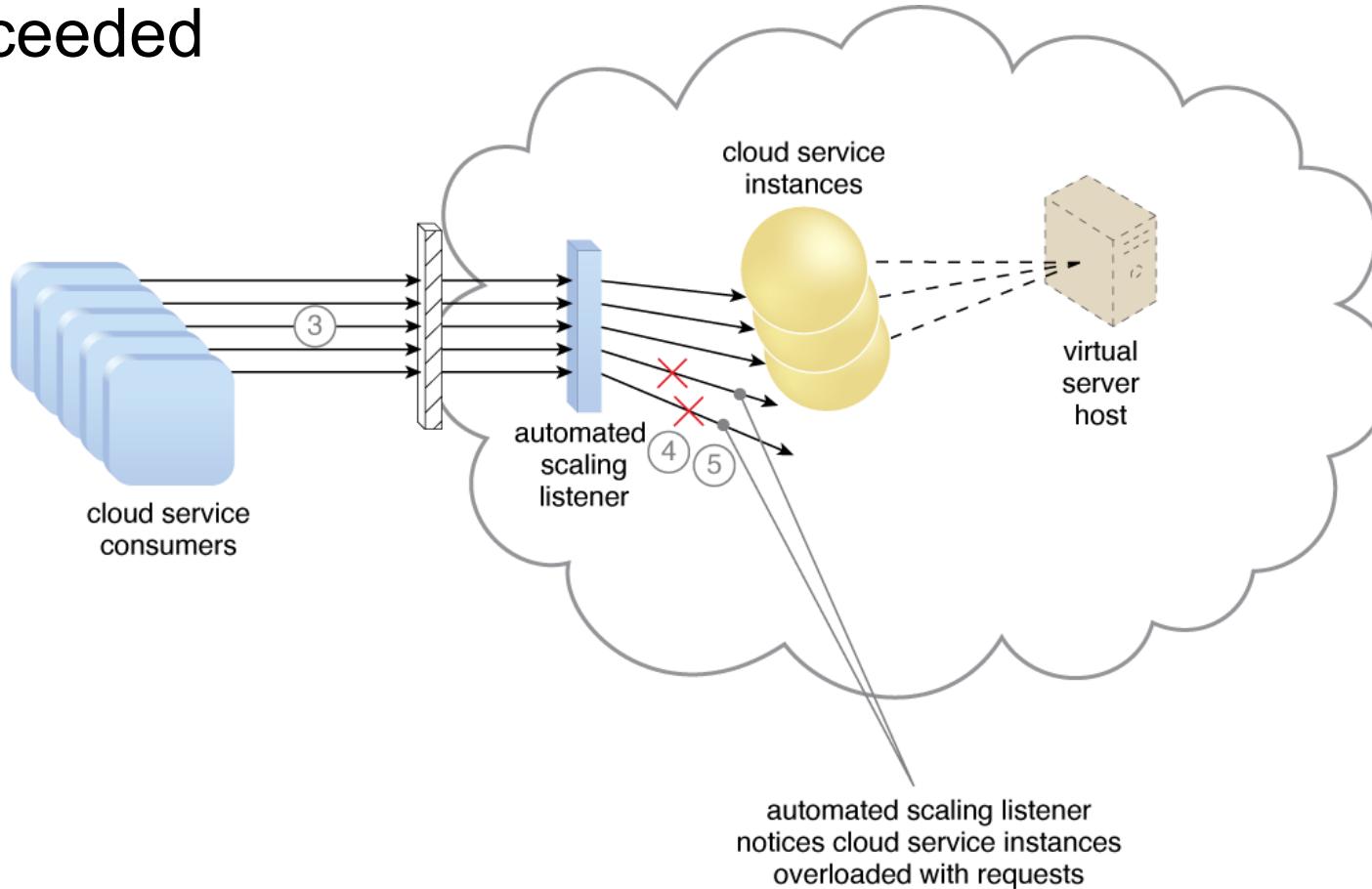
- Cloud consumers sending requests



Copyright © Arcitura Education

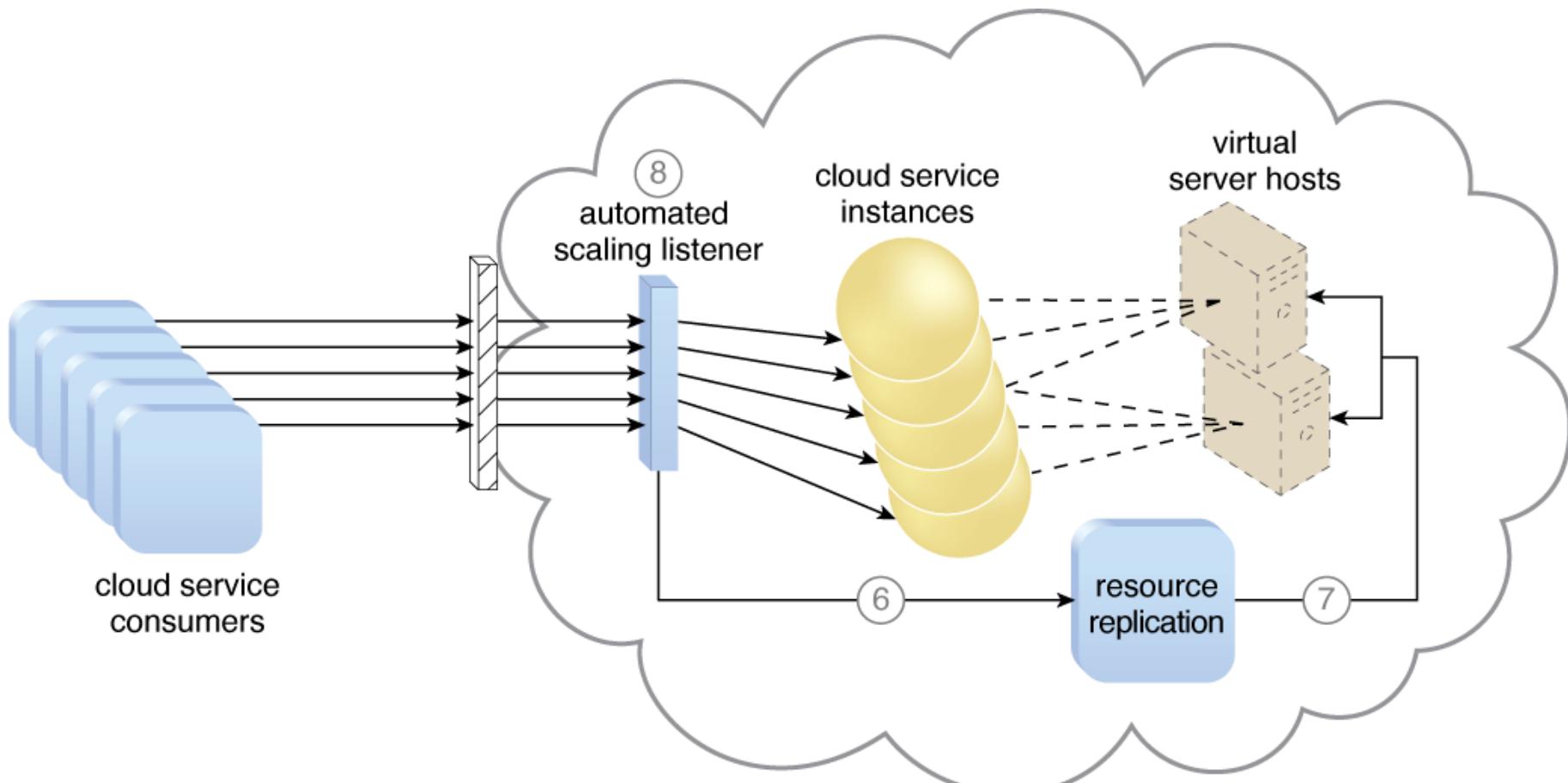
Simple Dynamic Scaling Architecture (2/3)

- Requests overload the existing resources, timeouts exceeded



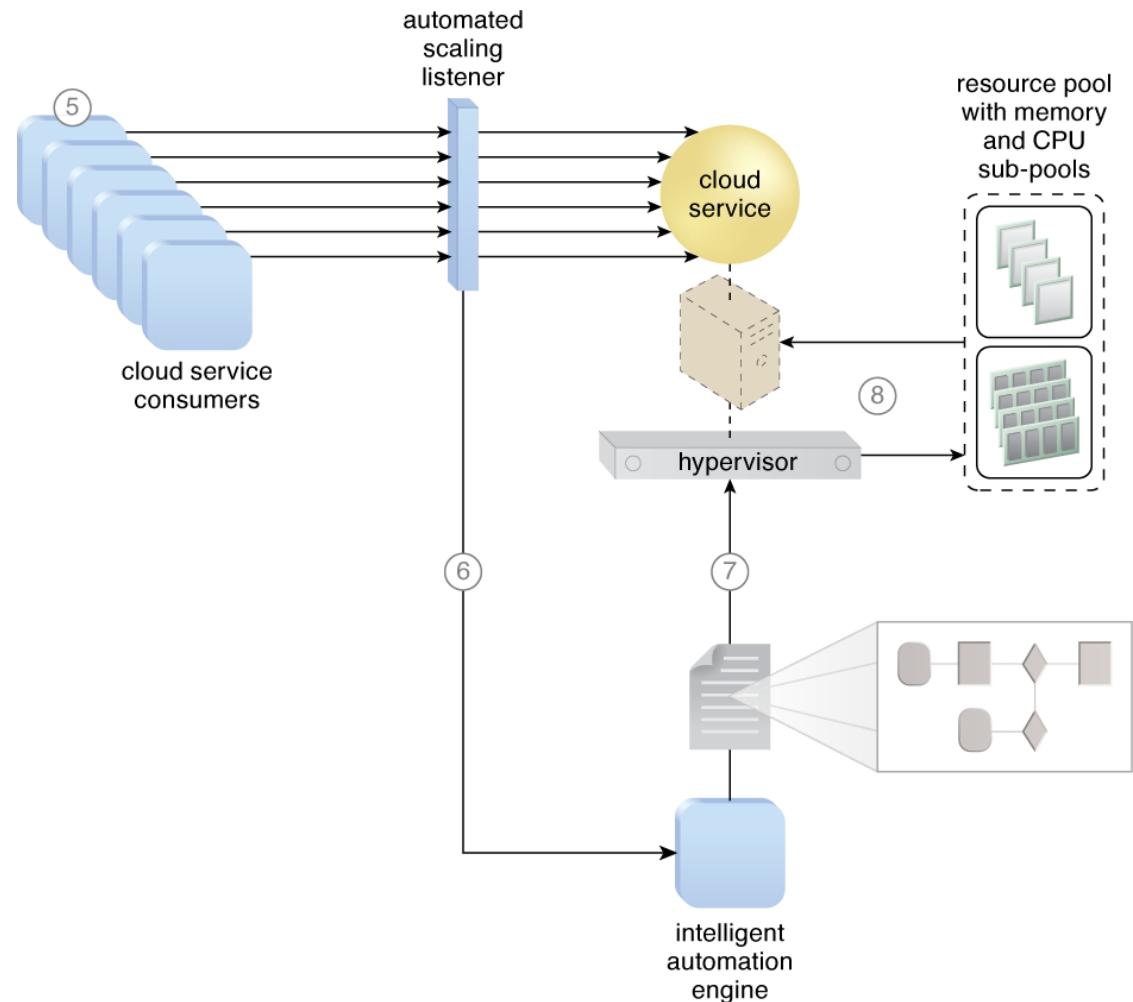
Simple Dynamic Scaling Architecture (3/3)

- Scaling up by deploying additional resources



More Advanced Elasticity Architecture

- Scaling up by deploying additional resources based on *intelligent automation engine*

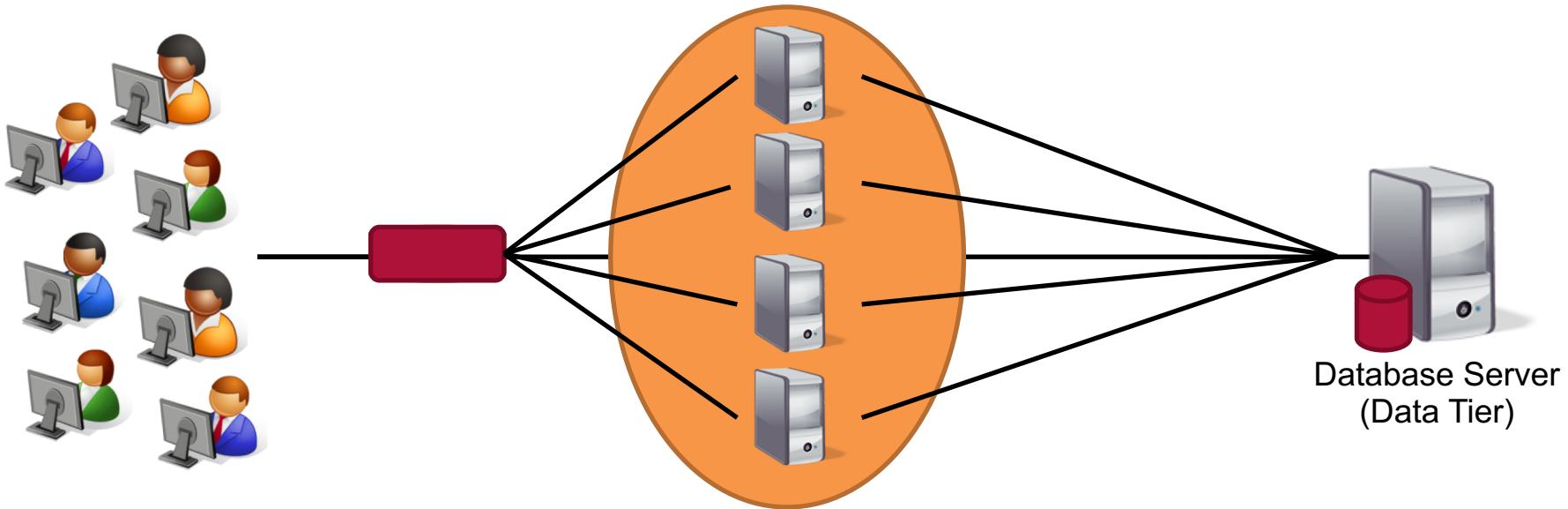


How to Achieve Scalability?

- Multi-tier application scalability entails two questions:
 - Where is the bottleneck in the architecture?
 - Is the bottleneck component stateful or stateless?
- Stateless components
 - Component maintains no internal state beyond a request
 - Examples: DNS server, web server with static pages, ...
- Stateful components
 - Component maintains state beyond request required to process next request
 - Examples: SMTP server, stateful web server, DBMS, ...

Scalability with Stateless Components

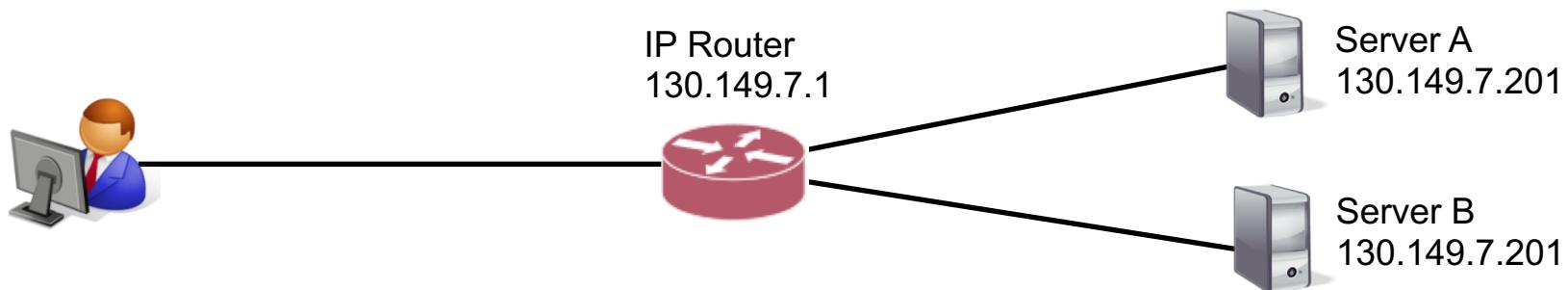
- Approach: More instances of the bottleneck component



- How do clients figure out which server to contact?
 - Clients know list of servers (e.g. P2P servers)
 - Introduction of a load balancer

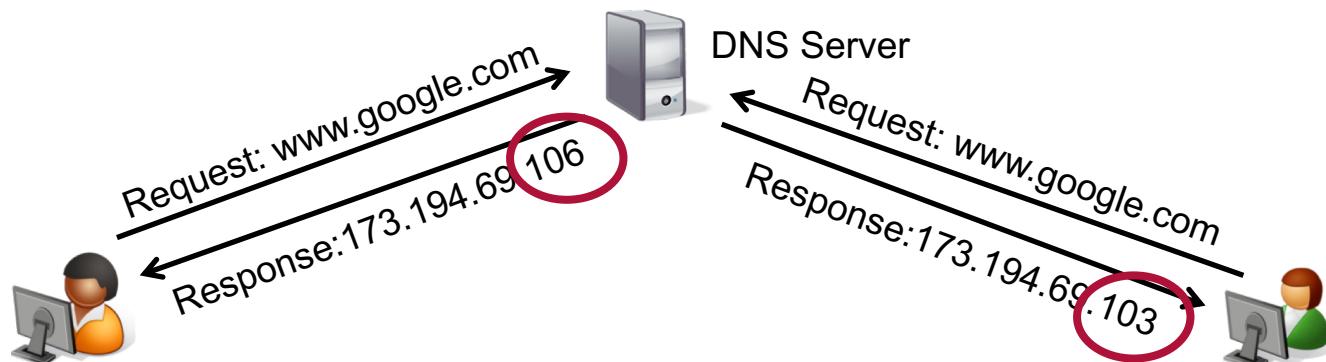
Possible Levels of Stateless Load Balancing (1/3)

- Load balancing on IP level
 - Balancing is implemented by IP routers
 - ◆ Multiple devices share one IP address (IP anycast)
 - ◆ Routers route packet to different locations
 - Requirements for applicability
 - ◆ Request must fit in one IP packet
 - ◆ Control over routers
 - Examples: DNS root server (mostly for reliability)



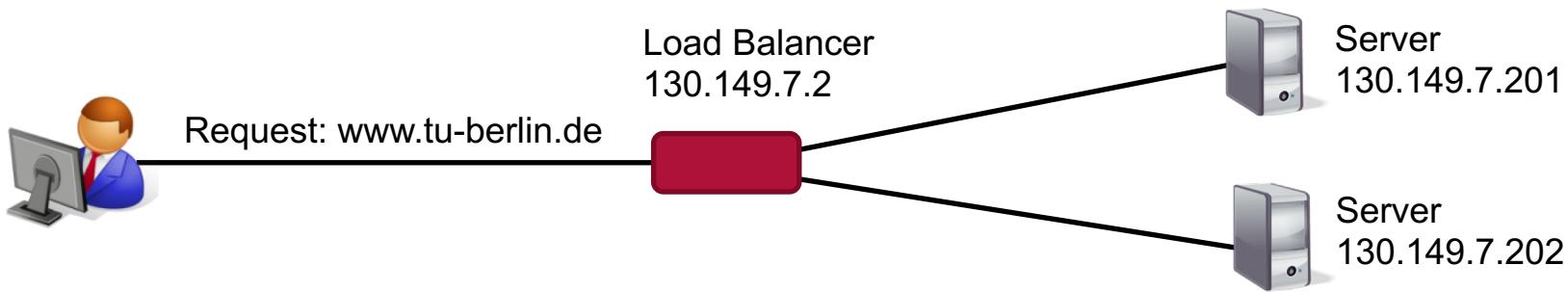
Possible Levels of Stateless Load Balancing (2/3)

- Load balancing on DNS
 - Balancing is implemented by DNS servers
 - ◆ DNS servers resolve DNS name to different IP addresses
 - Requirements for applicability
 - ◆ Control over DNS server
 - ◆ Stable load characteristics (think of DNS caching)
 - Examples: Various big websites, e.g. www.google.com



Possible Levels of Stateless Load Balancing (3/3)

- Load balancing by distinct load balancer
 - Deliberately distributes requests among machines
 - Clients first send request to load balancer, then to target
- Requirements for applicability
 - No network bottleneck
- Examples: Various websites, Amazon Elastic LB



Strategies for Stateless Load Balancing

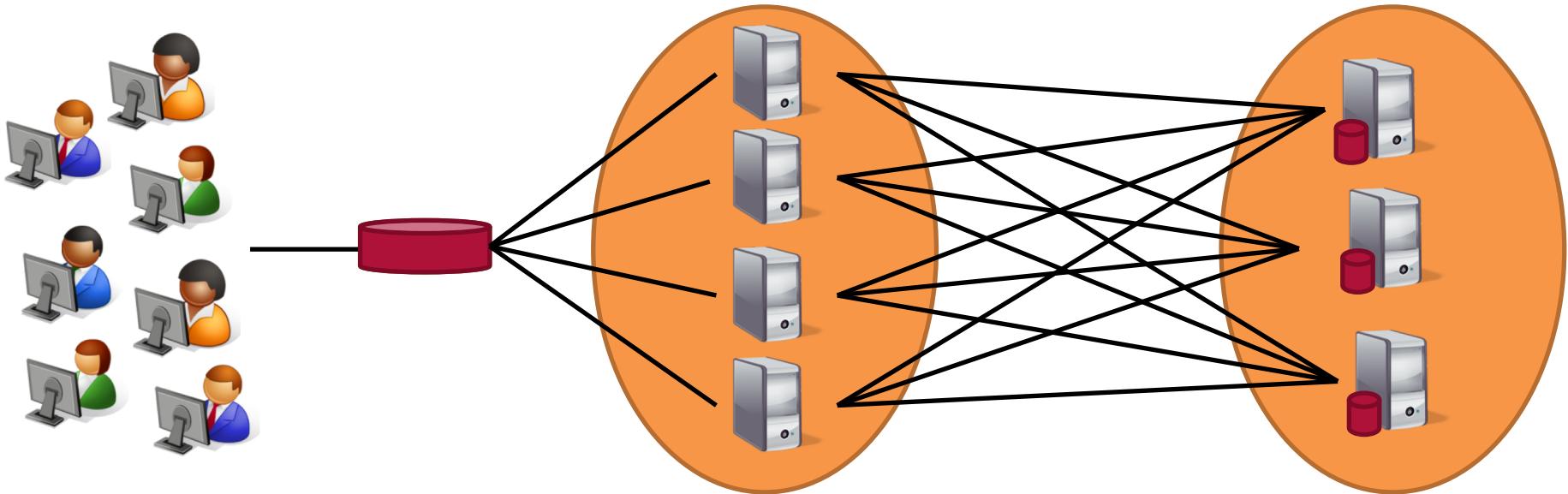
- Load balancing on different levels can be combined
 - For example, distribute network load among distinct load balancers with DNS load balancing
- Different strategies for the actual load balancing
 1. Round robin LB
 - ◆ Simple, good if all requests cause roughly the same load
 2. Feedback-based LB
 - ◆ Servers report actual load back to load balancer
 3. Client-based LB
 - ◆ Choose server with smallest network latency for client

Overview

- Intro
- **Partitioning**
- Replication and Consistency
- CAP Theorem
- Case Studies

Scalability with Stateful Components

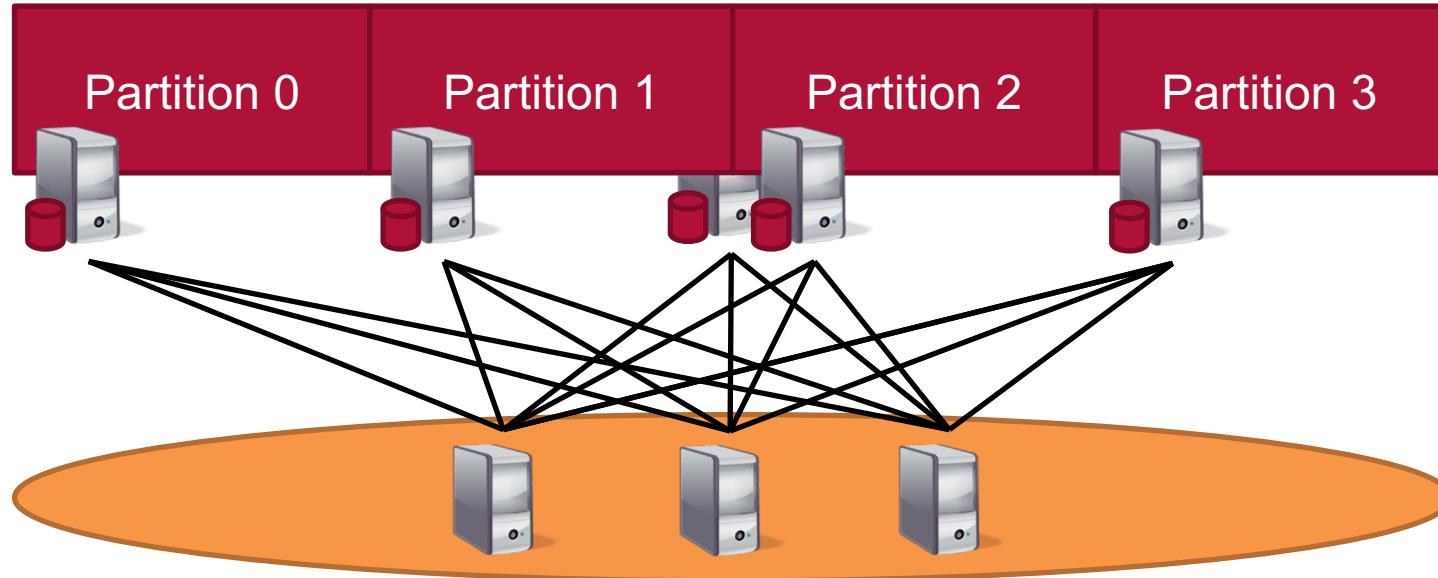
- Let's assume the data tier is the bottleneck now



- Database is stateful (stores data beyond requests)
 - Hence, requests from the same client must be handled by the same instance of the database server in this scenario

Scalability with Stateful Components: Partitioning

- Idea: Divide data into distinct independent parts
 - Each server is responsible for one or more parts



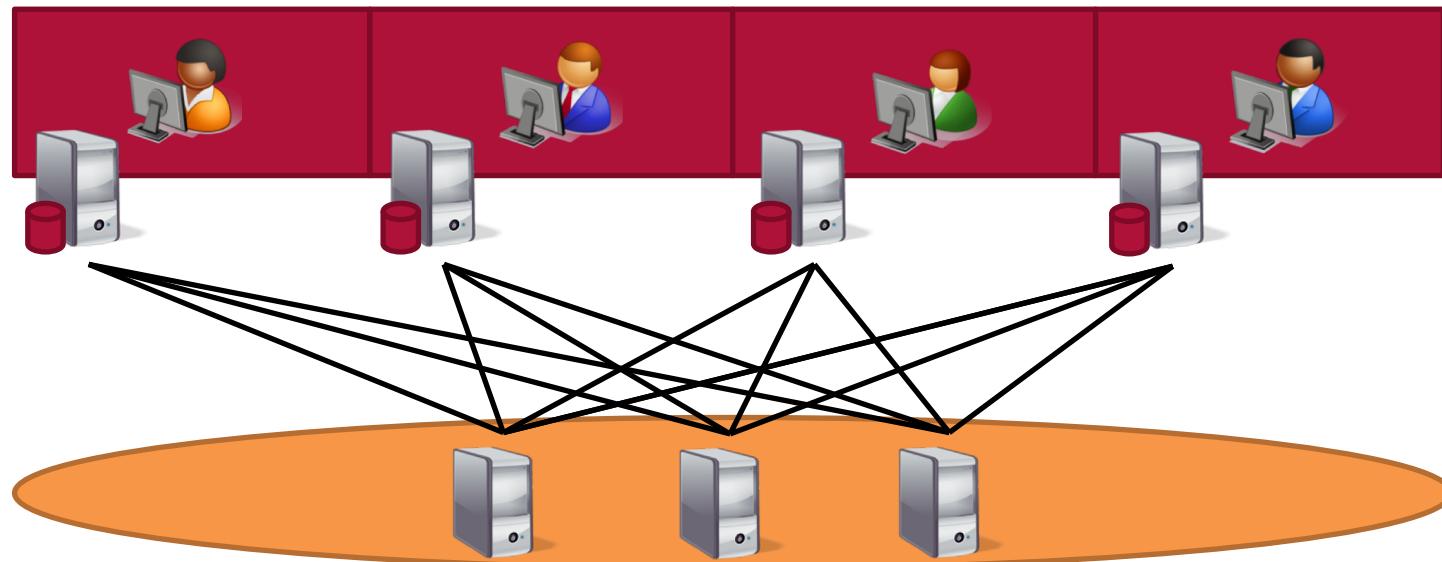
- Partitioning improves just scalability, not availability
 - Each data item is only stored in one partition!

How to Partition the Data?

- General consideration for partitioning and scalability
 - Data is now spread across several machines
 - Particular tasks might require to pull data from different places servers together → causes network traffic
- Network is a scarce resource with limited scalability
 - Becomes scarcer the more machines you add
- For good scalability, the goal of every partitioning scheme is typically to reduce network communication
 - However, this is highly application-specific ☹

Popular Partitioning Schemes (1/2)

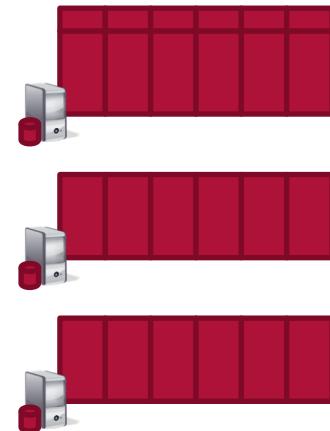
- Partitioning per tenant
 - Put different tenants on different machines
 - Pro: In clouds, tenants are expected to be isolated
→ No network traffic between machines, good scalability
 - Con: Tenants cannot scale beyond one machine



Popular Partitioning Schemes (2/2)

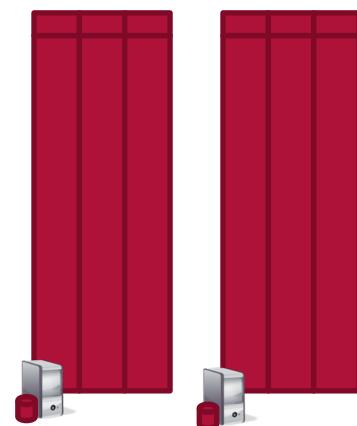
- Horizontal partitioning (relational databases)

- Split table by rows
- Put different rows on different machines
- Reduced number of rows, reduced indices
- Done by Google BigTable, MongoDB

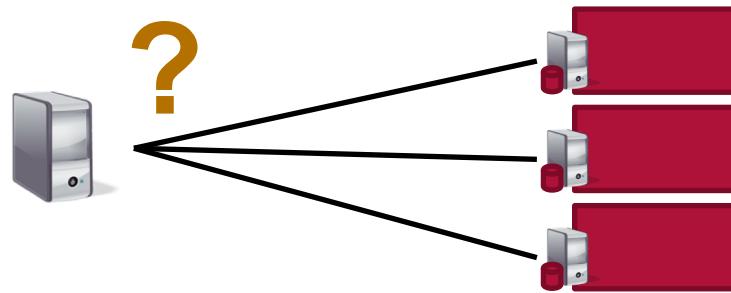


- Vertical partitioning (relational databases)

- Split table by columns
- Not very common to improve scalability (?)



How to Distribute Data Among Partitions?



1. Define key attribute $k \in K$ on the data item to be stored
2. Define a globally-known partition function p so that

$$p : K \rightarrow P$$

P is set of available partitions

- Characteristics of p have significant impact on
 - Load balancing
 - Scalability

Classes of Partition Functions

- Hash partitioning
 - Desired property: Uniform distribution
 - Pro: Good load balancing characteristics
 - Con: Inefficient for range queries, typically requires data reorganization when number of partitions changes
- Range partitioning
 - Desired property: If $k_1, k_2 \in K$ are close, $p(k_1), p(k_2) \in P$ shall also be close to each other
 - Pro: Efficient for range queries and partition scaling
 - Con: Often poor load balancing properties

Overview

- Intro
- Partitioning
- **Replication and Consistency**
- CAP Theorem
- Case Studies

Replication

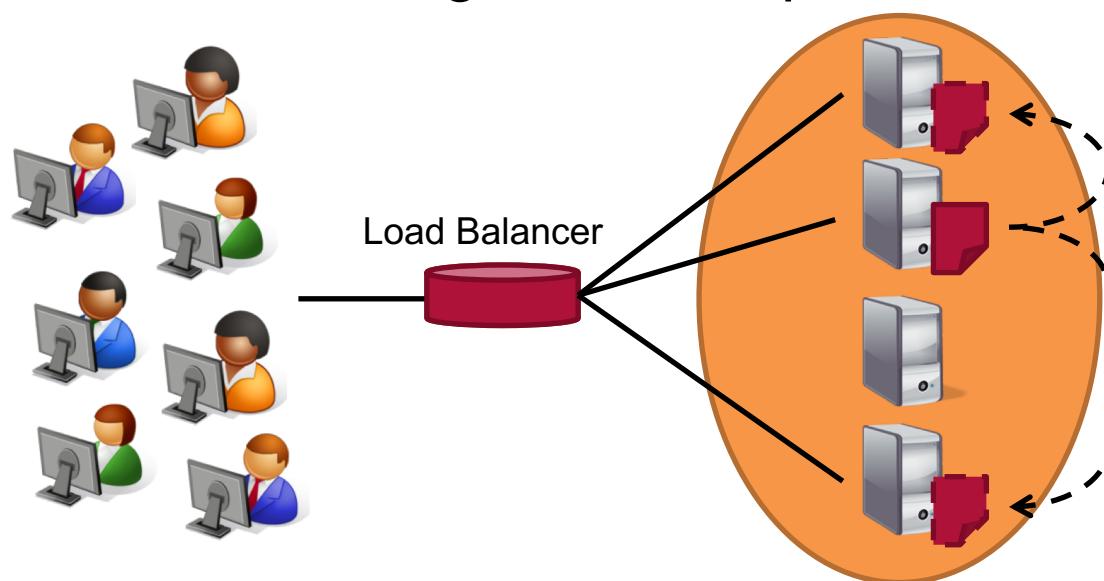
- Partitioning helps with scalability but not availability
 - Data is only stored in one location
 - If host goes down, data is gone
- Replication: Copies of the data on different machines
 - Where to place the copies?
 - Who creates the copies?
 - What happens when the data changes?
 - How do deal with inconsistencies among the copies?
- Replication can improve both scalability and availability!

Where to Place the Replicas?

- Within a cloud data center, replica placement is often done with respect to the network hierarchy
 - One replica on another machine
 - ◆ Guards against individual node failures
 - One replica on another rack
 - ◆ Guards against outages of the rack switch
- On a global scale, replicas are often distributed with regard to the client locations
 - e.g. typical with content distribution networks
 - Chosen to keep network transfers locally

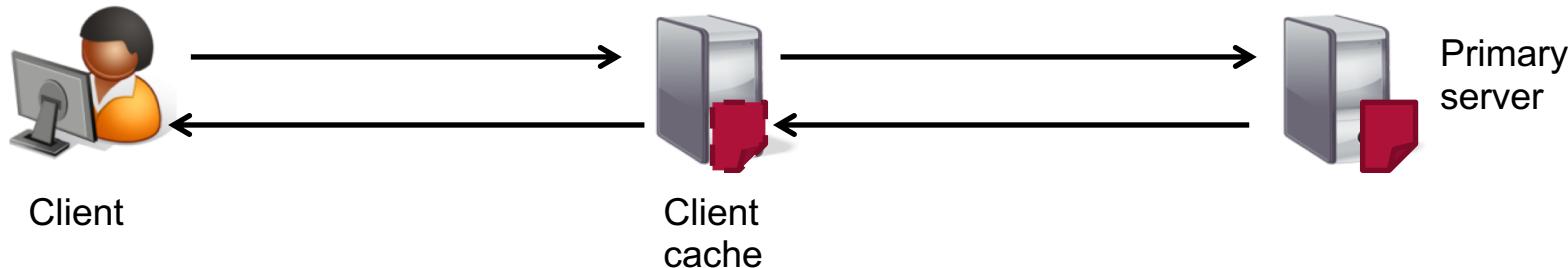
Who Creates the Copies? (1/2)

- Server-initiated replication
 - Copies are created by a server when popularity of data item increases
 - Mainly used to reduce server load
 - Server decides among a set of replica servers



Who Creates the Copies? (2/2)

- Client-initiated replication
 - Also known as client caches
 - Replica is created as result of client's response
 - Server has no control of cached copy anymore
 - ◆ Stale replicas handled by expiration date
 - Traditional examples: Web proxies



What Happens when the Data Changes?

1. Invalidation protocols
 - Inform replica servers that their replica is now invalid
 - Good when many updates and few reads
2. Transferring the modified data among the servers
 - Each server immediately receives latest version
 - Good when few updates and many reads
3. Don't send modified data, but modification commands
 - Good when commands substantially smaller than data
 - Assumes that servers are able to apply commands
 - Beneficial when network bandwidth is the scarce resource

Push vs. Pull (1/2)

- Push-based updates (server-based protocols)
 - Server pushes updates to replica servers
 - Mostly used in server-initiated replica setups
 - Used when high degree of consistency is needed

- Pull-based updates (client-based protocols)
 - Clients request updates from server
 - Often used by client caches
 - Good when read-to-update ratio is low

Push vs. Pull (2/2)

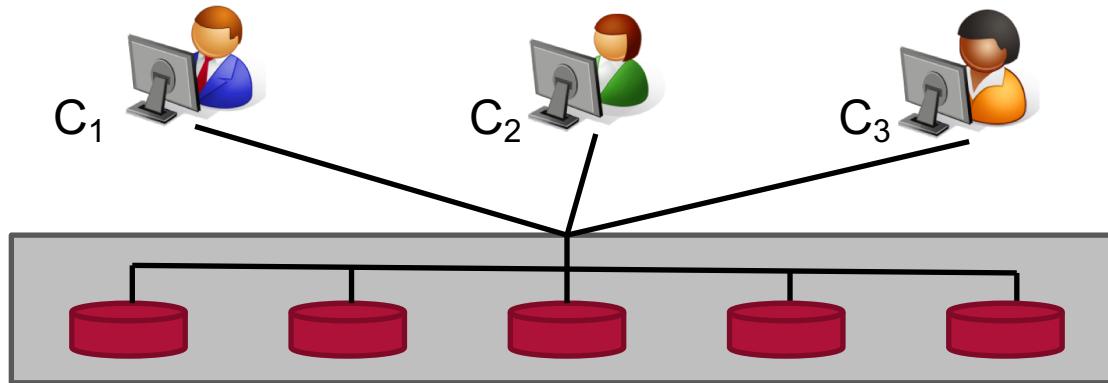
Issue	Push-based	Pull-based
State of server	Server must keep a list of all replicas and caches	None
Messages sent	Messages only when data changes	Caches poll server, possibly no updates
Response time at client	None, if not invalidation	Time it takes to fetch the modifications

How to Deal with Inconsistencies?

- When data lives in different locations, inconsistencies can occur, for example:
 - Client 1 updates data item X at replica server R1, Client 2 reads old value of X at R2
 - Client 2 updates X at R1, Client 2 updates X at R2
 - ◆ Which one is the correct value?
- Different levels of consistency can be enforced
 - Rule of thumb: the higher the consistency level (CL), the lower the performance
 - However, there must be a clear understanding what CL an application can expect from a data store

Consistency Models

- Assume a data store with replication
 - Clients can read/write data stored
 - Clients cannot influence which copy they access
 - Updates are propagated among replicas in data store



- Consistency model: Contract between client and data store → what version of the data can a client expect?

Two Views On Consistency

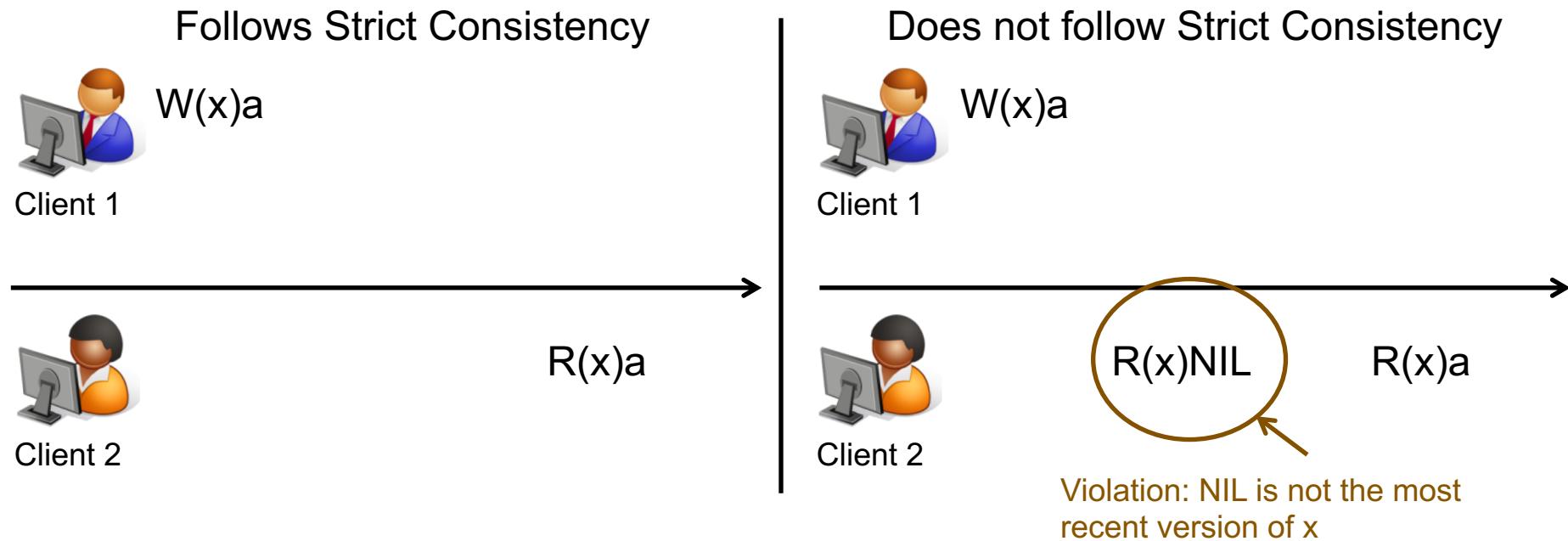
- Data-centric consistency models
 - Talk about consistency from a global perspective
 - Provides guarantees how a sequence of read/write operations are perceived by *multiple* clients
- Client-centric consistency models
 - Talk about consistency from the client's perspective
 - Provides guarantees how a *single* client perceives the state of a replicated data item

Data-Centric Consistency Models

- Strong consistency models
 - Operations on shared data is synchronized
 - ◆ Strict consistency (related to time)
 - ◆ Sequential consistency (what we are used to)
 - ◆ Causal consistency (maintains only causal relations)
 - ◆ ...
- Weak consistency models
 - Synchronization only when data is locked/unlocked
 - ◆ General weak consistency
 - ◆ Release consistency
 - ◆ Entry consistency
 - ◆ ...

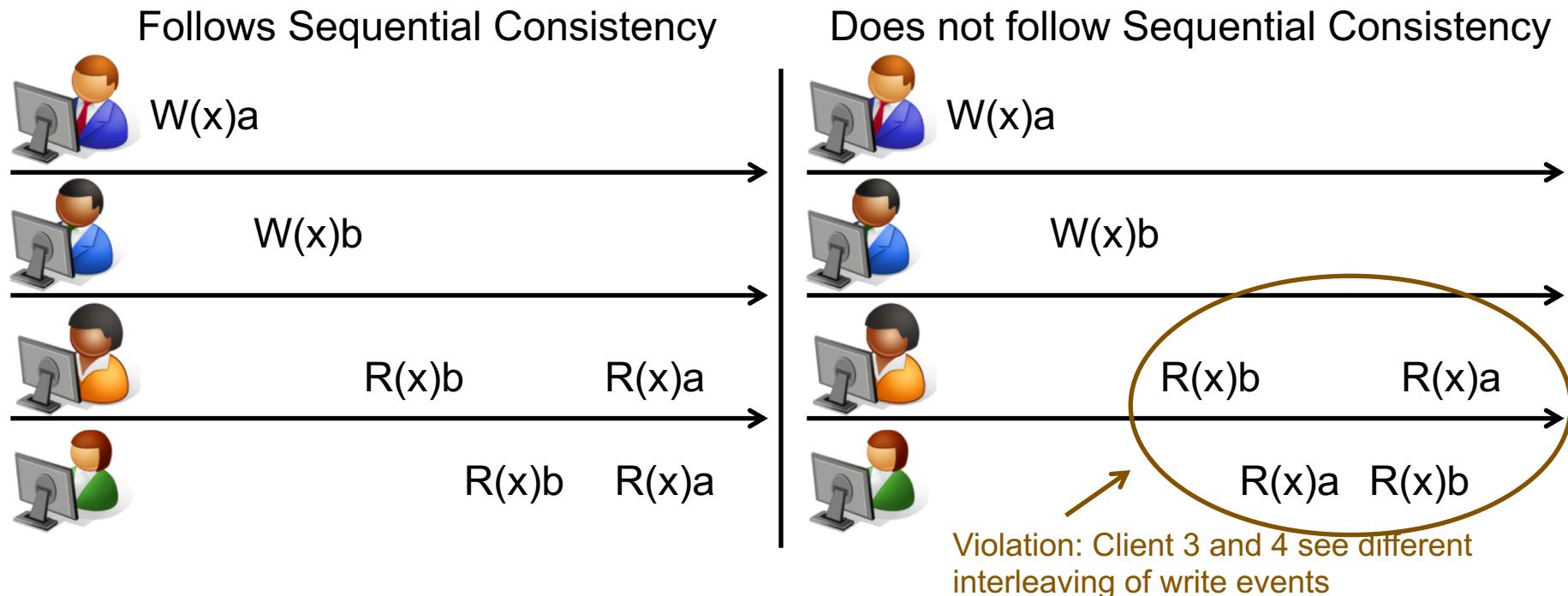
Strict Consistency

- Guarantee to clients:
 - Any read to a shared data item x returns the value stored by the most recent write operation on x
- Model only of theoretical interest in distributed systems



Sequential Consistency

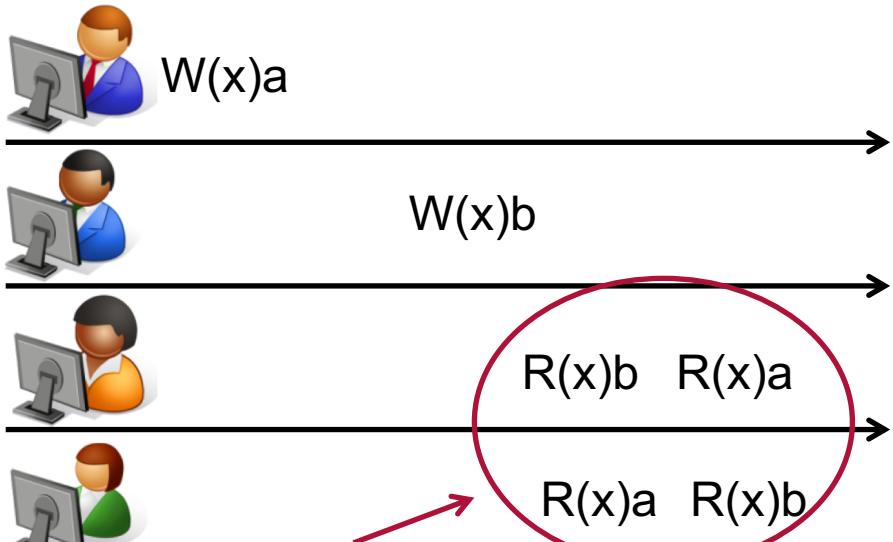
- Guarantee to clients: The result of any execution is the same as if the (read and write) operations of all processes were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program



Causal Consistency

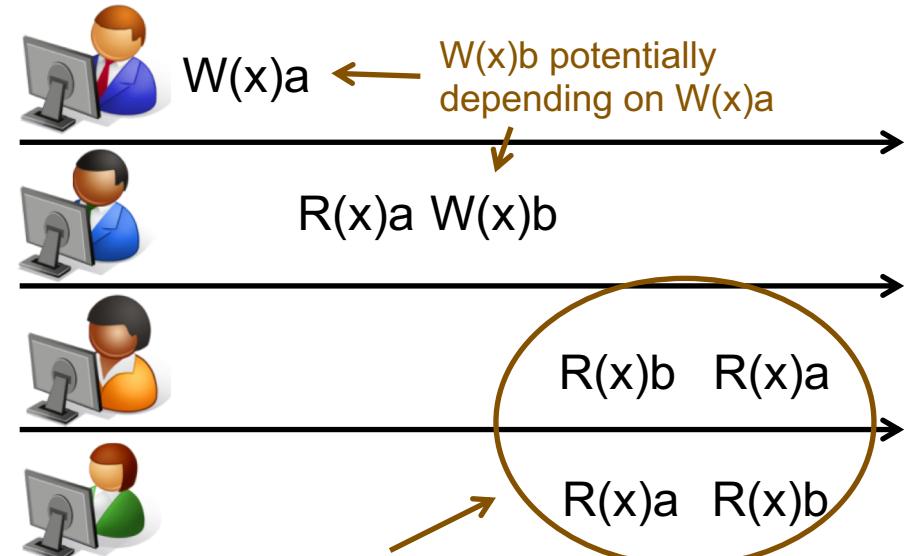
- Guarantee to clients: Writes that are potentially causally related must be seen by all processes in the same order. Independent writes may be seen in a different order by different clients.

Follows Causal Consistency



Now different read order is OK because $W(x)a$ and $W(x)b$ are concurrent writes

Does not follow Causal Consistency



Violation: Client 3 and 4 see write operations in different order

Client-Centric Consistency Models

1. Eventual Consistency

2. Monotonic Reads

3. Monotonic Writes

4. Read Your Writes

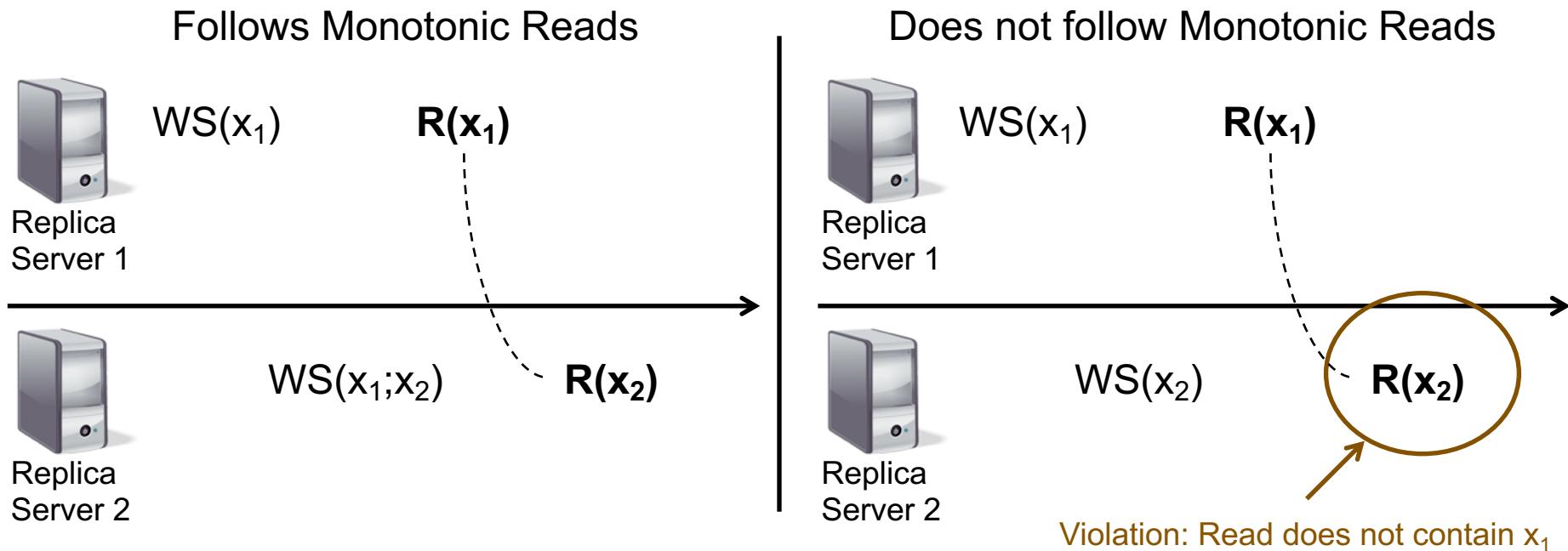
5. Writes Follow Reads

Eventual Consistency

- Guarantee for the client:
 - All replicas will eventually reach the most recent state
- Apart from that there is no guarantee
 - Client can read old data
 - Client can lose its own updates
- Model enjoys popularity in the cloud world as it can be implemented very cheaply

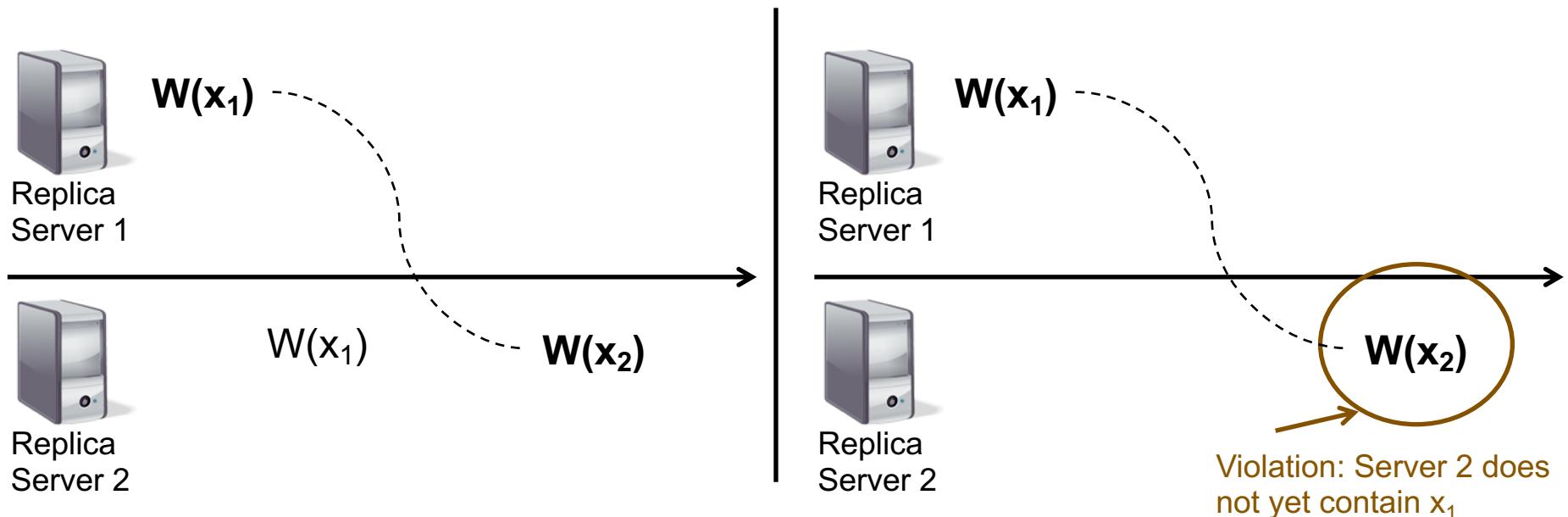
Monotonic Reads

- Guarantee for the client:
 - A read operation by one client is made only at a server containing all writes that were seen by previous reads



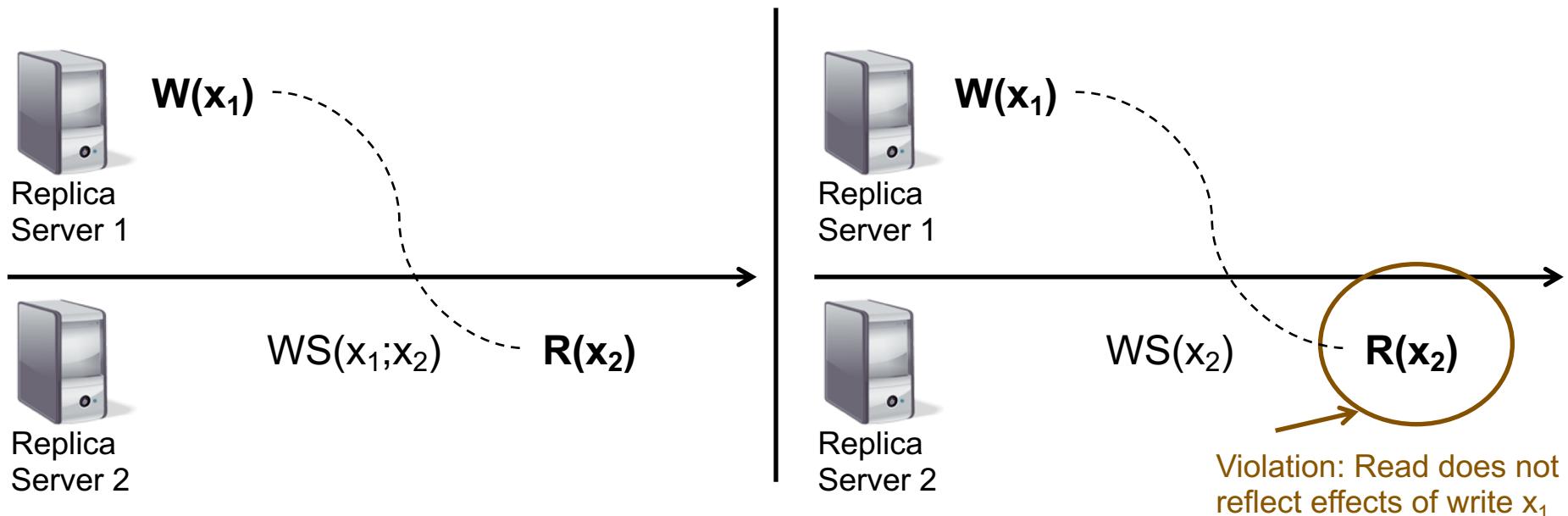
Monotonic Writes

- Guarantee for the client:
 - A write operation by a client on data item x is completed before successive write operations on x by the *same client*



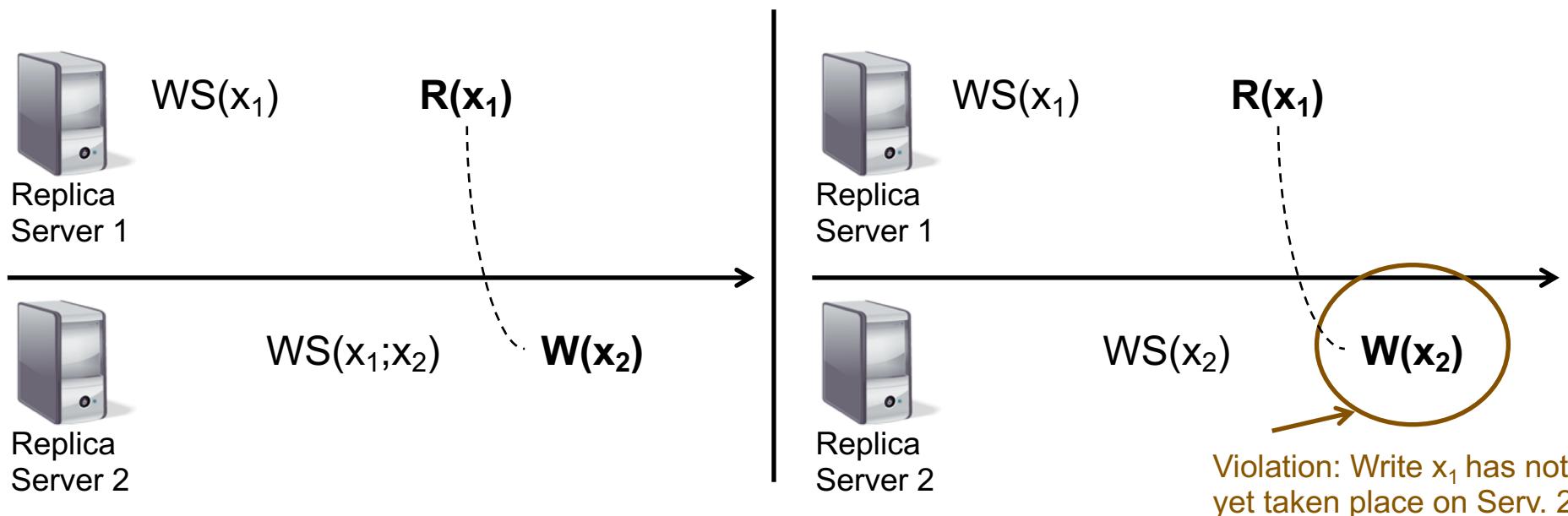
Read Your Writes

- Guarantee for the client:
 - The effect of a write operation by a client on data item x will always be seen by a successive read of x by the *same client*



Writes Follow Reads

- Guarantee for the client:
 - If a read on x precedes a write, then that write is performed after all writes that preceded the read



Summary Client-Centric Consistency Models

- Monotonic-read consistency
 - If a process reads x , any future read on x by the process will return the same or a more recent value
- Monotonic-write consistency
 - A write by a process on x is completed before any future write operations on x by the same process
- Read your writes
 - A write by a process on x will be seen by a future read operation on x by the same process
- Writes follow reads
 - A write by a process on x after a read on x takes place on the same or more recent value of x that was read

General Remark on Consistency

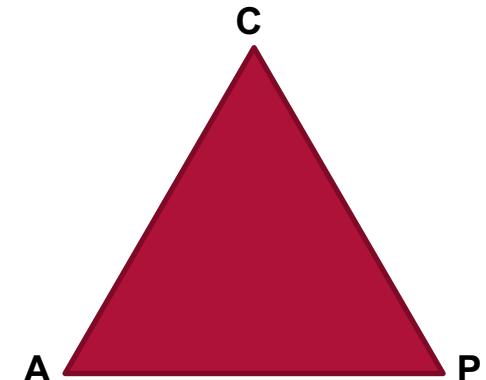
- In general, the stricter the consistency model, the more it impacts the scalability of a system
 - More consistency requires more synchronization
 - While the data is synchronized, some client requests may be answered
- Databases of the 80s and 90s put strong emphasis on consistency, lived with limited scalability/availability
- Today's cloud databases often sacrifice consistency in favor of scalability and availability

Overview

- Intro
- Partitioning
- Replication and Consistency
- **CAP Theorem**
- Case Studies

Brewer's CAP Theorem^[2]

- In a distributed system, it is impossible to provide all three of the following guarantees at the same time:
 - Consistency: Write to one node, read from another node will return something no older than what was written
 - Availability: Non-failing node will send proper response (no error or timeout)
 - Partition tolerance: Keep promise of either consistency or availability in case of network partition



- Proof by Seth Gilbert and Nancy Lynch^[3]

Illustration of CAP Theorem (1/2)

- Illustration of CA (consistency and availability)
 - Example: Replicated DBMS
 - Provided all servers can communicate, it is possible to achieve consistency and availability
- Illustration of PC
 - Example: Pessimistic locking of distributed databases
 - System can provide consistency even if some nodes are temporarily unreachable
 - However, some requests may not be answered due to unreachable nodes (violates availability property)

Illustration of CAP Theorem (2/2)

- Illustration of AP
 - Example: DNS system
 - System continues to work even when some DNS servers are offline (availability)
 - System tolerates network outages (partition tolerance)
 - DNS makes extensive use of caching to achieve partition tolerance
 - ◆ Non-authoritative DNS server may answer request with old data (violates consistency)

Overview

- Intro
- Partitioning
- Replication and Consistency
- CAP Theorem
- Case Studies

Literature and References

- Literature
 - A.S. Tannenbaum, M. Van Steen: “Distributed Systems: Principles and Paradigms”, Prentice Hall, 2016, Chapter 7
 - M. Kleppmann, “Designing Data-Intensive Applications”, 2017, Chapter 5 and 6
- References:
 - [2] Eric. A. Brewer: “Towards Robust Distributed Systems”, PODC Keynote 2004, <http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf>
 - [3] S. Gilbert, N. Lynch: “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”, ACM SIGACT News, 33 (2), 2002