Cloud Computing

Chapter 3: Platform as a Service

Summer Term 2017

Complex and Distributed IT Systems

TU Berlin

Overview

- **Motivation**
- Fundamentals for scalable/available applications
- Case studies

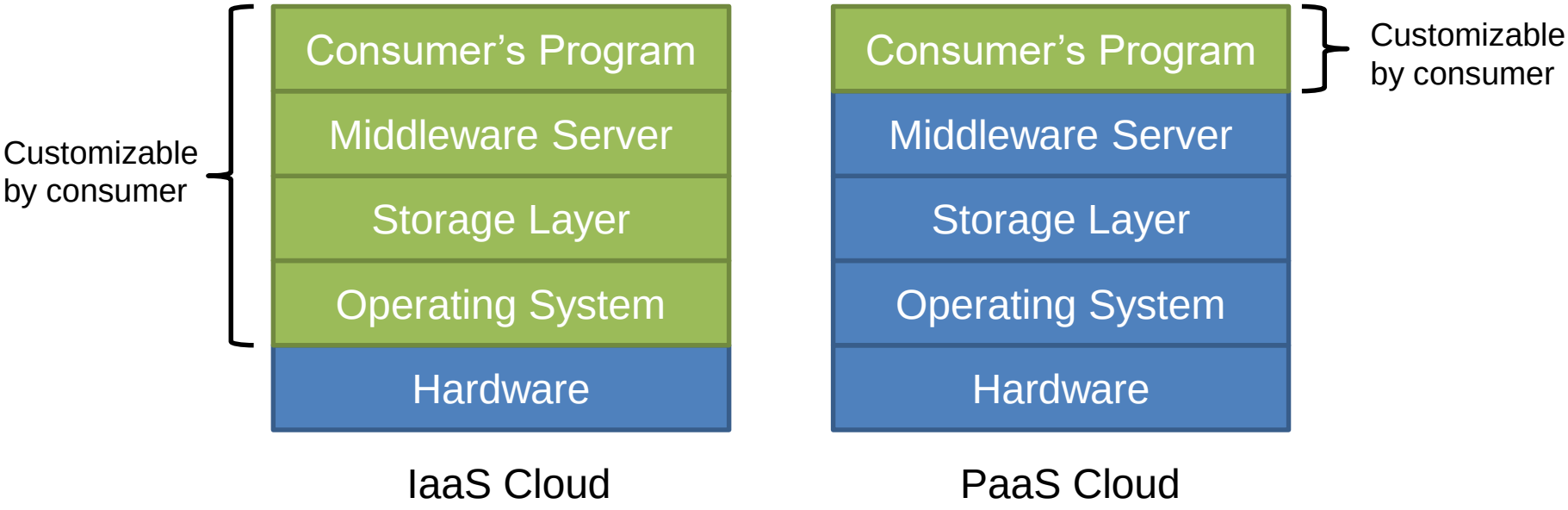
Recap: PaaS Clouds

- Services offered by PaaS clouds (according to NIST[1])
  - Programming languages
  - Libraries
  - Services
  - Tools

- Characteristics of services
  - Consumer can deploy custom applications on PaaS cloud using the provider's application model
  - Consumer does not directly control the operating system, storage, and deployed applications

IaaS vs. PaaS (1/2)

- PaaS offers higher abstraction level compared to IaaS
  - Less development/maintenance effort
  - Less flexibility, high provider dependence



IaaS vs. PaaS (2/2)

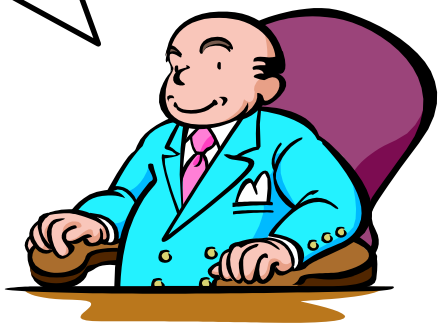
- Higher abstraction level enables other pricing models
- Service usage can be charged
  - by time
  - per query (e.g., for database services)
  - per message (e.g., for message queues)
  - per CPU usage (e.g., for request-triggered applications)
  - …
- Enables interesting scenarios for consumers
  - Deployed applications can have very low operational costs until they become popular

PaaS Value Proposition

1. Maintenance
   - Hardware maintenance
   - OS patches
   - Middleware updates
2. Availability
   - Application/service will be available despite maintenance/outages
3. Scalability
   - Application/service will scale to thousands of concurrent users

Developer, if you want to build and deploy a ***web application*** or ***service***, use our software stack.

It will give you the following ***three*** things:



Overview

- Motivation
- **Fundamentals for scalable/available applications**
  - **Partitioning**
  - Replication
  - Brewer's CAP theorem
- Case studies

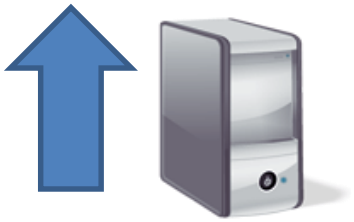
How to Achieve Availability/Scalability?

- 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? (1/2)

- Two principles methods to scale



Vertical Scalability (Scaling Up)	
Idea	Increase performance of a single node (more CPUs, memory, …)
Pro:	Good speed-up up to a particular point
Con:	Beyond that point, speed-up becomes very expensive

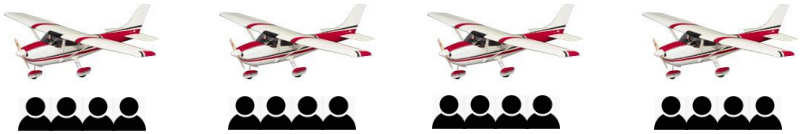
Horizontal Scalability (Scaling Out)	
Idea	Increase number of nodes
Pro:	Cheap to grow total amount of resources
Con:	Standard software is often not able to leverage resources, special software needed

Scalability
- Horizontal vs Vertical

Application: 

Host:  (Too small)

Scale horizontally:



Scale vertically:

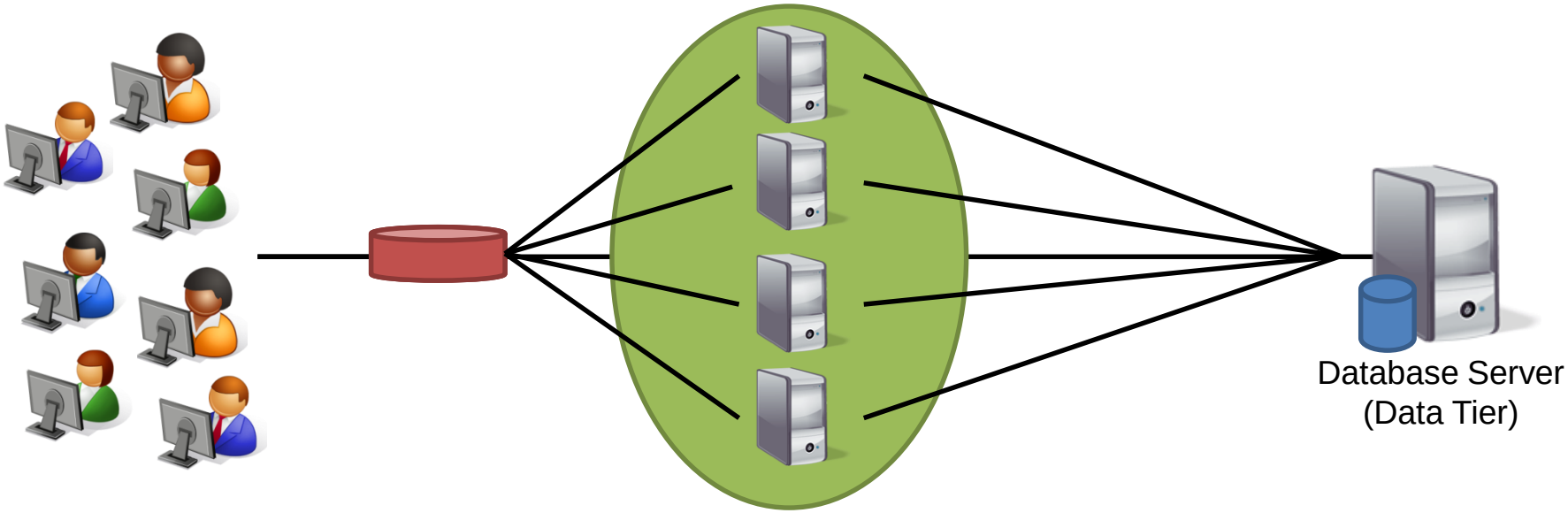
How to Achieve Scalability? (2/2)

- 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 WS, DBMS, …

Scalability with Stateless Components

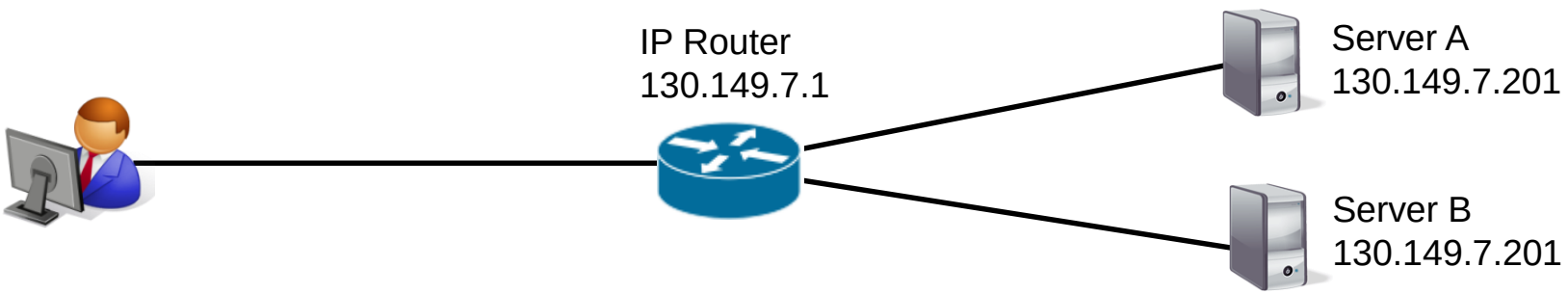
- Approach: More instances of the bottleneck component



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

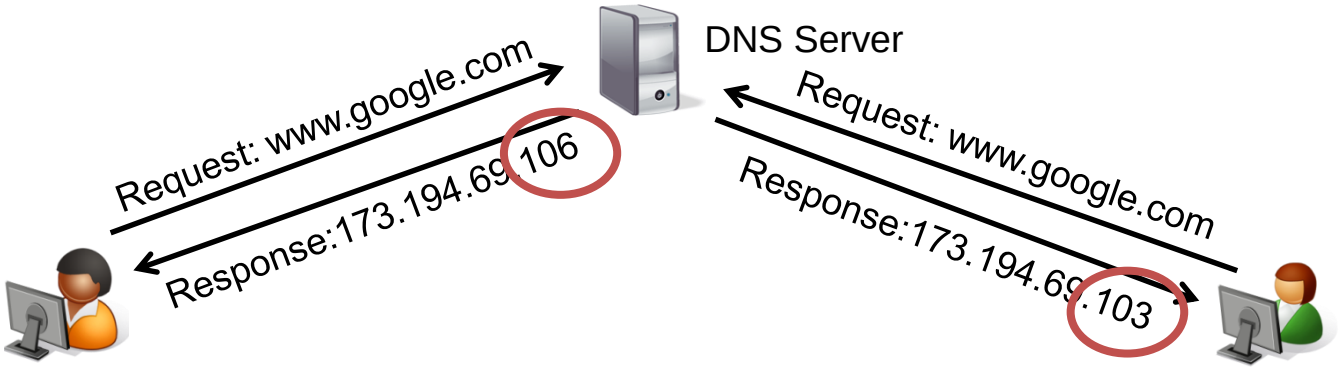
Possible Levels of Stateless Load Balancing (1/3)

- 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)



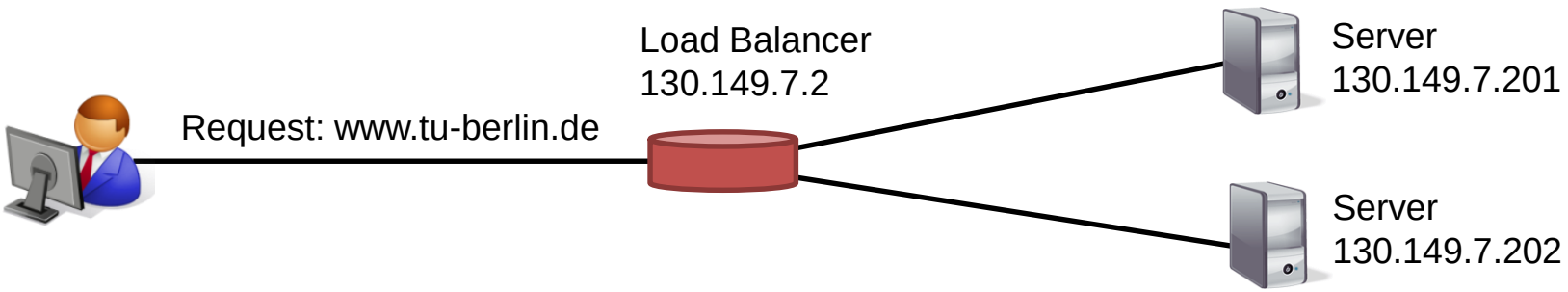
Possible Levels of Stateless Load Balancing (2/3)

- 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
  - Explicitly distributes request among available machines
  - Clients send requests to load balancer
- Requirements for applicability
  - No network bottleneck
- Examples: Various websites, Amazon Elastic LB



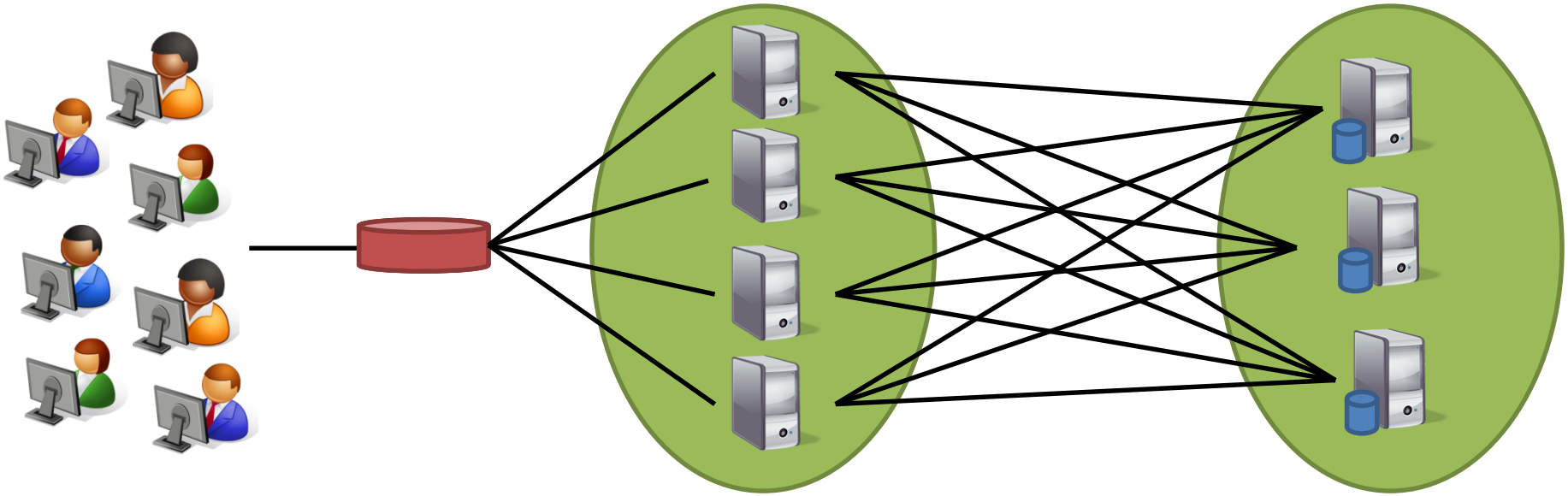
Strategies for Stateless Load Balancing

- 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 request 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

Scalability with Stateful Components

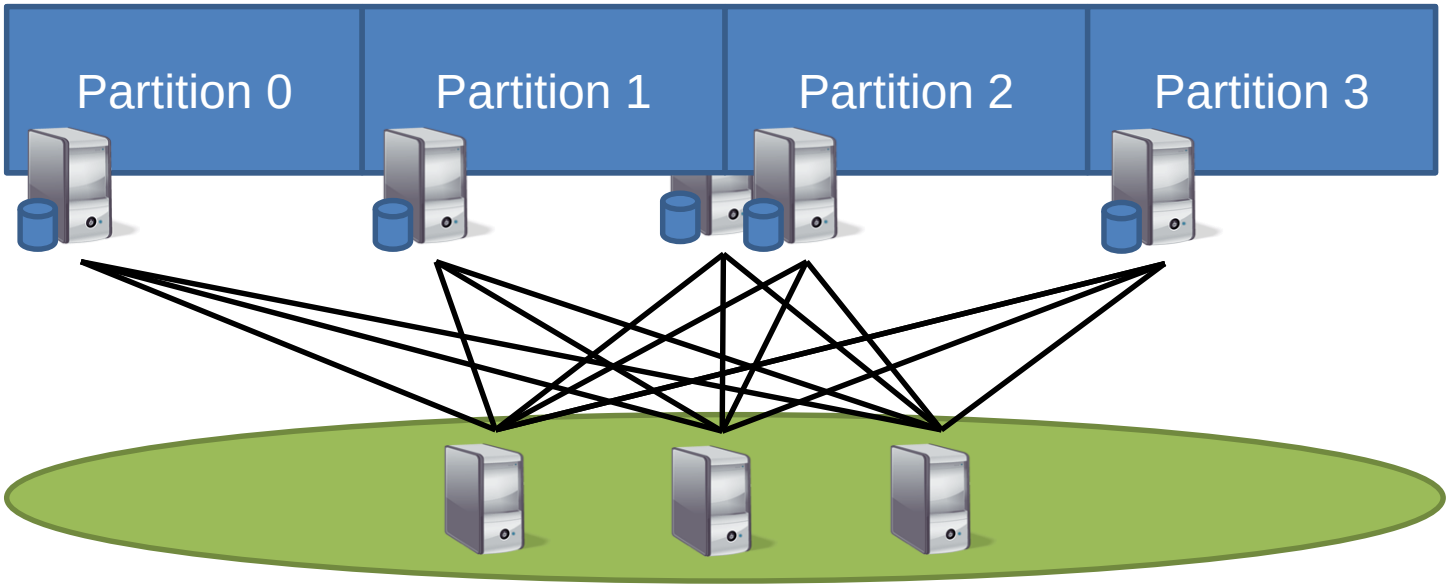
- Let's assume the data tier is the bottleneck now



- Database is stateful, store 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 of one or more parts



- Pure partitioning improve scalability but not availability
  - Each data item is only stored in one partition!

How to Partition the Data?

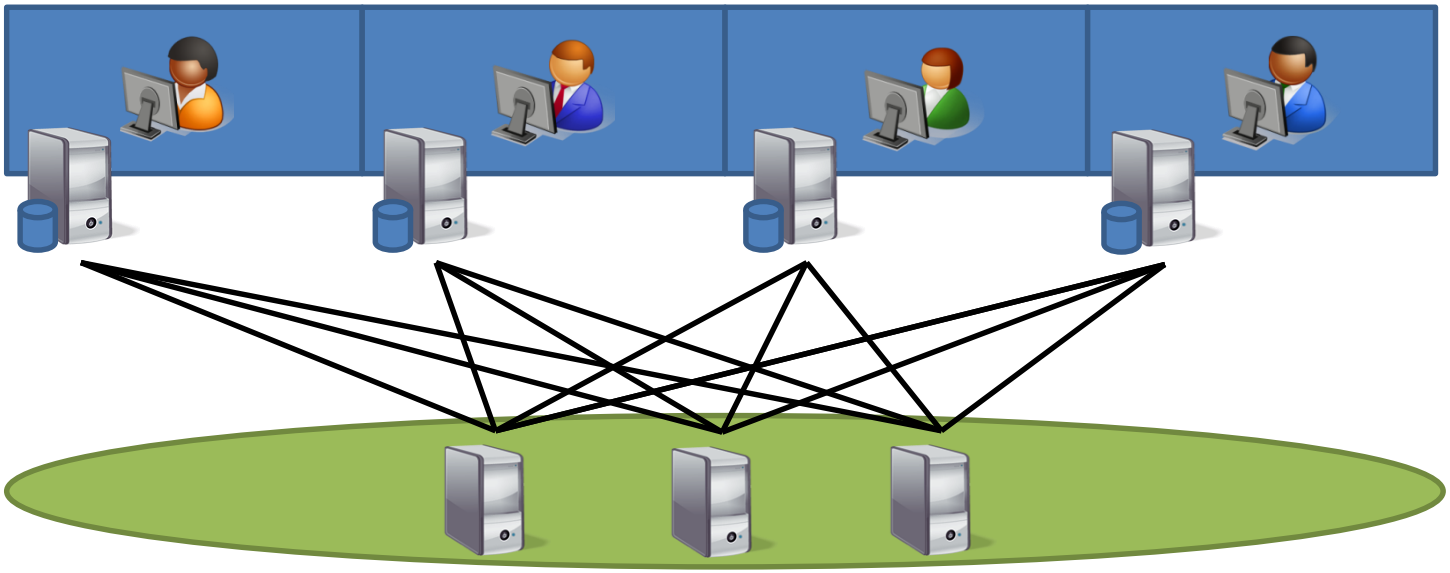
- General consideration for partitioning and scalability
  - Data is now spread across several machines
  - Particular tasks might require to pull data from different places servers together → causes network traffic
- Network is a scarce resource with limited scalability
  - Becomes scarcer the more 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 tenant on different machines
  - Pro: In PaaS clouds, tenants are expected to be isolated

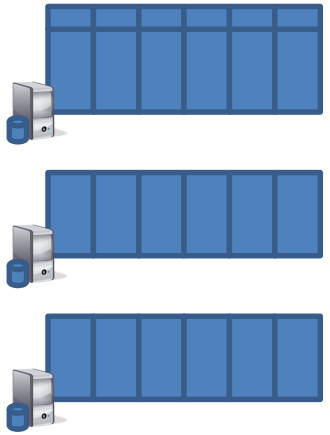
→No network traffic between machines, good scalability

→Con: Tenant cannot scale beyond one machine

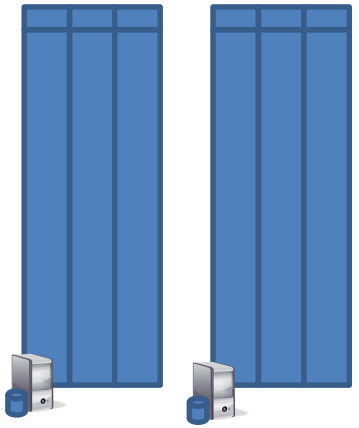


Popular Partitioning Schemes (2/2)

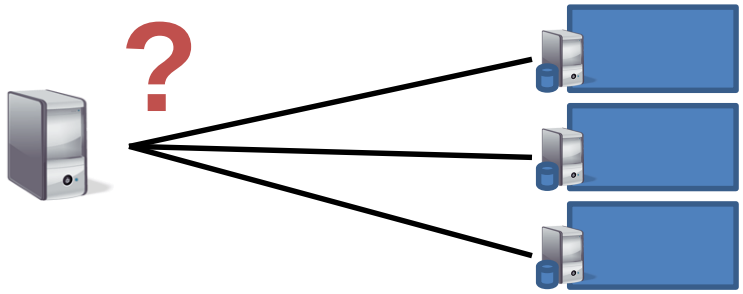
- Horizontal partitioning (relational databases)
  - Split table by rows
  - Put different rows on different machines
  - Reduced number of rows, reduced indices
  - Done by Google BigTable, MongoDB



- Vertical partitioning (relational databases)
  - Split table by columns
  - Not very common to improve scalability
    - Just mentioned here for the sake of completeness



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 $k_1, k_2 \epsilon K$ are close, $p(k_1), p(k_2) \epsilon P$ shall also be close to each other
  - Pro: Efficient for range queries and partition scaling
  - Con: Poor load balancing properties

Overview

- Motivation
- **Fundamentals for scalable/available applications**
  - Partitioning
  - **Replication**
  - Brewer's CAP theorem
- Case studies

Replication

- Partitioning helps with scalability but not availability
  - Data is only stored in one location
  - If machine 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?

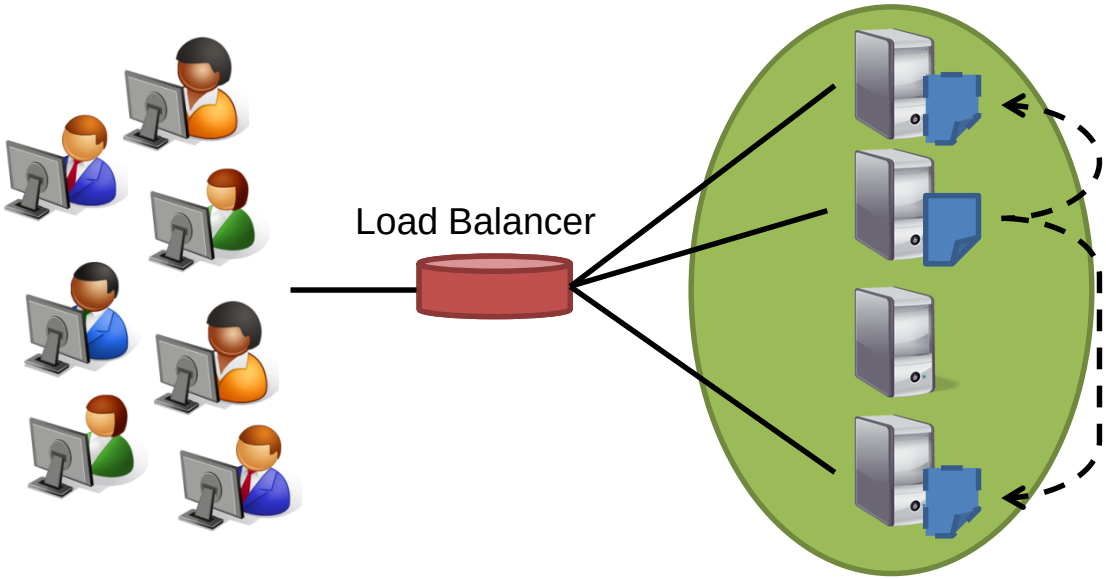
- Replication can improve both scalability and availability!

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
  - For example for content distribution networks (Akamai, …)
  - Chosen to keep network transfers locally confined

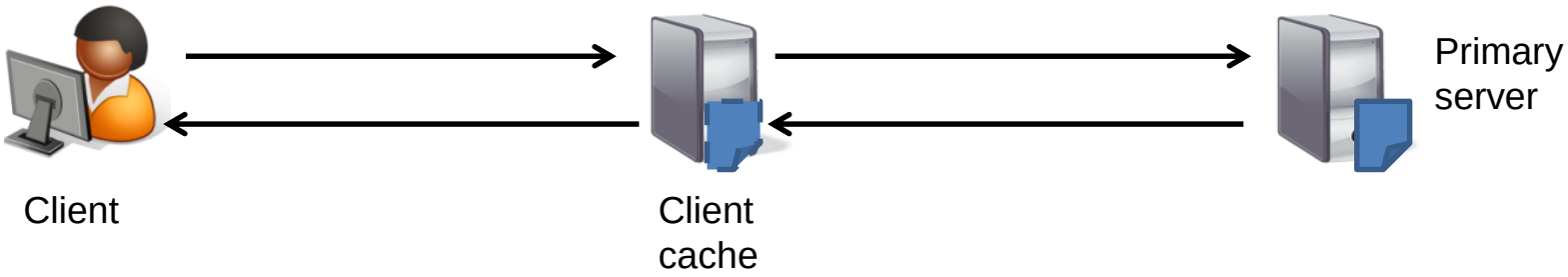
Who Creates the Copies? (1/2)

- Server-initiated replication
  - Copies are created by 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 the scarce

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

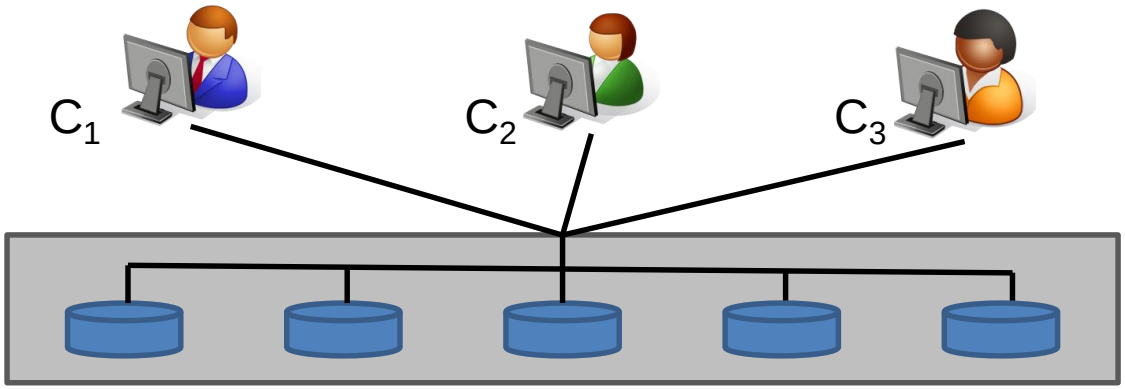
How to Deal with Inconsistencies?

- 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 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
  - However, the higher the consistency level (CL), the lower the performance typically
  - There must be a clear understanding what CL an appl. can expect from a replicated data store

Consistency Models

- Assume a data store with replication
  - Clients can read/write data in the store
  - Client cannot influence which copy it accesses
  - Updates are propagated among replicas in data store



- Consistency model: Contract between client and 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

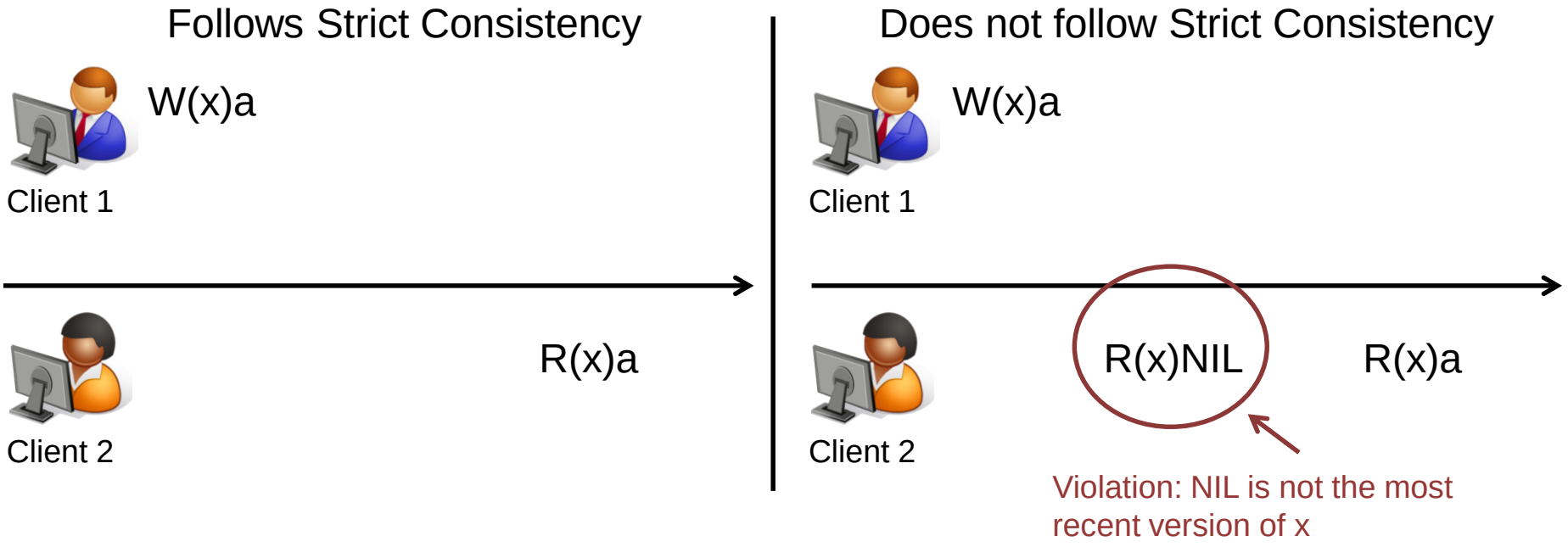
- Client-centric consistency models
  - Talk about consistency from the client‘s perspective
  - Provides guarantees how a *single* client perceives the state of a replicated data item

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
    - …

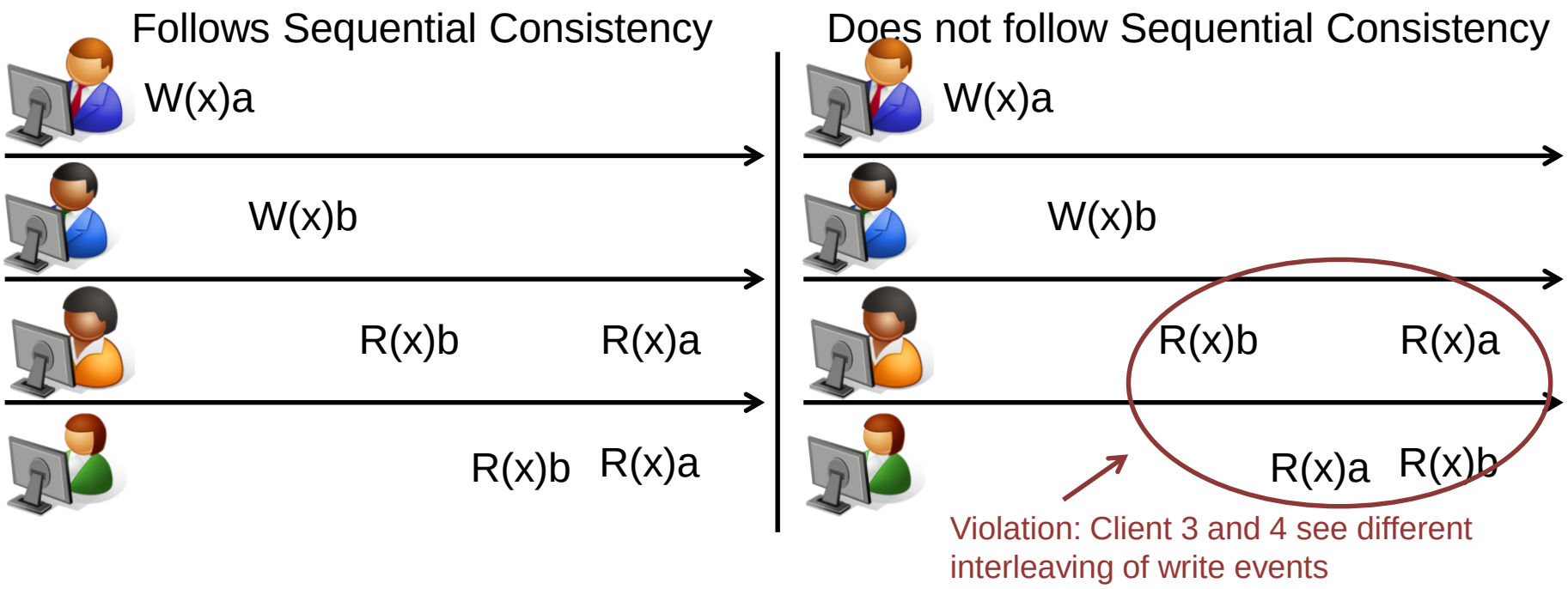
Strict Consistency

- 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



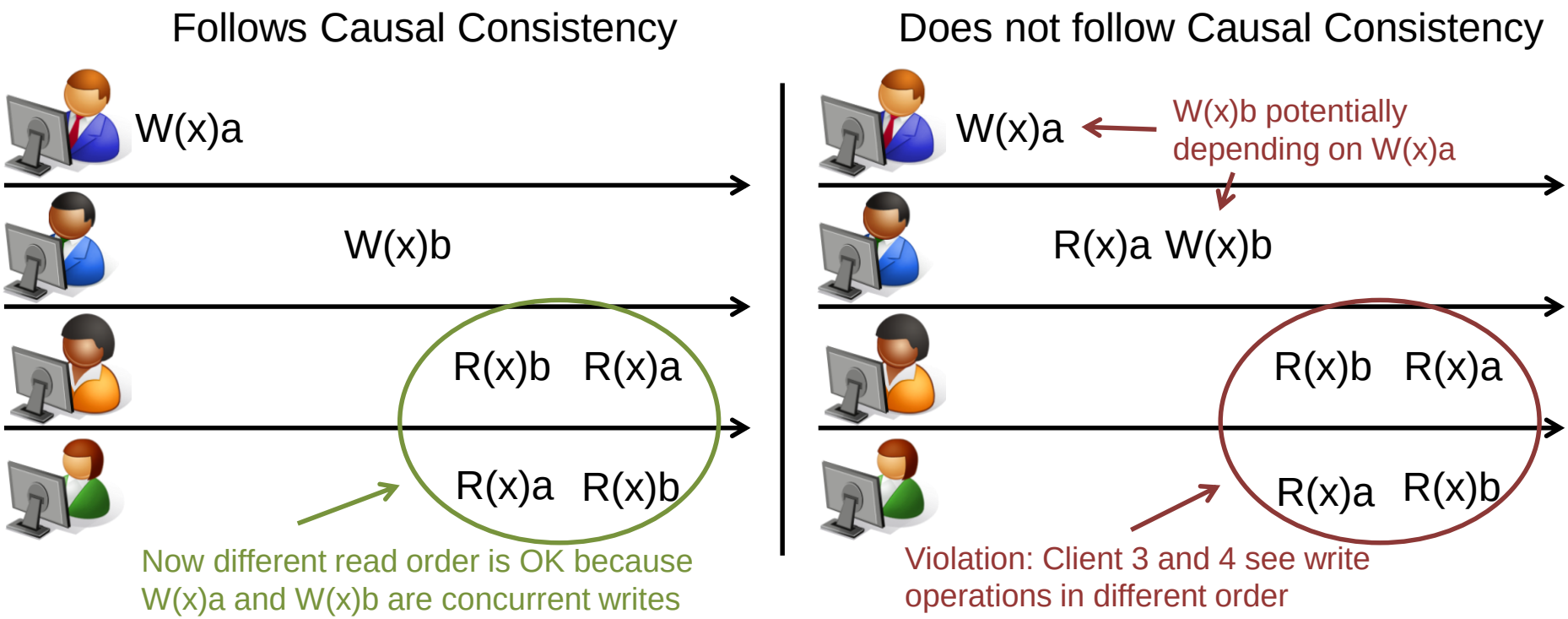
Sequential Consistency

- 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. Concurrent writes may be seen in a different order by different clients.



Client-Centric Consistency Models

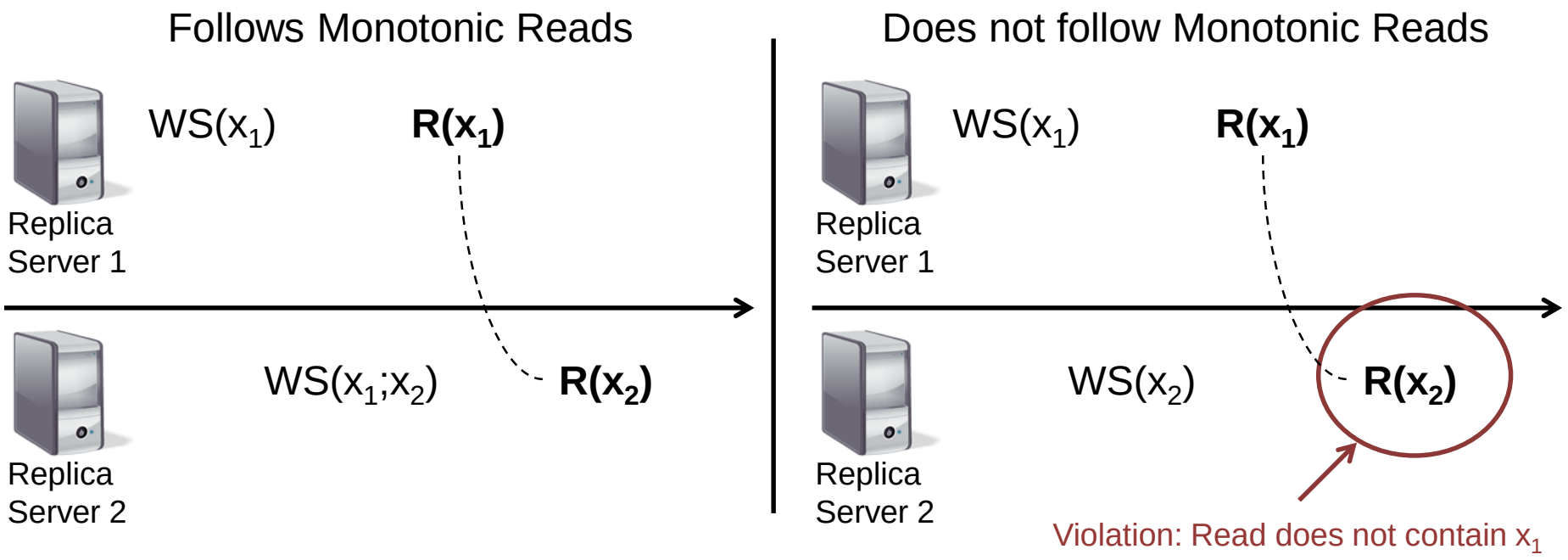
1. Eventual Consistency
2. Monotonic Reads
3. Monotonic Writes
4. Read Your Writes
5. Writes Follow Reads

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 loose its own updates
- Model enjoys popularity in the cloud world as it can be implemented very cheaply

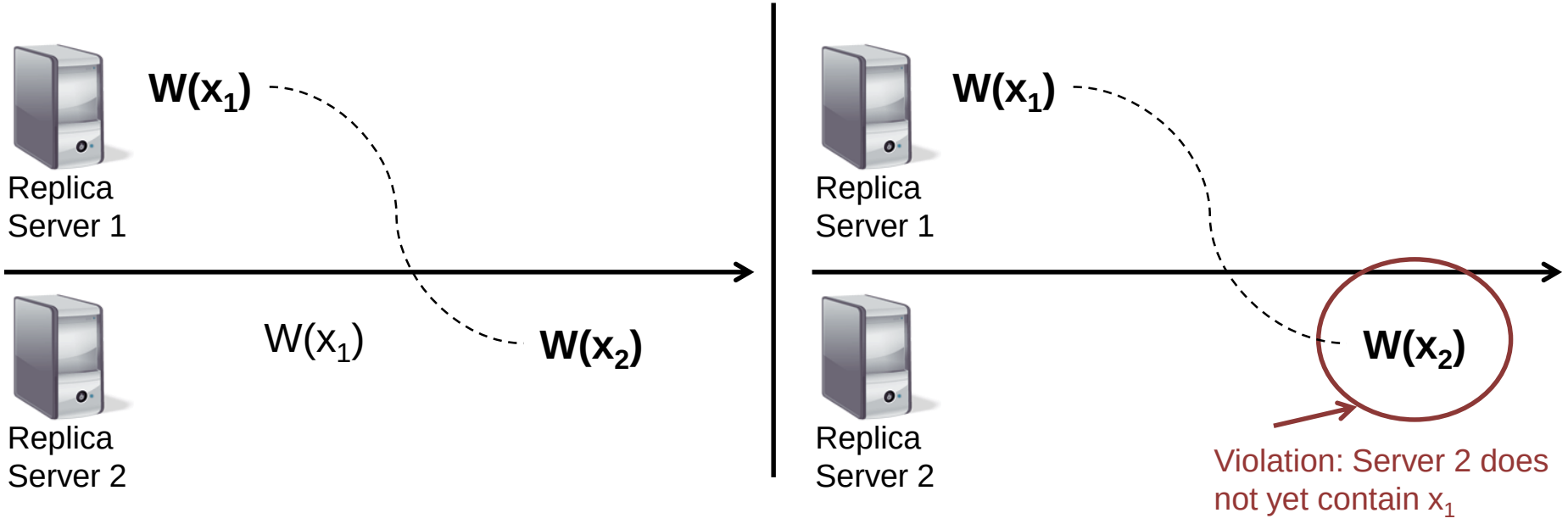
Monotonic Reads

- 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



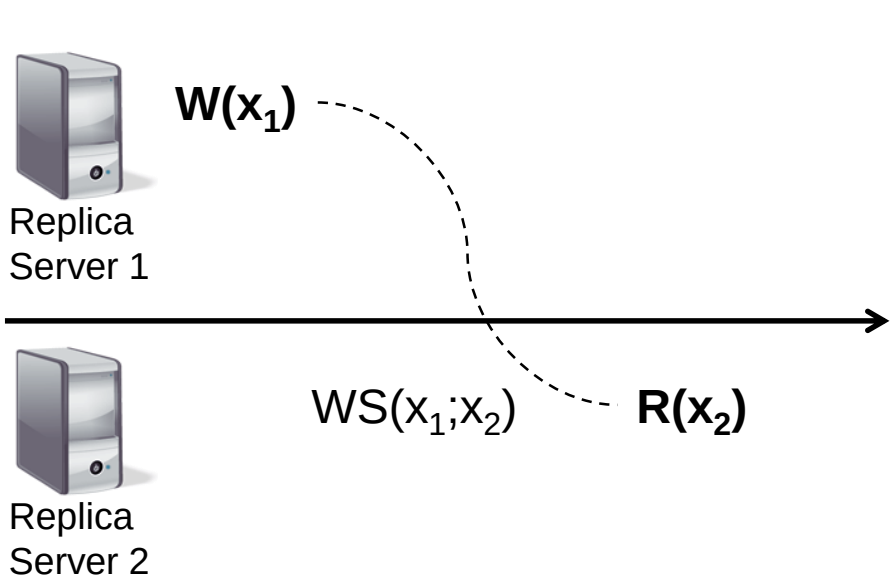
Monotonic Writes

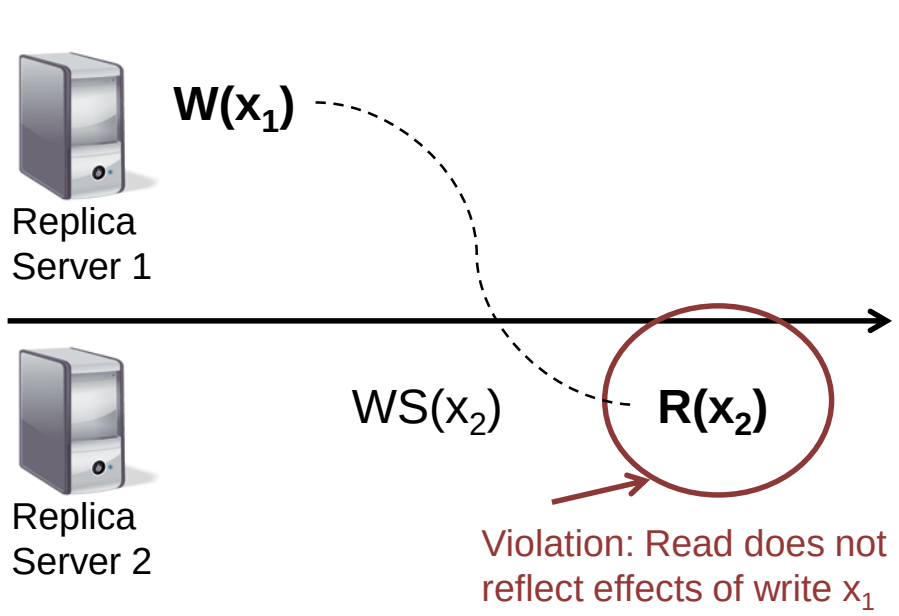
- Guarantee for the client:
  - A write operation by a client on data item x is completed before successive write operations on x by the same client



Read Your Writes

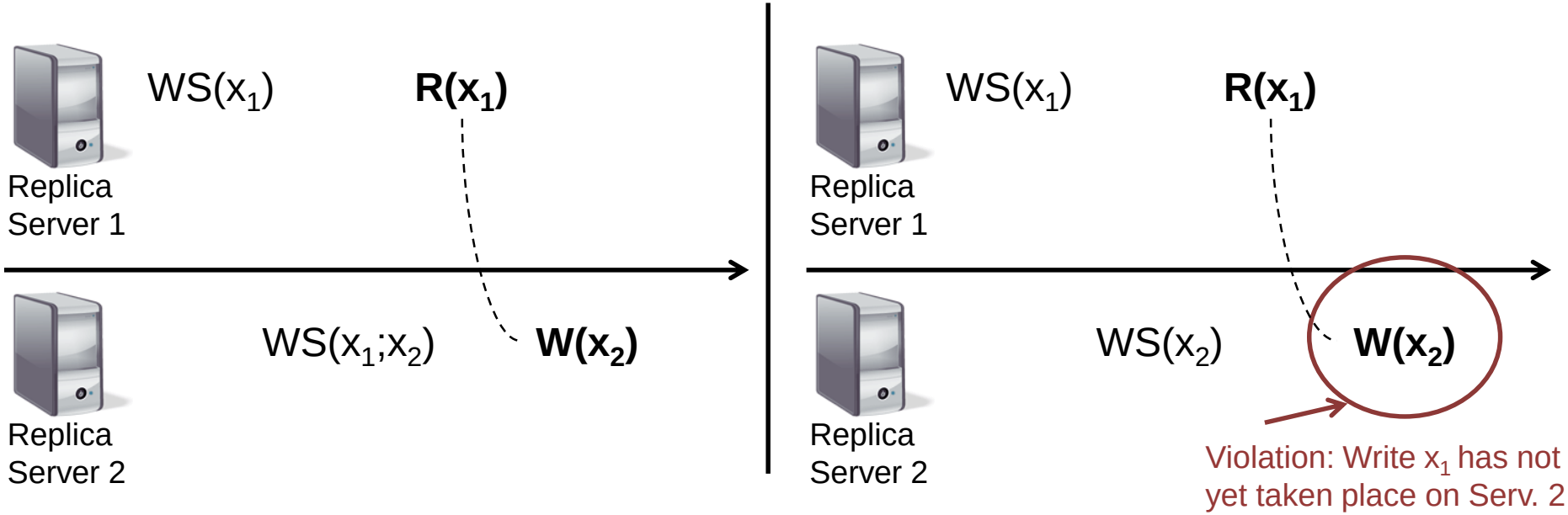
- Guarantee for the client:
  - The effect of a write operation by a client on data item x will be always seen by a successive read of x by the same client





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 reads on x by the process will returns 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 write
  - 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

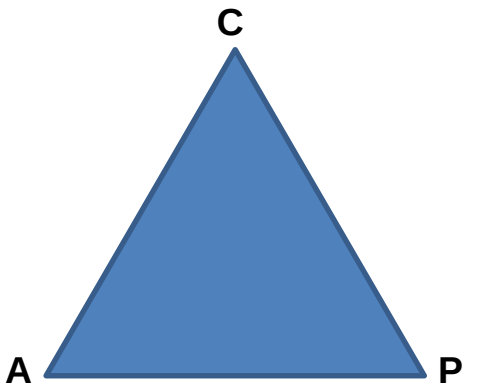
- In general, the stricter the consistency model, the more it impacts the scalability of a system
  - More consistency requires more synchronization
  - While the data is synchronized, some client requests may be answered
- Databases of the 80s and 90s put strong emphasis on consistency, lived with limited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

Overview

- Motivation
- **Fundamentals for scalable/available applications**
  - Partitioning
  - Replication
  - **Brewer's 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 unreachability (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

- Motivation
- Fundamentals for scalable/available applications
- **Case studies**
  - **Amazon Dynamo**
  - Microsoft Azure
  - Google App Engine

Amazon Dynamo[4]

- Highly-available key-value store used inside Amazon
  - Powers parts of AWS, such as Amazon S3[5]

- Primary design goals
  - High scalability (100-1000 servers)
  - High availability
    - Particularly, support for „always write“
  - High performance
  - Small latency

- Sacrifices consistency to achieve these goals

Amazon Dynamo Design Principles

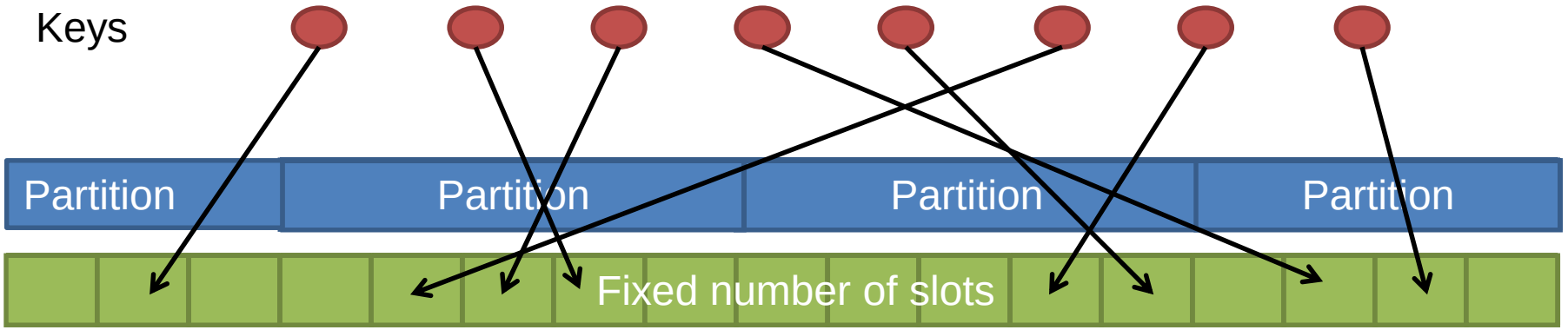
- Dynamo follows Peer-To-Peer approach
  - No server is more important than any other
  - No single point of failure
- Nodes can be added/removed incrementally at runtime
  - In terms of the CAP theorem, Dynamo is AP
  - Weak consistency guarantees (eventual consistency)
- No hostile environment, all servers obey rules
  - No security issues must be considered

How to Partition the Data Among the Servers?

- Dynamo designed to be a key-value store
  - →No support for range queries needed
  - →No need for range partitioning
- Hash partitioning has good load balancing properties
  - But how to avoid data reorganization when servers are added or removed?
- Solution: Hash function no longer maps to partitions
  - Instead function maps to a fixed number of slots
  - Variation of classic hashing called *consistent hashing*

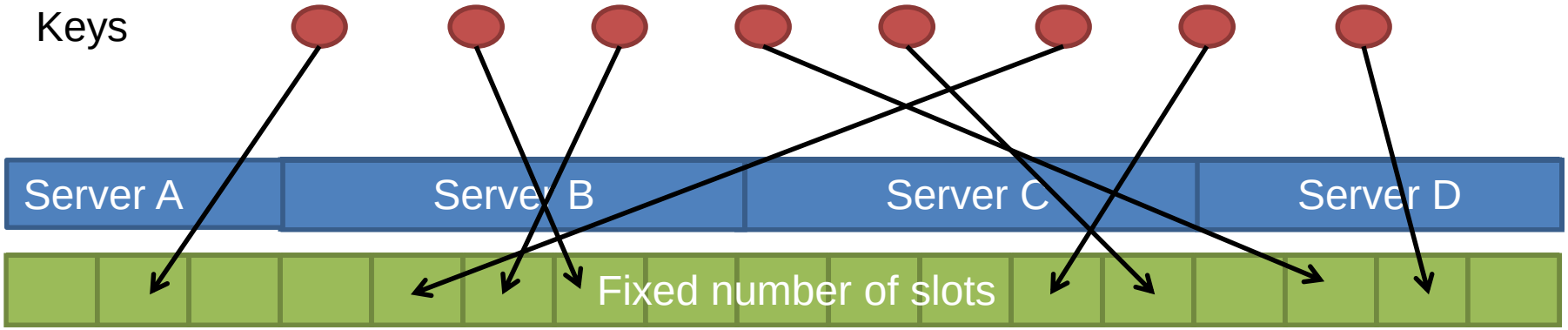
Consistent Hashing[6]

- Idea: Additional mapping between slots and partitions
  - Hash function maps keys to extremely large but fixed number of slots (for example 2^{128} slots)
  - A partition now covers a consecutive number of slots
    - A server can be in charge of one or more partitions
    - Size of a partition is variable



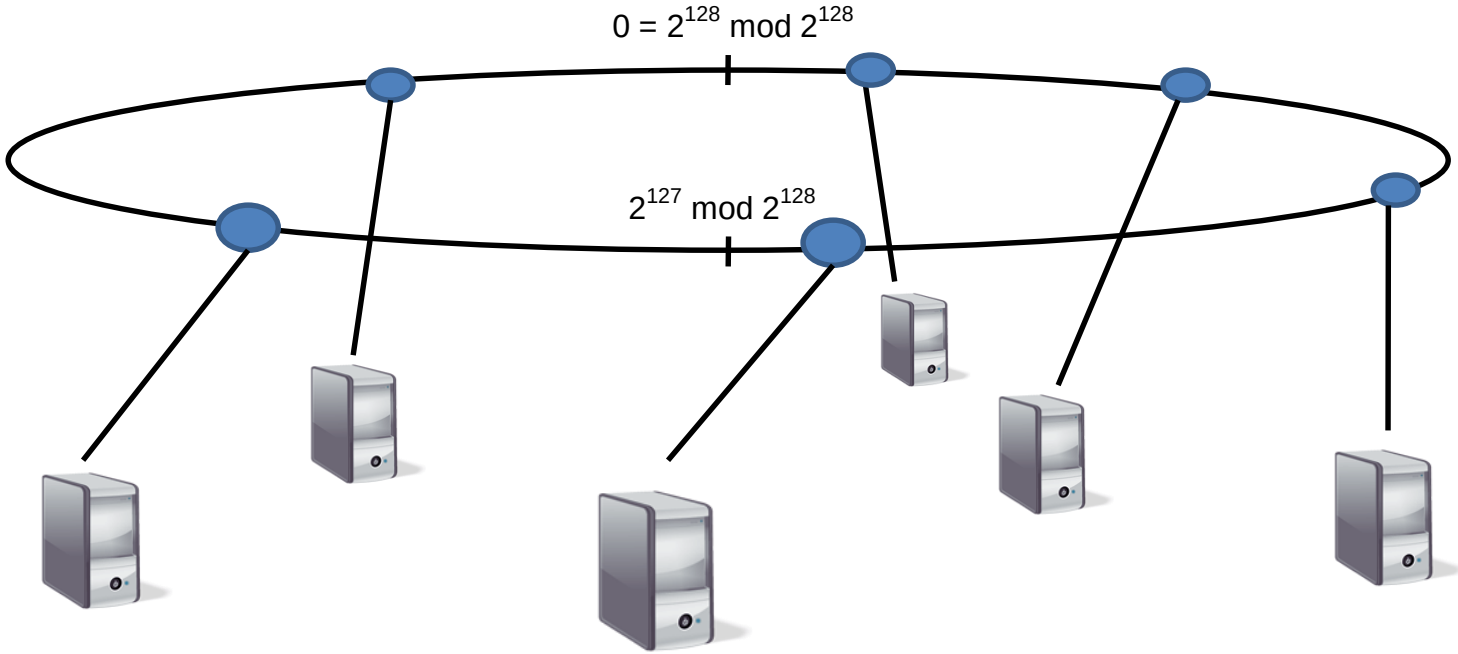
Distributed Hash Tables

- Consistent Hashing is basis for distributed hash tables (DHTs)
  - Each server takes at least one partition
  - Therefore each server is responsible for a continuous range of slots
  - Upon arrival/departure of server only O(#Keys/#Servers) data items must be reorganized



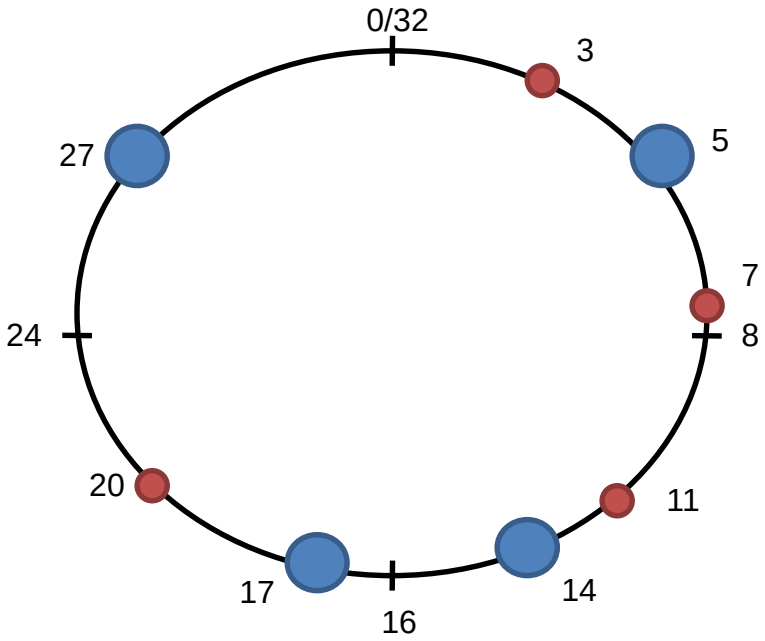
Amazon Dynamo‘s Partitioning Algorithm (1/2)

- Slots form a circular ID space
  - All servers hash their ID address with an MD5 function
  - Hashing result determines server‘s position on the ring



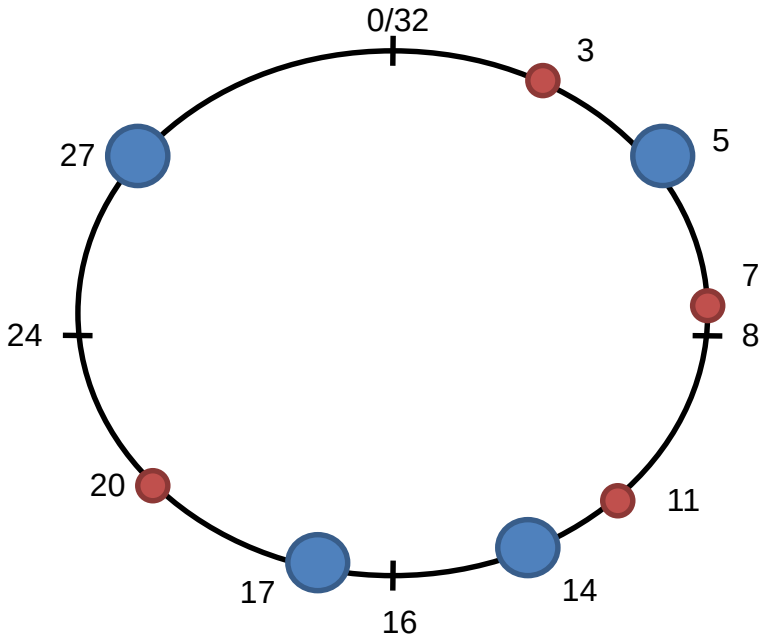
Amazon Dynamo‘s Partitioning Algorithm (2/2)

- To distribute data, key of data also hashed with MD5
  - Result of hashing is a position in the circular ID space
  - Rule: Server is responsible for all preceding IDs (i.e. slots) up to and including its own ID



Routing in Amazon Dynamo

- Each server maintains full routing table (ID to IP)
  - → Each server can determine which server is responsible for a data item based on routing table and mapping rule!
  - → One hop routing keeps latencies small

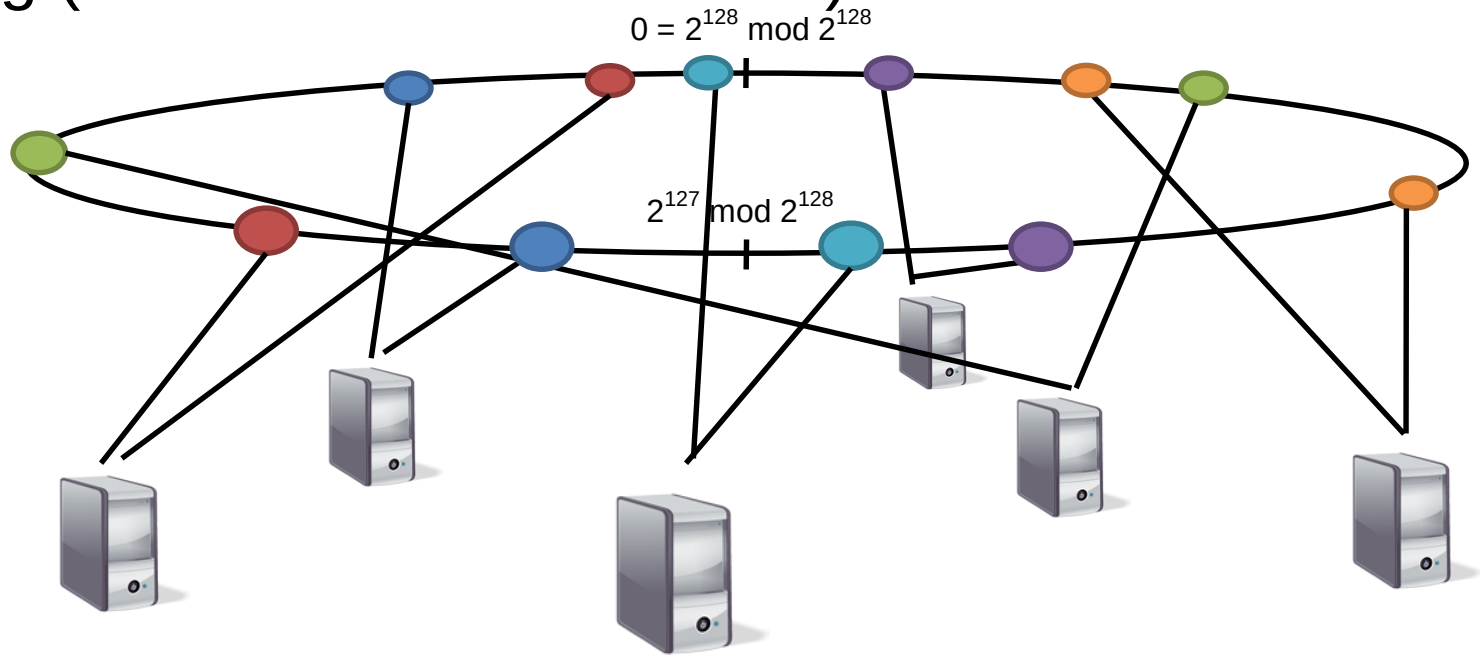


Example routing table in Dynamo

ID	IP Address
5	192.168.1.2
14	192.168.1.18
17	192.168.1.115
27	192.168.1.98

Virtual Servers for Load Balancing

- Despite uniform distribution over ID space, the servers may receive skewed number of requests
- Idea: Each server appears on multiple positions on the ring (known as virtual servers)



Replication in Amazon Dynamo

- Servers replicate data to their N successors on the ring
- Replication is adjusted whenever a server is added or removed from the ring
- Servers use heart beat protocol to determine availability
  - Periodic messages exchanged between ring neighbors
  - When server does not answer heart beat request in a given time span, it is considered gone
- Dynamo allows reads and writes on every replica!

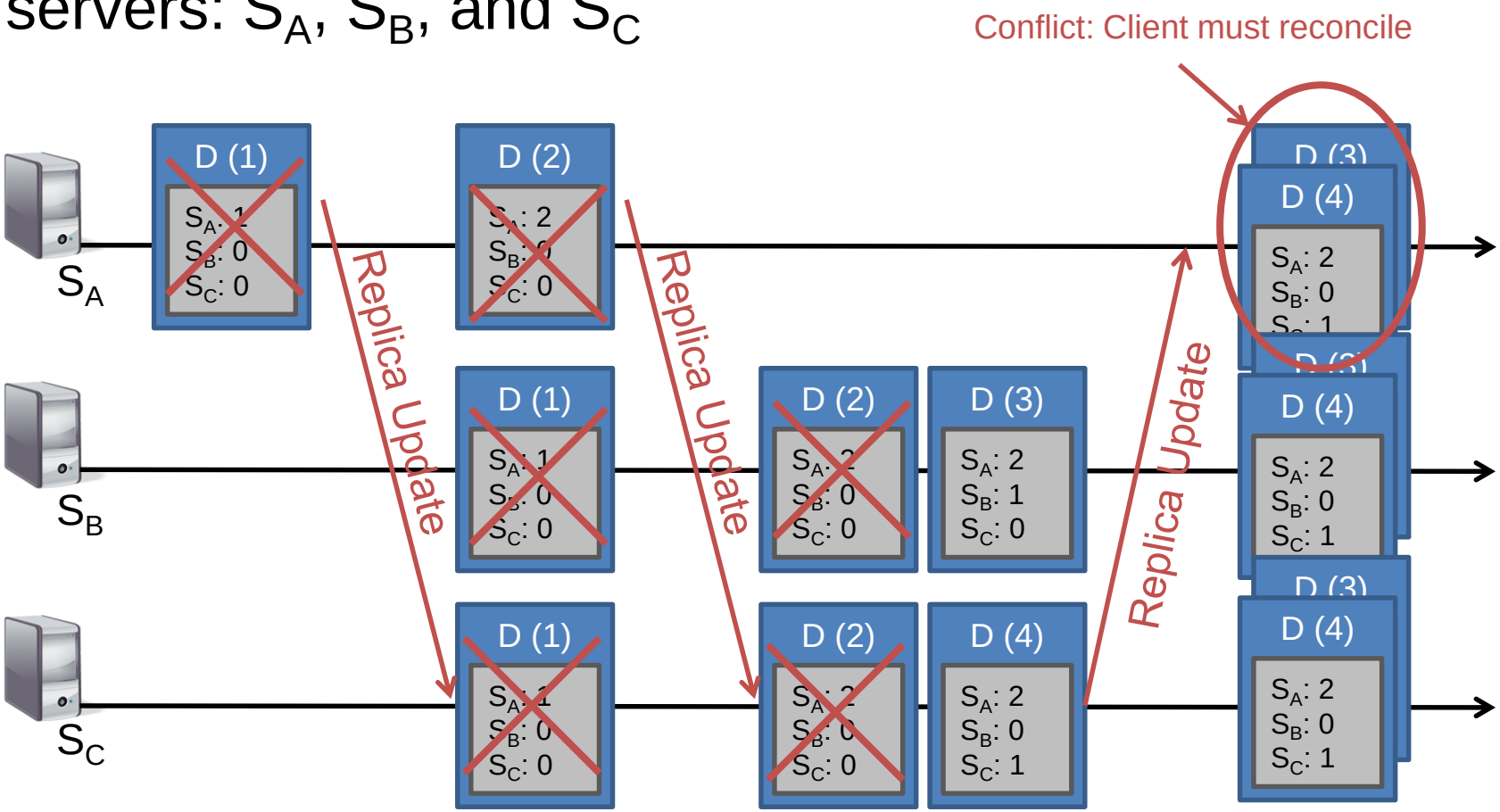
Data Versioning in Amazon Dynamo

- Dynamo allows „always write“ paradigm
  - Write operation allowed on every replica
  - put-() operation returns after one replica has been written
    - Lazy update of replicas in the background

- Result: Different version of a data item may exist
  - Dynamo treats each version of the data item as an immutable object
  - Vector clocks are used to reconcile different versions
  - When system cannot reconcile different versions, the versions are presented to client for reconciliation

Example of Version Reconciliation

- Let‘s assume data item D is replicated among three servers: S_A, S_B, and S_C

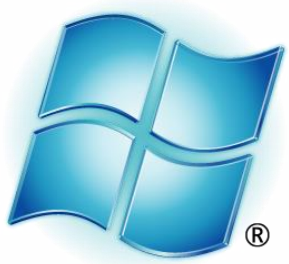


Overview

- Motivation
- Fundamentals for scalable/available applications
- **Case studies**
  - Amazon Dynamo
  - **Microsoft Azure**
  - Google App Engine

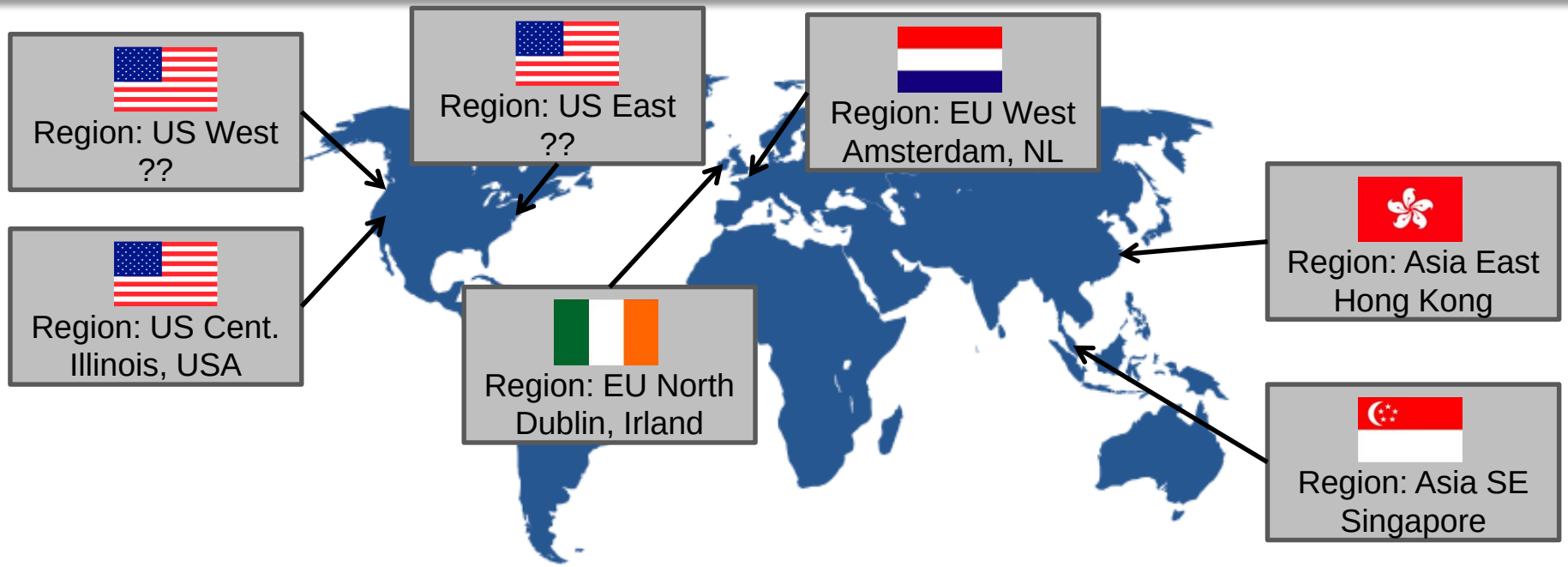
Microsoft Azure

- Microsoft‘s cloud computing platform



- Launched in 2010 as PaaS solution
  - Targeted at scalable, reliable web applications
- Initially, platform was composed of three components
  - Microsoft Azure: Compute and storage services
  - SQL Azure: cloud-based DBMS
  - Azure AppFabric: Tools to bridge gap between local and cloud-hosted applications

Geographic Distribution of Azure Data Centers



- Microsoft calls each geographic location a *region (38 regions around the world)*
  - Within a region, customers can create *affinity groups* to deploy services as closely together as possible

Microsoft Azure Affinity Groups

- Azure data centers are built from shipping containers
  - Each container can contain up to 2500 servers

Images from [7]

- Affinity group: Logical grouping of components
  - Azure tries to deploy components as closely as possible
  - For example within the same container if possible

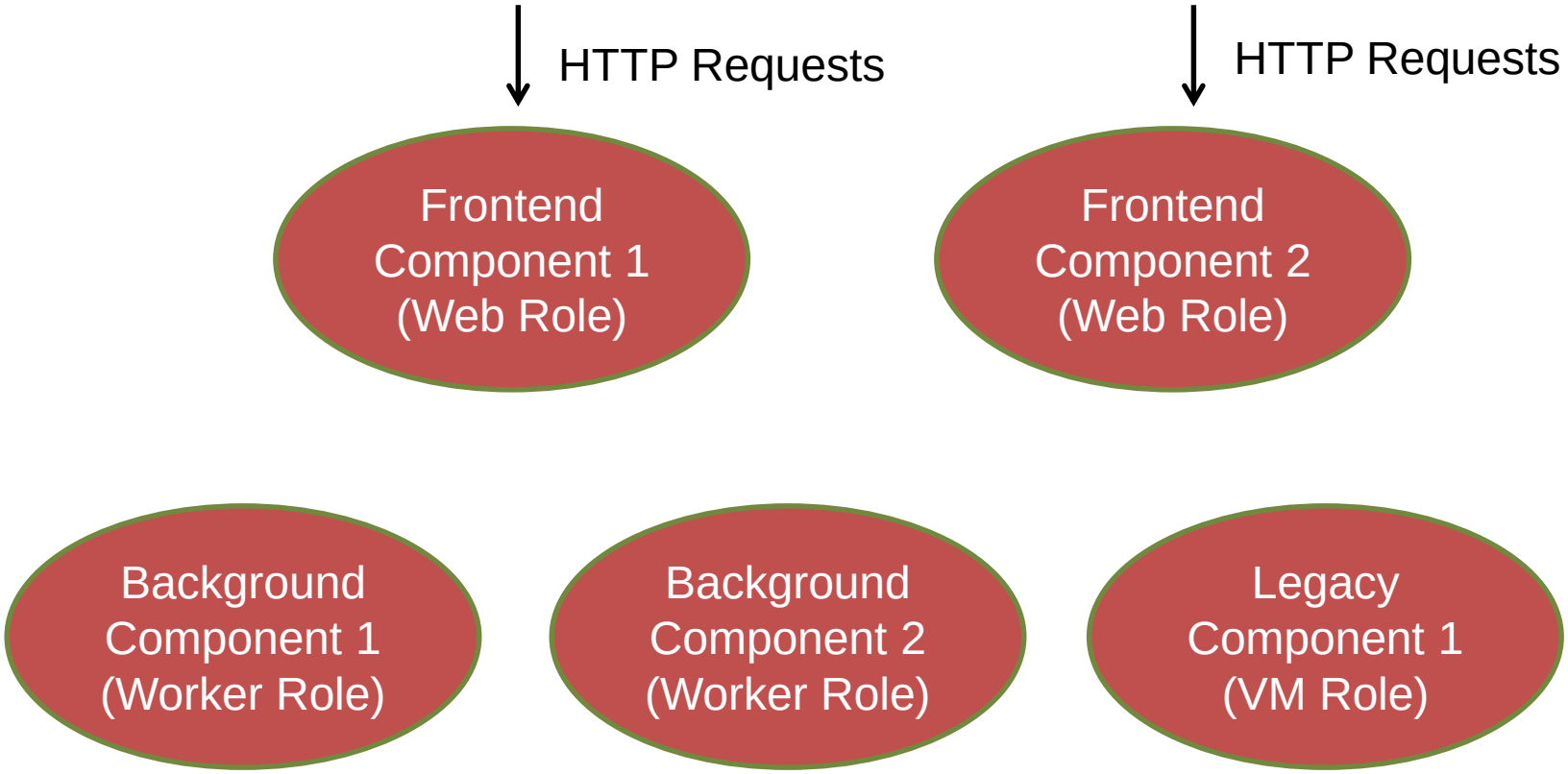
Microsoft Azure Programming Model (1/4)

- Azure applications separated into logical components
  - Each component as assigned to a so-called role
  - Multiple instances of the same component might exist

- Azure programming model offers three roles
  - Web role: Components facing the outer world
    - Accepting requests via HTTP, e.g. Web sites/services
  - Worker role: Components doing background tasks
  - VM role: Legacy components
    - Component which cannot be converted to the programming model

Microsoft Azure Programming Model (2/4)

- Example of an Microsoft Azure application according to the programming model



Microsoft Azure Programming Model (3/4)

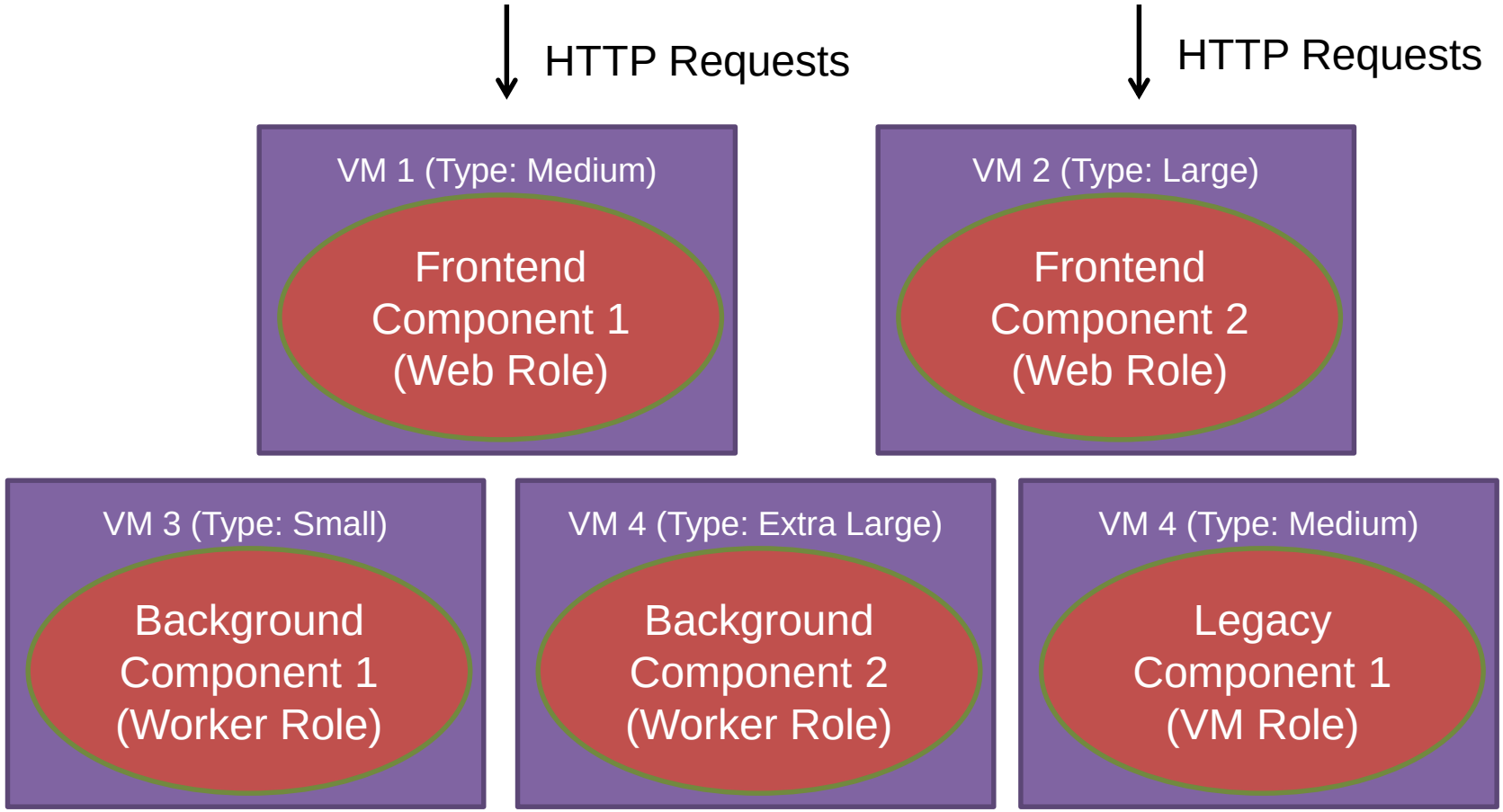
- Each instance of a component is executed in a separate virtual machine
  - Improves maintainability and isolation
  - Customer can choose between different types of VMs
  - VM usage is billed by the hour (like IaaS model)

Virtual Machine Size	CPU Cores	Memory	Price per Hour
Extra Small	Shared	768 MB	USD 0.02
Small	1	1.75 GB	USD 0.12
Medium	2	3.5 GB	USD 0.24
Large	4	7 GB	USD 0.48
Extra Large	8	14 GB	USD 0.96

Overview of VM types and pricing from https://azure.microsoft.com/de-de/pricing/ /

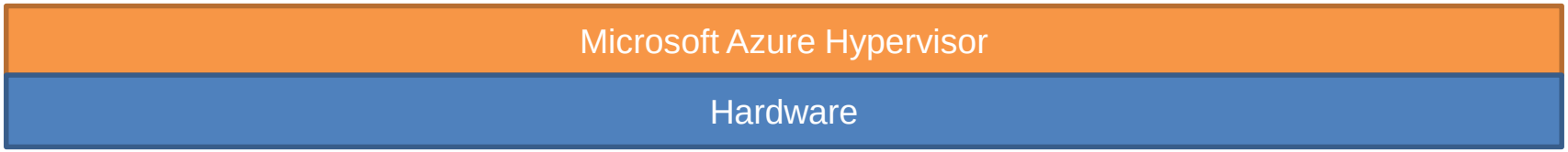
Microsoft Azure Programming Model (4/4)

- Example application with virtual machine isolation



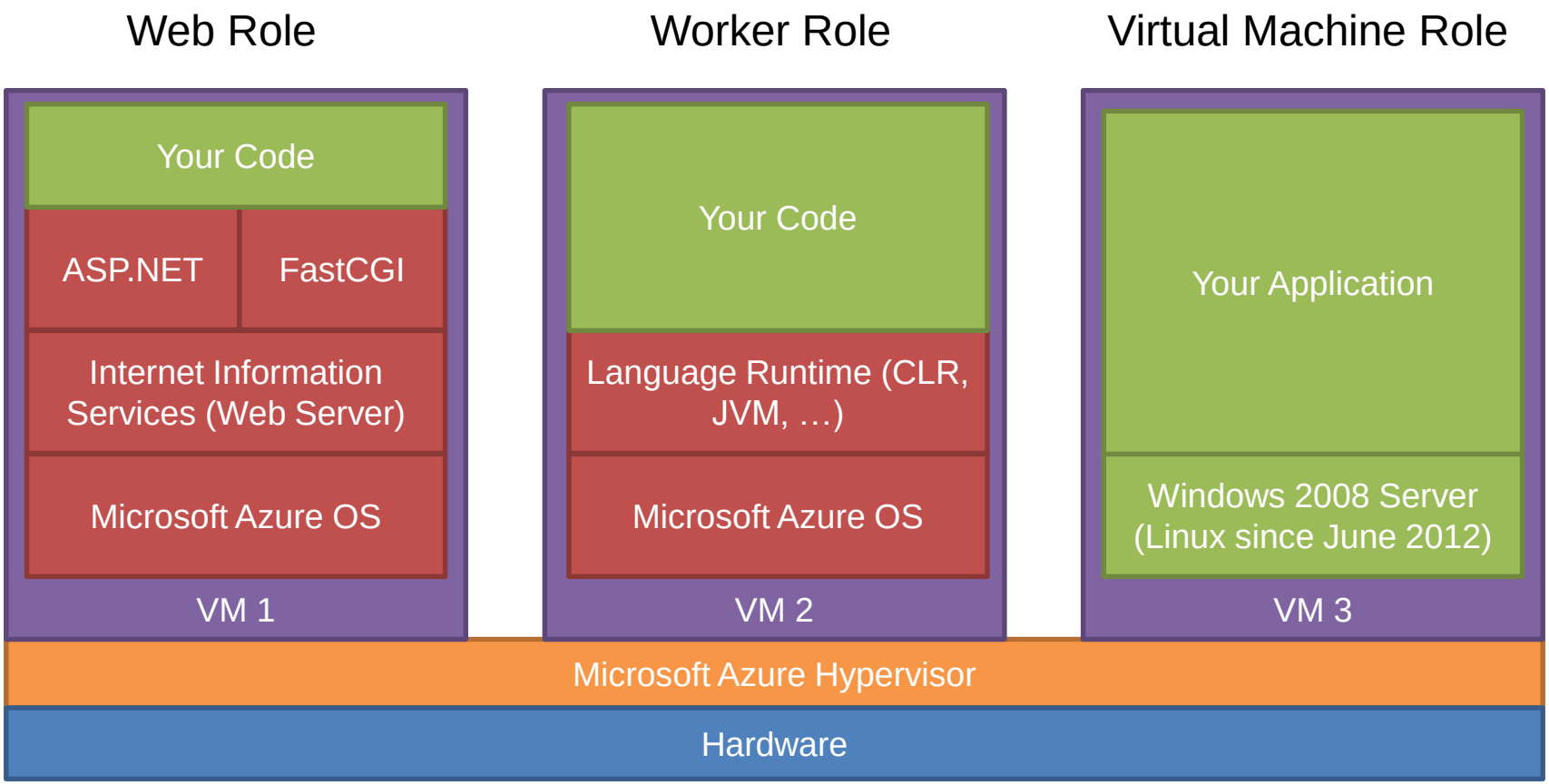
Why Are There Different Roles (1/2)?

- Different roles offer different levels of abstraction
- Some components are the same across all roles
  - Hardware
    - Commodity servers (physical specs are not disclosed)
    - MS tries to keep it as homogeneous as possible
  - Microsoft Azure Hypervisor
    - Few details known, but it is not Hyper-V
    - Relies on homogeneous HW to improve performance
    - Support for 2nd gen. hardware virtualization (NPT, EPT)



Why Are There Different Roles (2/2)?

- Different roles offer different levels of abstraction



Supported Programming Languages (1/2)

- Azure is basically a standard Windows environment
  - Every programming language for Windows Server also runs on Microsoft Azure
  - However, there are differences in the levels of integration
- Languages supported in Web Role
  - .NET
  - Node.js (through IIS extension)
  - Every language supporting the FastCGI interface
    - PHP, Ruby, …

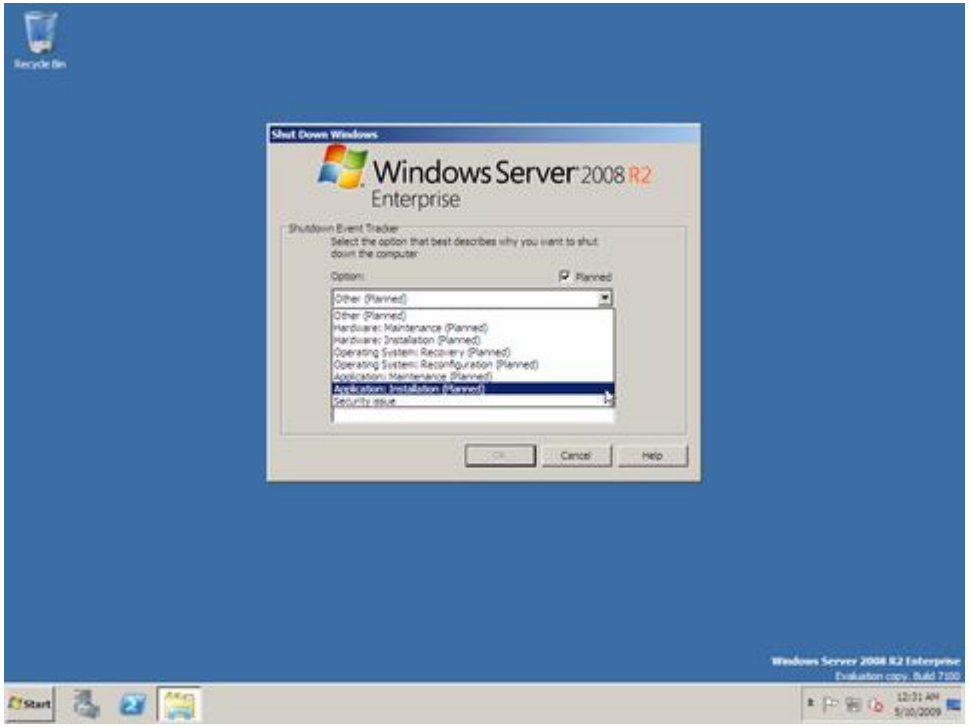
Supported Programming Languages (2/2)

- Languages supported in Worker Role
  - .NET
  - Possibility to run arbitrary Windows binaries
    - C++, Java, …
    - Java support for Worker Role
  - Also possible to bundle additional software with code
    - Example: Create Worker Role with Java code
    - Bundle Tomcat application server
    - → Java application server on Azure with similar behaviour to standard Web Role

Entry Points into Microsoft Azure Components (1/3)

- VM Role entry point: Precompiled VM image
  - Terminal server access

- Azure is agnostic of guest OS
  - No automatic patches or updates
  - Basically IaaS abstraction
  - Intended for legacy application only



Entry Points into Microsoft Azure Components (2/3)

- Worker Role entry point: Archive with intermediate code
  - Code must implement particular interface

```
namespace WorkerRole1
{
    public class WorkerRole : RoleEntryPoint
    {

        public override void Run()
        {
            // Code to be executed during role's life time
        }

    }
}
```

Entry Points into Microsoft Azure Components (3/3)

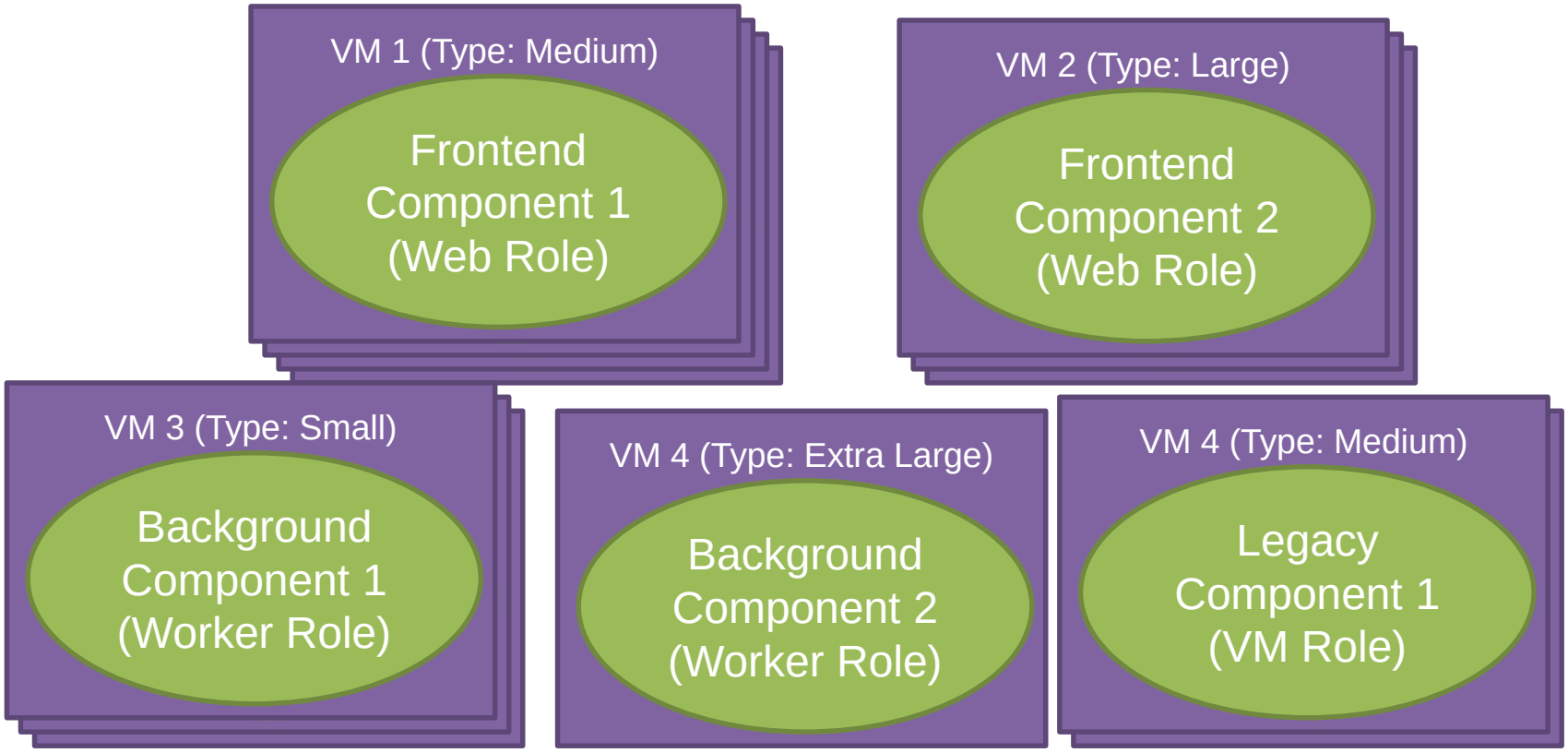
- Web Role entry point: Archive with web code
  - Either ASP.NET code or code compliant with FastCGI

```
@{
    // Some .NET code to be evaluated when page is rendered
    ViewBag.Title = "About Us";
}

<h2>Your heading</h2>
<p>
     The content of your website.
</p>
```

How To Achieve Scalability (1/2)?

- Azure‘s answer: Run many instances of each role



How To Achieve Scalability (2/2)?

- Load balancers distribute requests between instances
  - For Web Roles: HTTP load balancer for requests
    - Round-robin distribution
  - For Worker Roles: Dependant on communication scheme
- Prerequisite: Instances of roles must be stateless
- Questions:
  - Microsoft Azure does support stateful applications
  - Where does the state go?

Storing State in Microsoft Azure

- Azure offers distinct services to store state
  - Microsoft Azure Storage
    - Table
    - Blob
    - Queue
    - Drives
  - SQL Azure
- Idea: Storing state reliably is complex
  - Most customers don‘t care how it is done
  - Offer different persistent storage facilities as a service
  - Use standard interfaces to services (SOAP/REST)

Microsoft Azure Table Storage (1/2)[8]

- Abstraction similar to Excel sheet
  - Table: Set of entities
  - Entity: Basic data item, composed of properties (corresponds to a row in the sheet)
  - Property: Part of an entity (corresponds to a column in the sheet)
- Theoretical maximum size of a table: 100 TB
  - Maximum size of entity: 1 MB
  - Tables are partitioned horizontally (by special part. key)
  - Reads are load balanced across three replicas

Microsoft Azure Table Storage (2/2)

- Consistency:
  - Strong consistency
    - Writes go to a master replica
    - Further reads are only allowed if all replicas updated
  - Limited support for transactions

- Pricing:
  - Cost depend on total space occupied on storage service
    - No separation between table, blob, or queue
  - Microsoft charges per GB per month
  - Prices range from 0.14 to 0.085 USD

Three Rules of the Azure Programming Model[10]

1. A Microsoft Azure application is built from one or more roles.
   - Decoupled application from a logical perspective
   - Assign role depending on component's characteristics
2. A Microsoft Azure application runs multiple instances of each role
   - Key to scalability and availability
   - Allows Microsoft to silently update and restart instances
3. A Microsoft Azure application behaves correctly when any role instance fails
   - Instances are expected to be stateless

Microsoft Azure SLA

- Microsoft Azure SLA similar to SLA of Amazon EC2
  - Only covers availability of service (annual uptime)
  - Hardly any hard performance numbers in SLAs

- Different services have different SLAs
- Violation of SLAs also result in discount on monthly bill

$$\frac{\text{Maximum Connectivity Minutes} - \text{Connectivity Downtime}}{\text{Maximum Connectivity Minutes}} = \text{Monthly Connectivity Uptime Percentage}$$

Monthly Uptime Percentage	Service Credit
<99.95%	10%
<99%	25%

Example of Microsoft Azure SLA from http://www.microsoft.com/en-us/download/details.aspx?id=24434

Other Azure Services

- Microsoft Azure Compute
  - Web Role, Worker Role, VM Role
- Microsoft Azure Storage
  - Table, Queue, Blob, Drive
- SQL Azure
- Content Delivery Network
- Azure AppFabric
  - Access Control, Caching, Service Bus
- Azure Market Place
- Azure Virtual Network

Overview

- Motivation
- Fundamentals for scalable/available applications
- **Case studies**
  - Amazon Dynamo
  - Microsoft Azure
  - **Google App Engine**

Google App Engine

- Google‘s Platform-as-a-Service solution
  - Introduced in 2008

- **Value proposition to customers**
  - Write *scalable web applications* hosted on Google‘s infrastructure
  - Flexible payment model
    - Billing is primarily based on application‘s load
    - Time only influences usage fees indirectly

Basic Principles of Google App Engine

- **Primary goal**: Provide good response times to web clients
- Google's solution:
  - Provide (Java, Python, PHP, Go) APIs to write and deploy web applications
  - Automatic scale-out when number of requests increase
  - Automatic scale-down when request rate goes down
    - Google might evict applications from main memory to regain resources
    - Web Applications are automatically reloaded when new requests arrive
- How does this work?
  - Applications must not maintain internal state
  - Store any data in cloud storage services

Google App Engine Storage Services

- Overview of AppEngine Storage Services:
  - Datastore (scalable, schemaless key-value store)
  - Cloud SQL (managed MySQL)
  - Blobstore (large binary data objects)
  - Log Service
  - Memcache

App Engine Datastore

- Reliable data store for key-value structured data
- Builds upon two components fundamental to the Google infrastructure
  - Google File System (GFS)
    - Distributed, scalable, fault-tolerant
    - Makes use of local hard disks in thousands of servers
  - Google Bigtable
    - Flexible, distributed storage for structured data
    - Builds upon GFS
  - Both components inspired many open source projects

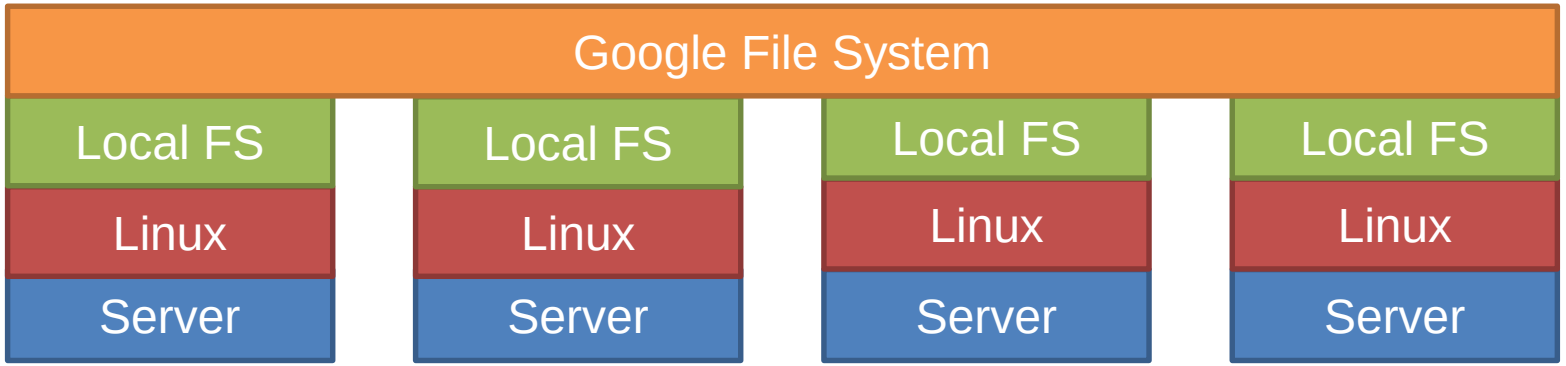
Google File System Design Principles (1/2)[12]

- GFS was designed to meet the following criteria
  - Scalability: Must store hundreds of terabytes
  - High performance: High data throughput over low latency
  - Support for commodity hardware (1000s of servers)
  - Fault-tolerance: Compensate individual server failures

- Google‘s workload characteristics
  - Individual files are expected to be big (MBs to GBs)
  - Billions of files must be managed by the file system
  - Most files are written only once, then read sequentially

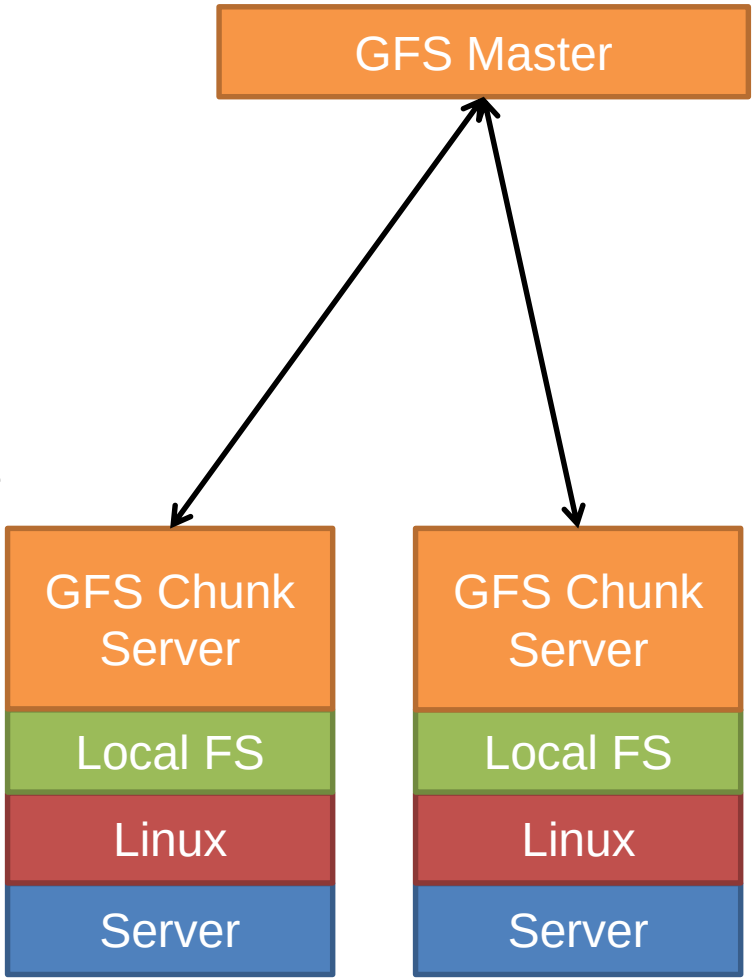
Google File System Design Principles (2/2)

- GFS is not a POSIX-compliant file system
  - Limited support for random writes (expected to happen rarely), data can be efficiently appended instead
  - Sits on top of regular file systems
  - Custom API to the developer



Google File System Architecture

- GFS follows master-worker architecture
  - Master stores metadata
  - Workers store actual data
    - Called chunk server
    - One chunk server per machine
- Files are split into fixed-sized chunks
  - Identified by GUIDs
  - Replicated across machines

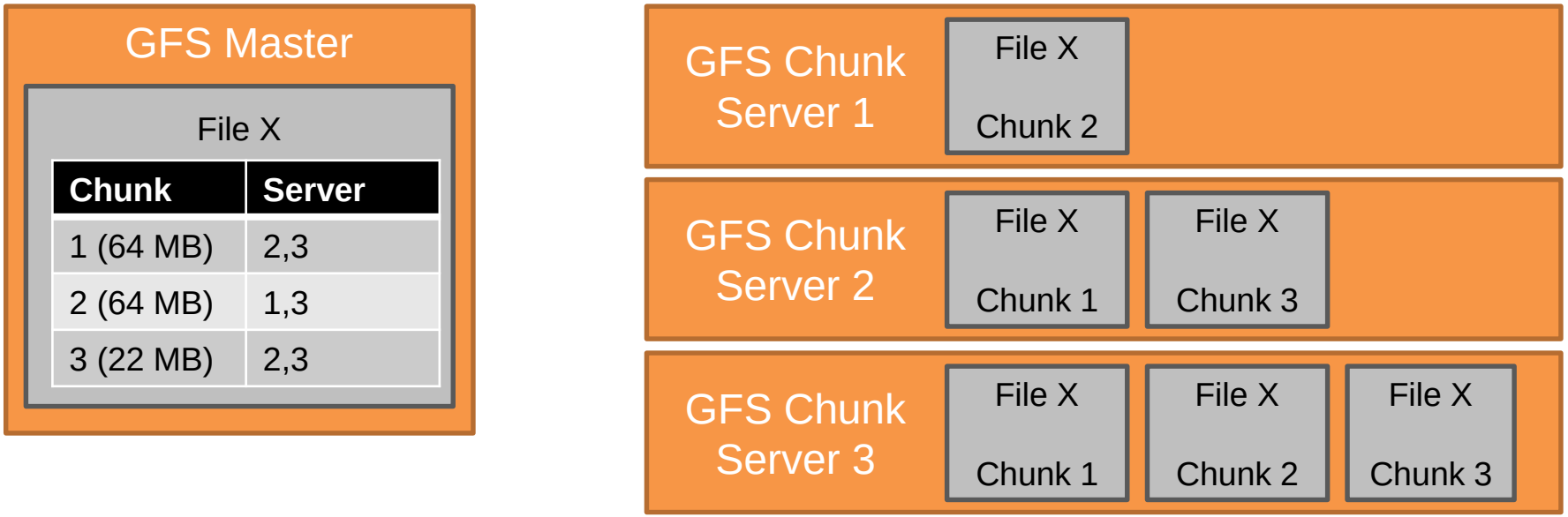


Example of Data Storage in Google File System

- Lets assume, we have file X with a size of 150 MB
  - Fixed chunk size is 64 MB

→File is separated in three chunks

→Chunks are stored across the available chunk servers



Reading File X From GFS

- Lets assume client wants to read X starting at MB 100
  1. Client knows chunk size, converts offset to chunk index
  2. Client contacts master with filename and chunk index
  3. Master returns chunk handle and list of replicas
  4. Client sends read request to closest replica
  5. Client caches chunk handle so further reads of the same chunk require no more client-master interaction
     - until cached information expires
     - file X is reopened

- Note: Master not involved in actual data transfer!

Design Considerations of the GFS Master

- Master has global knowledge of the file system
  - Simplifies design and response times
  - Easy garbage collection and data reorganization

- Meta data is kept in main memory for performance
  - Meta data for each chunk consumes 64 bytes

  → Chunk size is an important parameter of GFS

    → Determines the amount of meta data that fits into memory

    → Determines the frequency of the client requests

What Happens When the Master Fails?

- Meta data is crucial to the functionality of the FS
- GFS knows three types of meta data
  - Namespace mappings (files, directories)
  - File-to-chunk mappings
  - Replica locations

- Namespace/file-to-chunk mappings are written to log
  - Operation log is persistent on master‘s hard disk
  - Replicated to remote machine
- Replica locations are requested from chunk servers when the master starts or new chunk server joins

Operation Log

- Serves as logical timeline that defines the order of concurrent operations
  - Files and chunks (as well as their version) are all uniquely and eternally identified by their logical times at which they were created
  - Changes to the file system are only made visible to clients after the log has been flushed locally and remotely
- Master can recover FS state by replaying log
- When log grows beyond certain size the master creates snapshot of meta data to improve startup times

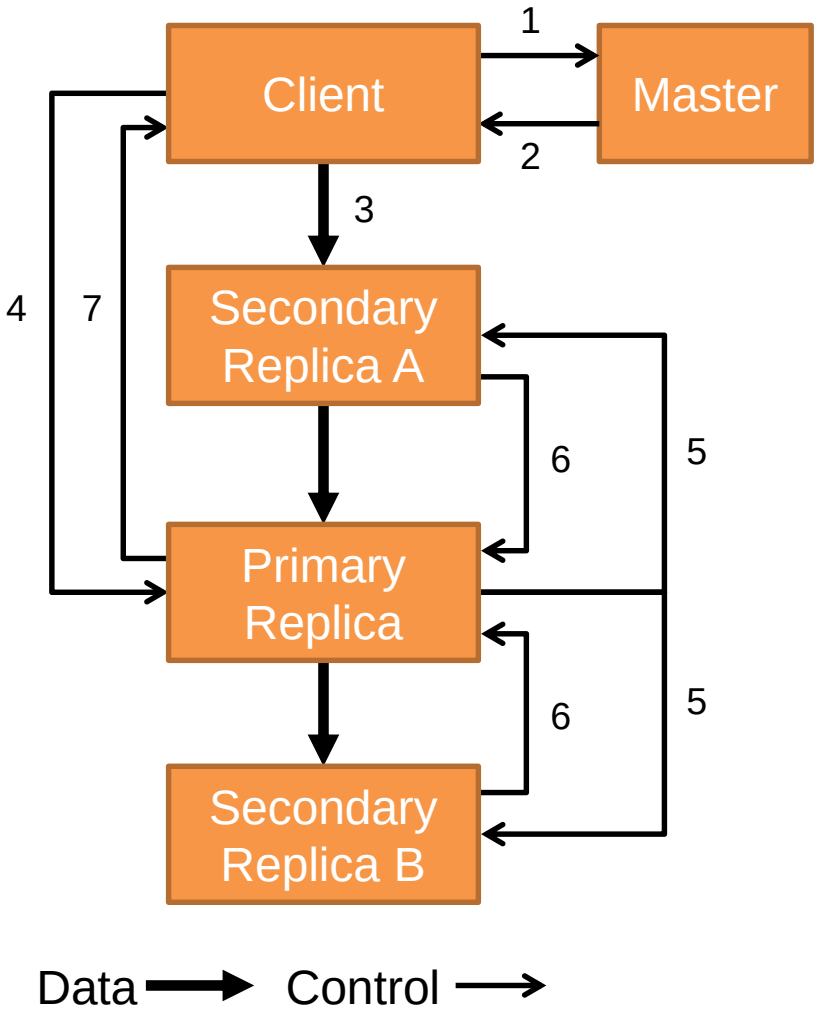
GFS Leases and Mutation Order

- A mutation (write/append) is performed at all the chunk‘s replicas
- Master grants chunk lease to one replica to ensure consistent mutation order across all other replicas
  - Replica with lease is called primary
- Primary chooses serial order for mutations
  - All other replicas follow this order

→Global mutation order is defined by

1. Lease grant order chosen by the master
2. Serial order chosen by primary within the lease

GFS Control/Data Flow for Writes/Appends

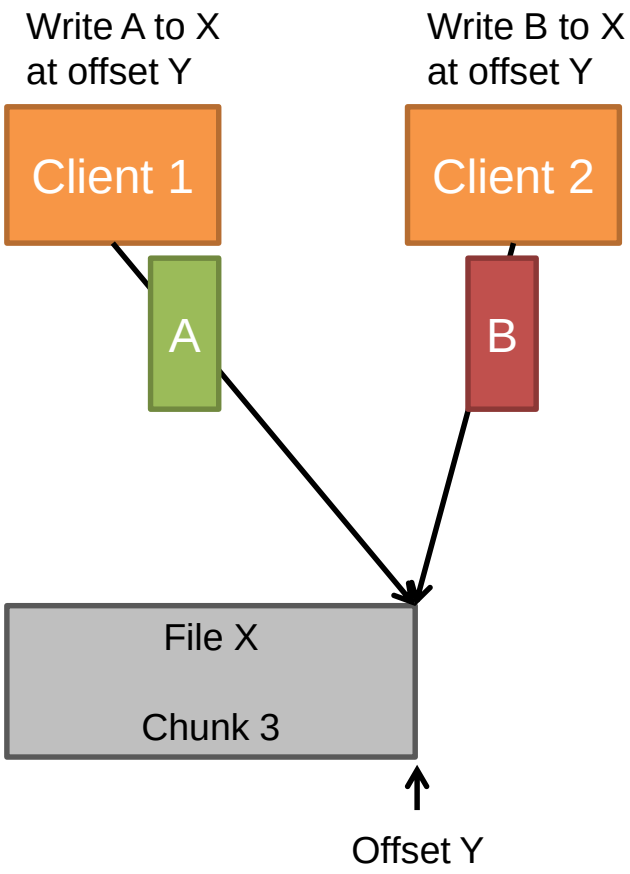


1. Request primary, list of replicas from master
2. Master‘s response
3. Data transfer to arbitrary replica server
4. Mutation request to primary, primary determines order
5. Replicas apply mutations in primary‘s serial order
6. Ack. to primary
7. Ack. to client

GFS Support for Atomic Appends (1/2)

- Technically, append is writing to a file at specific offset
  - Offset traditionally specified by client
  - Challenging in presence of concurrency
    - Example: Two application concurrently specify to write record at offset Y

    →One record would be destroyed
  - Google‘s application depend on atomic append, no loss of data



GFS Support for Atomic Appends (2/2)

- Special GFS operation to append data in atomic units
  - Operation guarantees that data is appended *at least once*
  - Append procedure
    1. Client specifies append operation and data (but no offset!)
    2. Primary serializes requests, chooses offsets
    3. Returns offsets to clients after data has been appended
  - Data appended to next chunk if size exceeded otherwise
    → Rest of previous chunk filled with padding bits!
  - Client retries operation if append fails at any replica
    → Appended data may occur more than once in some replicas
    → Replicas are not guaranteed to be bitwise identical!

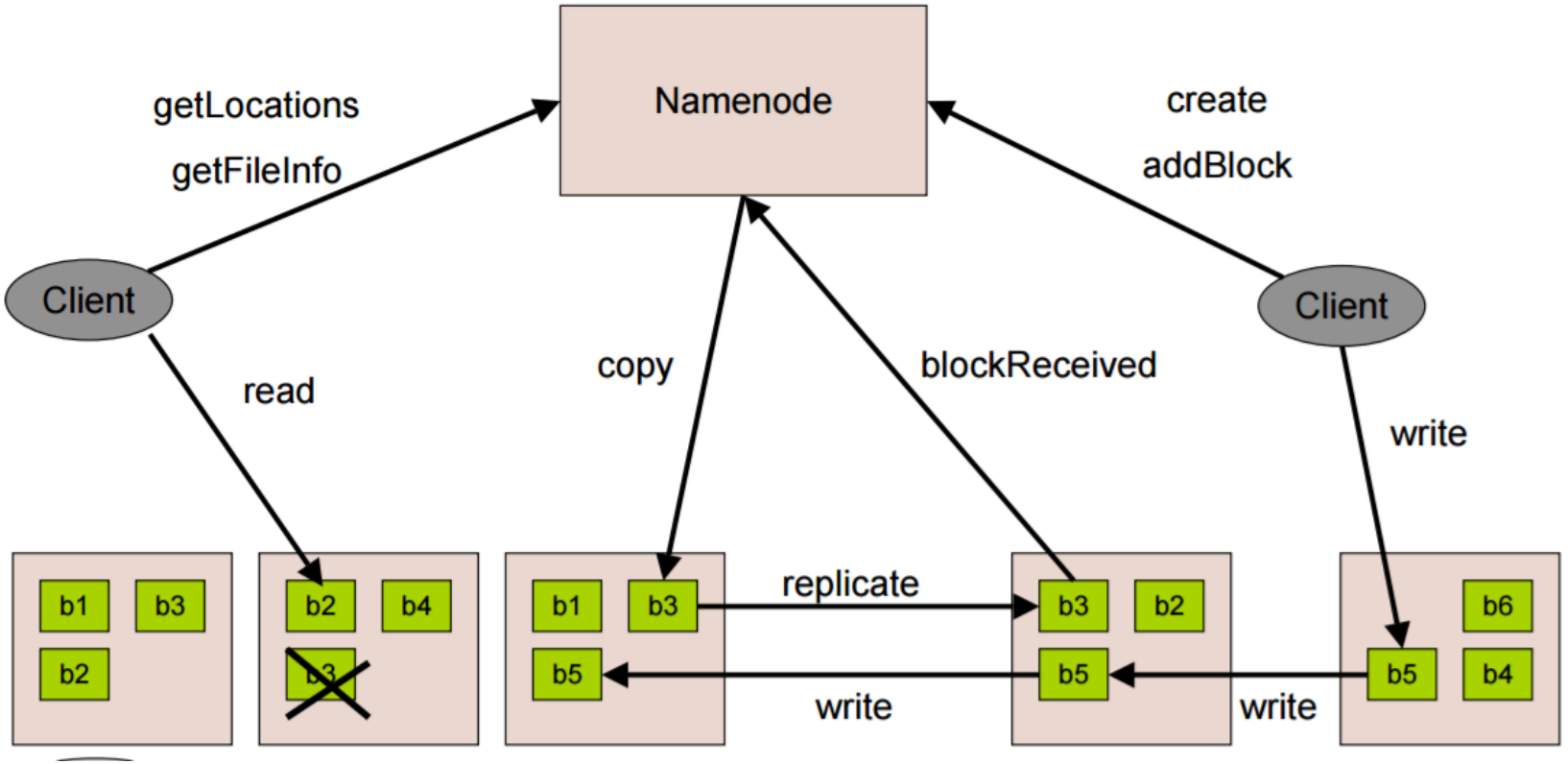
Hadoop Distributed File System (HDFS)

- A distributed replicated filesystem: All data is replicated on three different Data Nodes, inspired by the GFS
- Each node is Linux compute node with 4-12 hard disks
- NameNode maintains the file locations, many DataNodes (1000+)
- Any piece of data is available if at least one data node replica is up and running
- Rack optimized: one replica written locally, second in same rack, third replica in another rack
- Uses large block size, 128 MB is a common default designed for batch processing

Hadoop Distributed File System (HDFS)

- For scalability: Write once, read many filesystem
- All applications need to be re-engineered to only do sequential writes
- Example systems working on top of HDFS
  - HBase (Hadoop Database), a database system with only sequential writes, Google Bigtable clone
  - MapReduce batch processing system
  - Apache Pig and Hive data mining tools
  - Mahout machine learning libraries

HDFS architecture



http://assets.en.oreilly.com/1/event/12/HDFS%20Under%20the%20Hood%20Presentation%201.pdf

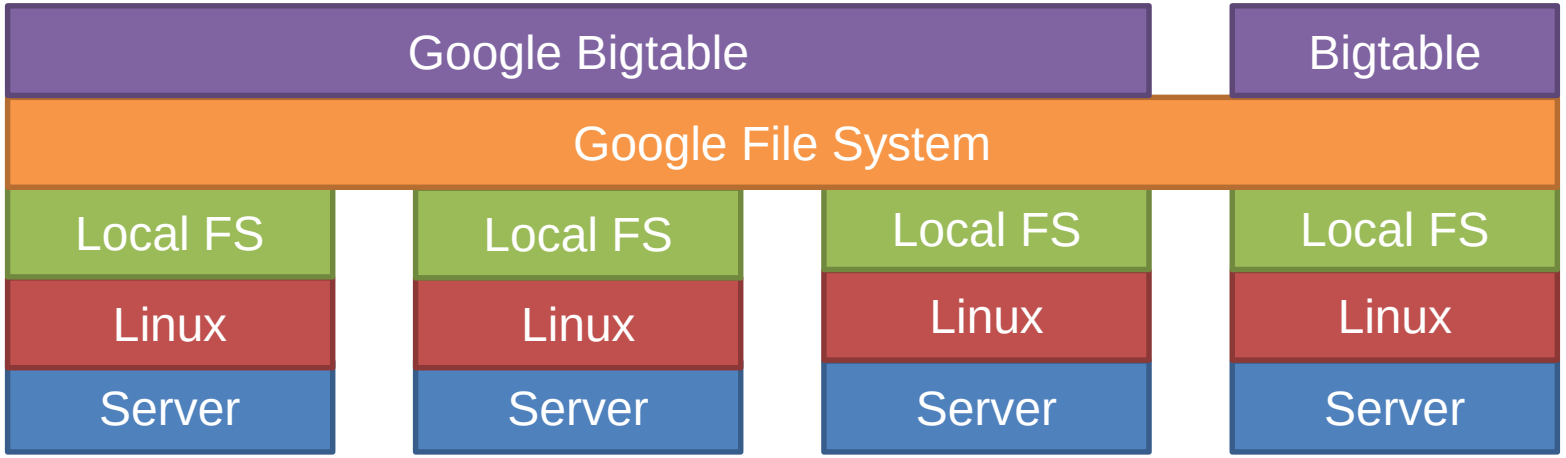
Hadoop Distributed File System (HDFS)

- NameNode is a single computer that maintains the namespace (meta-data) of the filesystem
  - Keeps all meta-data in memory, writes logs, and does periodic snapshots to the disk

- Rules
  - Replica writes are done in a daisy chained (pipelined) fashion to maximize network utilization
  - Reads from the nearest replica
  - Replication and block placement: file's replication factor can be changed dynamically (default 3)
  - Block placement is rack aware
  - Block under-replication & over-replication is detected by Namenode – triggers a copy or delete operation

Google Bigtable[13]

- GFS offers tremendous storage capacity, but limited support to efficiently retrieve structured data
- Solution: Google Bigtable
  - Abstraction: Multi-dimensional sorted map
  - Builds on top of the Google File System
  - Many use cases, backend for App Engine Datastore
  - Open source clone: Apache HBase

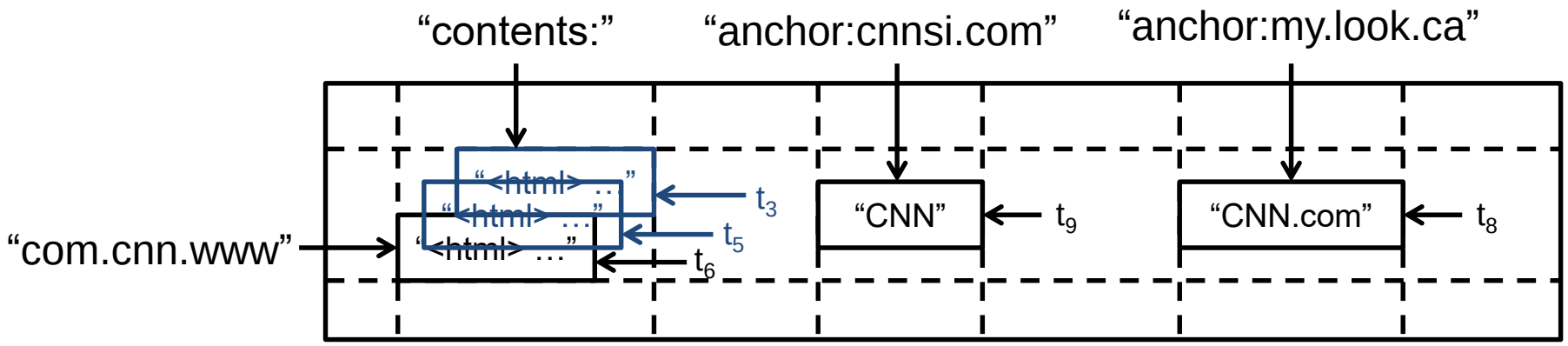


Google Bigtable

- Sequential writes are much faster than random writes
- Writes optimized over reads
  - Data is stored sorted by the row key, and the data is automatically sharded (split to different servers) in large blocks sorted by the row key, allowing for efficient scans in increasing alphabetical row key order
  - Columns are grouped in “Column families”, and all columns in a single column family are stored together in compressed form on disk
  - Some queries might not access columns in all column families => column families can be easily skipped
  - Timestamp field keeps track of versions, e.g., Website content changes over time

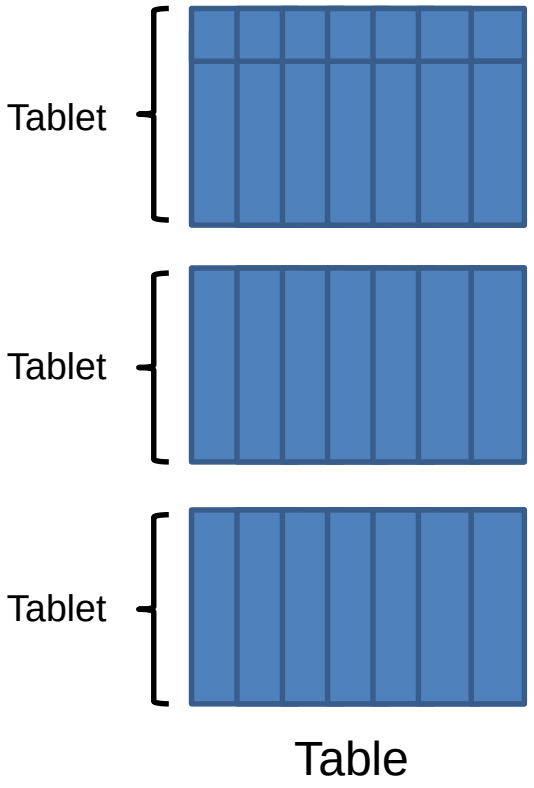
Bigtable Data Model

- (row:string, column:string, time:int64) → string
- No schema, each row can have arbitrary columns
- Columns are grouped to column families
  - Row key = reverse of URL to group pages from the same site
  - “Contents” column family stores Web site contents (certain version)
  - “Anchor” column family stores links pointing to the Web page
  - Easy to skip for example reading the Web page contents if we are only interested in links between Web pages



Bigtable Terminology

- Bigtable clusters stores several *tables*
- Rows with consecutive row keys are grouped together to “Tablets” = unit of distribution and load balancing
- A table consists of a set of *tablets*
  - Tablet contains all data associated with a row range
  - Initially, each table has one tablet
  - When tablet grows, it is automatically split into several tablets
    - Avg. tablet size is 100-200 MB



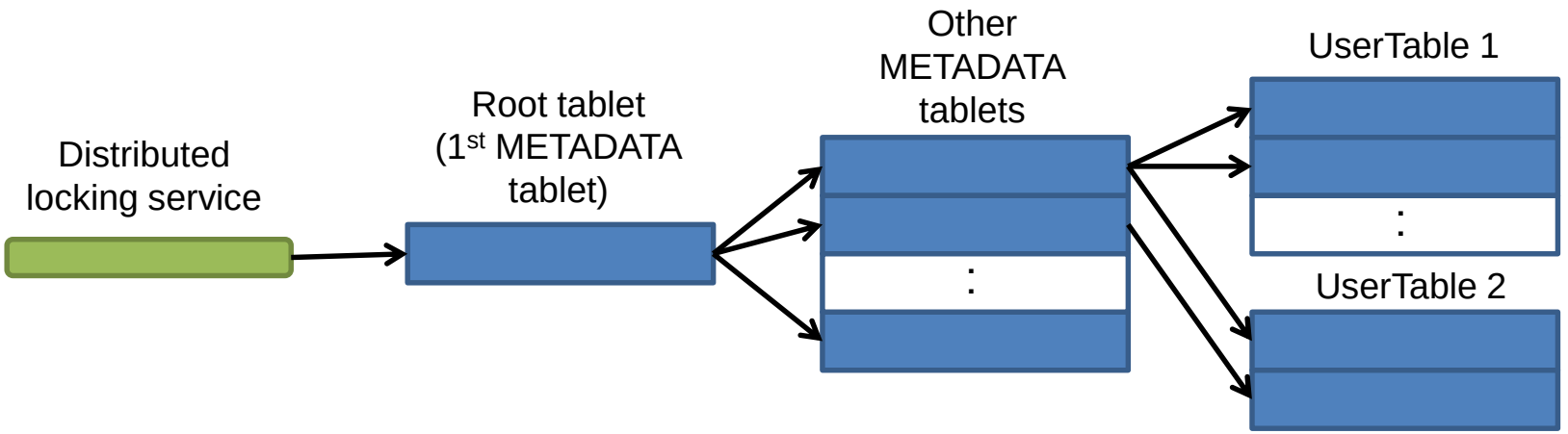
Bigtable Architecture

- Client
  - Sending requests (row, column, time), expecting value
- Master
  - Can manage several tables
  - Assigning tablets to tablet servers
  - Keeps track of addition/expiration of tablet servers
  - Garbage collection, load balancing
- Tablet servers
  - Store the actual tablets
  - Serve client requests
    - Clients rarely communicate with the master

Locating Tablets

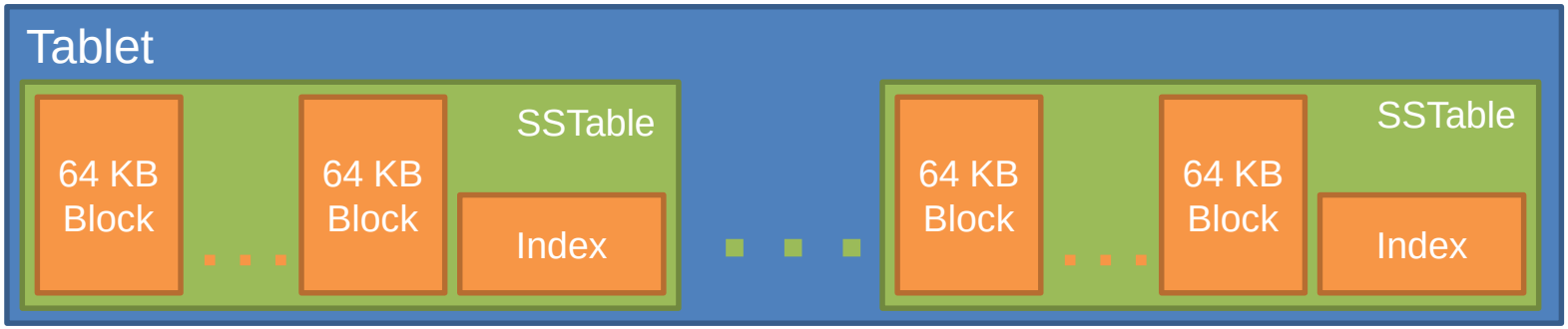
- Tree-level hierarchy to locate correct tablet
  - Client specifies table and row key
  - First lookup in root tablet (stored in dist. locking service)
- Information is stored in special METADATA tables
  - Each row in METADATA table consumes 1 KB

→2^{34} tablets addressable (assuming 128 MB tablet size)



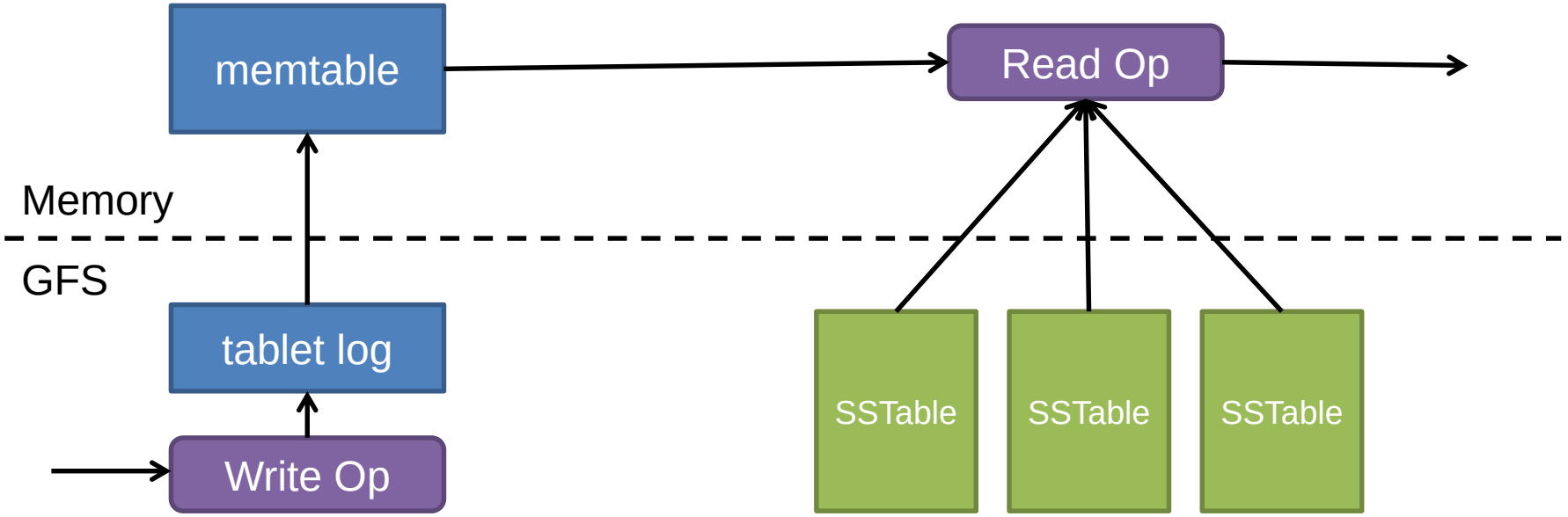
Internal Tablet Structure

- Actual data is stored in a structure called SSTable (Sorted String Table)
  - Persistent, ordered, immutable map
  - Each SSTable is file in GFS
- Each SSTable consists of several blocks and index
  - Each block has 64 KB size
  - Index is loaded in main memory when SSTable is loaded
    - Blocks can be loaded with a single disk seek



Serving Tablets

- Updates are stored to a commit log in GFS
  - When committed they are stored in a *memtable*
  - Older updates are stored in a sequence of SSTables
- Reads are served by merging memtable and SSTables



Google App Engine Production APIs

- Blobstore API (Python, Java, Go)
- Capabilities API (Python, Java, Go)
- Channel API (Python, Java, Go)
- Datastore API (Python, Java, Go)
- Datastore Async API (Python, Java)
- Images API (Python, Java)
- Log Service API (Python)
- Mail API (Python, Java, Go)
- Memcache API (Python, Java, Go)
- …

Google App Engine SLA

- Google App Engine also bases SLA on availability
  - Concrete meaning of “downtime” depends on service
  - In general “downtime” requires a certain percentage of requests to result in an error
  - Minimum downtime period is five minutes
    - Problems that remain less than that are not counted

Monthly uptime percentage	Percentage of discount on bill
99.00% – < 99.95%	10%
95.00% – < 99.00%	25%
< 95.00%	50%

Taken from https://developers.google.com/appengine/sla

References (1/2)

[1] P. Mell, T. Grance: “The NIST Definition of Cloud Computing”, Technical Report, National Institute of Standards and Technology, 2011, http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[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

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels: “Dynamo: Amazon's Highly Available Key-Value Store”, in Proc. of the 21st ACM SIGOPS symposium on operating systems principles, 2007

[5] W. Vogels: “Amazon DynamoDB – a Fast and Scalable NoSQL Database Service Designed for Internet Scale Applications”, 2012, http://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html

[6] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, D. Lewin: “Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web”, in Proc. of the 29th ACM symposium on theory of computing, 1997

[7] I. Fried: “Inside one of the world's largest data centers”, CNET, 2009, http://news.cnet.com/8301-13860_3-10371840-56.html

[8] J. Haridas, N. Nilakantan, B. Calder: “Windows Azure Table - Programming Table Storage”, 2009, http://go.microsoft.com/fwlink/?LinkId=153401&clcid=0x409

[9] B. Calder: “Windows Azure Queue - Programming Queue Storage”, 2008, http://go.microsoft.com/fwlink/?LinkId=153402&clcid=0x409

References (2/2)

[10] D. Chappell: “The Windows Azure Programming Model”, 2010, http://go.microsoft.com/?linkid=9751501&clcid=0x409

[11] http://www.google.com/about/datacenters/locations/index.html

[12] S. Ghemawat, H. Gobioff, S.-T. Leung: “The Google File System“, in Proc. of the 19th ACM symposium on operating systems principles, 2003

[13] F. Chang, J .Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber: “Bigtable: A Distributed Storage System for Structured Data”, ACM Transactions on Computer Systems, 26 (2), 2008

[14] A.S. Tannenbaum, M. Van Steen: “Distributed Systems: Principles and Paradigms”, Prentice Hall, 2006

[15] K. P. Birman: “Guide to Reliable Distributed Systems”, Springer, 2012

[16] S. Krishnan: Programming Windows Azure: Programming the Microsoft Cloud, O'Reilly, 2010