

CLOUD COMPUTING

Elasticity & Energy

Lorenzo Leonini
lorenzo.leonini@unine.ch

Objectives

- Introduce the notion of elasticity, discuss its objectives and its design space
- Discuss horizontal scaling and load-balancing solutions
- Describe an example of an application-agnostic elasticity solution for IaaS
- Discuss an example of application-specific elasticity for the NoSQL database HBase
- Discuss aspects related to energy efficiency
 - Energy proportionality and consolidation
 - Cloud environmental footprint

Introduction

- "Pay-as-you-go" pricing: key constituent of cloud model
- Workload typically evolve in time
 - Day/night patterns
 - Load spikes and idle periods
- Static resource provisioning is non optimal
 - Need to provision for the worst case (highest load)
 - Long idle periods with unused resources
 - Extra costs and energy consumption
 - An idle server still uses a considerable amount of energy
- Elasticity: dynamically adapt the amount of resources to actual needs

Scalability

- Scalability: ability of application or service to handle more requests when scaled up
 - Or handle existing requests with better response times (QoS: quality of service)
- Scaling **up**: adding more resource to support a cloud application of service
- Scaling **down**: removing resources

Vertical / horizontal scaling

- **Vertical** scaling
 - Scale up = add more resources to existing VMs
 - Virtual CPUs, RAM, network bandwidth, ...
 - Scale down = remove some resources
 - Generally fast and transparent to the application
 - Can only scale up to the hardware limits of the physical host
- **Horizontal** scaling
 - Scale up: add a new VM (or a new host)
 - Scale down: shut down a host and remove/migrate a VM
 - Takes time and often requires application support
 - Number of physical servers is the theoretical limit:
Only way to reach massive scales required by cloud services

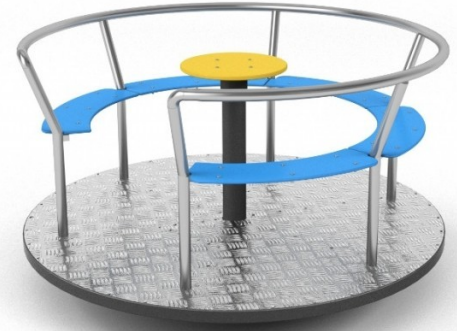
Horizontal scalability

- Ideal = linear scalability
 - N servers, L throughput \rightarrow $2 \times N$ servers, $2 \times L$ throughput
 - Example: object store for immutable data (Amazon Simple Storage Service)
- Scalability also depends on the workload nature
 - ZooKeeper:
 - N servers, 100% read throughput \rightarrow $2 \times N$ servers, $2 \times$ read throughput but ...
 - N servers, 100% write throughput \rightarrow $2 \times N$ servers, reduced write throughput
- Adding or removing servers often requires some reconfiguration and may impact performance for a time
 - Example: adding a server to a NoSQL DB may require data movement between servers to integrate new server and balance the load

Load-balancing

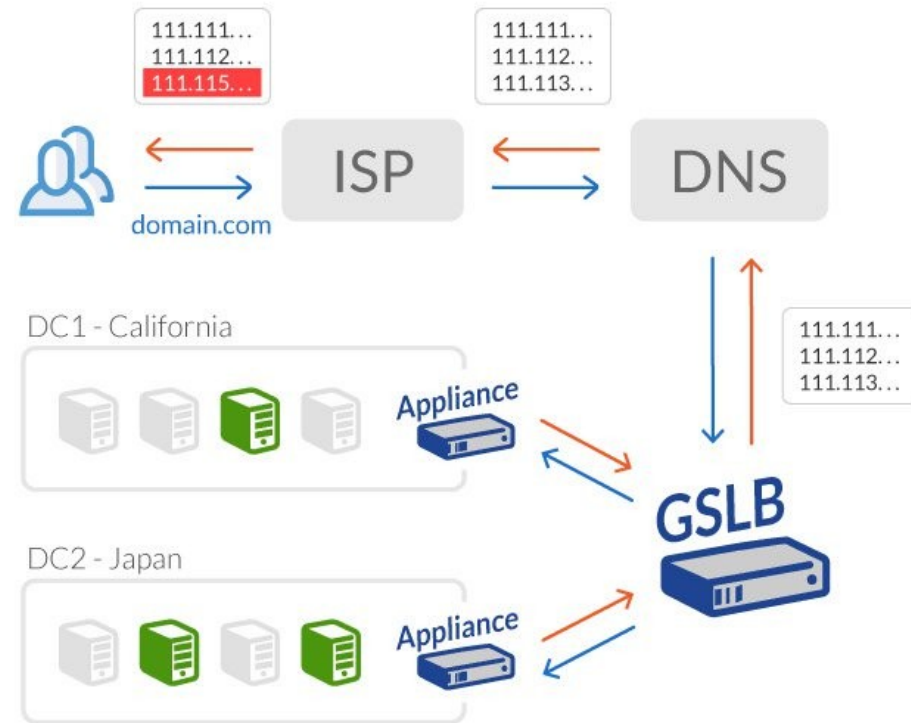
Load-balancing

- Repartition of the load / queries
- Different locations
 - Global (DNS) - Client or server side
 - Datacenter
 - On a single server
- Different layers
 - OSI model: network layers 4 (transport) or 7 (application)
- Different balancing strategies



Round-Robin DNS (1)

- One DNS name → different IPs provided
 - Subset of all IPs
 - Client side: random pick
- Advantages
 - DNS resolution close to the user (ISP)
 - Servers can be geographically dispatched
 - Even on different providers/datacenters
- Disadvantage
 - DNS response are cached by ISP
 - Need to provide low TTL to have frequent refreshes
 - Not load aware / failure detection
 - → Backed by datacenter LB



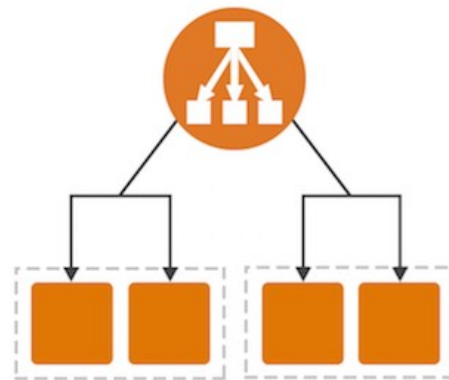
DNS Round-Robin (2)

```
leo@t480s: ~  
~ dig fr.pool.ntp.org  
;  
;<<>> DiG 9.11.4-3ubuntu5-Ubuntu <<>> fr.pool.ntp.org  
;; global options: +cmd  
;; Got answer:  
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 17980  
;; flags: qr rd ra; QUERY: 1, ANSWER: 4, AUTHORITY: 0, ADDITIONAL: 1  
;  
;; OPT PSEUDOSECTION:  
; EDNS: version: 0, flags:; udp: 65494  
;; QUESTION SECTION:  
;fr.pool.ntp.org.                IN      A  
;  
;; ANSWER SECTION:  
fr.pool.ntp.org.                150     IN      A      108.61.177.141  
fr.pool.ntp.org.                150     IN      A      80.74.64.1  
fr.pool.ntp.org.                150     IN      A      151.80.211.9  
fr.pool.ntp.org.                150     IN      A      51.255.141.154  
;  
;; Query time: 29 msec  
;; SERVER: 127.0.0.53#53(127.0.0.53)  
;; WHEN: Sun Nov 11 18:42:53 CET 2018  
;; MSG SIZE rcvd: 108
```

```
leo@t480s: ~  
~ dig fr.pool.ntp.org  
;  
;<<>> DiG 9.11.4-3ubuntu5-Ubuntu <<>> fr.pool.ntp.org  
;; global options: +cmd  
;; Got answer:  
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 38899  
;; flags: qr rd ra; QUERY: 1, ANSWER: 4, AUTHORITY: 0, ADDITIONAL: 1  
;  
;; OPT PSEUDOSECTION:  
; EDNS: version: 0, flags:; udp: 65494  
;; QUESTION SECTION:  
;fr.pool.ntp.org.                IN      A  
;  
;; ANSWER SECTION:  
fr.pool.ntp.org.                0       IN      A      62.210.28.176  
fr.pool.ntp.org.                0       IN      A      178.249.167.0  
fr.pool.ntp.org.                0       IN      A      212.51.181.242  
fr.pool.ntp.org.                0       IN      A      5.39.78.154  
;  
;; Query time: 0 msec  
;; SERVER: 127.0.0.53#53(127.0.0.53)  
;; WHEN: Sun Nov 11 18:48:55 CET 2018  
;; MSG SIZE rcvd: 108
```

Load-balancing - HTTP

- SSL/TLS
 - Sometimes using specific hardware - offload microservices from this task
- Compression
 - gzip - offload microservices from this task
- Authentication management
- Firewall
 - WAF (Web Application Firewall)
 - Detect "suspect" patterns: know CMS attacks, SQL injection, ...
 - DDoS protection
 - Syn cookies
- API Gateway is a load-balancer with additional routing capabilities
 - 404
- Caching



Load-balancing - TCP

- TCP Offload
 - Consolidate incoming HTTP requests inside a permanent pool of HTTP connections with the backend
 - Avoid TCP initialization delay (reduce latency)
 - Limit the total amount of connections to the backend (possibly serializing access)
 - Protect the backend from slow clients
- Health checking
 - Essential component for auto-scaling
 - HTTP configurable
- Rate shaping
 - Priority queue for different traffic

Load-balancing Strategies

- Round-robin
- Random
- Least connections
- Ressource based
 - Least response time
 - Lightweight way to detect backend ability to manage additional requests
 - Health check with load indication
- Sticky (HTTP)
 - "Stick" an user to the same backend server
 - Critical in case of HTTP stateful exchanges
 - HTTP session not shared between backends
 - Hash client IP to select the backend
 - Based on a session cookie
 - Set by backend
 - (or) set by load-balancer



- TCP/HTTP load balancer designed for High Availability and high traffic web sites
 - Support millions of SSL connections on a single server
 - <https://medium.freecodecamp.org/how-we-fine-tuned-haproxy-to-achieve-2-000-000-concurrent-ssl-connections-d017e61a4d27>
- Notables features
 - Single threaded event driven system
 - HTTP/2
 - Seamless configuration reload (hot reload)
 - Hardware SSL support
 - Pluggable architecture



- Static list of backend servers
 - `balance roundrobin`
 - `server lamp1 10.0.0.1:80 check`
 - `server lamp2 10.0.0.2:80 check`
- How elasticity is achieved ?
 - Scaling down
 - Nodes provide health checks, a node not responding is automatically (temporarily) removed
 - `option httpchk HEAD /`
 - Scaling up ?
 - HAProxy provides a Runtime API for dynamic reconfiguration
 - Configuration has first to be stored in a distributed fault-tolerant DB (e.g. etcd, ZooKeeper, ...)
 - Write a node daemon watching for configuration changes and reconfigure the local HAProxy

Load-balancing processes

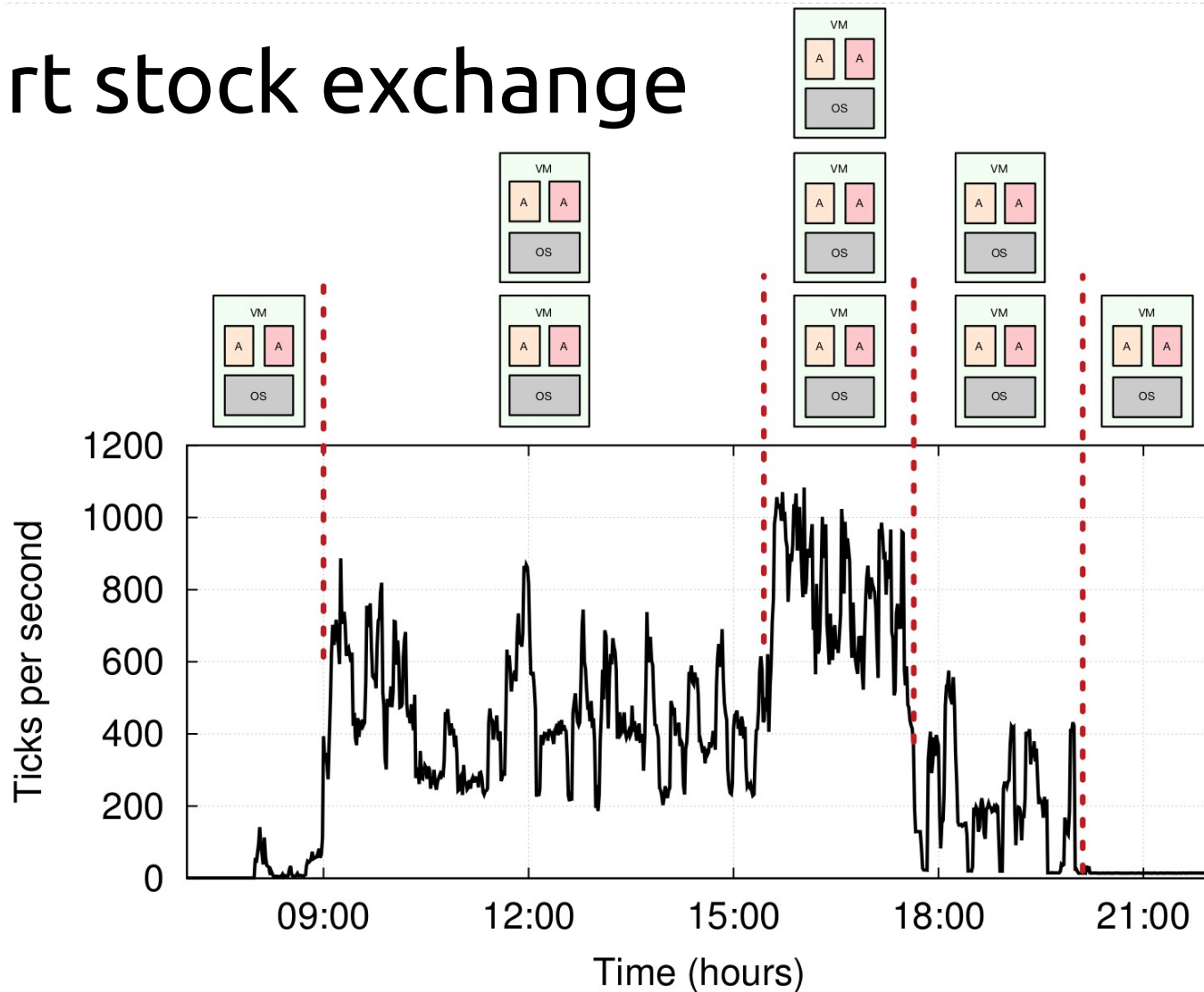
- Some languages (like Ruby and Python) have no support for parallelism
 - Global Interpreter Lock
 - Only one "thread" run at a point in time
 - Internal data structures are not thread safe
 - Some C extensions are not thread safe
 - https://en.wikipedia.org/wiki/Global_interpreter_lock
 - Webservers come to rescue
 - Launching a pool of processes (> number of physical CPUs)
 - Dispatching HTTP requests on them
- Some Ruby Webservers
 - Phusion Passenger: Fast web server & app server
 - Puma: A Modern, Concurrent Web Server for Ruby
 - Thin: Tiny, Fast, & Funny HTTP Server
 - Unicorn: Rack HTTP Server for Fast Clients and Unix

Elasticity principles

Elasticity

- Automated provisioning of cloud resources
- Based on a quality-of-service (QoS) objective
 - SLA: (formal) Service Level Agreement signed between a client and a provider
 - SLO: (informal) Service Level Objective set by the provider
- Visual example of the processing of streaming data from Frankfurt stock exchange
 - Using horizontal scaling (adding/removing VMs)

Frankfurt stock exchange



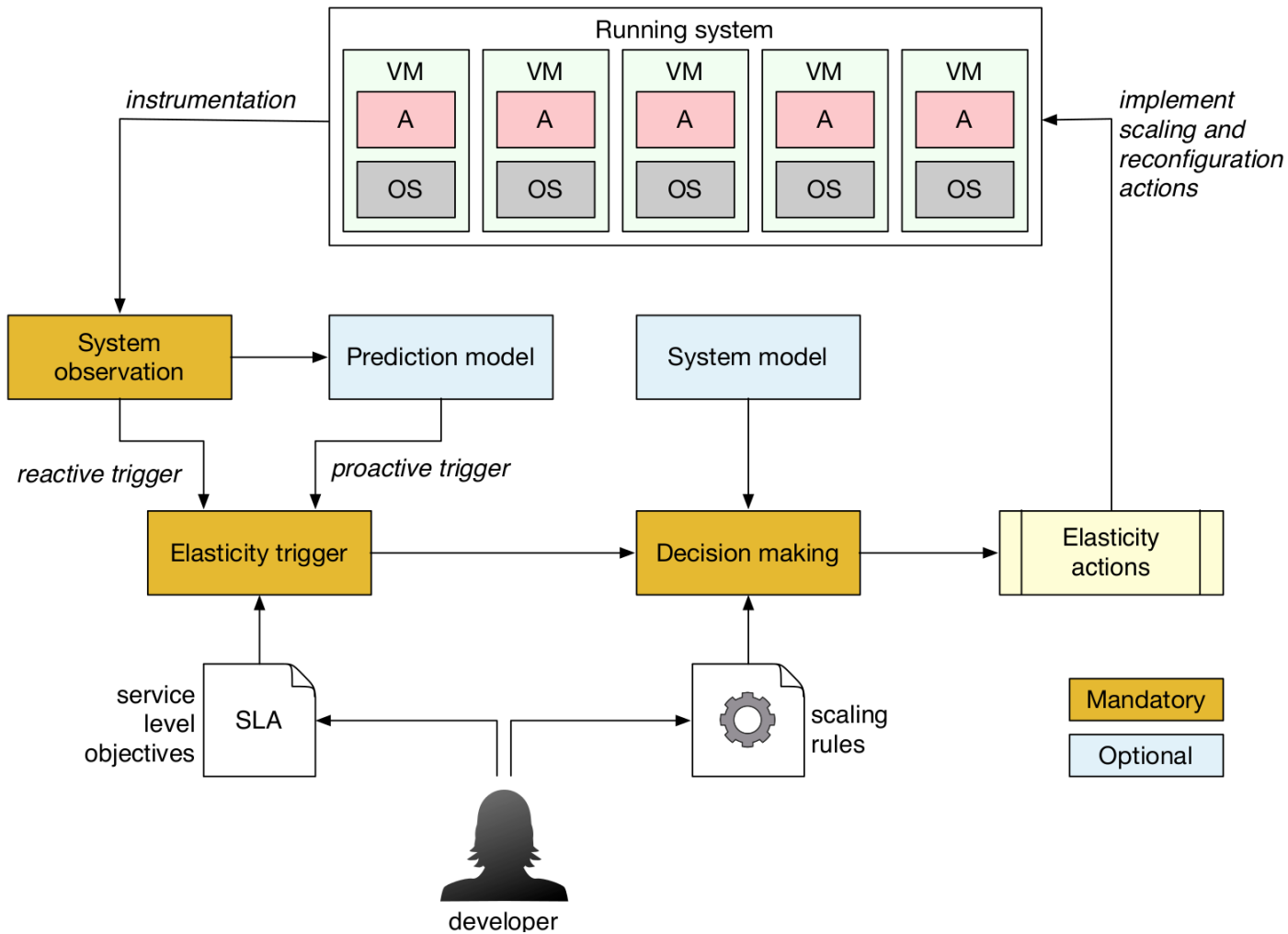
Elasticity enactors

- Who provides elasticity features?
 - Cloud provider
 - Supported by the cloud *as-a-service* offering (typically IaaS)
 - Generally with some hints by the developer
 - Service provider / cloud user
 - Implemented by the application developer herself
 - Typically using the underlying IaaS REST interface to reserve/release resources or update existing resources

Elasticity target

- Application-agnostic
 - No assumption about scalability behavior or model of application or simple (linear) scaling assumptions
 - Scale in/out may still use application-specific actions when scaling, provided by its developer, e.g.,
 - Add the new server to the outgoing links list of a load balancer
 - Remove a worker from the list maintained in ZooKeeper
- Application-specific
 - e.g., Relational / NoSQL DB, storage, multi-tier Web application
 - Knowledge of application scaling behavior
 - Ability to use fine grain information about workload and model the scalability and performance
 - Ability to better configure application in addition to adding and removing resources

Generic elasticity framework



Elasticity triggers

- Decides when to start a scaling action
- Reactive
 - Check measurements and compare with SLA
 - Elasticity trigger generated using threshold-based rules
- Proactive
 - Attempt to predict future load variations using a model
 - Typically based on machine learning techniques
 - Need to train the model beforehand
 - Allocate resources before they are actually needed
 - More complex
- Combined
 - Allow to deal with sudden load spikes but plan in the medium-term

Elasticity action

- When triggered, decide between available elasticity actions
 - Horizontal scaling: adding or removing VMs
 - Most common approach
 - Vertical scaling: changing VMs configuration
 - Combinations of horizontal and vertical
 - VM migrations
 - Application reconfiguration
- The appropriate action to take depends on a *decision making* mechanism

Decision making

- Simple: rules defined by developer
 - “If condition C is met, use action A”
 - Requires precise knowledge of the application
- Advanced: use of an application model
 - Estimate the effect of a scaling action on the performance of the application
 - As a function of the experienced workload
 - Model can be built gradually based on previous conditions and decisions, using machine learning techniques
 - Algorithms for decision making
 - Optimization theory and utility maximization
 - Control theory
 - Combination with rules

Examples of elasticity frameworks

Framework	Provider	Target	Trigger	Elastic actions	Decision making
Amazon Auto Scaling	IaaS provider	Application-agnostic	Reactive	Horizontal scaling	Rule based
AGILE	IaaS provider	App.-specific: Multi-tier Web applications	Proactive	Horizontal scaling (+ discussion on vertical)	Mathematical optimization
MeT	Service or PaaS provider	App.-specific: NoSQL data store HBASE	Reactive	Horizontal scaling and application configuration	Rule based and mathematical optimization

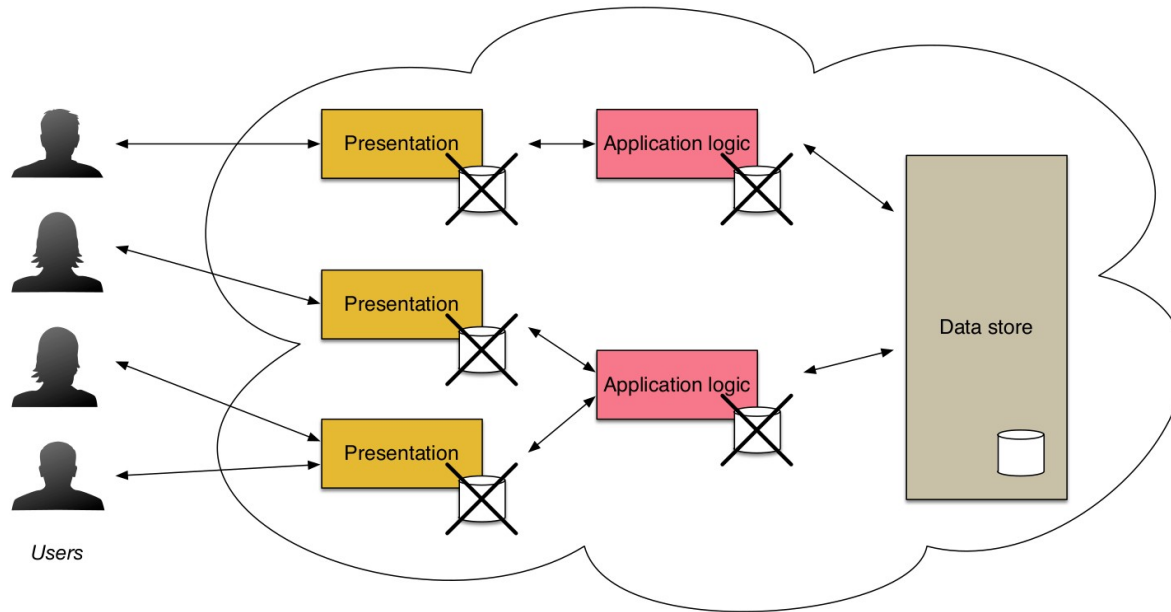
Example 1: Amazon Auto Scaling

Amazon AutoScale

- Facility for applications running on the Amazon EC2 IaaS Cloud, targeting primarily multi-tiered Web applications
- *Auto Scaling groups*: dynamically sized collection of EC2 VMs
- Elastic Load Balancer (ELB) balances incoming requests between EC2 VMs in the group
- Cloud Watch: collects metrics at the ELB and EC2 levels
 - Amount of requests, load of the instances, requests latency
- Scaling policies: user-provided scaling rules
 - Based on thresholds on the metrics monitored by CloudWatch and defined as CloudWatch alarms
 - Auto Scaling cooldown period allows current operation to terminate before next decision is taken

Multi-tier Web applications

- A common and relatively simple elasticity target
- Adapted when all servers can handle any request (no data inside the server)
- Also a good model for microservices



Example of Cloud Watch alarm

Create Alarm



You can use CloudWatch alarms to be notified automatically whenever metric data reaches a level you define.

To edit an alarm, first choose whom to notify and then define when the notification should be sent.

☒ Send a notification to: AddCapacityNotification [cancel](#)

With these recipients: mymail@example.com

Whenever: Average of CPU Utilization

Is: >= 80 Percent

For at least: 1 consecutive period(s) of 5 Minutes

Name of alarm: AddCapacityAlarm

CPU Utilization Percent



[Cancel](#)

[Create Alarm](#)

Example of Auto Scaling scale out rule

Increase Group Size

Name:

Execute policy when: [AddCapacityAlarm](#) [Edit](#) [Remove](#)
breaches the alarm threshold: CPUUtilization ≥ 80 for 300 seconds
for the metric dimensions AutoScalingGroupName = my-asg

Take the action: when \leq CPUUtilization $< +\infty$

[Add step](#) ⓘ

Add instances in increments of at least instance(s)

Instances need: seconds to warm up after each step

[Create a simple scaling policy](#) ⓘ

Example of Auto Scaling scale in rule

Decrease Group Size

Name:

DecreaseCapacity

Execute policy when:

DecreaseCapacityAlarm [Edit](#) [Remove](#)
breaches the alarm threshold: CPUUtilization <= 40 for 300 seconds
for the metric dimensions AutoScalingGroupName = my-asg

Take the action:

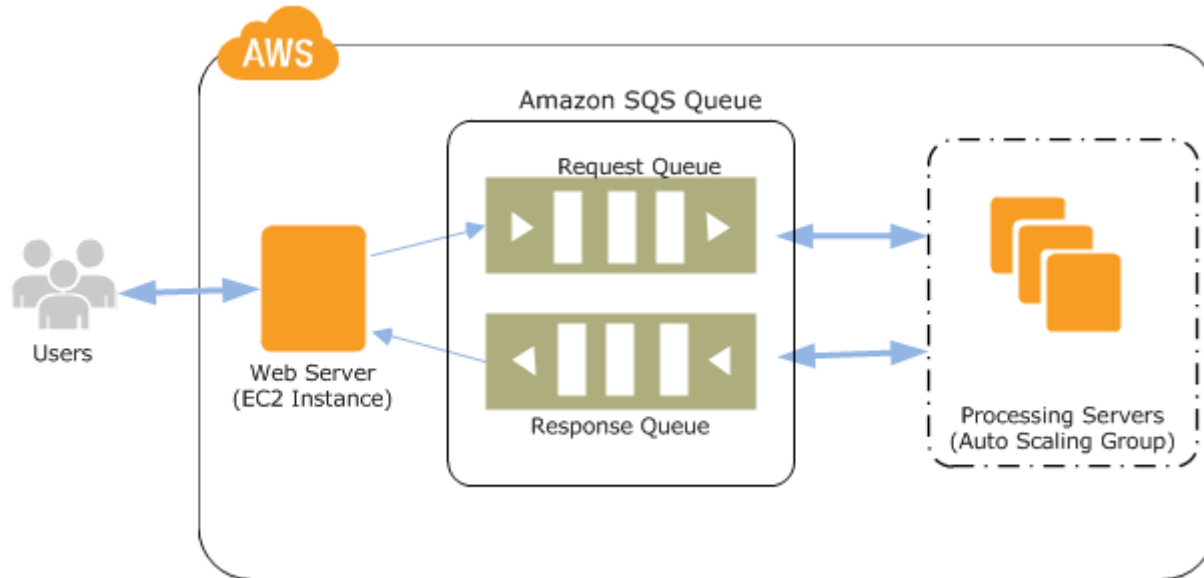
Remove ▾ 2 instances ▾ when 40 >= CPUUtilization > -infinity

[Add step](#) ⓘ

[Create a simple scaling policy](#) ⓘ

Scaling rules based on queues

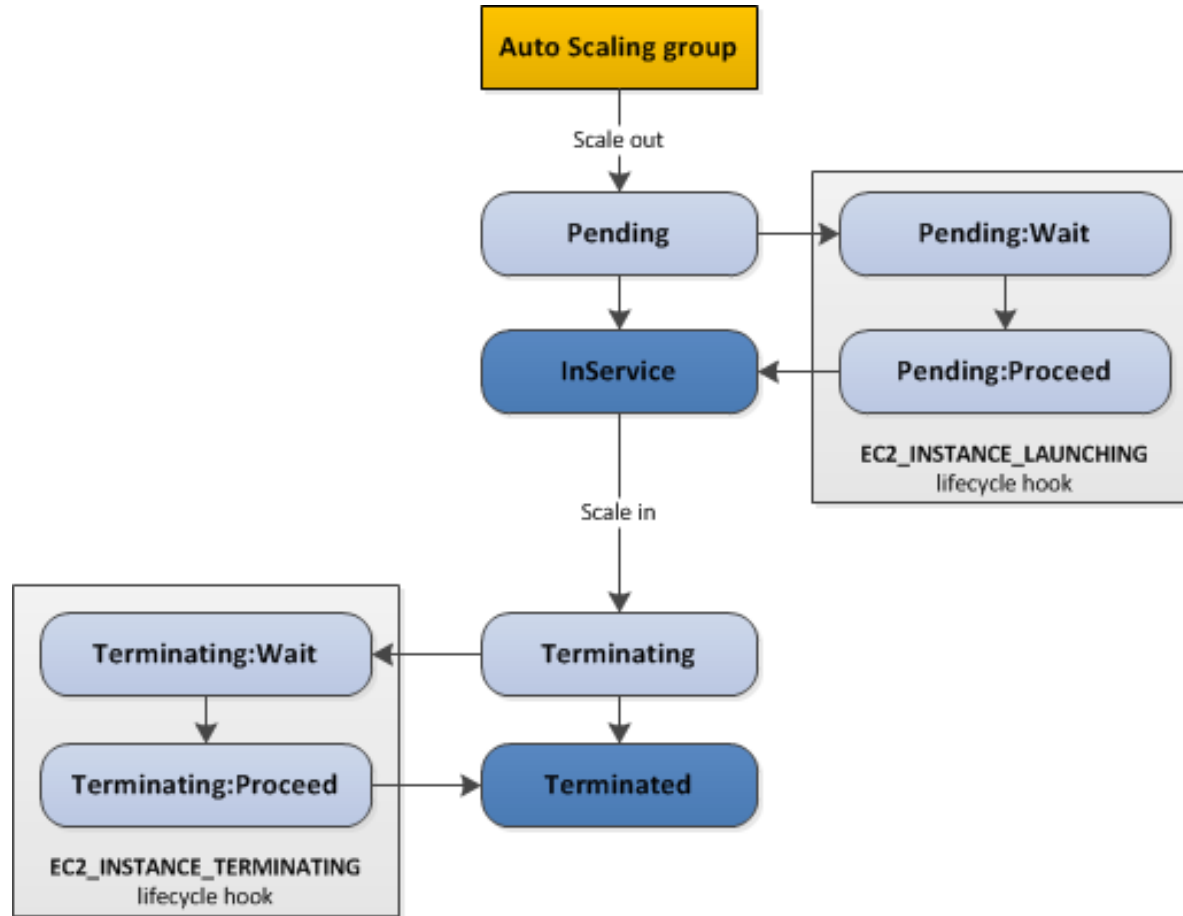
- Amazon Simple Queue Service: specialized coordination service for FIFO queues
 - Size of the queue indicates load pressure
 - Can be used as a metric for scaling in/out



Auto Scaling *Lifecycle* hooks

- In many cases, adding or removing a VM to the application group requires some configuration action
 - Registering the new VM to the previous tier (if not using the provided ELB)
 - Terminating gracefully and notifying other application instances of the shutdown
 - Trigger a new leader election if shut down VM was the leader
 - Many more examples possible
- Lifecycle hooks allow to register to Auto Scaling events and perform specific actions
 - If a hook is registered, the instance is in a wait state until the hook returns
 - Implement configuration actions in the hook

Auto Scaling *Lifecycle* hooks



Amazon Auto Scaling: limitations

- Application performance is implicitly considered to be linearly scaling with the number of VMs
 - No application model possible
 - Adapted to stateless Web application tiers
 - Assumes the storage uses managed and scalable platform services
 - DynamoDB, S3, EBS, etc.
- Only reactive approaches: not possible to plug in a predictive model for a proactive triggering

Integrating vertical scaling

- Amazon Auto Scaling only targets horizontal scaling
 - For short load spikes, vertical scaling can be more appropriate
- VMs on the same physical host may have different needs at a given time
- Temporarily augment a VM with more resources
 - Either by "stealing" from the co-located VM
 - self-healing scaling
 - Or from unallocated resources on the same host
 - often available due to fragmentation
- Can be used as a measure to handle load spikes while waiting for horizontal scaling to happen
- Implementable using EC2 REST API calls for IaaS management

Example 2: application-specific elasticity

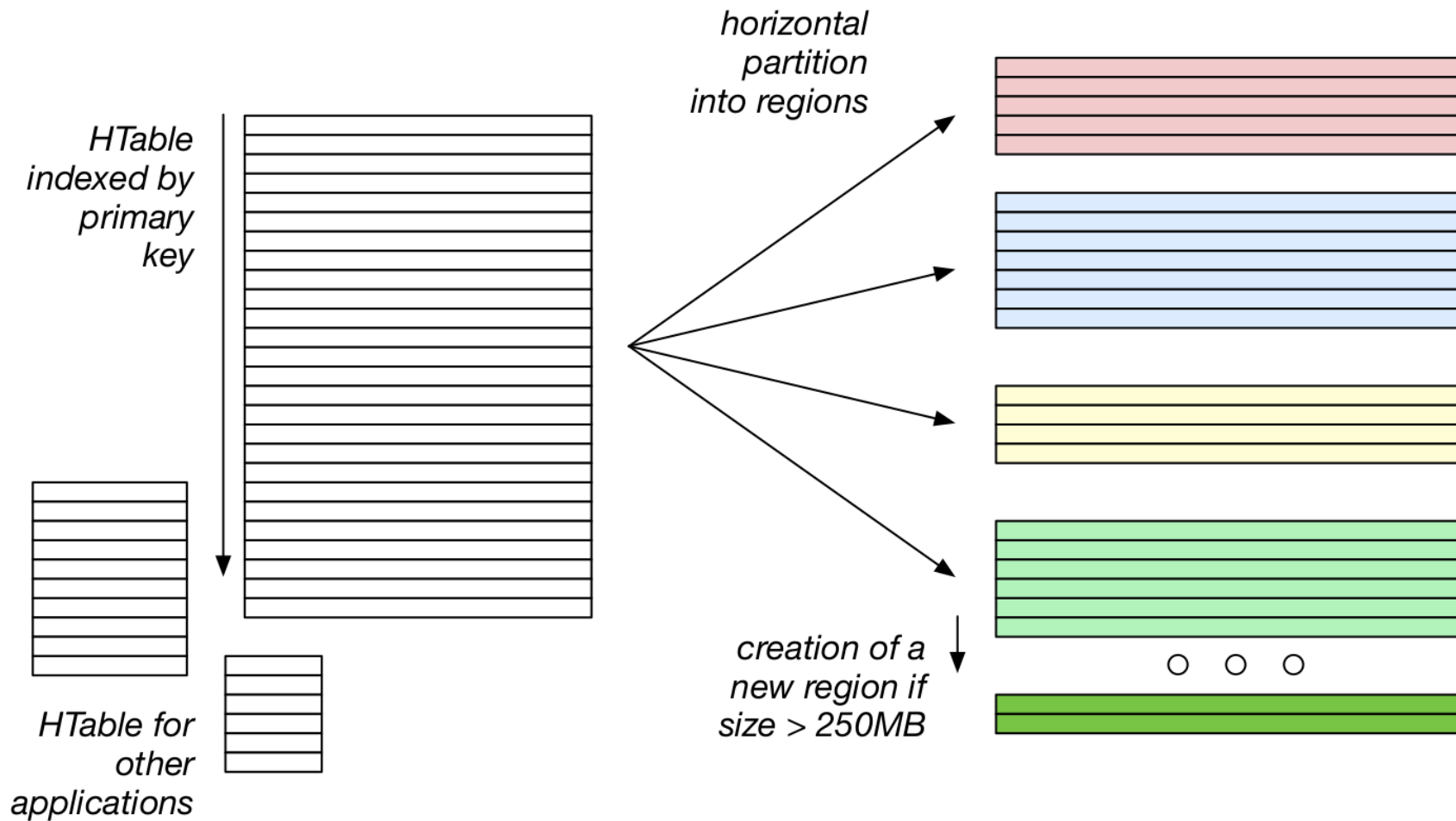
Application-specific elasticity

- Generic elasticity solutions may not be the most efficient for all applications
 - Load per VM might not be naturally balanced
 - Application-scaling model may be difficult to build or inaccurate
 - Elasticity and adaptation to workload may require more than just adding and removing resources: might also require to configure applications
- Application-specific solution can leverage knowledge about the application
 - Scaling and performance behavior
 - Configuration parameters and their expected effects

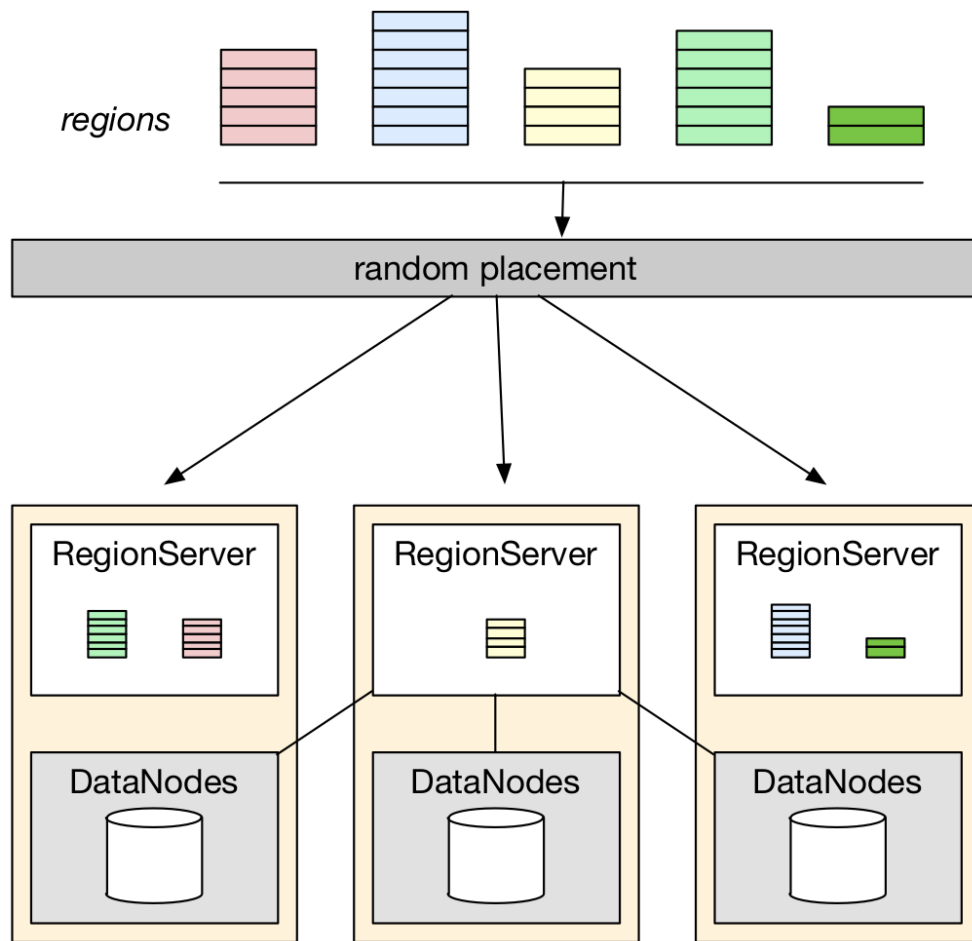
Application-specific elasticity

- Example of an application-specific elasticity solution for the HBase NoSQL database
 - HBase supports scale out and scale in operations thanks to dynamic partitioning
 - A given configuration of storage nodes might be more appropriate for a certain type of workload
 - Automatically drive addition and removal of resources as well as the allocation of data and configuration of nodes
- MeT: Workload aware elasticity for NoSQL

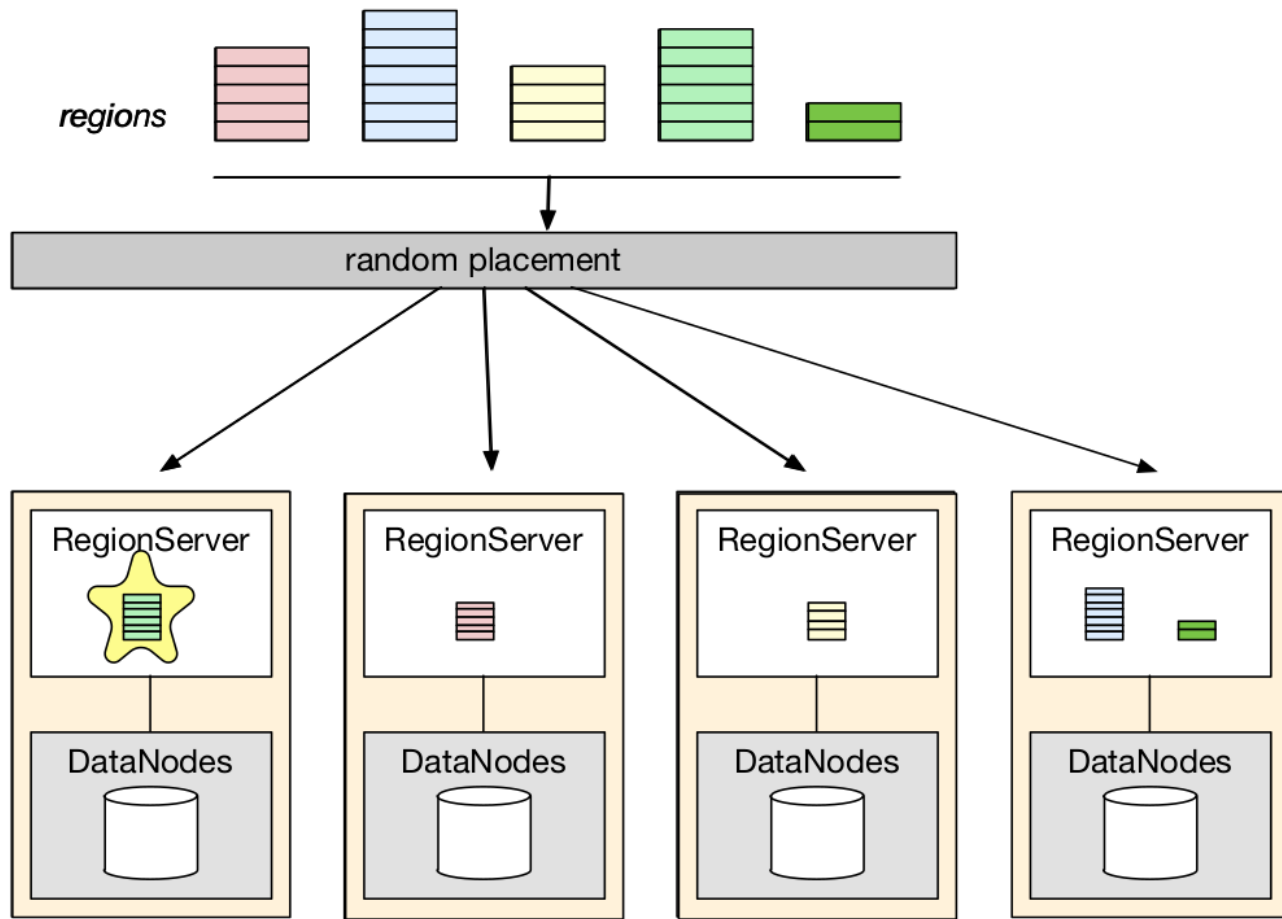
Principles of HBase (1)



Principles of HBase (2)



Data hotspots



Impact of configuration parameters

- Reading and writing performed in blocks
- 2 configuration parameters significantly impact performance
 - **Block cache** size
 - Amount of memory available to cache blocks from other regions
 - Higher value means higher **read** throughput
 - **Memstore** size
 - Amount of memory available to store updated blocks before flushing to disk
 - Higher value means higher **write** throughput
- These two parameters expressed as a percentage of java heap space
 - Total recommended to be < 65% of application heap space
 - Default 45% block cache, 20% memstore
- Block size also impacts (better large for scans, small for reads)

Evaluation of the impact of placement and configuration

- Comparison of random and manual placement of regions in a 5-node cloud
- Using the YCSB NoSQL benchmark
- 6 concurrent workloads with different read/write/scan operations balances
 - From 7 GB to 13 GB of data in 30 minutes
- Each workload HTable (but 1) has 4 regions with unbalanced popularities
 - Using the hotspot distribution of YCSB
 - One partition gets 34% of requests, the second gets 26%, and the two last get 20% each
- Each region replicated twice

Workloads characteristics

	Read	Write	Scan	Regions	Replicas
A	50%	50%		4	8
B		100%		4	8
C	100%			4	8
D	5%	95%		1	2
E		5%	95%	4	8
F	50%	50% _(with prior read)		4	8
total				21	42

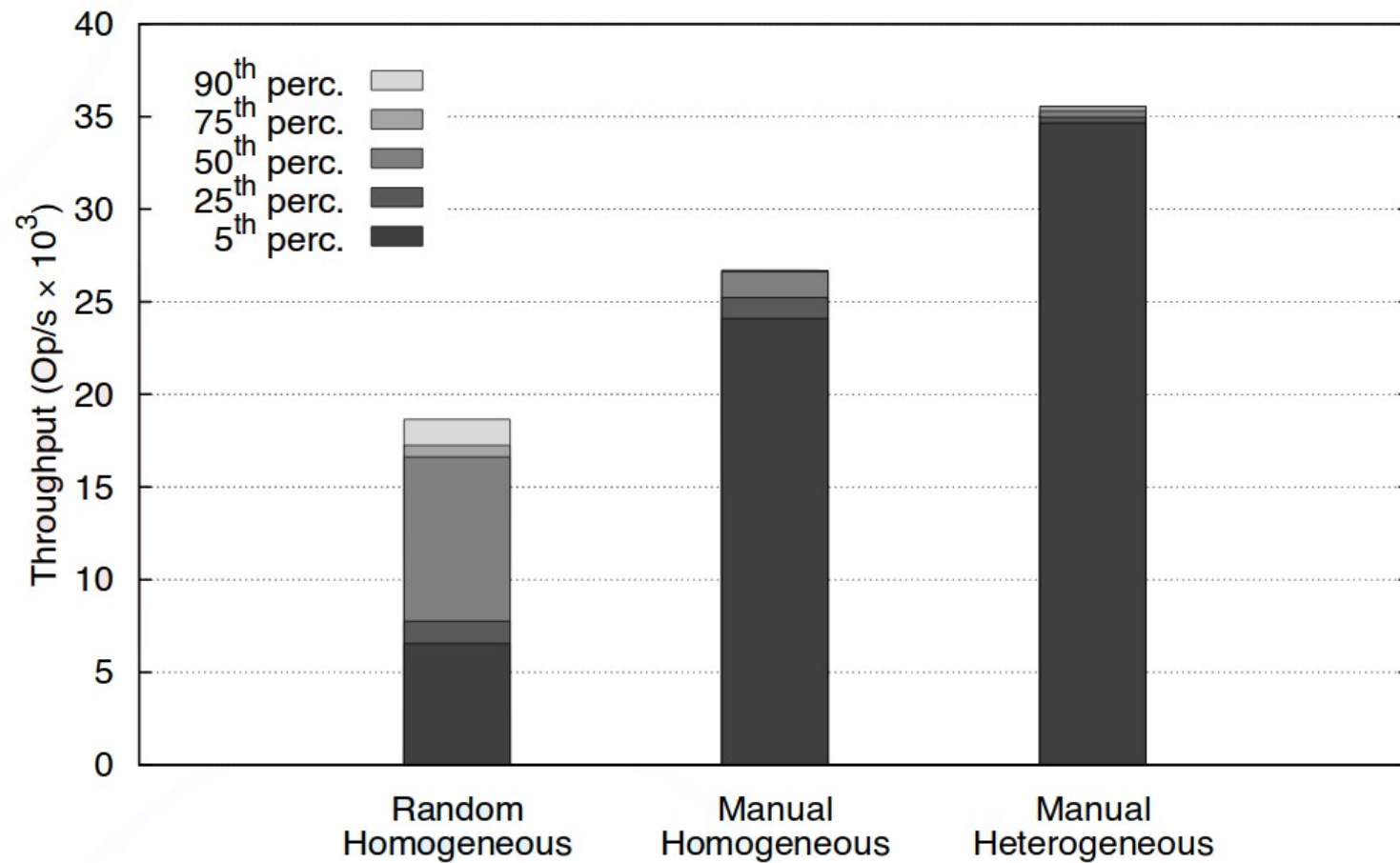
Placement and configuration

Strategy	Placement	Configuration
Random-Homogeneous	Random across RegionServers	Homogeneous (default 45%/20% optimized for read/write)
Manual-Homogeneous	Manually to balance load across RegionServers	
Manual-Heterogeneous		Heterogeneous using different region server configurations

Heterogeneous region server configurations

	Main operations	Total regions	Total replicas	Region servers	Region servers configuration
A & F	R and W	8	16	2	balanced (45%/20%) block size 32 KB
B & D	mostly W	5	10	1	write-optimized, large memstore (10%/55%) block size 64 KB
C	only R	4	8	1	read-optimized, large block cache (55%/10%) block size 32 KB
E	only Scan	4	8	1	scan-optimized large block cache (55%/10%) block size 128 KB

Results

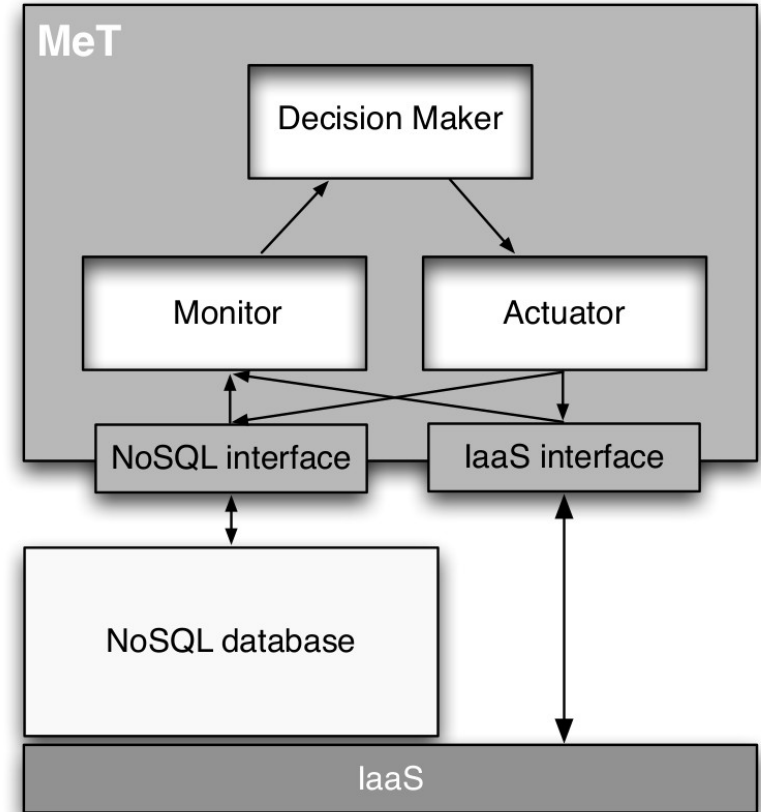


MeT - goals

- Heterogeneous and workload-adapted configuration boosts performance
 - But would be impossible to maintain manually
- Implement elasticity for HBase
 - Add and remove nodes based on observed load
 - Reactive and threshold-based as in Amazon Auto Scaling
 - Analyze the nature of the workload
 - Set up heterogeneous configurations
 - Add nodes with configurations adapted to the workload they are expected to serve
 - Match placement of regions to configurations of region servers

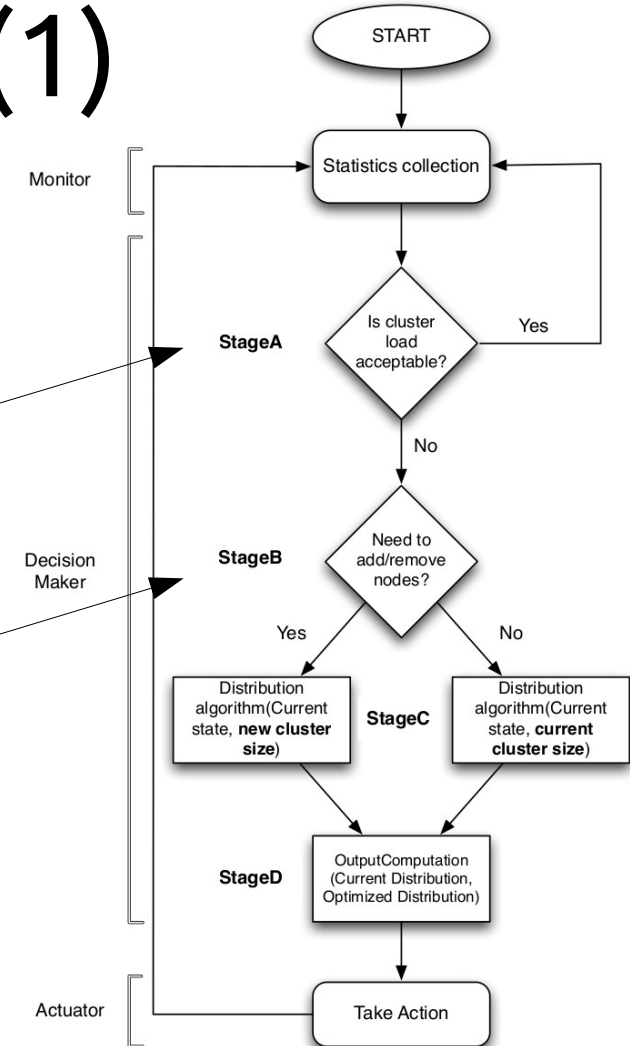
System architecture

- Need an interface to both the NoSQL DB API and the IaaS system
- Scaling up elasticity actions combine
 - Adding a node with the IaaS API
 - Setting up a region server configuration with the HBase API



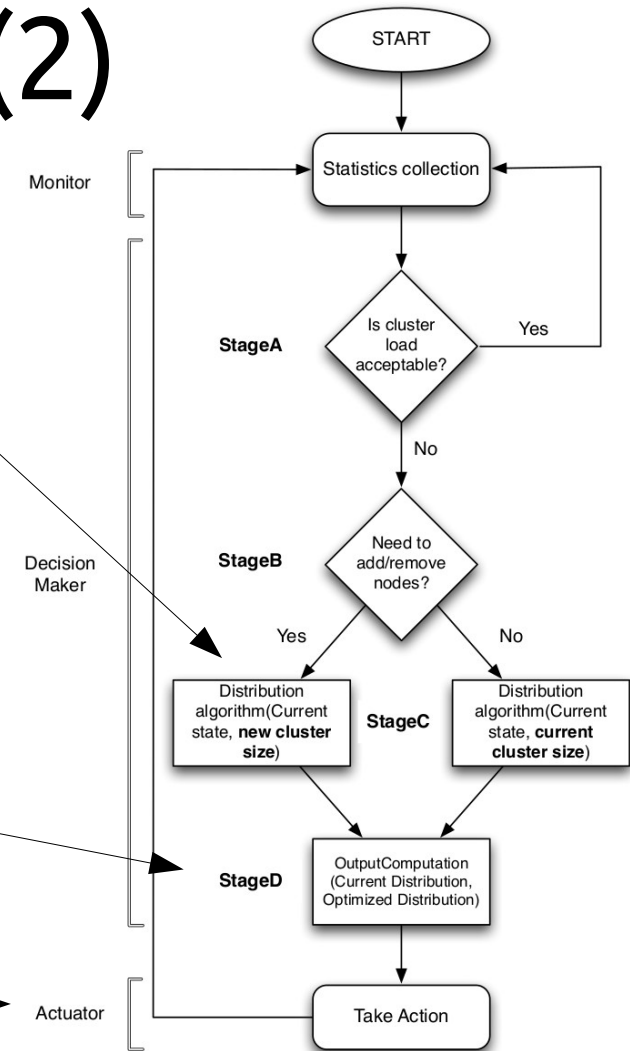
MeT flowchart (1)

- Statistics collection
 - From IaaS: CPU, I/O wait and memory
 - From HBase: amount of read, write, scan operations; locality of data accesses
- Reactive threshold-based trigger
 - Both for individual servers (unbalance) and for whole cluster (capacity)
- Decide to add/remove node(s)
 - Is redistributing regions enough or not?
 - Quadratic scale out: multiply # of nodes * 2
 - Linear scale in: remove 1 node



MeT flowchart (2)

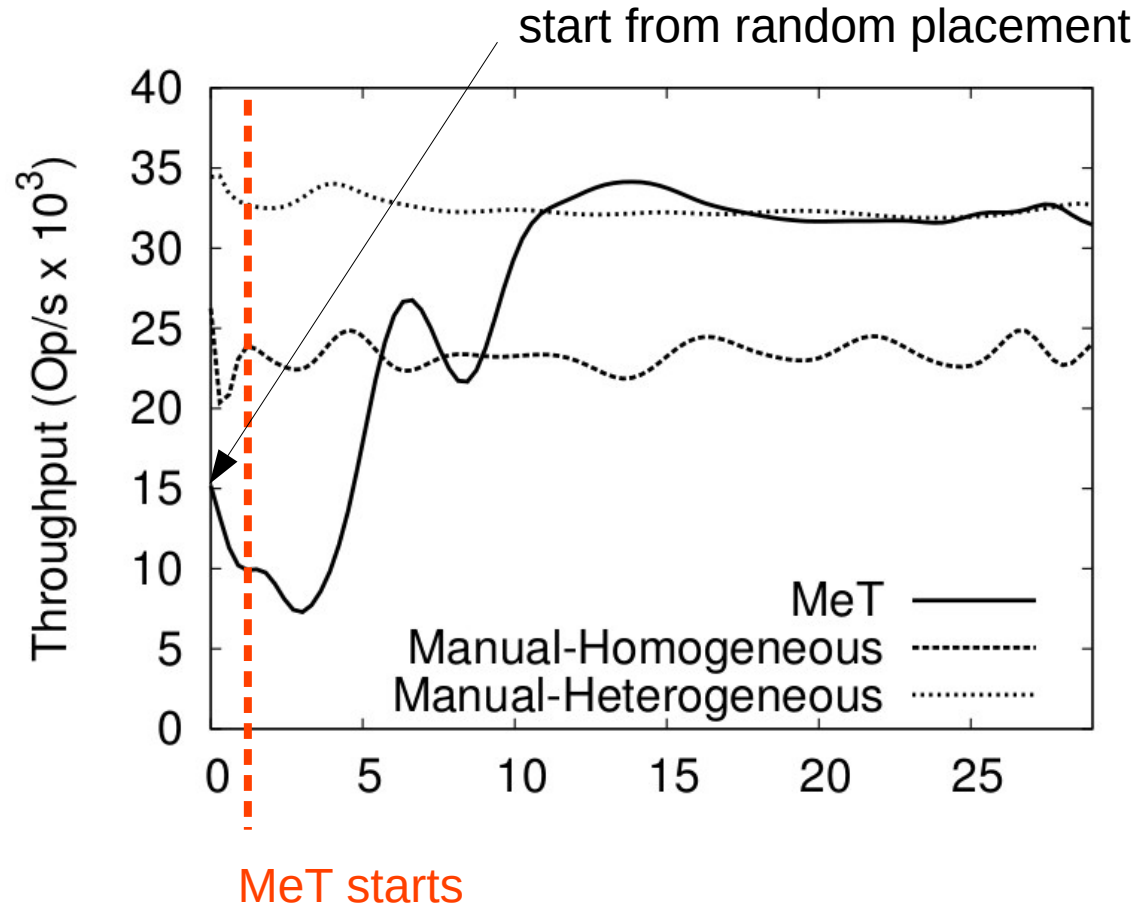
- Decide on new distribution of regions to region servers
 - Classify regions in different classes according to the natures of their access workloads
 - Read, Write, Read/Write, Scan
 - Determine how IaaS many region servers for each class are needed
 - Balance regions across region servers for their class, balancing the load
- Determine minimum set of actions
 - Minimal number of region movements and reconfigurations
 - Configuration resulting in most local accesses
- Actions use IaaS and NoSQL APIs



Evaluation

- Using previously defined workloads
- Comparison with manual-heterogeneous
 - Starting from a random homogeneous state
 - Static cluster: evaluates ability to reconfigure

Comparison with manual-*geneous

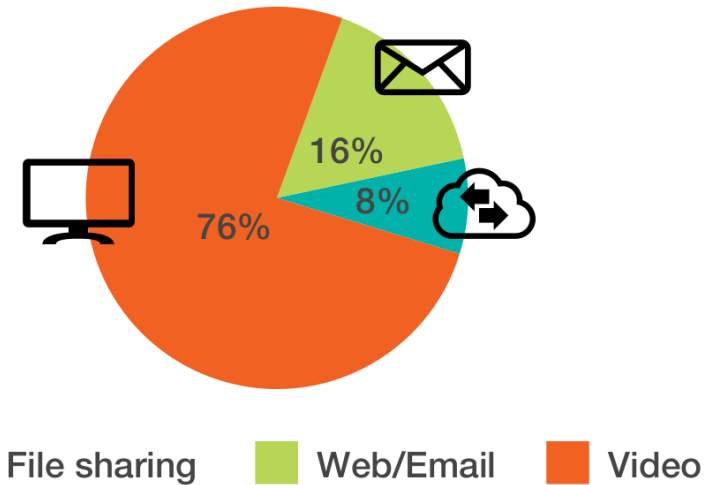


Cloud computing and energy efficiency

Introduction

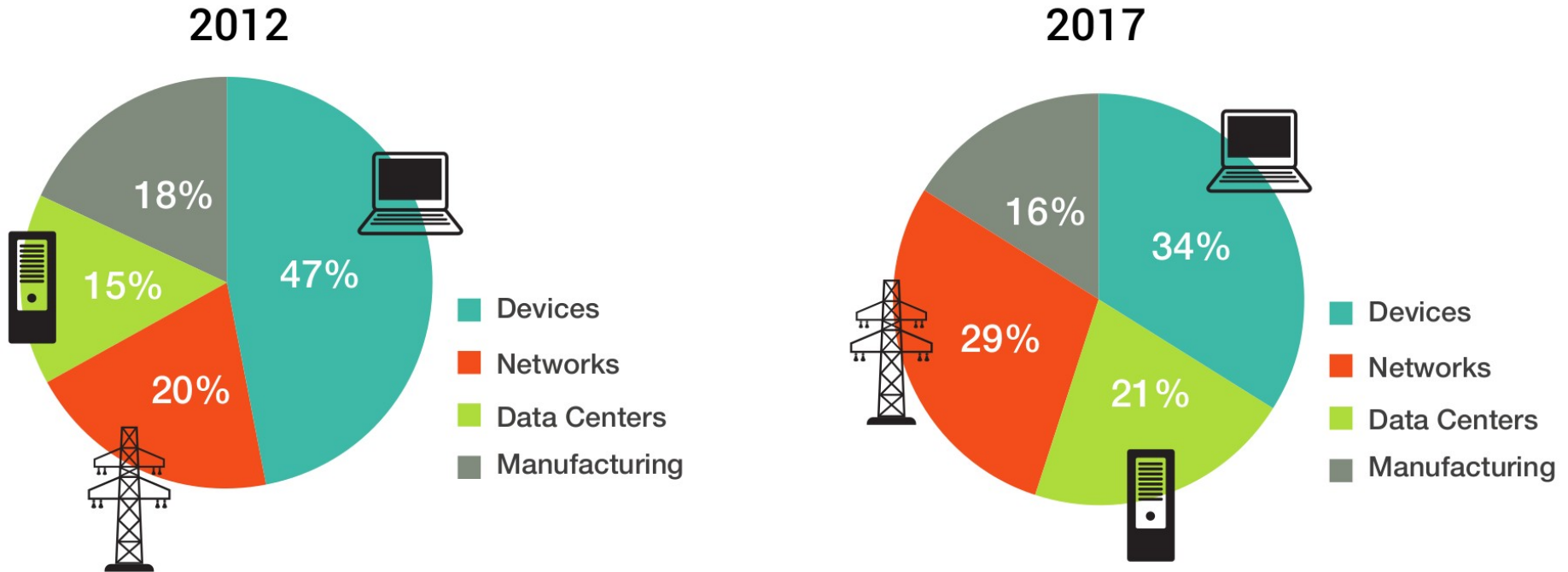
- Internet data: 20% more every year
 - Mobile data: 69% growth rate 2014-2019
 - Primarily due to video content
- More and more complex operations have to be handled by cloud backends
 - More lightweight clients (mobile phones, tables) → heavier cloud services
 - Outsourced storage
 - Outsourced signal processing (Siri, ...)
 - Outsourced rendering for video games

Expected internet consumer traffic: 2018⁷



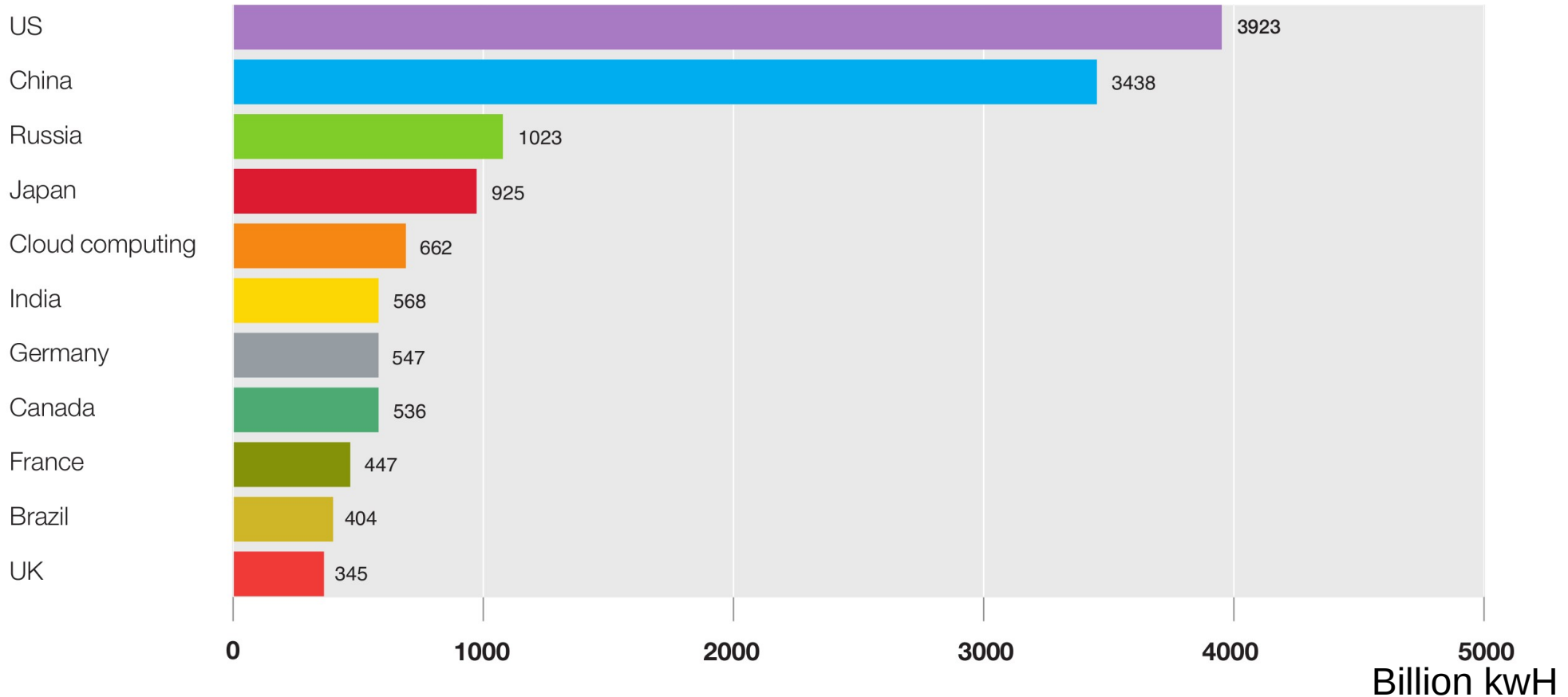
Cisco Visual Networking Index: Forecast and Methodology, 2013–2018.

Cloud consumption on the rise

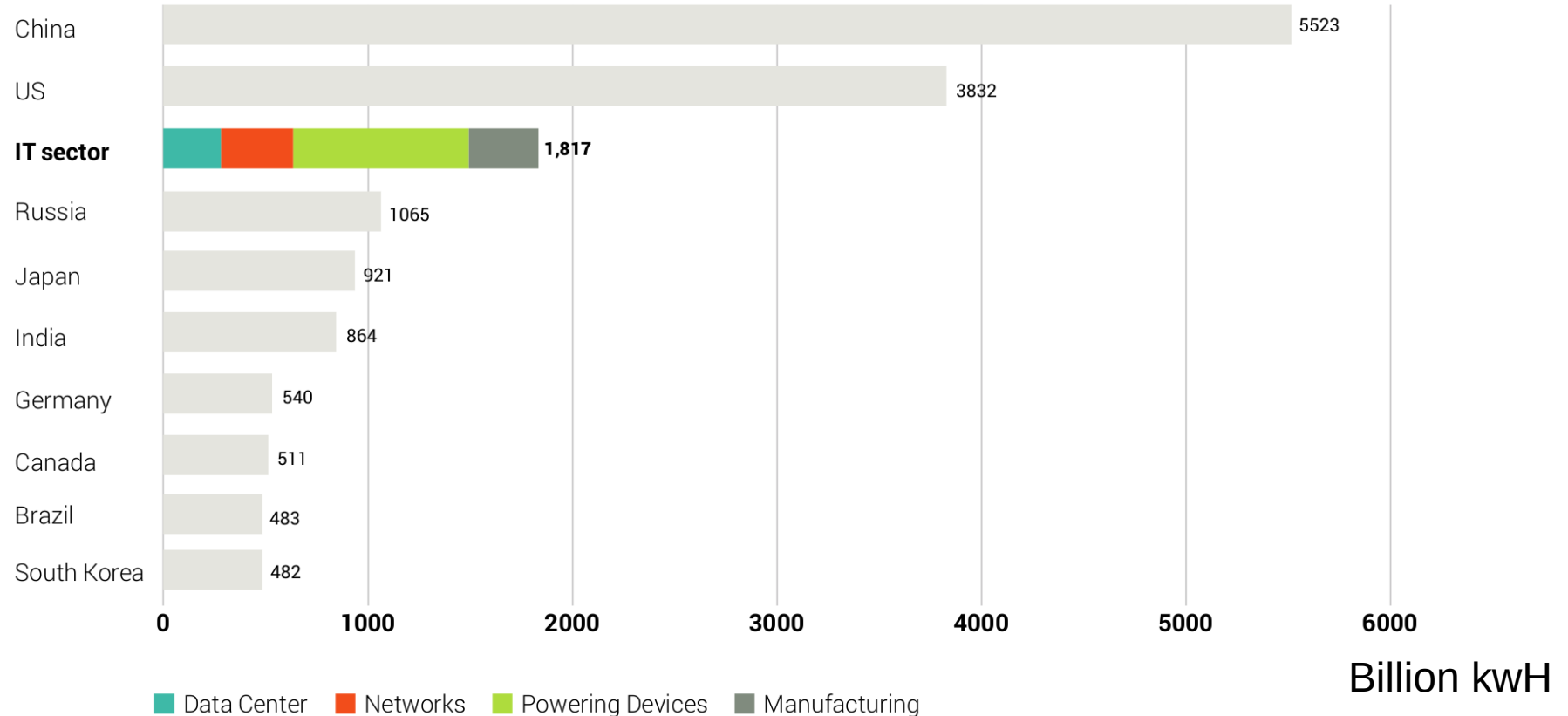


Main components of electricity consumption for the IT sector, 2012. From "Emerging Trends in Electricity Consumption for Consumer ICT"

If the Cloud of 2007 was a country



If the Cloud of 2012 was a country

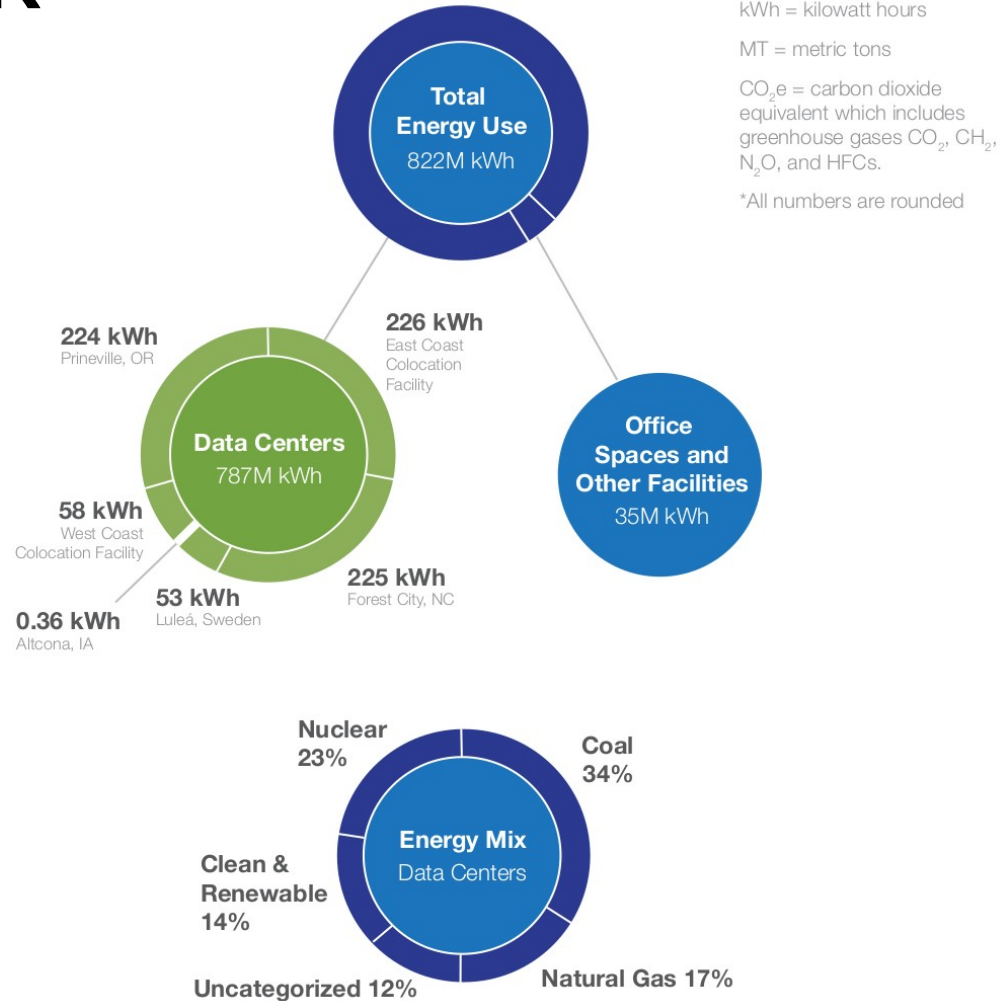


Data centers of the world will consume 1/5 of Earth's power by 2025 !

Facebook

Total Energy Use

822M kWh (2013)



Source: facebook

Using energy better

- For a given workload, use IaaS management systems to minimize the energy used
 - Techniques linked with elasticity mechanisms
 - And typically combined
- Two main and complementary goals
 - Use less resources, turn off unused resources:
energy proportionality
 - Collect VMs onto a smaller number of physical hosts:
workload consolidation

Energy proportionality

- Adapt the resources allocated to an application to the minimum amount required to meet its SLO (Service-Level Objective)
 - An application of vertical elasticity principles
 - Turn off CPU cores or memory banks
 - Adjust CPU power/capacity parameters
 - DVFS (Dynamic Voltage and Frequency Scaling)
 - Frequency boost modes
 - Use of HyperThreading

Workload consolidation

- Consolidate multiple applications onto the smallest possible set of physical machines
 - Usage level in average data center estimated around 10 to 20%
- Turn off servers hosting no VM
 - An application of horizontal elasticity principles
 - Energy utilization is not linearly proportional to workload: servers consume energy even when unused
 - Technical challenges for turning the servers back on if feature not planned by constructor (not a problem for major operators who build their own custom servers)

Example of workload consolidation: Quasar

- Combines elastic IaaS management with the goal of minimizing the number of machines
- Users express their SLO directly to the system
- Quasar uses classification techniques (based on machine learning) to "learn" the effect of each parameter on the application performance
 - Based on classification techniques, group applications with similar behavior together and assume these will have the same scaling behavior
 - Requires VM migration capabilities
 - Similar to algorithms used for Netflix recommendation systems!
- Solves a bin-packing problem finding the minimal number of servers for a certain set of applications

Exploiting heterogeneity

- A computation might be more energy-efficient on some hardware than on some other
 - Typically, CPU vs. GPU computation
 - GPU better at highly parallel but simple tasks
 - CPU better at I/O and complex code paths
- Monitor apps performance and deploy versions based on optimal power consumption
 - Still more a research direction than a concrete feature in cloud offerings















Use better energy

- The source of energy used by a data center matters
 - Renewability
 - Environmental costs
 - Availability
 - A source of clean energy is less useful if it requires falling back to coal-based energy when unavailable
- Not renewable: nuclear, natural gas, coal
- Renewable: hydropower, geothermal
- In-between: biogas, biomass

Trend towards sustainable energy

- Survey by Greenpeace
 - "Clicking Green" report 2016
- Half a dozen IT companies committed to being 100% renewably powered
 - Including Apple, Facebook and Google
 - Apple leading the movement
 - Oracle and Amazon lagging behind

Greenpeace 2016 company scorecard

	Final Grade	 Clean Energy Index	 Natural Gas	 Coal	 Nuclear	Energy Transparency	Renewable Energy Commitment & Siting Policy	Energy Efficiency & Mitigation	Renewable Procurement	Advocacy
 Adobe	B	23%	37%	23%	11%	B	A	B	B	A
 Alibaba.com	D	24%	3%	67%	3%	F	F	C	F	D
 amazon.com web services	C	17%	24%	30%	26%	F	D	C	C	B
 Apple	A	83%	4%	5%	5%	A	A	A	A	B
 Baidu 百度	F	24%	3%	67%	3%	F	F	D	F	F
 f	A	67%	7%	15%	9%	A	A	A	A	B
 Google	A	56%	14%	15%	10%	B	A	A	A	A
 hp	C	50%	17%	27%	5%	D	B	C	B	C
 IBM	C	29%	29%	27%	15%	C	B	C	C	F
 Microsoft	B	32%	23%	31%	10%	B	B	C	B	B

Conclusion

- Elasticity: Adapt the amount of resources to the actual workload
 - Enabler for pay-per-use pricing model
- Application agnostic elasticity
 - Apply to VM based applications, most often used for multi-tier Web applications
 - Machine learning techniques can allow predicting load spikes
- Application-specific elasticity
 - Can be paired with the problem of finding optimal configuration
- Energy efficiency features are close in principles and mechanisms to those used for elasticity

References (1)

- A Survey on Cloud Computing Elasticity
 - Guilherme Galante, Luis Carlos E. de Bona
 - UCC '12 Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing
- AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service
 - Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, John Wilkes
 - ICAC 2013: 69-82
- SnowFlock: Virtual Machine Cloning as a First-Class Cloud Primitive
 - H. Andrés Lagar-Cavilla, Joseph Andrew Whitney, Roy Bryant, Philip Patchin, Michael Brudno, Eyal de Lara, Stephen M. Rumble, M. Satyanarayanan, Adin Scannell
 - ACM Trans. Comput. Syst. February 2011
- Lightweight Resource Scaling for Cloud Applications
 - Han, Rui and Guo, Li and Ghanem, Moustafa M. and Guo, Yike
 - CCGRID '12
- MeT: workload aware elasticity for NoSQL
 - Francisco Cruz, Francisco Maia, Miguel Matos, Rui Oliveira, João Paulo, José Pereira, Ricardo Vilaça
 - EuroSys 2013

References (2)

- Automated, Elastic Resource Provisioning for NoSQL Clusters Using TIRAMOLA
 - Dimitrios Tsoumakos, Ioannis Konstantinou, Christina Boumpouka, Spyros Sioutas, Nectarios Koziris
 - CCGRID 2013
- Energy proportionality and workload consolidation for latency-critical applications
 - George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, Edouard Bugnion
 - SoCC 2015
- Quasar: resource-efficient and QoS-aware cluster management
 - Christina Delimitrou, Christos Kozyrakis
 - ASPLOS 2014
- Clicking Clean Report 2015, Greenpeace
 - <http://www.greenpeace.org/usa/global-warming/click-clea>
- <https://aws.amazon.com/autoscaling/>
- <https://www.cisco.com/c/en/us/solutions/service-provider/visual-networking-index-vni/index.html>