Middleware and Web Services

Lecture 11: Cloud Computing Concepts

doc. Ing. Tomáš Vitvar, Ph.D.
tomas@vitvar.com • @TomasVitvar • http://vitvar.com

Czech Technical University in Prague
Faculty of Information Technologies • Software and Web Engineering • http://vitvar.com/courses/mdw

Evropský sociální fond
Praha & EU: Investujeme do vaší budoucnosti

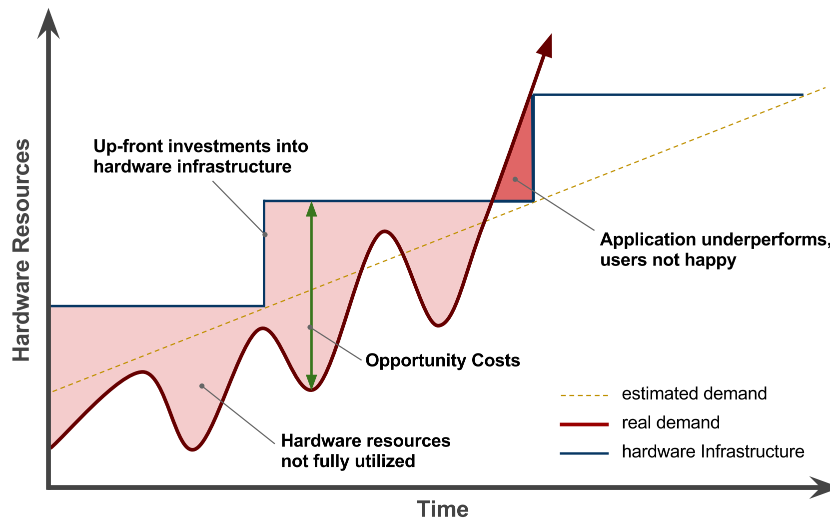
Modified: Mon Dec 12 2011, 22:31:47
Humla v0.2.2

Overview

- Cloud Computing
  - *Cloud Layers*
- Service Performance
- Load Balancer
- Messaging Systems

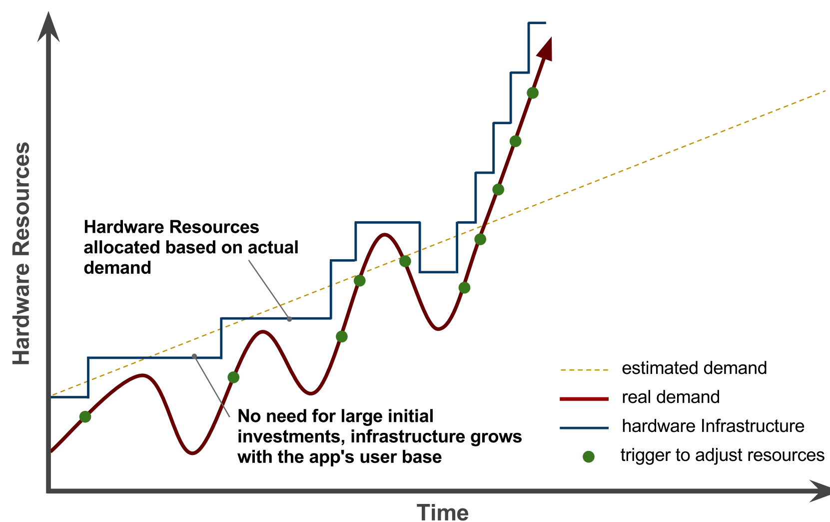
Traditional Solution to Infrastructure

- Traditional hardware model
  - *Up-front hardware investments*
  - *Hardware not optimally utilized*



Cloud Solution to Infrastructure

- Cloud computing model
  - *No up-front hardware investments*
  - *Hardware optimally utilized*



Cloud Computing

- Outsourcing of application infrastructure
  - *Reliability and availability*
  - *Low costs – pay-as-you-go*
  - *Elasticity – can dynamically grow with your apps*
- Different way of thinking
  - *Got your grand mum's savings under your pillow?*
    → probably not, you better have them in your bank
  - *Data is your major asset*
  - *you better have them in a "bank" too*
  - *Someone can abuse your data?*
  - *banks bankrupt too, sometimes – it is a risk you take*
  - *there is a market and a competition*

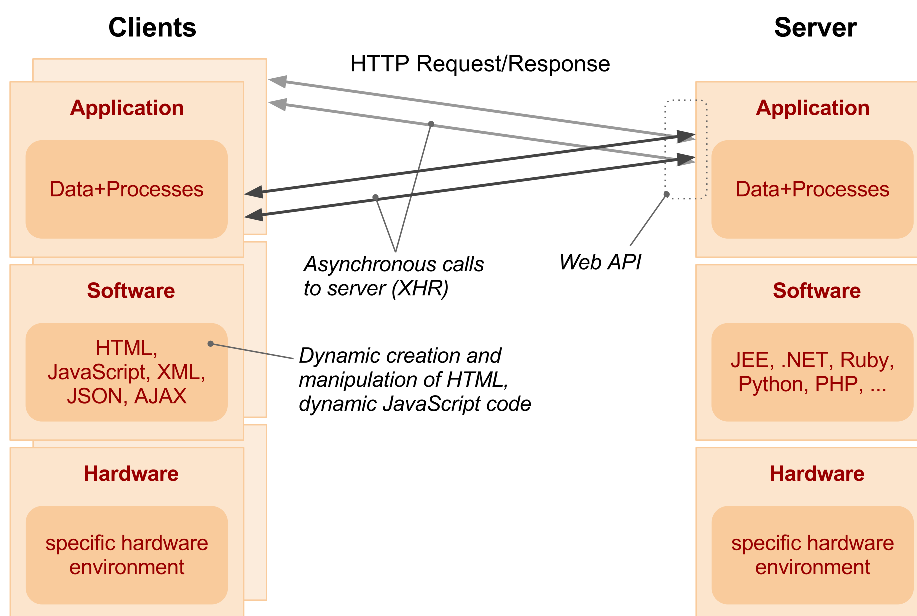
What is a Cloud?

- Any app you access over the web?
- A datacenter?
  - *Offers virtualization*
  - *Any company having a datacenter wants to move to*
- Three layers: IaaS, PaaS, SaaS
  - *access programmatically, pay-as-you-go, low up-front costs, initial version for free, micro payments etc.*
- Cloud computing offers services
  - *scalability, storage*
  - *Possible to configure programmatically*
    → integration to enterprise administration processes
    → usually REST interface

Overview

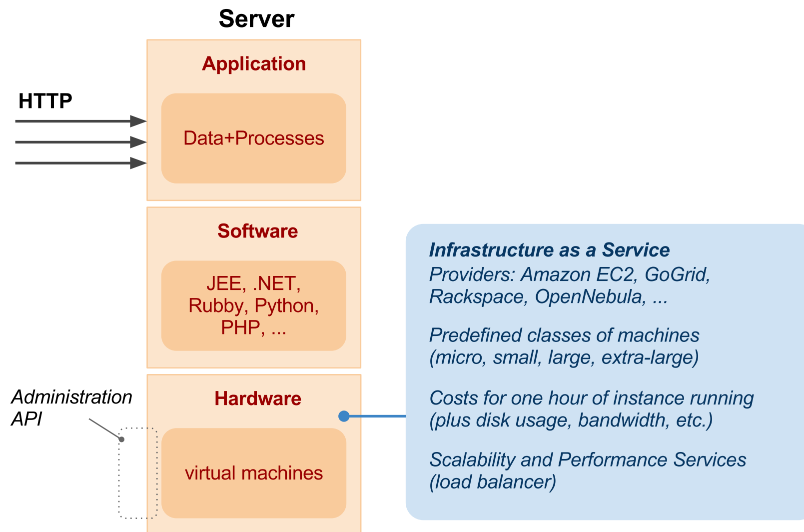
- Cloud Computing
  - *Cloud Layers*
- Service Performance
- Load Balancer
- Messaging Systems

Web 2.0 App Architecture



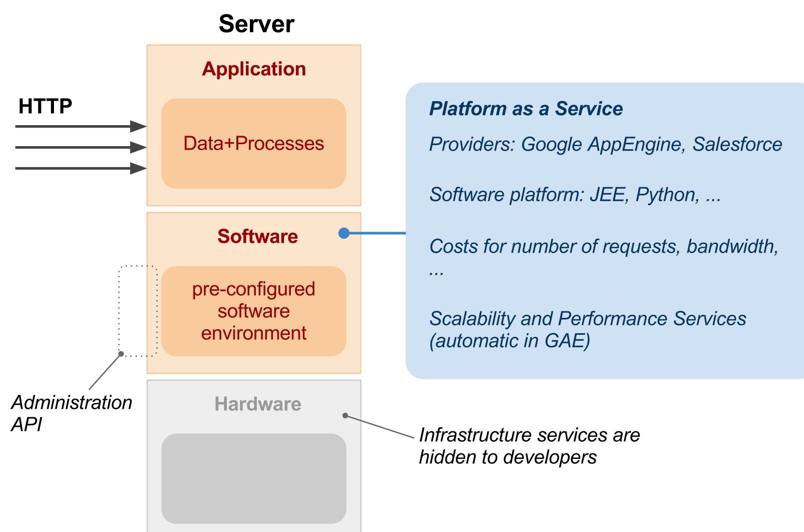
Move to the Cloud: IaaS

- Infrastructure as a Service
  - *Services for application providers*



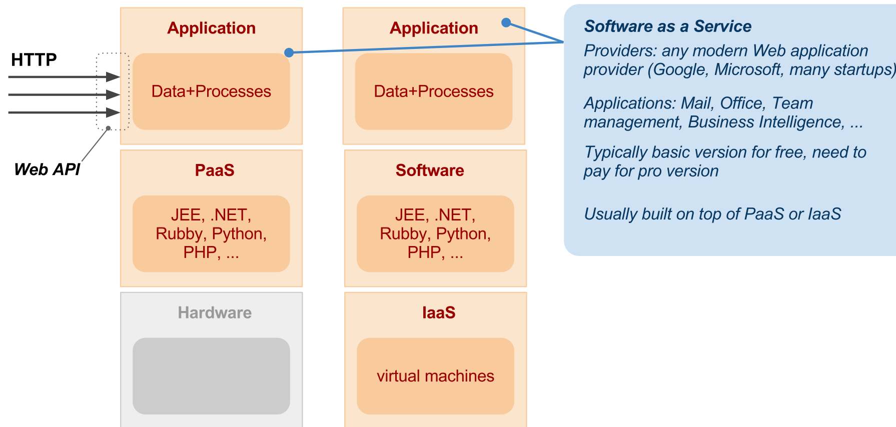
Move to the Cloud: PaaS

- Platform as a Service
  - *Services for application providers*

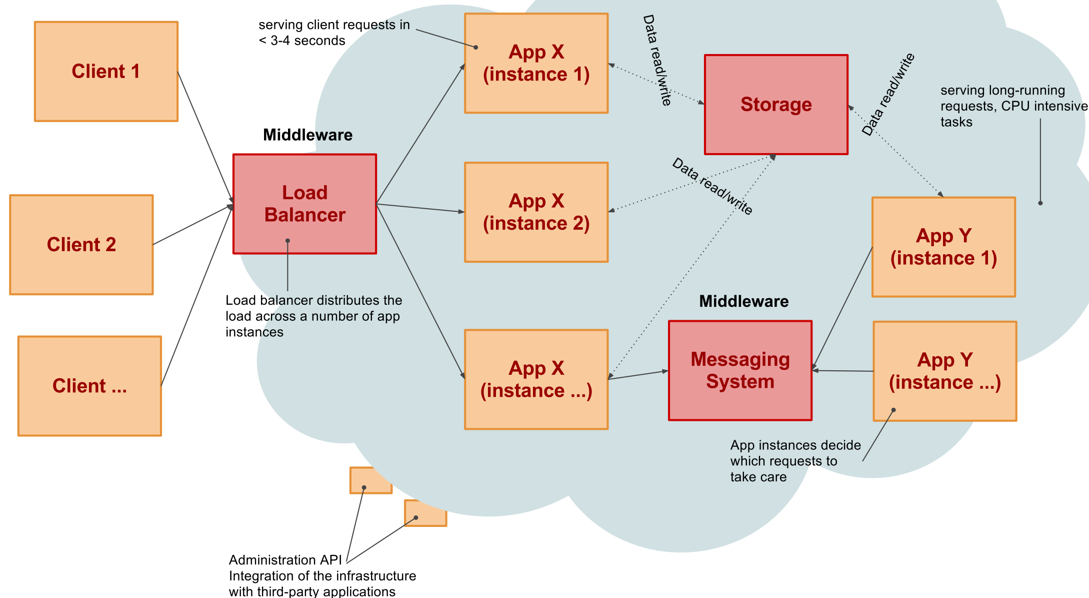


Move to the Cloud: SaaS

- Software as a Service
  – *Services for end-users*



Cloud Computing Services



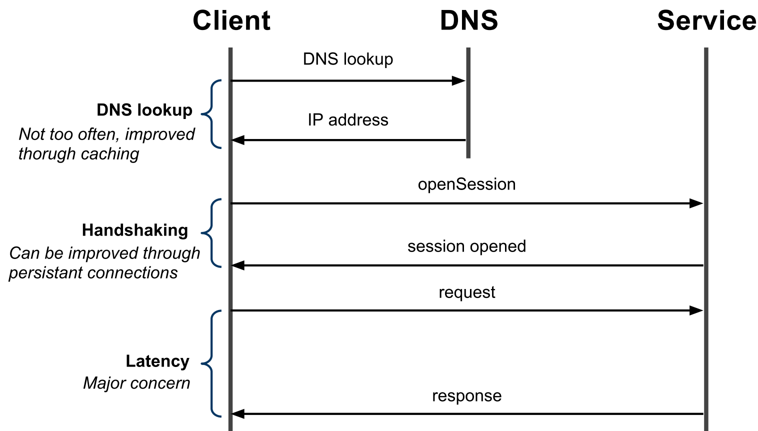
Overview

- Cloud Computing
- Service Performance
- Load Balancer
- Messaging Systems

Good Performance

- Scalability
  - *server scalability*
    - → *ability of a server to react on changes in loads*
    - → *a user should not feel the change in loads*
    - → *horizontal and vertical scaling*
  - *network traffic*
    - → *not all about network bandwidth capacity*
    - → *service needs to limit the network traffic through caching*
- Availability
  - *probability that a service is operational at a particular time*
    - → *e.g., 99.9987% availability – downtime ~44 seconds/year*
- Metrics
  - *Latency*
  - *Queries per Second (QPS)*

Latency



- High latency influenced by
  - *CPU intensive service or a bad implementation of a service*
    - *→ consider asynchronous processing when CPU intensive*
  - *Writing to a data store*
  - *No or bad caching mechanism*
    - *→ even if data changes often, with high QPS caching improves a lot*
    - *→ stateful service implementation may have influence too*

CPU Time

- The time a CPU spends on executing a service
- CPU Time influences
  - *latency of your service*
  - *your bills you pay for running the service*
- Reference CPU
  - *system's infrastructures are complex, times may vary*
  - *all times calculated wrt this CPU*
  - *billing on a reference CPU in the cloud*
  - *Example in GAE:*
    1.2GHz CPU, 3600 MIPS (3 instructions per cycle)
- How to determine CPU time
  - *profiling (depends on programming environment)*
  - *analysis of service's computational complexity*

Profiling in GAE

- Example profiling code

```
// QuotaService gives us an access to CPU cycles
QuotaService qs = QuotaServiceFactory.getQuotaService();

// initialization
// 10000 numbers; worst-case scenario - in reverse order
int[] A = new int[10000];
for (int i = 0; i < A.length; i++)
    A[i] = A.length - i;

// current number of cycles
long start = qs.getCpuTimeInMegaCycles();

// insertion sort
insertionSort(A);

// current number of cycles
long end = qs.getCpuTimeInMegaCycles();

```

- Analysis
  - *Number of CPU mega cycles* $c = end - start \approx 400$
  - c *is in mega cycles, 1200 mega cycles is 1 CPU second*
  - *Total CPU seconds for the insertion sort* $T'_{cpu} = c/1200 = 0.35$

Service Analysis

- Running time

$$T(n) = k_1 \cdot g(n) + k_2$$

  - k_1, k_2 *are constant factors (independent on* n*)*
  - *for functions with* $O(g(n))$ *complexity,* $T(n)$ *describes worst-case running time*
  - *CPU-intensive functions: large* n
- Constant factors
  - *overhead costs when running the function:*
    (a) k_1 *denotes efficiency of function's algorithm implementation,*
    (b) k_2 *denotes one-time costs such as reading, writing, serializing data, simple calculations*

CPU Seconds

- Running time in CPU seconds

$$T_{cpu}(n) = \frac{k_1 \cdot g(n) + k_2}{c_m}$$

where c_m is a CPU's Million Instructions per Second (MIPS)

- Example (function analysis)
  - *sort 10,000 items using insertion sort on a 1.2 GHz x86 CPU with 3 instructions per clock cycle (~ 3600 MIPS)*
  - *insertion sort complexity is $O(n^2)$*
  - $k_1 = 12$ *(~12 instructions per operation in avarage),*
    $k_2 = 0$ *(no additional costs)*
    $c_m = 1200 \cdot 3 = 3600$
  - *CPU Time in seconds:*

$$T_{cpu}(10^4) = \frac{12 \cdot (10^4)^2}{3600 \cdot 10^6} \approx 0.33$$

Overview

- Cloud Computing
- Service Performance
- Load Balancer
- Messaging Systems

Load Balancer

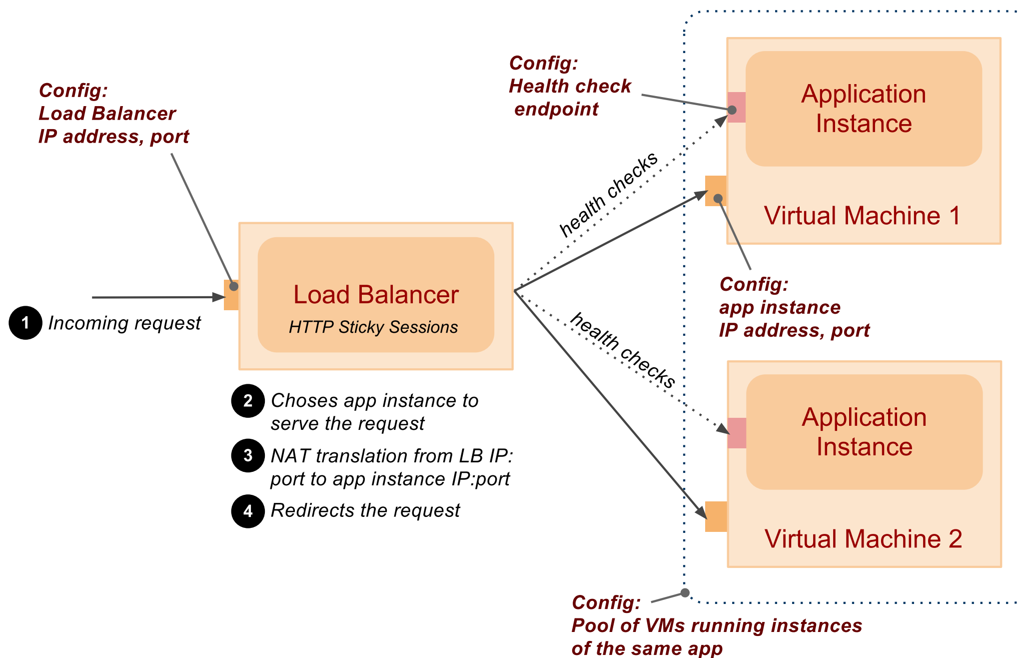
- Distributes the load to multiple app instances
  - *App instances run on different machines*
  - *Load sharing: equal or with preferences*
  - *Health checks*
- Types
  - *DNS-based load balancer*
    - *→ DNS Round Robin*
  - *NAT-based load balancer*
    - *→ app protocol layer*
  - *HTTP Sticky-sessions*

DNS-based Load Balancer

- DNS Round Robin
  - *Very first load balancing mechanism*
  - *A DNS record has multiple assigned IP addresses*
  - *DNS system delivers different IP addresses from the list*
  - *Example DNS A Record:*

    `moon.com A 147.32.100.71 147.32.100.72 147.32.100.73`
- Advantages
  - *Very simple, easy to implement*
- Disadvantages
  - *IP address in cache, could take hours to re-assign*
  - *No information about servers' loads and health*
- Variation of DNS Round robin
  - *a load balancer assigns domain names*
    - *→ each domain has a different IP in DNS*
  - *a health-check is possible, IP re-assignment problem remains*

NAT-based Load Balancer



Load Balancer Configuration API

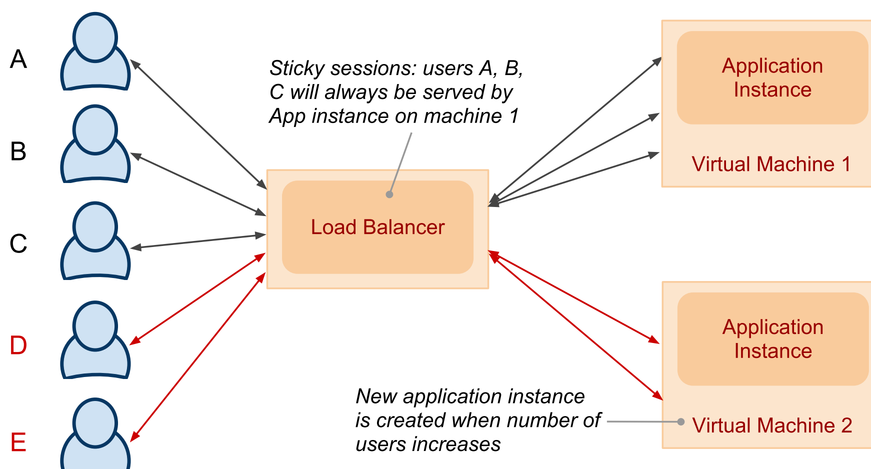
- `CreateLoadBalancer`
  - `LoadBalancerName` – *unique name of the load balancer*
  - `Protocol` – *protocol such as HTTP*
  - `LoadBalancerPort` – *port on which LB listens for incoming requests*
  - `InstancePort` – *app instance TCP port such as 8080*
- `ConfigureHealthCheck`
  - `LoadBalancerName` – *name of existing LB*
  - `Target` – *in a form of* `HTTP:port/PathToPing`
  - `Interval` – *amount of seconds to perform health checks*
  - `Timeout` – *amount of seconds when no response to health check endpoint means unhealthy app instance*
  - `HealthyThreshold` – *number of successful health checks that will mark the app instance as healthy*
  - `UnhealthyThreshold` – *number of unsuccessful health checks that mark the app instance as unhealthy*
- `RegisterInstancesWithLoadBalancer`
  - `LoadBalancerName` – *name of existing LB*
  - `Instances`– *app instances to put to the pool*

App Instance Usage

- Sticky sessions
  - *Identified by cookies created by the load balancer*
  - *Information in the request about the app instance*
- Steps to chose an app instance to serve the request
  - ***if a sticky session exists to an app instance***
  - *always use that app instance*
  - ***if a sticky session does not exist***
  - *Use round robin for the next app instance*
  - ***health check***
  - *if a number of health checks exceeds the unhealthy threshold*
    → discard the app from the list
  - *if an app was unhealthy and a number of health checks exceeds the healthy threshold*
    → add the app to the list

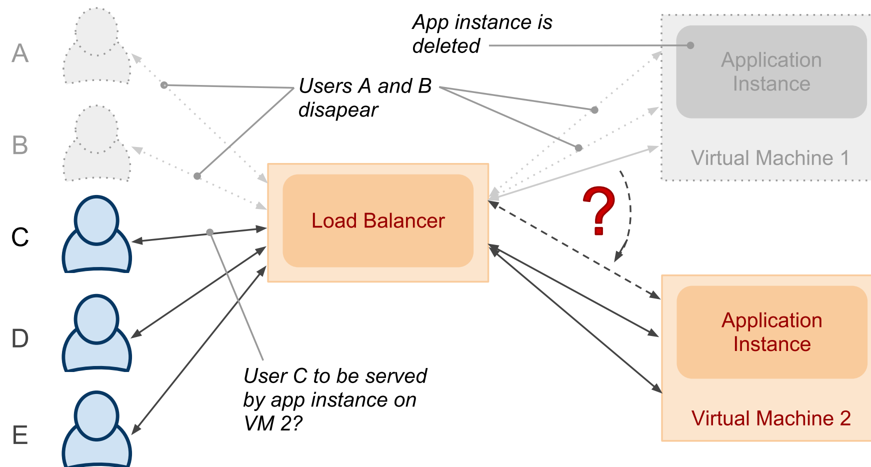
Scalability

- Scaling Up
  - *Number of users increases, need for more resources*
  - *Replication of app instances*



Scalability

- Scaling Down
  - *Number of users decreases, no need for many resources*
  - *Deleting app instances*

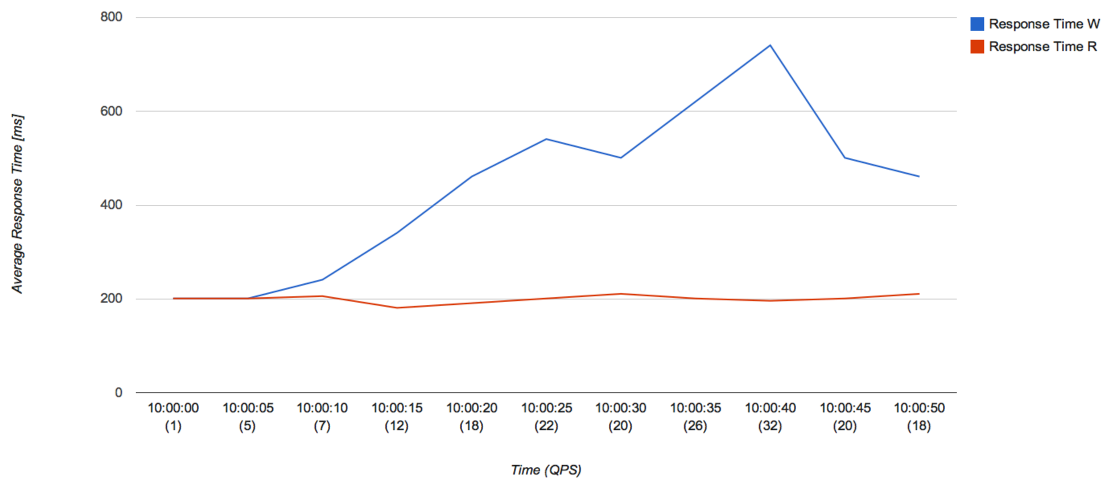


Scaling Down Issues

- Session in a persistent storage
  - *Stateful server*
  - *No sticky sessions – auto scaling*
- GAE implementation
  - *New or updated session data*
    $\longrightarrow$ *written to BigTable and the memcache*
  - *Reading the data*
    $\longrightarrow$ *only from the cache, is fast*
- Limitations on data writing
  - *GAE allows to update a single entity 5 times/sec.*
  - *When QPS > 5 (GAE scales up to 500 QPS)*
    $\longrightarrow$ *the latency grows*

Example Application Scalability

- Response times × QPS for reading (R) and writing (W)
  - *Writing scales well up to 5 QPS*
  - *Reading scales well for all QPS*



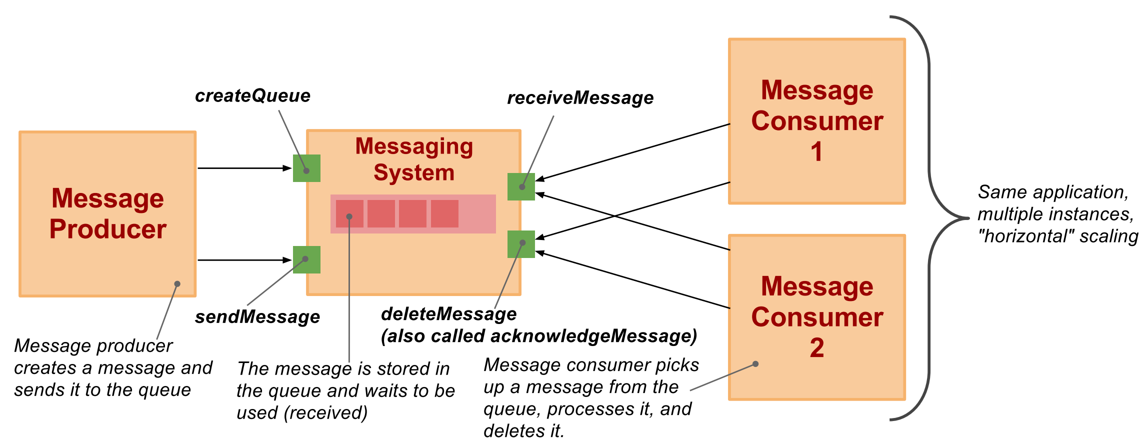
Overview

- Cloud Computing
- Service Performance
- Load Balancer
- Messaging Systems
  - *Message Queues*
  - *Publish/Subscribe*

Messaging Systems

- Messaging Middleware
  - *aka Message-Oriented Middleware (MOM)*
  - *Message consumer and message producer*
  - *asynchronous communication*
  - *"anonymity" between producers and consumers*
    → no matter who, where, when a msg was produced
  - *Ensures reliability, scalability*
- Loose coupling of applications
  - *a kind of "peer-to-peer" relationship between applications*
- Two types (Messaging Domains)
  - *Point-to-Point (message queue — MQ)*
  - *Publish/Subscribe (event-based)*

MQ System

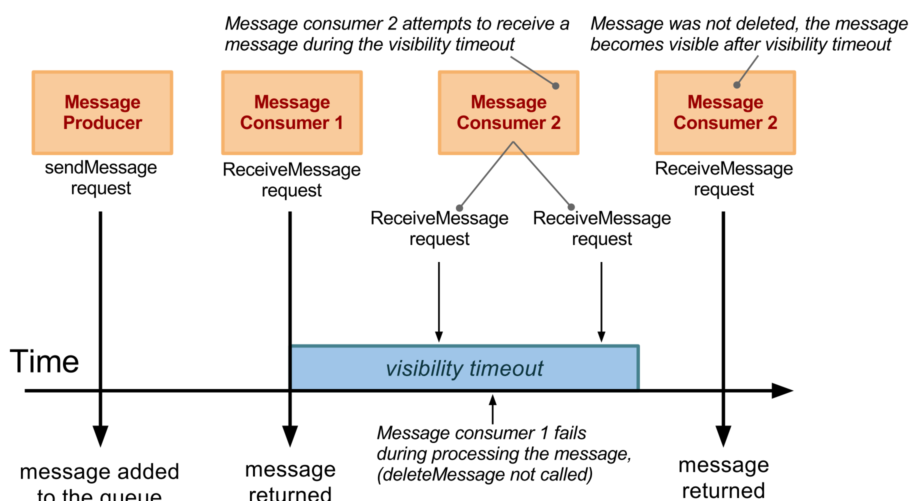


- *a "1 : 1" relationship between producer and consumer*
  → ***one message must be processed by one consumer***
- ***no time-dependency*** *between message producer and consumer*
  → consumer does not need to exist when producer sends a message
- *Message exists in the queue until it is used by a consumer*
- *message consumers take as many messages as they are able to serve*

MQ API

- `createQueue`
  - *creates a queue in the MQ system*
  - *called by the message producer*
- `sendMessage`
  - *send a message to the queue*
  - *called by the message producer*
- `receiveMessage`
  - *request to receive (read) a message from the queue*
  - *called by the message consumer*
- `deleteMessage`
  - *deletes a message from the queue*
  - *called by the message consumer when the message is successfully processed*

Message Consumption from the Queue

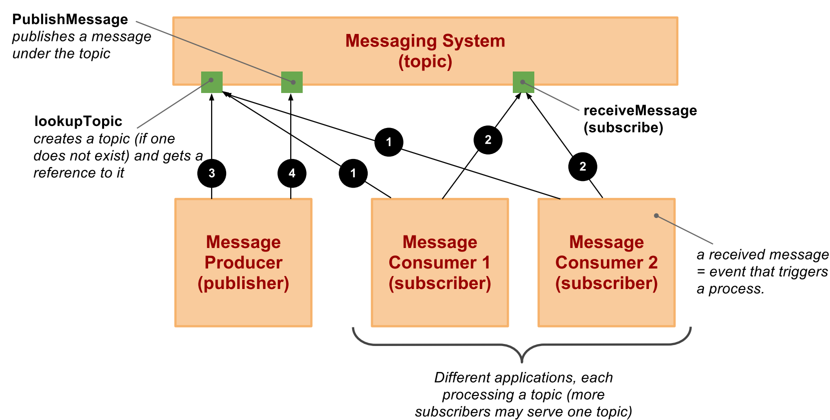


- *Amazon Simple Queue System (SQS), part of EC2*
- *message consumers may fail anytime*
- *multiple instances of message consumers increases reliability*
- ***visibility timeout*** *– time during which the message exist in the queue, and need to be deleted by the consumer (~ 30 seconds in SQS)*

Overview

- Cloud Computing
- Service Performance
- Load Balancer
- Messaging Systems
  - *Message Queues*
  - *Publish/Subscribe*

Publish/Subscribe System



- *occurrence of a message = event that triggers one or more processes*
- *a "1 : N" relationship between producer and consumer*
  *→ **one message can be processed by many different subscribers***
- ***time-dependency** between publisher and subscriber*
  → subscriber must first subscribe to a topic and then publisher can publish a message under that topic
- *a message is deleted when all its subscribers consume it*

Publish/Subscribe API

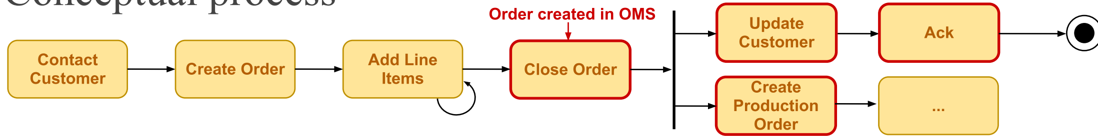
- `lookupTopic`
  - *lookups or creates a topic*
  - *called by the subscriber first and then by the publisher*
- `receiveMessage`
  - *request to receive (read) a message under the topic*
  - *called by the subscriber*
  - *Implementation specific:*
    - → *synchronous – blocking, with timeout*
    - → *asynchronous – through event listener*
- `publishMessage`
  - *publishes a message under the topic*
  - *called by the publisher*

Event-driven Communication

- Event
  - *Occurrence of a message with certain topic*
- Event-driven Process
  - *events trigger actions*
  - *one event may trigger more actions*
  - *loose coupling – not all actions need to be known at design time*
- Service Oriented Architectures (SOA)
  - *Event-Driven Architectures (EDA) – new trends in realization of SOA*

Event-driven Process Example

- Conceptual process



- Event-driven process implementation