# 1 Motivation

### Motivation

There are applications for which it is critical to establish certain availability, consistency, performance etc.

- Banking
- Web mail
- KOS, CourseWare (to some degree)

**Questions**

- **How can we define/measure such non-functional application requirements?**
- **What techniques/tools can we use to provide such applications?**

# 2 Core concepts

### Understanding Core Concepts

- **Mission-critical application** is an application that is essential to the survival of a business or an organization, i.e., failure or interruption of the application significantly impacts business operations.
- Important properties of such an application
  - How well can it be adapted to handle bigger amounts of work?
    - *scalability*
  - How well does it provide useful resources over time period?
    - *availability*
  - What is the rate of processing of the specified workload over the specified time period?
    - *performance*

### Scalability of an application

- **Scalability** is a property of an application which defines
  - how easily it can be expanded to satisfy increased demand for network, processing, database access, file-system resources etc.
  - how well it handles the increased amount of work

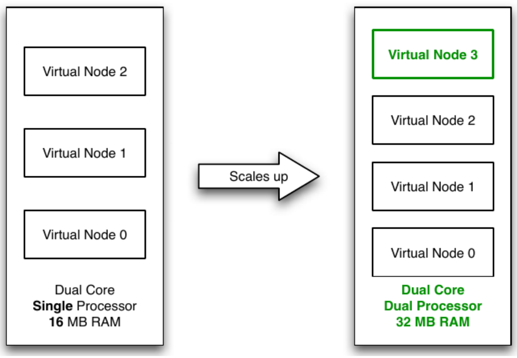


Figure 1: Virtualization Example – vertical scaling of hosting services by increasing number of processors, the amount of main memory to host more virtual servers.

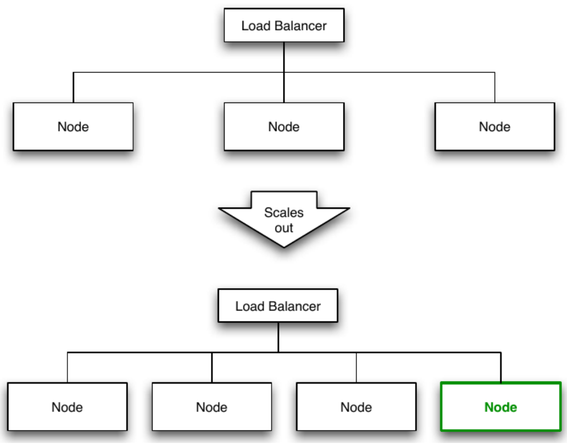


Figure 2: Clustering Example – horizontal scaling of SOA systems/web services by adding more servers nodes to a *load-balanced network*.

- There are 2 ways to scale an application
  - **vertically (scaling up)** – expanding by adding processor units, main memory, storage or network interfaces to a node.
  - **horizontally (scaling out)** – expanding by adding new nodes with identical functionality to existing ones.

### Vertical Scaling Example

### Horizontal Scaling Example

### High-availability of an application

- **Uptime (downtime)** is time during which application is running (not running).

| Availability | Downtime per year | Downtime per week | Downtime per day |
|---|---|---|---|
| *90% ("one nine")* | *36.5 days* | *16.8 hours* | *2.4 hours* |
| 95% | 18.25 days | 8.4 hours | 1.2 hours |
| 97% | 10.96 days | 5.04 hours | 43.2 minutes |
| 98% | 7.30 days | 3.36 hours | 28.8 minutes |
| *99% ("two nines")* | *3.65 days* | *1.68 hours* | *14.4 minutes* |
| *99.9% ("three nines")* | *8.76 hours* | *10.1 minutes* | *1.44 minutes* |
| *99.99% ("four nines")* | *52.56 minutes* | *1.01 minutes* | *8.66 seconds* |
| *99.999% ("five nines")* | *5.26 minutes* | *6.05 seconds* | *864.3 milliseconds* |
| *99.9999% ("six nines")* | *31.5 seconds* | *604.8 milliseconds* | *86.4 milliseconds* |
| *99.99999% ("seven nines")* | *3.15 seconds* | *60.48 milliseconds* | *8.64 milliseconds* |

Table 1: Measuring Availability – vendors typically define availability as given number of "nines".

- **Availability** is defined as the percentage of time an application provides its expected functionality $A = (1 - \frac{t_{unplanned\_downtime}}{t_{uptime}}) * 100$
- Note, that *uptime* and *availability* are different concepts.
- **High-availability** characterizes applications that are obliged to have availability close to 100%.

#### Measuring availability

#### Service Level Agreement (SLA)

**Service Level Agreement (SLA)** defines obligations of involved parties in delivering and using an application, for example:

- minimal/target levels of availability
- timing of reaction (reply to client, fix; based on urgency; e.g. issue A, 1 hour reaction, 8 hours fix)
- maintenance windows
- performance and metrics for its evaluation
- billing
- consequences of not meeting obligations

# 3 Techniques

## 3.1 Load Balancing

#### Load Balancing

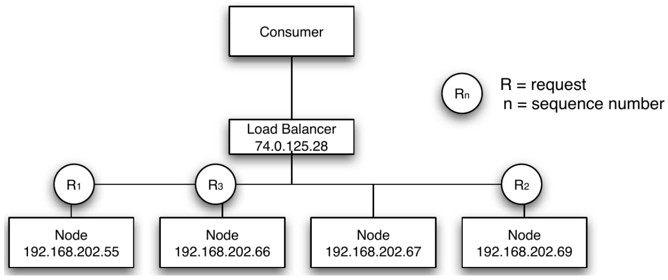


Figure 3: Load Balancer using the Round Robin algorithm.

- **Response time** defines the amount of time it takes a system to process a request after it has received it
- **Latency** is often used to refer to the response time lowered by the processing time of the request on the server
- **Throughput** defines the number of transactions per second that an application can handle
- **Load balancing** is a technique for minimizing *response time* and maximizing *throughput* by delegating requests among multiple nodes
- **Load balancer** is responsible for routing requests to available nodes based on scheduling rules

### Load Balancing

- Distributes client requests or network load efficiently across multiple servers
- Hardware vs Software load balancers
- Load balancing strategies:

  **Round Robin** distribute requests to servers sequentially

  **Least connections** incoming requests are routed to servers with the least load (factoring in server strength)

  **IP hash** IP address of the request client determines target server

### Round Robin Load Balancer

### Persistent/Sticky Session

Stateful applications with server-side sessions require requests from one session to go to the same server.

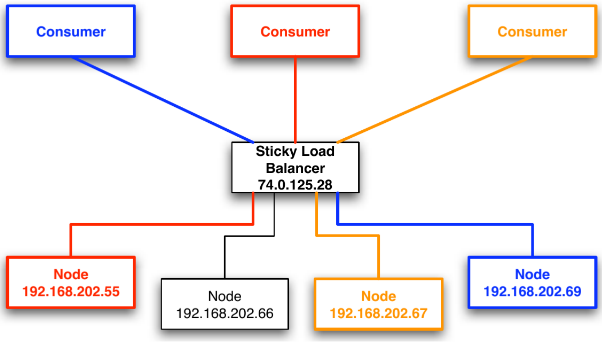


Figure 4: Sticky Load Balancer.

### Common Features of Load Balancers

- *asymmetric load distribution* – different loads are assigned to different nodes
- *priority activation* – if the load gets too high, some standby nodes are activated
- *dynamic configuration* – add/remove servers in server pool quickly at runtime
- *content filtering* – modifies traffic on the way through
- *firewall* – deciding whether traffic might pass through an interface or not based on security rules

## 3.2 Caching

### Caching

**Caching** is a technique for sharing data among multiple data consumers. It is useful for data that are expensive to compute or fetch or do not change often.

- implemented by index tables where *key* is used to retrieve cached entry (datum)
- query for datum using cache can lead to *cache hit* or *cache miss*
- Cache is transparent for its client

### Cache Types

- **application cache**
  - implicit vs. explicit application caching – with little/no participation of a programmer (e.g. Ehcache) vs. using caching API (e.g. Memcached)
- **web cache**
  - *client side* (browser) vs. *server side caching*

  - *web-accelerators* – operates on behalf of the server of origin (e.g. content distribution networks, Akamai)
  - *proxy caches* – serve requests to a group of client accessing same resources. Used for content filtering and reducing bandwidth usage (e.g. Apache)
- **distributed cache** – implemented across multiple systems that serves requests for multiple customers and from multiple resources (e.g. distributed web cache Akamai, distributed application cache Memcached)

#### Cache Strategies

**Read-through** Data are read through cache, if miss, data are read from storage and put into cache

**Write-through** Data are written through cache, i.e., update occurs synchronously in cache and in data storage

**Write-behind** Data are written into cache, update in storage occurs asynchronously after configured delay/when another update to the data occurs

**Write-allocate/No-Write-Allocate** Writing data allocates (does not allocate) cache as well

#### Eviction

- *Index-based* – delete at specified index
- *Random, Round Robin* – delete at random/computed position
- *FIFO (TTL)* – replace oldest item (regardless of access frequency)
- *LRU* – replace least recently used item

#### Write-through with No-write Allocation

#### Write-behind Cache with Write Allocation

## 3.3 Clustering

#### Clustering

- **Cluster** is group of computer systems that work together in a form that appears from the user perspective as a single system
- *Load-balancing cluster* (Active/Active) – distributes load to redundant nodes, while all nodes are active at the same time offering full-service capabilities
- *High-availability cluster* (Active/Passive) – improves service availability by redundant nodes eliminating single points of failures

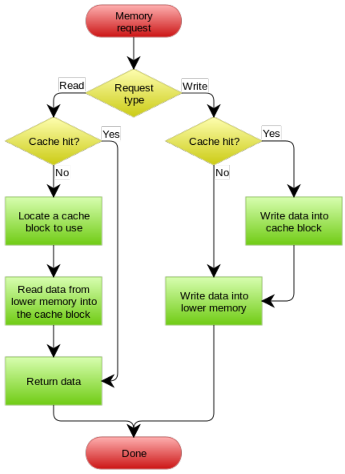


Figure 5: A write-through cache with no-write allocation taken from `https://en.wikipedia.org/wiki/Cache_(computing)`

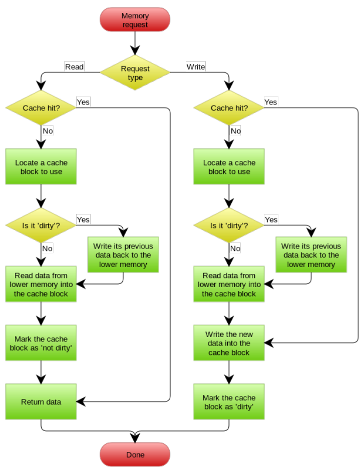


Figure 6: A write-behind cache with write allocation taken from `https://en.wikipedia.org/wiki/Cache_(computing)`

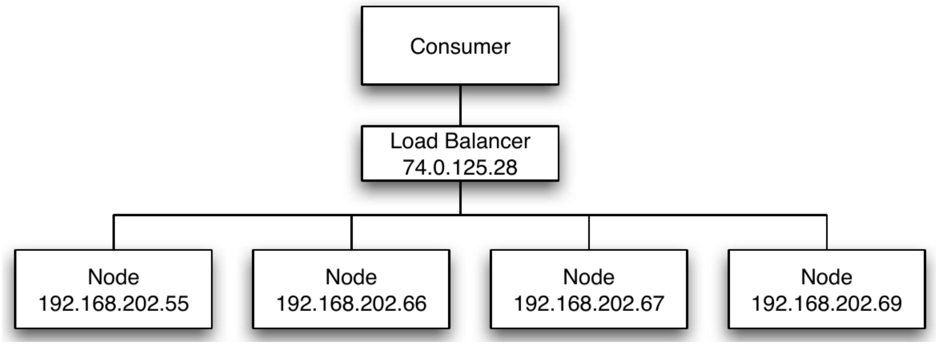


Figure 7: Load-Balancing Cluster (Active/Active)

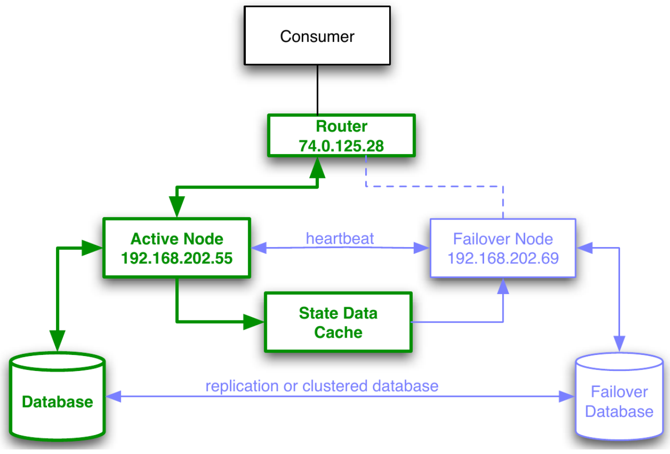


Figure 8: High-Availability Cluster (Active/Passive). It uses "heartbeat" to detect if nodes are ready and routing mechanism to switch traffic if a node fails.

### Load-Balancing Cluster

### High-Availability Cluster

### Principles to Achieve High Availability

- Elimination of single points of failure – adding redundancy so failure of a component does not cause failure of the entire application
- Reliable crossover – ability to switch to from failing node to new node without loosing
- Detection of failures as they occur – failing node should maintain activity, not user's attention.

### Docker + Kubernetes

- Docker allows simple and reproducible node setup
  - same image, central repository
  - containers are configured
  - docker-compose allows bind several services (Java server, nginx, PostgreSQL)
- Kubernetes provides cluster, runs services on multiple nodes
  - configuration
  - load balancing
  - monitoring, fault tolerance (self healing) – restarts services
  - automatic cluster sizing

## 3.4 Cloud Computing

### Cloud Computing

- Keeping a reliable environment with well-configured high-availability is hard!
- **Cloud Computing** is a type of internet-based computing where applications are running on distributed resources owned and operated by a third-party like Amazon.
- *Pay-as-you-go billing*
- Service models within cloud computing :

  **Infrastructure as a Service (IaaS)** use provided infrastructure – virtual machines, servers, load balancers, network, e.g., Amazon EC2

  **Platform as a Service (PaaS)** using provider's services, libraries, tools with control over deployed application – execution runtime, database, web-server, development, e.g. Google AppEngine, MS Azure

  **Software as a Service (SaaS)** using providers application with limited control over the application, e.g., Office 365, email

### On Premise vs IaaS vs PaaS vs SaaS

### System performance testing

- **Performance** refers to application throughput with specified workload and period of time.
- Performance specifications are typically documented in SLA document
- Troubleshooting performance issues requires multiple types of testing such as

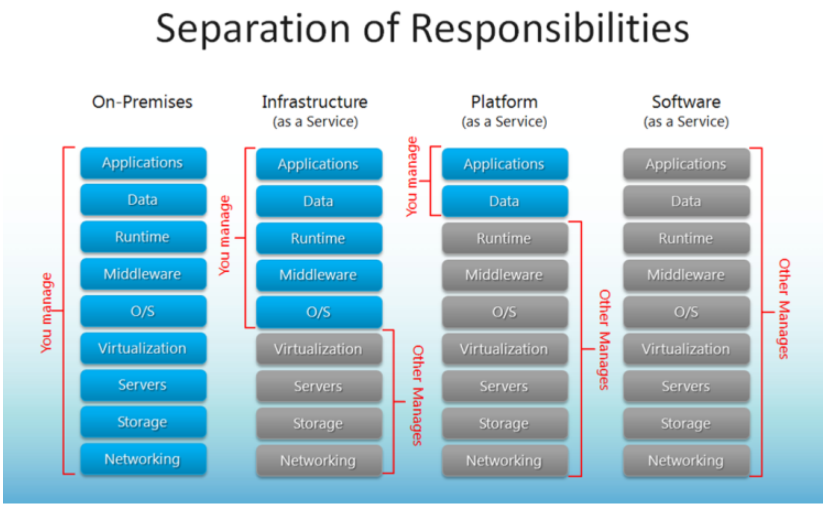


Figure 9: Cloud computing models. Source: `http://robertgreiner.com/2014/03/windows-azure-iaas-paas-saas-overview/`

  - *endurance testing* – identifies resource leaks under the continuous, expected load
  - *load testing* – show application behavior under a specific load
  - *spike testing* – shows application behaviour under dramatic changes in load
  - *stress testing* – identifies the breaking point for the application under dramatic load changes for extended periods of time

# 4 Tools

### Caching

- Java specification SJR 107 – *JCache*
- Spring caching support older, it has its own set of cache-related annotations
- Application cache implementations – Ehcache, Memcached

### Tools for critical-mission applications

- Netbeans Profiler, IntelliJ IDEA Profiler
- JConsole, VisualVM, Java Flight Recorder
- Apache JMeter or Gatling (performance testing by scripts)
- Apache HTTP Server, nginx, IIS (caching, high availability, load balancing)
- EC2 Elastic Load Balancing

| Spring | JSR-107 |
|---|---|
| @Cacheable | @CacheResult |
| @CachePut | @CachePut |
| @CacheEvict | @CacheRemove |
| @CacheEvict(allEntries=true) | @CacheRemoveAll |
| @CacheConfig | @CacheDefaults |

Table 2: Alternative annotations within Spring and JSR-107

## Demo

### Load testing with JMeter

- Testing EAR e-shop with/without Ehcache
- Using JMeter with a number of concurrent clients
- Code in `https://gitlab.fel.cvut.cz/ear/b211-eshop`, branch *ehcache*
- JMeter call: `jmeter -n -t ear-eshop.jmx -l results.log -e -o eshop`

### JConsole, VisualVM

- Connect to EAR e-shop with JConsole, see what is going on there
- Connect to EAR e-shop with VisualVM, see what is going on there

## The End

# Thank You

## Resources


1. https://www.nginx.com/resources/glossary/load-balancing/
2. https://aws.amazon.com/caching/
3. https://docs.oracle.com/cd/E13924_01/coh.340/e13819/readthrough.htm
4. https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html\#cache
5. https://jmeter.apache.org/