Cloud Computing - Swarm communication and programming

THE TECHNICAL UNIVERSITY of CLUJ-NAPOCA

Mircea-Florin Vaida, TUC-N
ETTI- COM Department
Mircea.Vaida@com.utcluj.ro

Overview

- Information Society- Cloud computing evolution

- Integration/Interoperability

- Swarm communication and programming

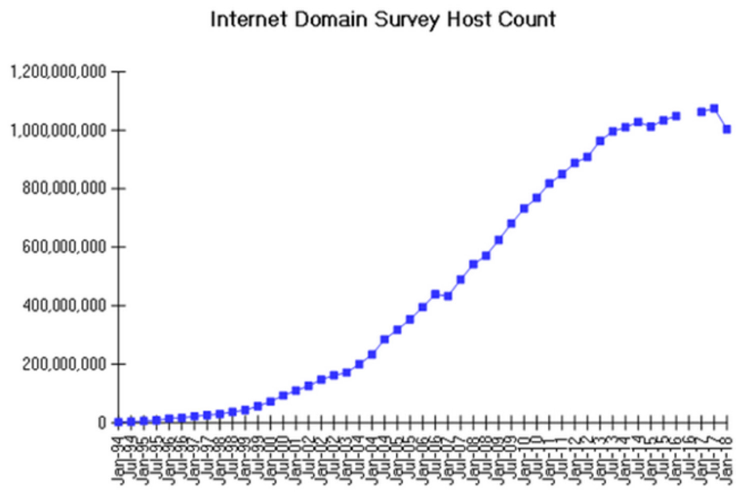
Information Society-
Cloud computing evolution

- Information Society - a new stage with multiple phases in the evolution of human (consumer) society, involving the use of information and communication technologies
  - ***Cloud Computing***
  - *Business Analytics* - ***Gartner*** defines *business analytics* as the use of a set of software applications to build statistical models that help leaders look at data on past business performance, understand the current situation, and predict future scenarios.
  - *Social Web*
  - *Mobile applications development*
  - …

Cloud Computing Evolution

- Cloud Computing is a conglomerate of technologies, and the current form would not have been possible without key moments in the evolution of computer science and distributed systems.
- Between 1943 and 1994 there have been critical developments in the field of computers and computer networks. In 1937 we have the starting point with **Universal Turing Machine (UTM)**.
- From the perspective of the processing unit, the available internal memory, the storage space, the number of available nodes in the network with the emergence of the distributed mechanism and the Internet, we have a spectacular evolution.

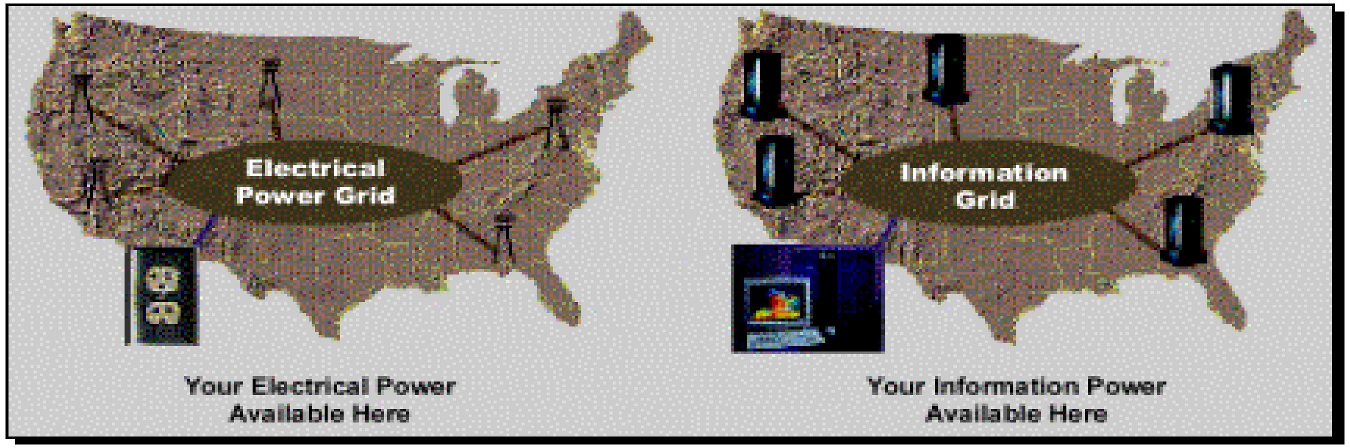
- 1937 - UTM
- 1943 - ENIAC
- 1954 - Fortran
- 1962 - Programma 101
- 1967 - ArpaNet
- 1970 - C
- 1972 - Unix
- 1975 - IBM 5100
- 1978 - TCP/IP
- 1983 - Ethernet
- 1989 - WWW
- 1991 - Internet
- 1994 - Clusters
- 1999 - Grid computing
- 2006 - Cloud Computing



Source: Internet Systems Consortium (www.isc.org)

Information Society| Grid Computing

- 1960: “to enable men and computers to cooperate in making decisions and controlling complex situations without inflexible dependence on predetermined programs.”(Licklider 1960)
- The term Grid emerged at the end of 1990s as public/private, research, P2P, botnet, et. all.
- Analogy to power grids


Electrical Power Grid
Your Electrical Power Available Here
Information Grid
Your Information Power Available Here

Necessity:

- Using unoccupied resources

- About 90% of the power of a processor is not used

- The ability to solve a wide variety of problems at a reasonable cost, Cost/Performance ratio vs. Super computers performance (HPC - High Performance Computer)

- Distributed computing infrastructure originally designed for military, scientific and then industrial projects:

  -Provides support for *searching* and *retrieving* information, regardless of their physical location

  -Allows task *execution on multiple machines*, viewed as a unique computer

  -*Flexible*, *secure* and *coordinated* resources sharing between dynamic collections of individuals, institutions and resources

- Ability to create Virtual collaborative Organizations (VO):

  - possibly *dynamically formed*

  - *sharing applications* and *data* in a heterogeneous open environment to solve various complex problems

  - the existence of a *hardware* and *software* infrastructure that offers permanent, inexpensive access from anywhere in a consistent manner to computing resources

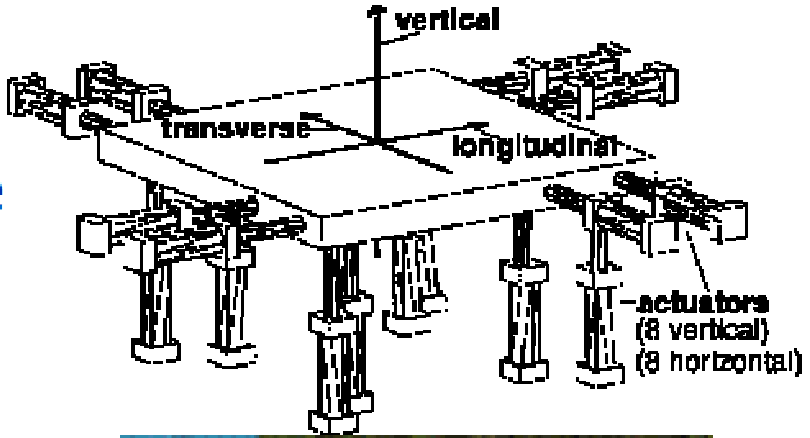
  - a way to *process the information* available on the Internet in a distributed manner

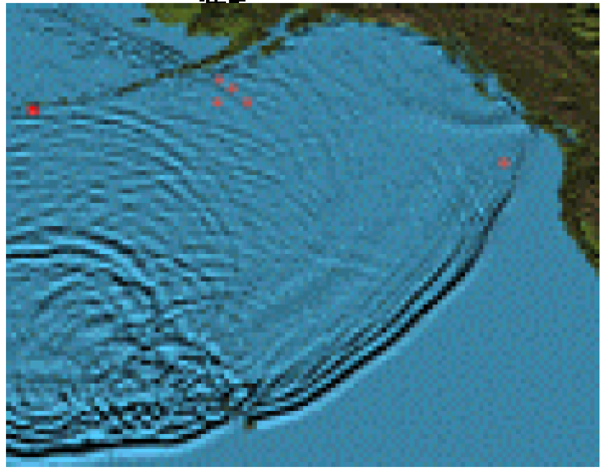
Share: *Computing/processing power, Data storage/networked file systems, Communications and bandwidth, Application software, Scientific instruments*

Grid Computing - Applications

- Earthquake Engineering Simulation

  - NEESgrid: national infrastructure to couple earthquake engineers with experimental facilities, databases, computers, & each other
  - On-demand access to experiments, data streams, computing, archives, collaboration

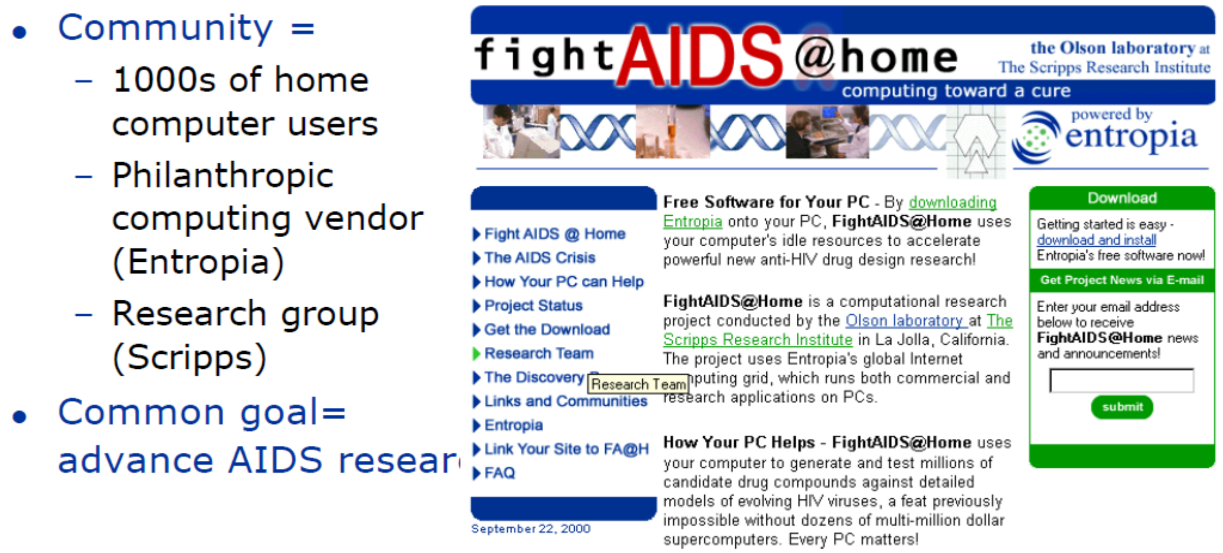




NEESgrid: Argonne, Michigan, NCSA, UIUC, USC

[http://www.nesc.ac.uk/talks/talks/Grids_and_Globus.pdf]

- *Home Computers evaluate AIDS Drugs*



- Community =
  - 1000s of home computer users
  - Philanthropic computing vendor (Entropia)
  - Research group (Scripps)
- Common goal= advance AIDS resear

fightAIDS@home
computing toward a cure
the Olson laboratory at The Scripps Research Institute
powered by entropia

- Fight AIDS @ Home
- The AIDS Crisis
- How Your PC can Help
- Project Status
- Get the Download
- Research Team
- The Discovery P
- Links and Communities
- Entropia
- Link Your Site to FA@H
- FAQ

September 22, 2000

Free Software for Your PC - By downloading Entropia onto your PC, **FightAIDS@Home** uses your computer's idle resources to accelerate powerful new anti-HIV drug design research!

FightAIDS@Home is a computational research project conducted by the Olson laboratory at The Scripps Research Institute in La Jolla, California. The project uses Entropia's global Internet [Research Team]puting grid, which runs both commercial and research applications on PCs.

How Your PC Helps - **FightAIDS@Home** uses your computer to generate and test millions of candidate drug compounds against detailed models of evolving HIV viruses, a feat previously impossible without dozens of multi-million dollar supercomputers. Every PC matters!

Download

Getting started is easy - download and install Entropia's free software now!

Get Project News via E-mail

Enter your email address below to receive **FightAIDS@Home** news and announcements!

submit

[http://www.nesc.ac.uk/talks/talks/Grids_and_Globus.pdf]

- 3D photorealistic viewing
- Medicine: Pharmaceutical Companies (Novartis), Virtual Vascular Surgery (CrossGrid, http://www.crossgrid.org)
- Solving problems requiring high computing, optimization, etc. (e.g., CERN)

Information Society| Grid Computing

- Impact
  - Great in industry and research
  - Very small among ordinary users

  - Generation 1 Grid Computing architecture consists of *protocols*, i.e., services required to ***describe*** and ***share*** available ***physical resources***

  - Sharing ***vision*** and *interoperability* at the *hardware* level

- **Generation 2** (OGSA (*Open Grid Services Architecture*))
- Standards have been developed for *software* sharing and *interoperability* (WSDL, SOAP, BPEL4WS, ...)

  => **Grid services** can be described in a standardized way

  => ***shareable applications***

Features of services:

-Dynamic and Volatile - Multiple Compound Services created, invoked and eliminated "on the fly"

-Ad-hoc - there is no central location or central control

-Large-scale orchestration of a large number of services (> 100) must be done at any time

-Available, potentially long-term (e.g., a simulation may take weeks)

SaaS Paradigm

- **How can the impact on users be increased ?**
- **SaaS Paradigm** (Software-as-a-Service)
  - Designates software that is owned, provided, and managed by a vendor
  - It is consumed on the *pay-per-use* principle via a Web browser or APIs
  - It's a new solution versus traditional software
  - It has developed in parallel with grid computing
  - The *user pays the usage time functionality*
  - The user does not own the software, has not made investments in infrastructure, licenses, etc.

Information Society| Cloud Computing

- **Next step?**

Cloud Computing has gained momentum due to technological competition and the global economic environment emerging from the 2008 crisis. Goes to:

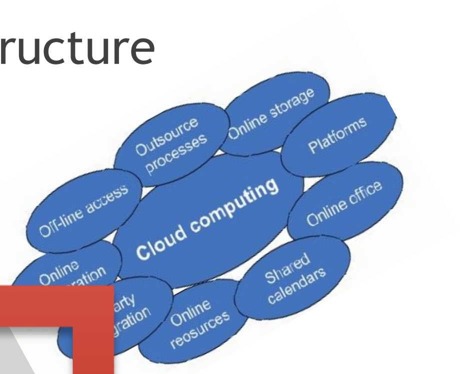
-Scalable, flexible, robust, and reliable **physical infrastructure**

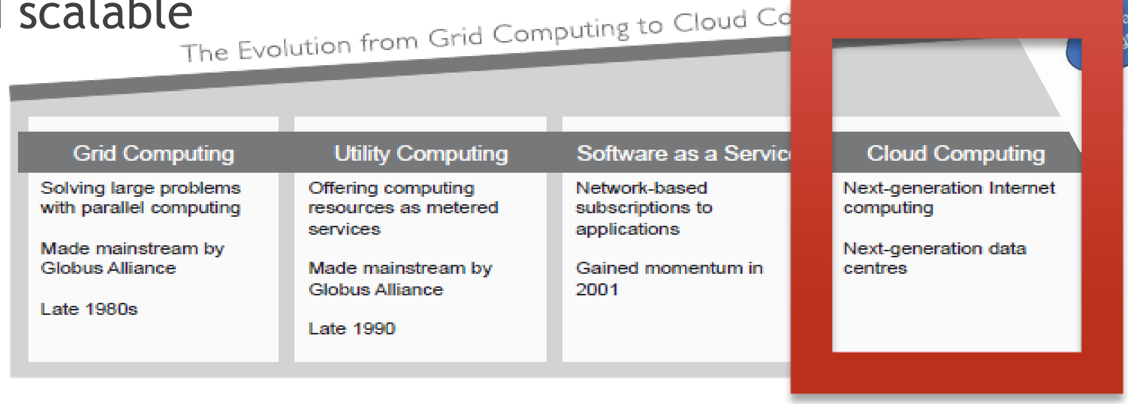
-Services that provide programmers with access to the infrastructure

physics by manipulating **abstract interfaces**

-SaaS developed, deployed, and running on an infrastructure

flexible and scalable





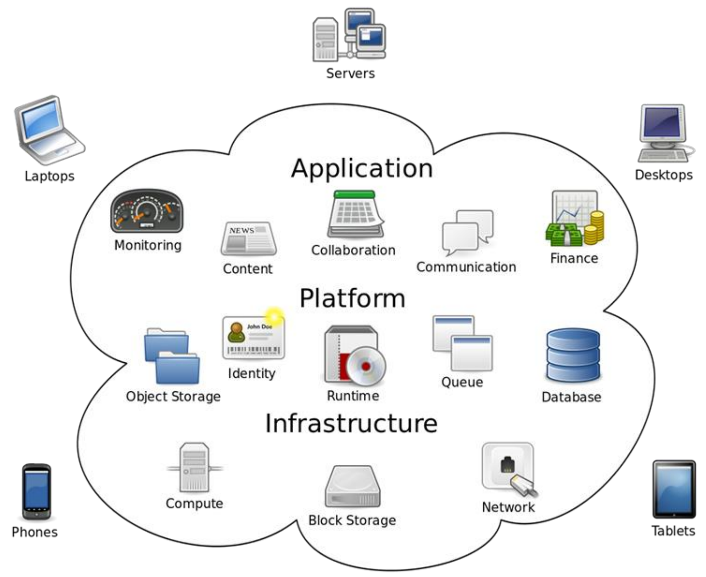
The Evolution from Grid Computing to Cloud Co

Grid Computing	Utility Computing	Software as a Servic	Cloud Computing
Solving large problems with parallel computing	Offering computing resources as metered services	Network-based subscriptions to applications	Next-generation Internet computing
Made mainstream by Globus Alliance	Made mainstream by Globus Alliance	Gained momentum in 2001	Next-generation data centres
Late 1980s	Late 1990		

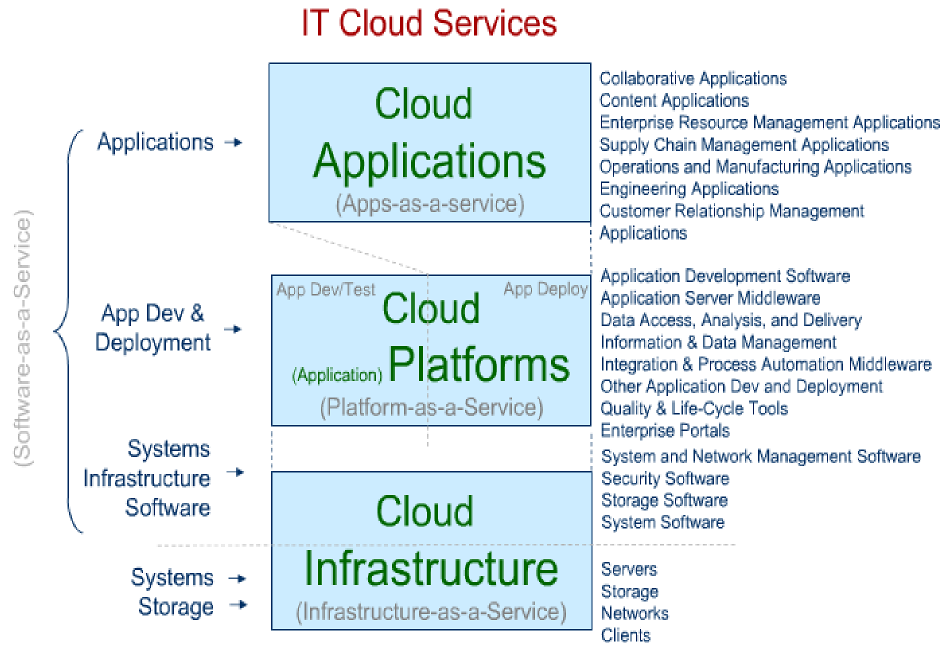
The Evolution to Cloud Computing (adapted from IBM 2009)

[Grid and Cloud Computing **- A Business Perspective on Technology and Applications**, 2010]

- According to Gartner, Cloud Computing is a way to *provide services* that rely on infrastructures with scalability and high reliability
- Manner: pay-per-use
- The services are addressed to all categories of users



Cloud computing


IT Cloud Services
(Software-as-a-Service)
Applications →
App Dev & Deployment →
Systems Infrastructure Software →
Systems →
Storage →
Cloud Applications
(Apps-as-a-service)
App Dev/Test
App Deploy
Cloud (Application) Platforms
(Platform-as-a-Service)
Cloud Infrastructure
(Infrastructure-as-a-Service)
Collaborative Applications
Content Applications
Enterprise Resource Management Applications
Supply Chain Management Applications
Operations and Manufacturing Applications
Engineering Applications
Customer Relationship Management Applications
Application Development Software
Application Server Middleware
Data Access, Analysis, and Delivery
Information & Data Management
Integration & Process Automation Middleware
Other Application Dev and Deployment
Quality & Life-Cycle Tools
Enterprise Portals
System and Network Management Software
Security Software
Storage Software
System Software
Servers
Storage
Networks
Clients

- **Cloud Computing** is defined as "a *large-scale distributed computing paradigm* that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers *over the Internet*."
- Cloud Computing has evolved from Grid Computing and has the Grid reference and infrastructure as support.
- The evolution followed from the provision of infrastructure (computing and storage resources) to the provision of abstract resources and services.
- When Cloud Computing is being discussed, a number of factors are meant to be considered.

Cloud Essential aspects:

- ○ **Utility Computing**: For utility computing to be successful, an *accessible*, *comprehensible* and *practical interface* for the app developers is needed
- ○ **SOA – Service Oriented Architecture**: is a methodology and architecture *designed to maximize reuse of multiple services* (possibly deployed on different platforms and using multiple programming languages).
- ○ **Web Service**: "is self-describing and stateless modules that perform discrete units of work and are available over the network"
- ○ **APIs** -makes it possible to *access the provider`s services* (e.g., Amazon's EC2-Elastic Compute Cloud- API, is a *SOAP and REST* API, used to send commands to create, store, manage AMIs (Amazon Machine Images))
- ○ **Micro-services:** are a software development approach derived from Service Oriented Architecture (SOA). The general principle underlying a micro-service architecture is that you have *many individual services* that are freely coupled, implemented independently, and communicating with each other through protocols such as HTTP.
- ○ **SLA (Service Level Agreement)**: is *a contract between a service provider and a customer* that specifies in measurable terms various characteristics for supplied services. Measurable terms refer to QoS (Quality of Service) that contains a set of mechanisms which point both to the evaluations related to the final client and to technical evaluations.(e.g., bandwidth, availability, latency et.al.)

Cloud properties and features

- **Scalability and Elasticity** - Scalability represents a characteristic of a system, network or process which indicates the ability to *easily handle the increase of data* management.

Elasticity represents an infrastructure *ability to adapt to the requirements* based on real-time analysis methodologies of the entire system.

Elasticity is an ability to *increase the system performance* when the *demands are high* and to *decrease* the performances when demands are *low*.

Scalability and Elasticity are sustained by factors such as dynamic provisioning and multi-tenancy design.

- **Availability and Reliability** - Availability represents the degree to which a system/subsystem equipment is in a *functional state* and is *ready to work* any time. Usually, cloud systems offer high availability (~ 99.999%).

Reliability refers to the ability of a system or component to *perform the required duties* under specified conditions for a specified period of time (fiability).

Availability and Reliability are sustained by factors such as: fault tolerance, resilience, security.

- **Manageability and Interoperability** - Manageability refers to issues closely related to *enterprise management* features tailored to cloud computing systems.

Interoperability represents a *system property to own interfaces that are fully understood* and that enable present and future *interaction with other systems*, without access restrictions or implementation limits.

Manageability and Interoperability are sustained by factors such as: control automation, system monitoring that also imply a billing mechanism.

- **Accessibility and Portability** - Accessibility describes *the level* to which a *product, device, service or environment is accessible* to more customers.

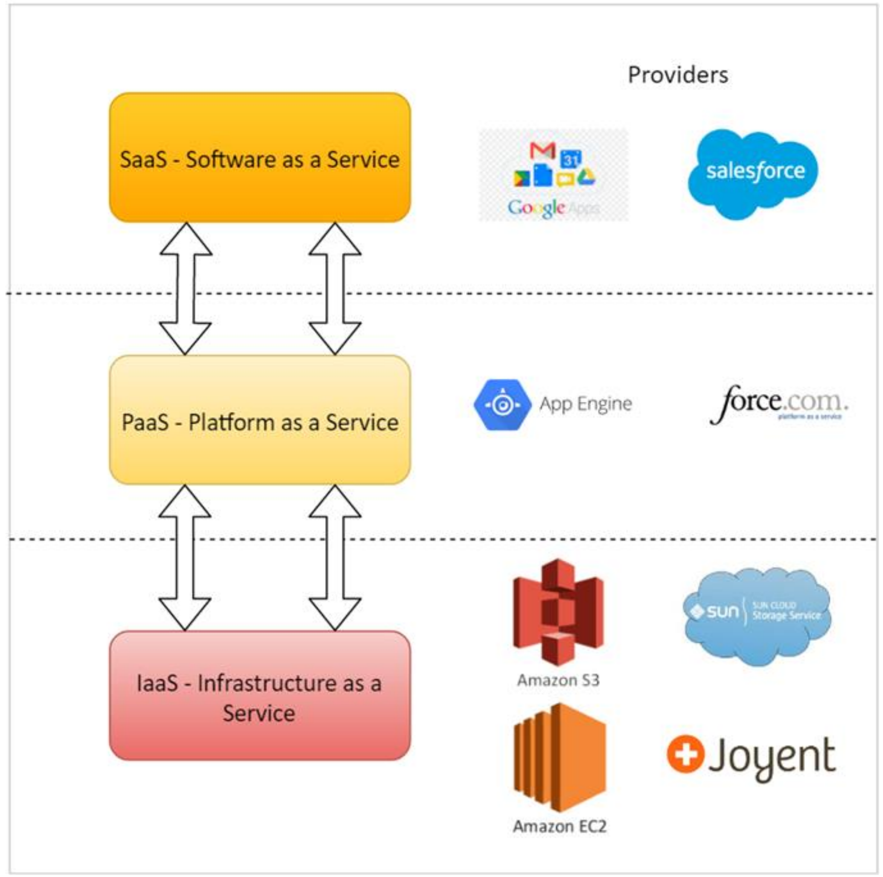
Portability is the ability to *access* the service *using any device*, anywhere, continuously and *in an adaptive manner* to the resource availability.

Accessibility and portability are sustained by factors such as uniform access no matter the operating system/platform/used device for thin client.

- **Performance and Optimization** - Performance in Cloud systems is assured through various approaches from *parallel* and *concurrent processing* mechanisms, *distributed* techniques (e.g., load balancing, tasks planning) to *virtualization* or *Docker* containerization

Cloud main services

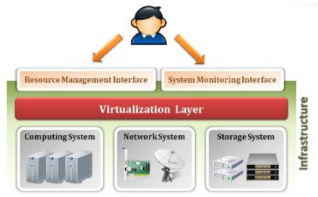
- Cloud Computing Services are divided into three main types of services: Infrastructure as a Service(IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). A cloud architecture can be viewed as a stack formed at first sight from these


Providers
SaaS - Software as a Service
Google Apps
salesforce
PaaS - Platform as a Service
App Engine
force.com
IaaS - Infrastructure as a Service
Amazon S3
sun Cloud Storage Service
Amazon EC2
Joyent

- **IaaS** abstracts Fabric level (raw hardware) named also *HaaS* (Hardware as a Service). IaaS offers virtualized infrastructure as a service, for processing, storage and communication. The consumer has no access to the fabric, but has control over operating systems, storage, application development and configurations related to network.
- **PaaS** is designed for software developers who develop applications according to the specifications of a platform as a set of specific API's, without involving factors related to hardware infrastructure. The platform is one that dynamically allocates resources if the application is widely used. The consumer does not have access to the Management of cloud infrastructure (network, server, operating system or storage), but has control over the applications developed and, on some configurations, related to application hosting. PaaS provides a standardized interface and services for SaaS level.
- **SaaS** is the highest visible level of the Cloud for end-users; it provides software applications exposed as Web interfaces or Web services. (ex. Outlook, Skype, Drop Box, Google Drive, etc.)

Applications are available on a wide variety of thin clients. "SaaS is software that is owned, delivered and managed remotely by one or more providers and that is offered in a pay-per-use manner". Usually, SaaS users do not know the details of the infrastructure.

Cloud Computing | Services



IaaS - Infrastructure as a Service - Virtualization

- Key technology: virtualization
- Virtualization - is the emulation of one or more workstations in a single physical computer

▶ Transform or virtualize computer hardware resources (CPU, RAM, hard disk, network controller) => functional virtual machine that can run its own OS

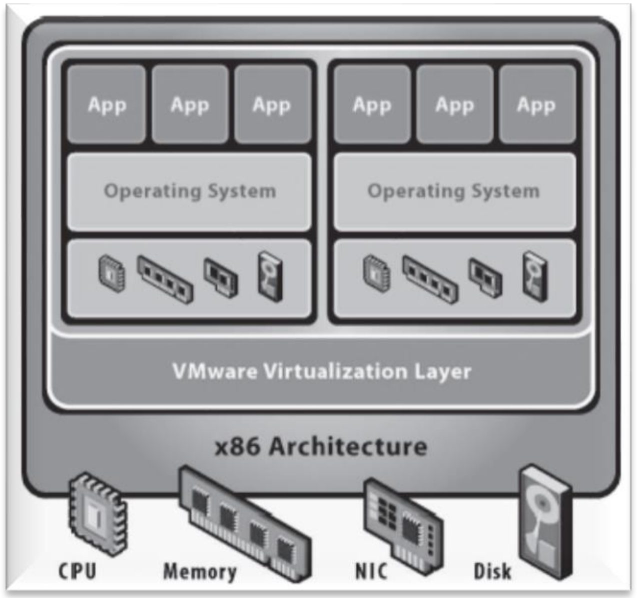
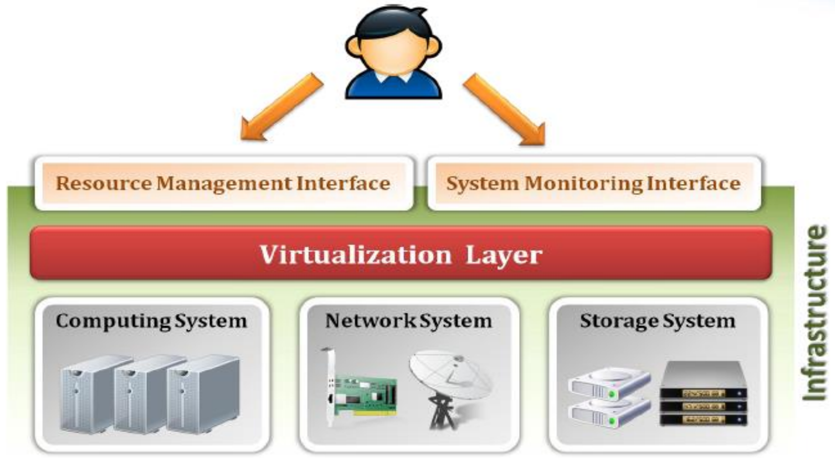


Figure. The architecture of a virtual machine using VMware on an X86 architecture

- 1999 VMware introduced the first virtualization application for systems X86

[Cloud Computing Virtualization Specialist Complete]

Cloud Computing | Services



IaaS - Infrastructure as a Service

The IaaS provider can provide services:

RMI (Resource Management Interface)

- *Virtual Machine* – operations: creation, suspension, restart, termination
- *Virtual Storage* – operations: allocating space, freeing up space, writing or reading
- *Virtual Network* – operations: IP address allocation, domain registration, connection establishment, etc.

SMI (System Monitoring Interface) - examples of monitoring metrics:

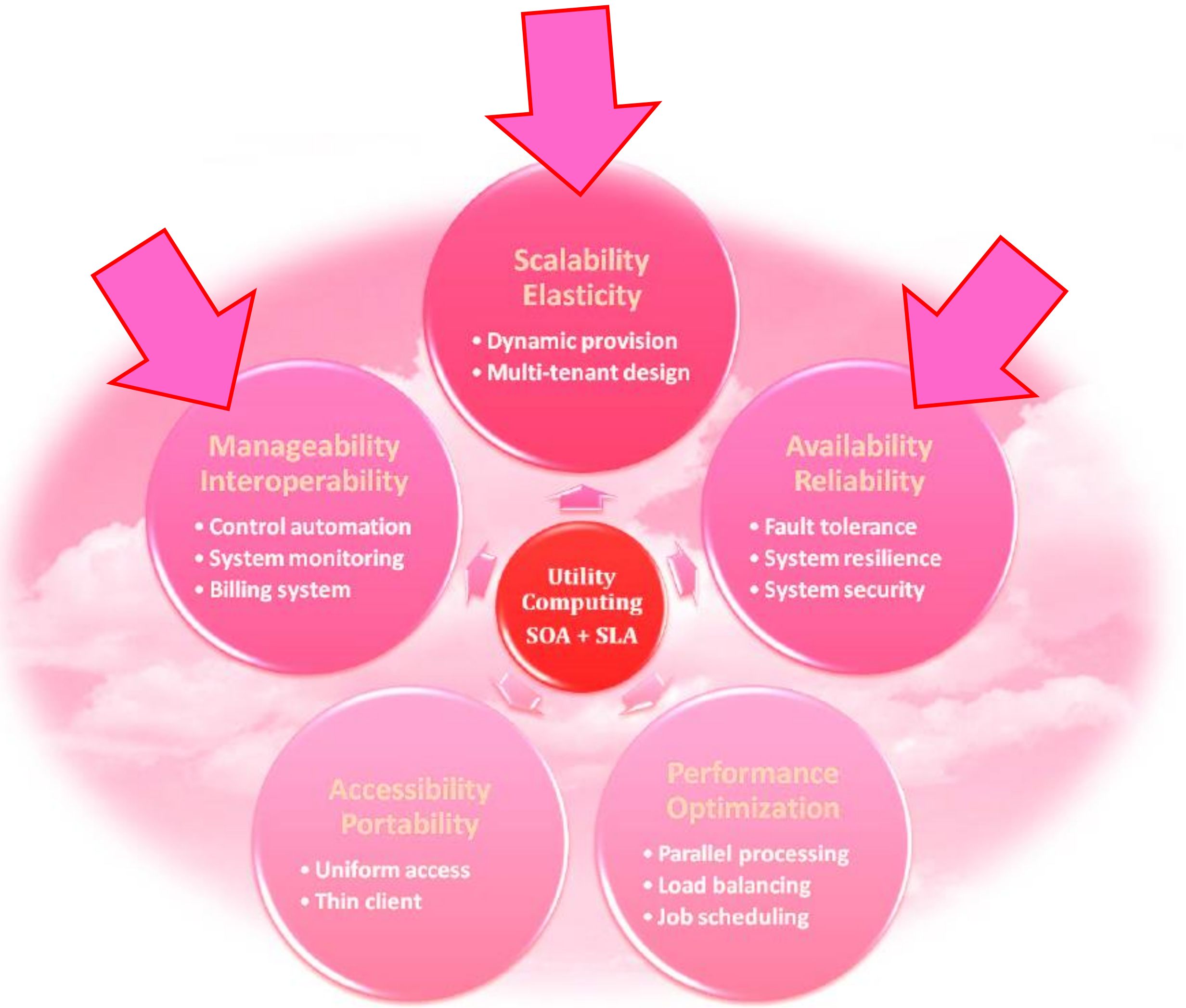
- *Virtual Machine*: CPU loading, memory usage, IO loading, internal network loading, etc.
- *Virtual Storage*: use of virtual space, duplication of data, bandwidth for access to the storage device
- *Virtual Network*: bandwidth for the virtual network, the degree of network load

Cloud Computing

IaaS -Infrastructure as a Service

Virtualization provides support for properties such as:

- *scalability and elasticity*
- *availability and reliability*
- *manageability and interoperability*



Cloud deployment models

- **In terms of data center owner there may be:**

● **Public Cloud**

A Public Cloud contains *data centers managed by third parties*, e.g., Google, Microsoft, Amazon, which expose their services to companies and consumers via the Internet

● **Private Cloud**

For Private Cloud the cloud infrastructure is used for a *single organization*, but the cloud management can be done by another organization. Also called *internal cloud*, or *on-premise cloud*, it is based on the virtualization of the existing infrastructure of the organization and this implies more efficient use of resources. The Private cloud appeared due to the requirement to limit the risks associated with a Public Cloud

- **In terms of how many cloud environments are integrated (*multiple-Cloud environments*):**

● **Community Cloud**

Multiple organizations share their cloud infrastructure to achieve a common goal. Also called *Federation of Clouds*, each cloud is independent, but can *interoperate* (exchange data and computing resources) with other clouds through standard interfaces

● **Hybrid Cloud**

The infrastructure consists of *multiple clouds* (private, community, public) that remain unique entities, but they are linked together to standardized or proprietary technologies that ensure data and application portability (e.g., in situations where load balancing is required)

Integration Platform as a Service

- If, until recently, the main focus was on intra-organizational integration and ESB (Enterprise Service Bus) that were playing an essential role, the focus has changed on a new type of cloud service named iPaaS - ***integration Platform as a Service***.
- iPaaS is *included in the stack of cloud services* at an essential level. As such, during *complex applications*, it could benefit from the characteristics offered at IaaS and PaaS level (*workload distribution, availability, redundancy, et. al.*)
- An iPaaS system is designed to be optimized for different kinds

of *processing*, for example transaction of huge amount of data

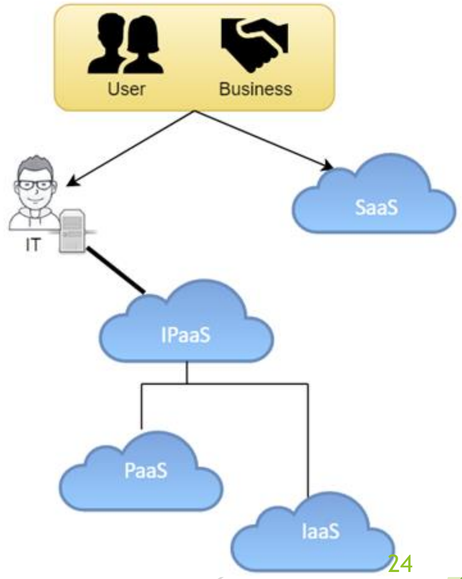
with low latency.

The *limitations* and *risks* associated with iPaaS

(vendor lock-in, privacy issues, governance, et.al.)

go hand in hand with the limitations of a Cloud platform

in general.



Cloud Applications

- The app architecture determines the level of performance and in many cases, its level of security and privacy in a program. Despite the fact that there are numerous programming techniques ensuring high level of proficiency, some architectural decisions may lead to their limitation. There is no universal architectural solution, and the choice of the model should be made depending on the needs of the application.
- In distributed systems, *a first step* towards IT outsourcing comes with the idea of

Web Applications that can be provided by a central supplier (one-to-many delivery model). The main problem was:

- the *inability to provide personalized services*
- problems (issues) regarding *scalability, robustness*.

In this case, we are talking about the paradigm of Application Service Provisioning (ASP) that *failed*, while, in the background, the first steps in the evolution of the internet and the web technologies were being taken.

- But ASP problems could be solved by using Grid Computing and Web Services.
- Web Services allow services *personalization* and Grid Environment offers *flexibility and scalability*.
- Therefore, the *background to design applications* with many-to-many delivery models has been created.

Furthermore, providing computational power in pay-per-use manner leads to new business models for utility computing.

Also, there were many initiatives at software level, which led to the concept of SaaS.

- The *next step*, the one of Cloud apps, has come naturally and we can talk about the *following characteristics*: *scalability, robustness* and *reliable* physical infrastructure; *services* that provide developers access to infrastructure by manipulating abstracted interfaces; SaaS running on a flexible and scalable infrastructure.
- All these steps of architectural and technological evolution have determined the presence of various architectural patterns like:

  - monolithic architectures and,

  - micro-services architectures.
- The monolithic architectures contain components which are tightly-coupled (interconnected and interdependent, therefore compilation/and the execution of the app must be accomplished if *all its components are present*).

Many of traditional web applications follow such an architectural pattern, since many of IDEs and developing tools were meant to be created like this. Only some frequently used schemes are to be subsequently mentioned: a *Java Web* application consists of a WAR (Web application ARchive) file that runs on Tomcat, or a *PhP* application that runs on Apache/Nginx server and has access to a MySql database.

- The fact that these types of architectures are still to be used on a large scale is due to a series of *benefits* such a solution may bring:

● **simple to deploy**

● **scalability with additional costs**

● **cross-cutting concerns**

● **test** and **debug** on a monolithic application implies fewer variables to be taken into account

- The trend in which tightly-coupled apps are replaced by *loose-coupled* apps appeared due to some aspects which are difficult to be solved in a traditional manner.
- Tight-coupling implies:

  - an increase in complexity because the change of a component may lead to the behavioral change in a different component and thus it is getting more and more complicated for maintenance and developing

  - continuous deployment: although the *simplicity in terms of the delivery process* is an advantage for this architecture, the recompilation and reloading of the application requires time and computational resources. This advantage turns into a *disadvantage in case of frequent reloading's* or in case you have to redeploy just one component, but you are forced to stop background tasks which have nothing in common with the changed carried out. There are situations when these not updated components can fail at a new restart. The easiest example for continuous deployment is user interface programming that implies actions such us iterative changes and frequently re-deploys.

- Basically, a monolith architecture is *beneficial* to apps which do not require any scaling, present tight-coupling properties, do not have a highly complex structure and modification/upgrade operations seldom occur.

- In *the opposite case*, when these requirements are not met, the architectural choice must be based on services, or micro-services oriented in order to address the above-mentioned aspects, or additional issues such as:

-a monolithic application is difficult to understand and modify and therefore the development activities of new team members are severely impeded

-resources overload - a more complex code or application implies IDE slowing down or wasted time until the application container starts. Diminishing impact on development is registered in this case, also.

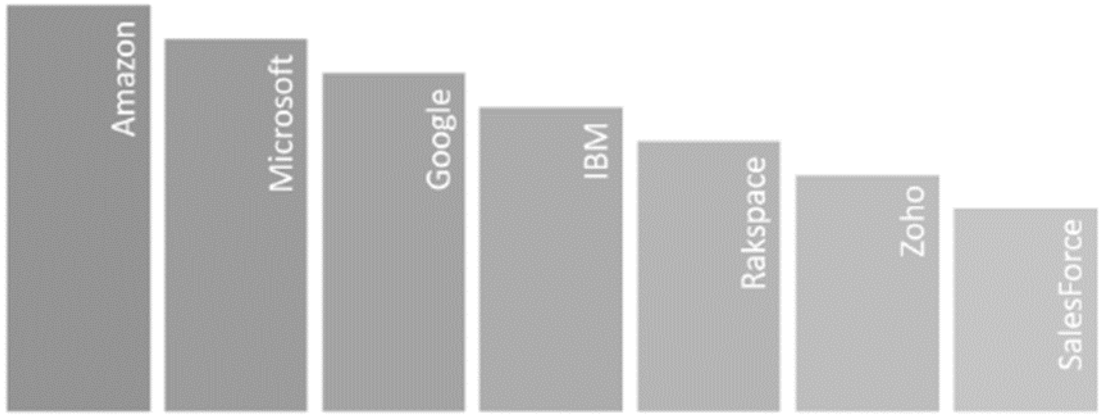
-scaling development impediment - when an application reaches a certain level of complexity, there are *necessary teams* able to work independently, covering *various domains*: UI team, security team, database team, et.al. In a monolithic application a special team should exist in order to coordinate development activities and re-deployments.

-lock-in in a technology stack - a monolithic application constricts you within a technology (or a special version of that technology) and the adoption of a new one is a painstakingly and risky process.

- In this context, the SOA architecture-based systems as well as micro-services-based applications are able to be used.

Cloud Computing Providers and Consumers

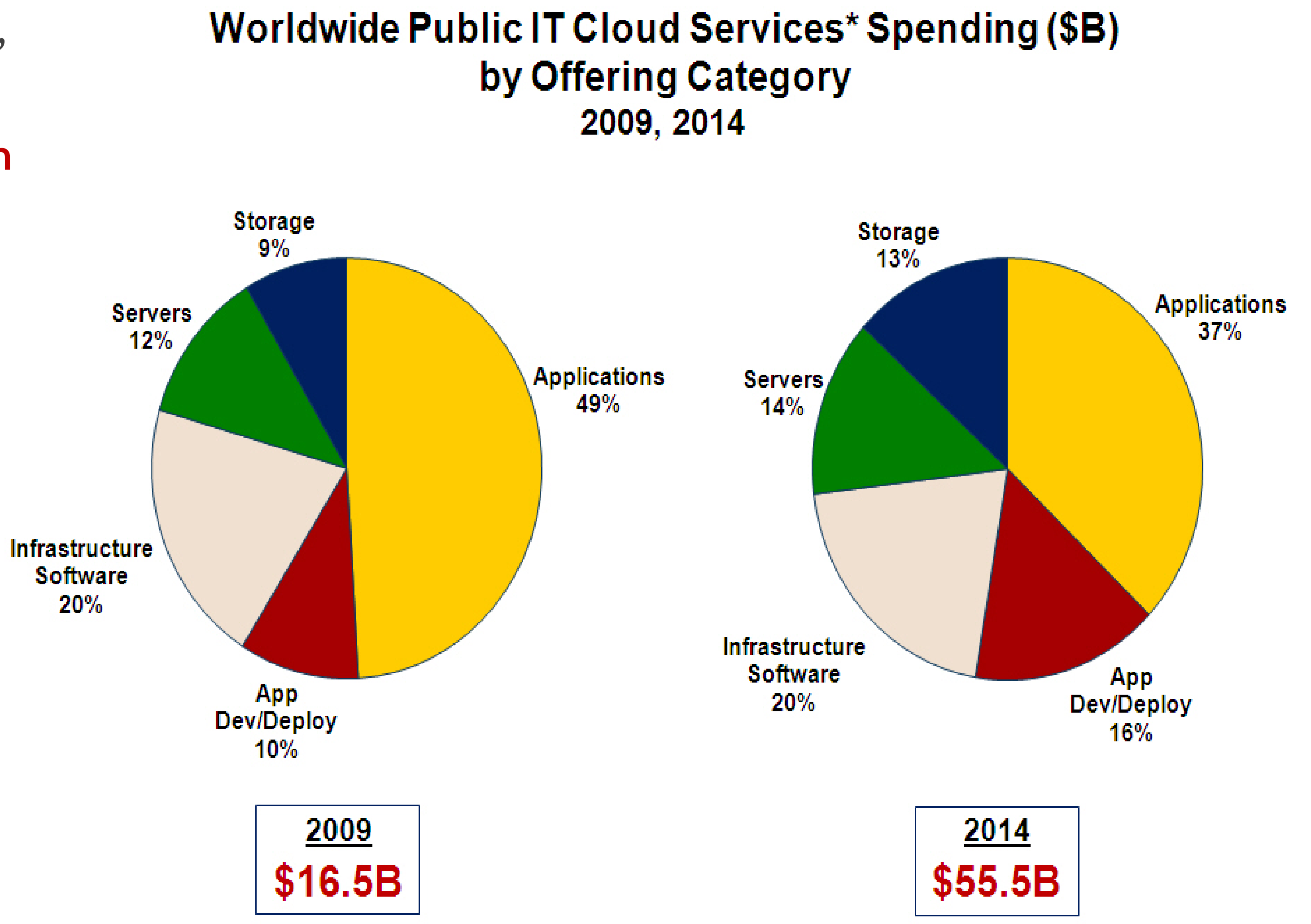
- **Cloud main providers:**



- **Cloud consumers:**
  - *End consumers*
    - Uses cloud services via a Web browser
  - Example: Google Apps (Google Mail, Google Drive, Google Spreadsheets, ….), Live Mesh (Microsoft), Social Networks (Facebook, Twitter,...), AEC2 - Amazon Elastic Compute Cloud, etc.
  - *Business customers (e.g., Companies)*
    - Access Cloud services to improve their own infrastructure with on demand resources or to use various available applications as services
  - *Researchers/Developers*
    - Access cloud services at different levels and become service providers

Cloud Computing

- Perspective
- From beginning, and **in 2018 at $160 Billion**

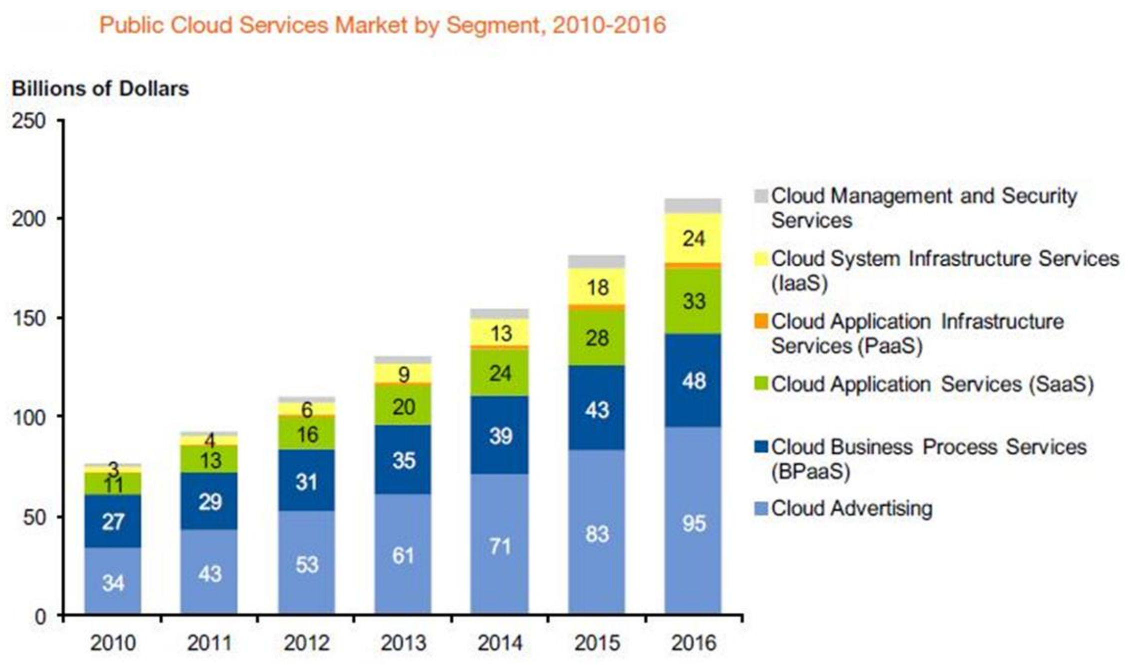


Source: IDC, June 2010

* Includes spending on Applications, Application Development & Deployment Software, Systems Infrastructure Software, Server capacity and Storage capacity provided via the public Cloud Services delivery model.

Cloud Computing Services Market. Social Impact

If you have a mobile device (tablet, smartphone, ...) then you are a few clicks away from the information you want



[http://blog.jaguarpc.com/general/personal/top-5-examples-of-cloud-computing/]

Cloud Computing | Impact on education and research

Research and Education Institutes have adopted or want to use Cloud infrastructures

Reality: Infrastructures with high computing power are not found in the most elitist institutions

What Cloud brings:
-accessing data and complex applications by researchers/teachers /students regardless of location
-the use of external cloud infrastructures - using academic licenses: Amazon, Microsoft (Azure), Google (GAE)

Cloud Computing Perspective

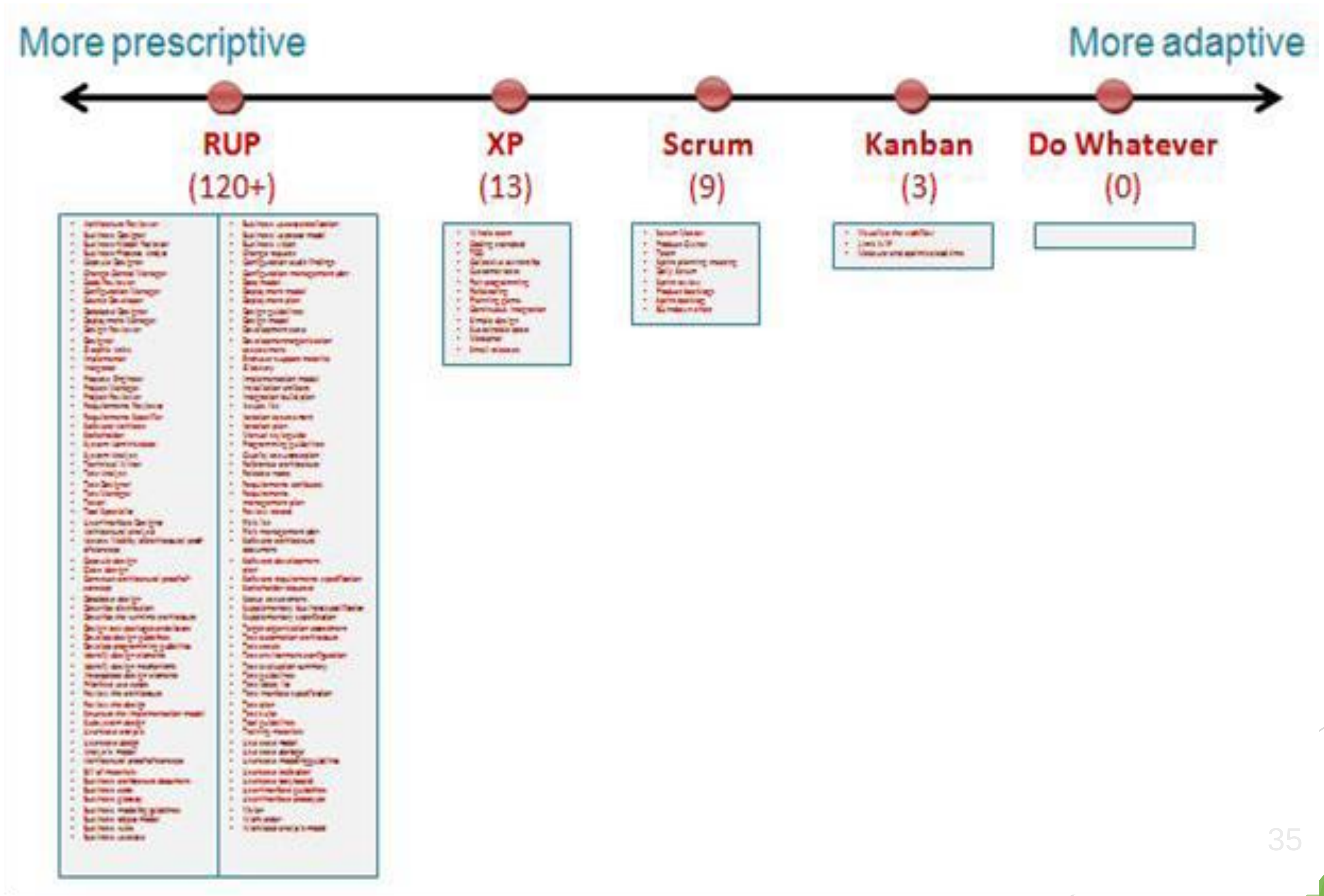
- Perspective:
  - Large companies are building private cloud to solve security problems
  - Coding and development skills will take a less prominent position in relation to project management, quality assurance, business analysis
  - IT departments will shrink
  - Cloud information will require security measures equivalent to money security in the bench
  - Small to mid-sized businesses will switch to cloud
  - Large businesses can become part-time cloud providers
  - The browser will be the desktop we need
  - Games will be of great interest as cloud applications

[http://www.focus.com/briefs/top-10-cloud-computing-trends/]

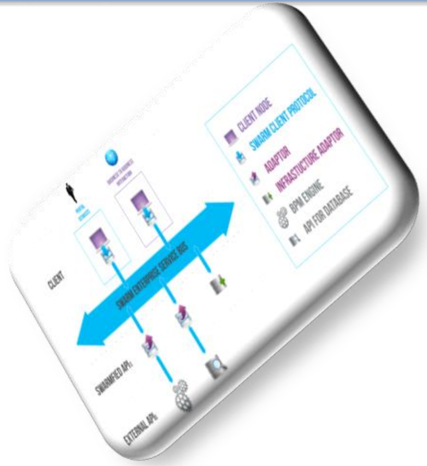
Integration - a starting point key concept

- Started from the ‘90ies, the concept of Integration was implied in the syntagm Continuous Integration coined by Kent Beck, as part of eXtreme Programming.
- eXtreme Programming (XP) is a methodology employed in software development intended to *increase software quality* and also *to adapt dynamically* to the client requirements. XP appeared against RUP (Rational Unified Process) (https://techterms.com/definition/rup)
- Continuous Integration refers actually to a first step in a *Continuous Delivery* process, which assures that the work-in-progress code belonging to a project is continually integrated with code belonging to other developers from the same project
- Integration is defined in **Gartner** as follows: “Integration services are detailed design and implementation services that *link application functionality* (custom software or package software) and/or data with each other or with the established or planned IT infrastructure. *Specific activities* might include project planning, project management, detailed design or implementation of application programming interfaces, Web services, or middleware systems.”

Software methodologies development



Integration



- Import/Export files from one system to another
- Using the same database
- Enterprise Systems Integration:
    - Complex Ad-Hoc Middleware
    - MOM (Message Oriented Middleware)
    - ESB (Enterprise Service Bus)
- Cloud System Integration: iPaaS
    – MuleESB, SwitchYard, SnapLogic Integration Platform , …

So, another proposal using various specifications for orchestration?....No☺

Integration/Interoperability

- Many times, concepts like *integration* and *interoperability* are used interchangeably, but they *are different*.
- When we talk about interoperable systems, there is *an environment* in which *sharing the information* is carried out in the original context, *without* the need of any *middleware*.
- This *middleware* appears in the *integration* process and it may be provided by a third party as *iPaaS*, for instance.
- Even though, at first sight, interoperability seems a good solution, there are situations when a(n) change/upgrade of a party can cause interoperability issues or even the end of communication.
- Furthermore, *interoperability* implies *general principles of communication* rather than a technical specification.
- This latter case is to be discussed within integration area which ensures a full functional system that provides backwards and forwards compatibility with various versions of the product, a reliable data transfer, automated maintenance and other characteristics which vary depending on the integration systems.
- In conclusion, interoperability offers *an exchange* and integration offers *a full functionality.*
- Next, we will focus on the Swarm communication and programming.

Swarm Proposal

– Swarm Communication has the same goal as SOA: compose services behaviors

– Differences:

- In Swarm based systems
  - there is no central controller that manages the business logic and messaging sequence
  - the nodes that represent the participant services *do not have predefined interfaces* regarding communication with other nodes
    - » There is no node that know the whole business logic as in case of orchestration

– The business logic is an orthogonal aspect for *nodes* represented by *executable choreographies*

SOA to Swarm development steps

- The expertise in the field of Cloud Computing and distributed architectures has been a long-term process and the solutions suggested for different contexts have kept up with the technical evolution worldwide. The years 2011-2012 represented an anticipation moment concerning the technological trend combining Cloud Computing (iPaaS, serverless architectures) with micro-services.
- The development of an e-learning system in collaboration with an education specialized research team and a company, took two directions: contributing to the *design of the system's* architecture and, identifying a *solution to integrate* a new system among other software enterprises developed by the company up to the point of ERP - Enterprise Resource Planning and CRM - Customer-Relationship Management systems and a document management system.
- That has represented a starting point in finding an approach meant to use an existing SOA solution and the design of an alternative aimed at offering a practical environment for the entire team.
- SOA solutions are usual of a huge cost and not suitable for low-cost applications.
- *A SOA project* needs programmers, SOA architects, business process modelers, ESB - Enterprise Service Bus experts, user interface designers, tests, security experts, testers, administrators of the tools within the stack, data architects and many other skills related to the specific business best practices.
- That is why it was proposed a swarming approach which is not only *a method of programming using asynchronous messages* but an architecture for ESB like systems.
- Swarm based architecture introduces only a few intuitive concepts like phases, adapters, groups, swarm description and swarm primitives, and it has a huge benefit for rapid learning.

Swarm Communication

- Orchestration is seen like a system where a conductor coordinates every person in an orchestra. The coordinator (the intelligent node) is the one who indicates during a concert (achieving a task) the way each member in the orchestra (each node in a distributed system) must act (accomplish an action). Therefore, we have a centralized system.
- Choreography is seen like a space where dancers (the nodes in a distributed system) accomplish some actions according to a set of rules that they have previously learned and now they collaborate in order to perform a dance (accomplish a task). In this situation, the system is not centralized anymore.
- Swarm communication, aligns with this last scenario and also adds a new element: *the nodes* (dancers) that are part of the system, act based on *rules that they do not initially know*, rules brought by *smart messages* that will visit each node. Metaphorically, we can see the dancers as being ”inspired” and acting according to the specifications from the messages.
- After finishing the task, they will return to the initial state or they will be ”inspired” again (by a new message). This represents a new approach.
- Swarm communication is inspired by Nature and Mobile Code and involves Internal Component Modeling, Software Service Composition, Micro-Services, Asynchronous Control, Integration, Business Processes, Smart Contracts

- Swarming, as was defined, is different from service orchestration because there is no central controller that manages the business logic and there is no messaging sequence.
- It is also different from service choreography because the individual nodes that represent the participant services do not fully control the answers to requests, they do not know the entire business logic as in the case of choreography.
- We can say that swarming is somewhere in the middle, with more *smart messages* and less intelligent service nodes. The logic is distributed in *swarm descriptions* and the effective execution and decisions take place in individual nodes.
- The *lack of a central orchestrator* makes a swarm-based system inherently more *scalable*.
- By using swarm communication, a system will have *the ability* to *dynamically scale*, meaning it will be capable of handling variations in capacity requirements in an automated mode.

- All the existing commercial and open-source solutions that address the integration problem were developed based on the ESB architecture, like MuleESB, Apache ServiceMix, SwitchYard (former JBoss ESB), OpenESB, TalendESB.
- There are many similarities, but also differences between them, because they take different approaches to achieve the same goal.
- A comparison can be made regarding many key features of an ESB, like *transports* and *connectors'* support (JMS, XMPP, AMQP, SOAP, TCP, FTP, JDBC etc.), *security* plugins, *web services* support (Axis, CXF, REST, WSDL etc.), *transformation mechanisms* (XSLT, Smooks, etc.), *BPM integration*, etc.
- In *[AAP13]* it was proposed a *new paradigm regarding communication mechanisms and composability of services*. It brings a new perspective compared to service orchestration and choreography and also to MOM (Manufacturing Operations Management) and ESB systems. This approach was named swarm communication.
- The goal of swarm communication, as in the case of service integration patterns from SOA, is to *compose services' behaviors*.

- *[AAP13] L., Alboaie, S. Alboaie, and Panu, A., Swarm Communication - A Messaging Pattern Proposal for Dynamic Scalability in Cloud, 15th IEEE International Conference on High Performance Computing and Communications (HPCC 2013). IEEE, pp. 1930 – 1937 (2013)*

Swarm programming

- The syntactic description of a swarm closely resembles the description of a class.
- We have in the swarm description declarations of *variables (attributes)* and membership *functions (methods)*, but we have a special concept called *phase*.
- A *phase* is a *function*, but its execution is done only by calling a special primitive, even called *swarm()*.
- Basically, *instantiating a swarm* creates an object running in a *node*, but every *call of the swarm primitive will clone the current object* and thus an instance, *followed by the migration* of the new object to another location. (like a *stub* in Java RMI/IIOP based on RPC protocol)
- The concept of location is more elaborated in *PrivateSky* project, as the support for swarm concept.
- In this way, we can perceive *swarm programming* as a simple extension of *Object-Oriented Programming, OOP,* through *mobile code* facilities.
- However, the difference is greater because many of the typical OOP programming patterns do not have meaning, because the semantics of execution is changed by code migration.
- Typically, *an instance of a swarm* is not a single object but *a swarm of objects*.
- The concept of identity gets a new and interesting technical meaning and maybe even a philosophical dimensions.

Swarm communication

- swarm description *(variables, functions, **phases**)*
- swarm instances
- ***swarm()*** primitive
- explicite locations
  - *Adapters (SwarmESB)*
  - *PrivateSky project support:*
    - *Security Context - agents*
    - *Other companies systems*
    - *Intern sub-components*

```
vars:{
 x:int,
  ....
}
phase1: function(){
 ...
 swarm("phase2")
...
}
phase2: function(){
  ..
}
...
```

Object-Oriented Programming?

OOP extensions : with **mobile code**

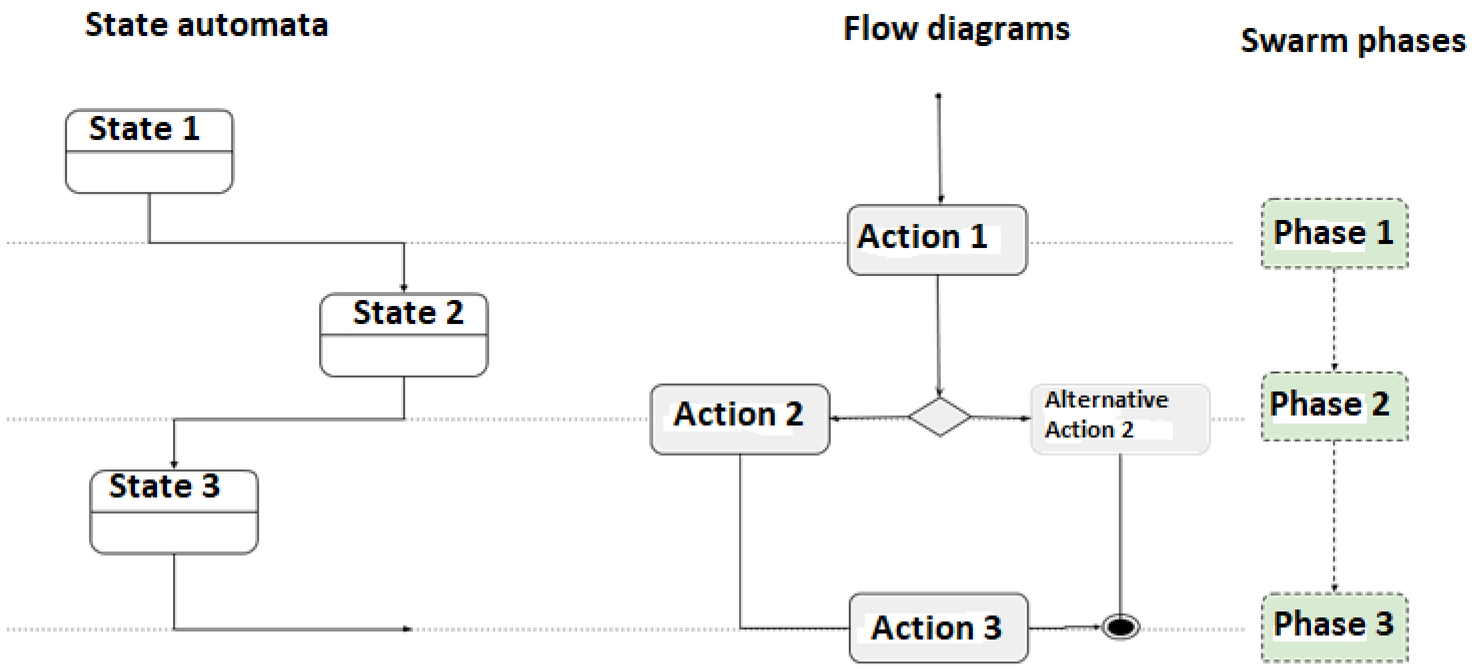
Benefits of Swarm programming

- To explain one of the benefits of swarm programming, we call for comparison with state automata and flow diagrams.
- State automata suffer from a problem called *the "explosion" of the state*, being somehow *a syntactic artifact*.
- Flow diagrams express a very *detailed level of semantics*, but it shows a problem similar to the state explosion.
- Swarm programming manages to better *reflect the level of pragmatics* and is therefore better suited to use by programmers.

On the other hand, the explicit existence of the

phases, *highlighting* on the one hand *the* most important *scales* in the state machines and,

the code place (location) where the primitive *swarm* is used, *highlights the actions* that may affect security, or private data protection.



Swarm communication

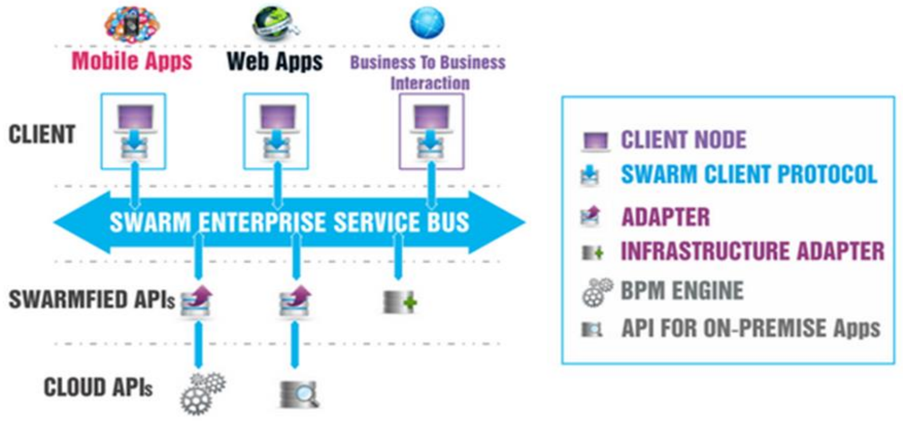
Swarm System

- Swarm's can be both a code-writing technique (description of algorithms and data structures, creation of abstraction on the Object-Oriented Programming line), as well as a general method of easy implementation of *Executable Choreographies* and *Smart Contracts*.
- A swarm system is doing real time processing for a stream of requests using swarm descriptions.
- Swarm descriptions are used to launch swarms that are visiting distributed nodes to perform computation. A *swarm description* can be seen as a ”program” describing swarm processes that are executed when a request is made.
- Swarms can be integrated with existing queueing technologies, database technologies, or web services.
- A swarm-based system is similar in some respects to *Storm*, a distributed real-time computation system. Storm reliably processes unbounded streams of data using a network of intelligent nodes, called topology. A Storm topology represents a complex multi-stage stream computation which runs indefinitely.
- A key difference between Storm topologies and Swarm description is that a swarm will always finish and will not generate more communication and computation, whereas a Storm topology processes messages forever or until it is killed.
- In practice it was met many communication patterns (e.g., request/reply, request/reaction etc.), but *swarms bring something new*: when a use-case appears, all related messages can be automatically *grouped* in a simple representation.
- This is a significant advantage to code maintenance and offers support for developer intuition regarding messages flow. When we mention ”use-case”, we envision that a communication use-case represents a set of complementary messages sent to solve a specific task. This *set of complementary messages forms a swarm*.

Swarm communication description

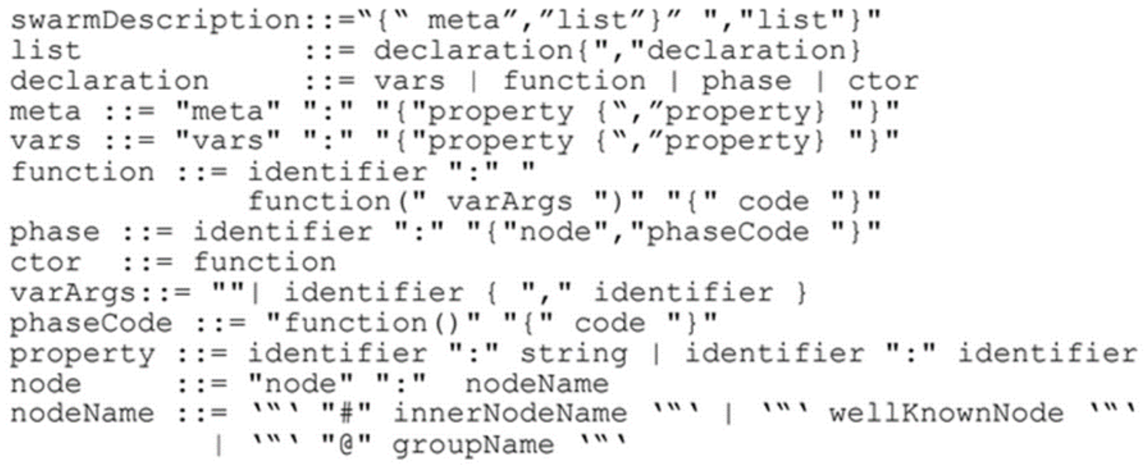
- The communication is performed between nodes of the distributed system.
- In this environment a ***node*** is a *software entity* that can *receive* or *send swarm messages*.
- From SOA perspective, a node includes an *endpoint* functionalities having a unique address, a specific contract and of course containing services, which provide specific functionalities.
- Swarm concept of nodes is nesting all these aspects, making it easier for developers to mentally handle them.
- In the swarm communication model, each node has a *unique identifier* that we usually refer to as name of the node (***nodeName***).
- The communication mechanism is accomplished through the nodes ”talk”: a node is talking to another node by sending a *swarm()* primitive.
- Sending a *swarm()* primitive means that it sends a message that is part of the set of swarm messages. Returning to our metaphor inspired by nature, the ”talk” is made not by words but with ”small beings” that *have intelligence* and *take decisions* instead of being just a method for information representation.
- In other words, in time, a swarm goes through a set of states (called them ***phases***) to accomplish a goal.

- Such *phases* contain code applications that change the internal state of the swarm or the state of the nodes in which the swarm got executed. It is important to notice that *the phase's code is not part of the node*, but the code of the swarm itself, even if it is executed in the context of a node, is. Except for having a name (*phase* name) and a node hint, a *phase* is the behavior (code) that should happen (execute) when a message is received by a node.
- We distinguish three types of nodes (see Figure): *adapter* nodes, *client* nodes and *infrastructure* nodes.
- *Adapter nodes* are *server-side nodes* that provide some services or APIs for the existing applications (e.g., CRM, ERP etc.).
- The nodes that offer *system core functionalities* are called *infrastructure adapters*.
- *Client nodes* are *logically connected to an adapter* by communication protocols created over TCP sockets or other protocols.



Swarm DSL (Domain Specific Language)

- Swarm DSL as a Domain Specific Language *is a first proposal*, which depicts the flow of messages that occurs between nodes in an implicit and dynamic way. This language enables an *intuitive* description of communication patterns, based on intuitive primitives we are going to present below.
- For *syntax* description, it was used BNF(*Backus-Naur Form)* notation.
- Swarm DSL is an internal DSL so all JavaScript syntactic and semantic rules should be considered. By using an internal DSL, we can benefit from the existing tools for debugging, IDEs (Integrated Development Environment) and programming expertise, therefore we reduce adoption risks for this new technology. Swarm DSL language is presented below:



```
swarmDescription::=“{“ meta”,”list”}” ","list"}"
list                ::= declaration{","declaration}
declaration         ::= vars | function | phase | ctor
meta ::= "meta" ":" "{"property {“,”property} "}"
vars ::= "vars" ":" "{"property {“,”property} "}"
function ::= identifier ":" "
             function(" varArgs ")" "{" code "}"
phase ::= identifier ":" "{"node","phaseCode "}"
ctor  ::= function
varArgs::= ""| identifier { "," identifier }
phaseCode ::= "function()" "{" code "}"
property ::= identifier ":" string | identifier ":" identifier
node     ::= "node" ":"  nodeName
nodeName ::= `"` "#" innerNodeName `"` | `"` wellKnownNode `"`
           | `"` "@" groupName `"`
```

- Therefore, a swarm description can contain four construction types:
- *vars section* in swarm description is *a way to document and initialize* the variables used by the swarm messages
- *swarm phases* represent *the swarm progress* towards accomplishing a goal. Such phases contain code applications that change the internal variables of the swarm or the state of the adapters in which the swarm got executed. It is important to notice that the phase's code is not part of the adapter, but code of the swarm itself, even if it *is executed in the context of the adapter*. Except that having a *name* (phase name) and a *node hint* (an indication of the node where the swarm should be executed), a *phase* is the *behavior (code) that should happen* (execute) when a message is received by a node (adapter). A single swarm can execute code *concurrently* in multiple nodes and therefore multiple phases can be concurrently "alive". Swarm v*ariables* are *not automatically shared between those concurrent executions*, so that programmers exercise to implement communication between swarms. This characteristic turns our system into an environment meant to *develop oriented architectures on micro-services*.
- *meta variables* are special variables that *are reflecting the current execution of the swarm* and can be handled by programmers to influence or control the execution of the swarm in various ways
- *functions* and *ctors*: *ctor* functions, or *ctors,* are functions that are called when a swarm is started (similar to *constructors* in OOP languages). A *ctor* function will initialize the swarm variables and will start swarming in one or multiple phases. Normal functions are *utility functions* made available to swarms' code for a greater modularity and code reuse between phases: *swarm(), startSwarm(), startRemoteSwarm(), home(), broadcast()*, et al.

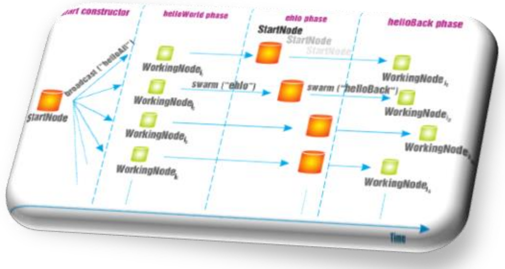
- As described in BNF grammar, for *nodeName*, it is possible to have three possible values:
- 1) *innerNodeName* - represents *the name* for inner nodes. Inner nodes use resources from a normal node, but they *do not have any visibility outside* that node. Sending a swarm to an inner node does not produce network traffic. Sending a swarm to an inner node is exactly like sending the swarm to a current node, and this feature can be used during development to compare execution times between remote and local communication or when planning what services (adapters) are required.
- 2) *wellKnownNode* - can take a well-known node name *as value*. Such nodes represent the *infrastructure nodes* for our application (e.g., *Core* - the node that is the source authority for all swarm descriptions, *Logger* - the node where logging information is stored from all nodes, *SessionManager* that is responsible for managing nodes that keep information about sessions).
- Well known nodes can be problematic for scalability (on high load, we could cause bottlenecks) and can be used for a while until the load increases and should be replaced by groups of nodes. Therefore, *to enable load balancing, replication and high availability* we have introduced the concept of groups. Each adapter can join a number of groups by using a *join()* primitive. By joining a group, a node becomes accessible to nodes that know the group name but are not aware of their name.
- 3) *groups* - to enable different grouping of nodes or to avoid the above-mentioned situations, we have introduced the concept of groups. As notation, we set that group *names will always start with character "@"*. Therefore, nodes can be grouped in *fleets* or *groups of nodes* with similar functionality or by geographic location or other criteria. Sending a swarm to a group with a *swarm()* primitive means that one node is chosen, and the execution will continue there. If the chosen node is down, another node is tried until the swarm succeeds in finishing the task;

- In the grammar, we assume that *identifier* and *code* are valid JavaScript identifiers and respectively valid JavaScript source code that can appear in valid JavaScript functions.
- Below is presented a ”swarmified” Hello World example using Swarm DSL:

```
    start : function(){ this.swarm("concat");}
    concat    : { //begin phase
        node : "Core",
        code : function (){
          this.message = this.message + "The swarming has begun!";
          this.swarm("print"); }},
    print : { //print phase
        node :"Logger",
        code : function (){
        console.log(this.message);},}
}
```

- In this example the swarm execution contains two *phases*: *concat* and *print*.
- The phase’s execution is performed in two different nodes *Core* and *Logger*. These nodes are different processes, possibly located on different machines, depending on system configuration. A swarm starts through a *ctor* execution and this can be done on any node of the system.
- For this example, the constructor is called *start* and his role is to lead the swarm in *concat* phase using the *swarm()* primitive.
- According to our discussion, as we can notice in this example, each *phase* declaration has two components: *node* and *code*.
- In *node*, we specify the location where the phase will be executed.
- To exemplify *concurrency,* we should understand that even if a new phase is running, the current phase could still exist for a while. Of course, the variables in these two concurrent phases are not shared (are just copies) but this mechanism usually serves us well without any risks of interferences or performance penalties.

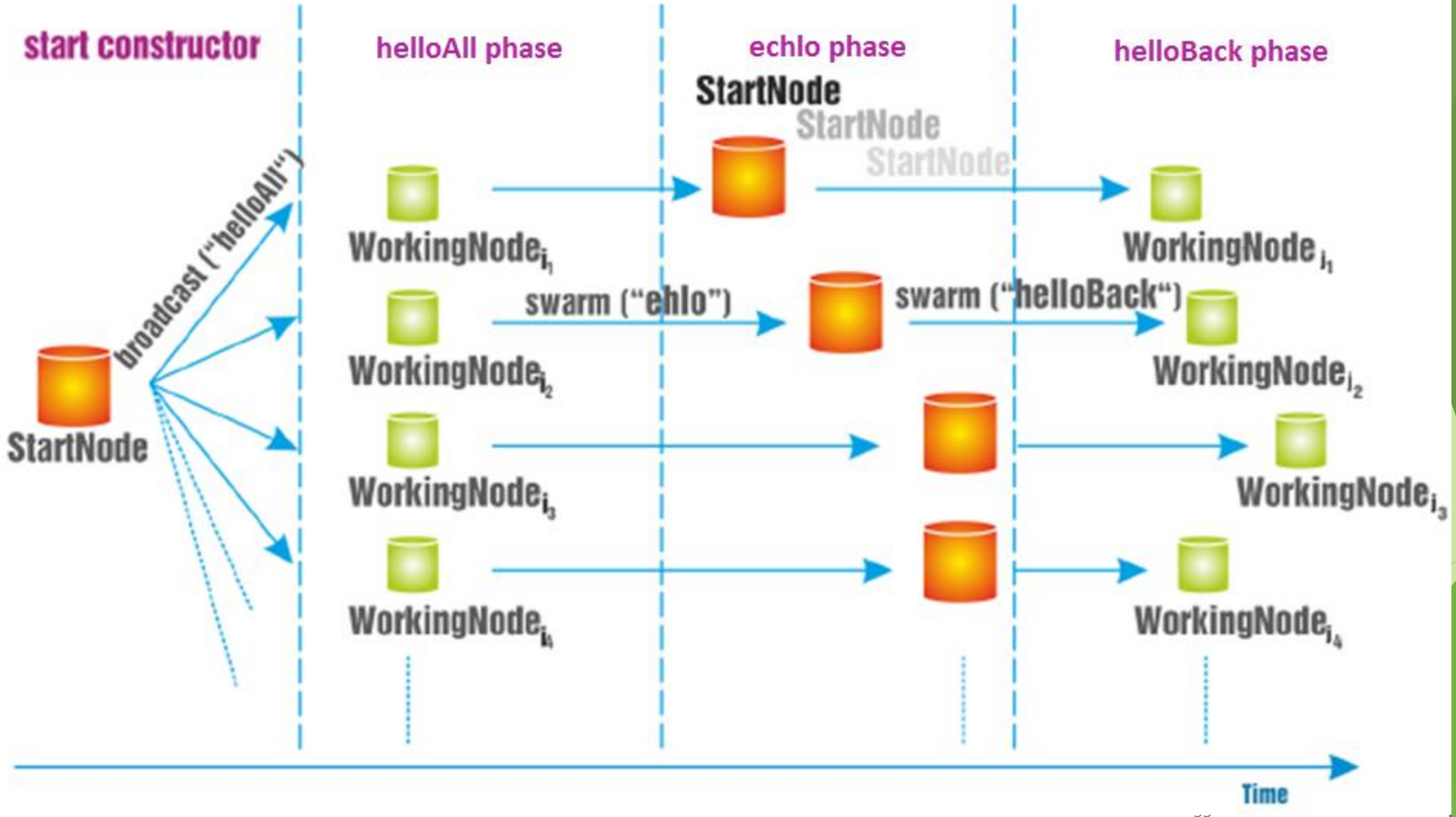
Swarm System – as programming approach - *[AAP13]* Echo example

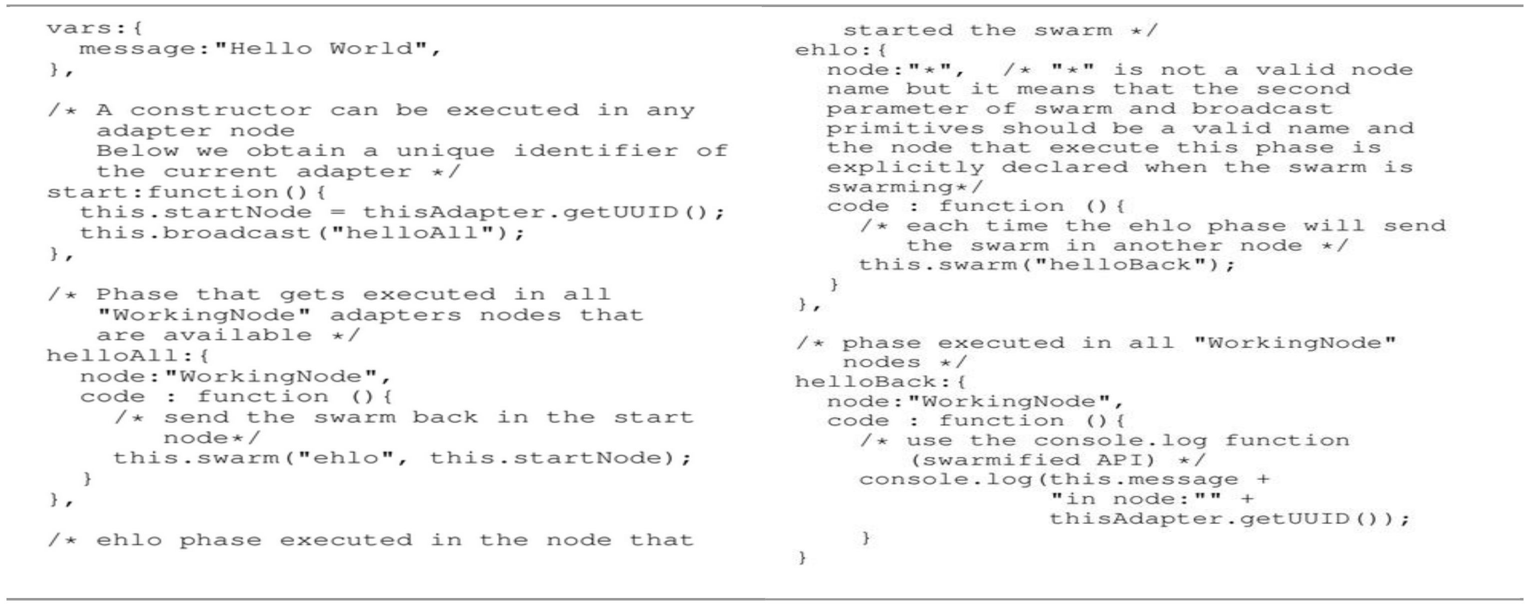


```
vars:{
      message:"Hello World",
},
start:function(){
     this.startNode = thisAdapter.getUUID();
     this.broadcast("helloAll");
},
helloAll: {/*phase */
      node:"WorkingNode",
      code : function (){
      this. swarm("ehlo", this.startNode);
     }
},
```

```
ehlo: {/*phase */
     node:"*",
     code : function (){
     this. swarm("helloBack");
     }
},
helloBack: {/*phase */
      node:"WorkingNode",
      code : function (){
     console.log(this.message +
          "in node:"" +
          thisAdapter.getUUID());
      }
}
```

It is assume that it was already started an arbitrary number of nodes with name *WorkingNode* and the following swarm is launched in the system:





```
vars:{
  message:"Hello World",
},

/* A constructor can be executed in any
   adapter node
   Below we obtain a unique identifier of
   the current adapter */
start:function(){
  this.startNode = thisAdapter.getUUID();
  this.broadcast("helloAll");
},

/* Phase that gets executed in all
   "WorkingNode" adapters nodes that
   are available */
helloAll:{
  node:"WorkingNode",
  code : function (){
    /* send the swarm back in the start
       node*/
    this.swarm("ehlo", this.startNode);
  }
},

/* ehlo phase executed in the node that
   started the swarm */
ehlo:{
  node:"*",  /* "*" is not a valid node
  name but it means that the second
  parameter of swarm and broadcast
  primitives should be a valid name and
  the node that execute this phase is
  explicitly declared when the swarm is
  swarming*/
  code : function (){
    /* each time the ehlo phase will send
       the swarm in another node */
    this.swarm("helloBack");
  }
},

/* phase executed in all "WorkingNode"
   nodes */
helloBack:{
  node:"WorkingNode",
  code : function (){
    /* use the console.log function
       (swarmified API) */
    console.log(this.message +
                "in node:"" +
                thisAdapter.getUUID());
    }
}
```

In implementation, a swarm description is a source code file written in a subset of JavaScript language. A *swarm description* is for a swarm what a *class* is for objects (class instances).
The name of this file will give a common characteristic to all instances created on it and we refer to this characteristic as *the type of the swarm*.

Swarm Benefits

- Swarm programming is using familiar technologies (NodeJS)
  - Many SOA projects based on orchestration are failing because used technologies are hard to master by big mass of programmers
- Approaching the limits of Moore’s law, swarms can provide an appealing programming approach for parallel computing
  - *Asynchronous message programing* is recognized as a better programming method compared with *threads-based programming*

Executable Choreographies as a Proposal

- What is Choreography?

Metaphor: choreography is for a dance show what orchestration is for a concert

It's a service composition method without a centralized conductor

- Why Executable Choreography?

Unlike WS-CDL (Web Services Choreography Description Language), it was proposed and implemented a language that can be employed and executed on a large scales

Proposed Solution for Integration Problems

- Intra-Organizational implementation: **SwarmESB**
- Inter-Organizational (Federated) implementation: **PrivateSky Platform** - https://github.com/PrivateSky

Usages:

- Swarm ESB, was used in **Industry Projects** as an Enterprise Service Bus (e.g., the Redpoint S.A. - Document Management Project (2012), S.C. Axiologic SaaS - The BuzzCenter(2013))
- **Research projects**:
    - SwarmESB was used in USMED (Operational Sectoral Program for Increased Economic Competitiveness) (2013-2015)
    - SwarmESB is used in a research European Project OPERANDO – H2020 (http://www.operando.eu/) (2015 -2018)
    - PrivateSky Project – POC (2016-2021)

OPERANDO

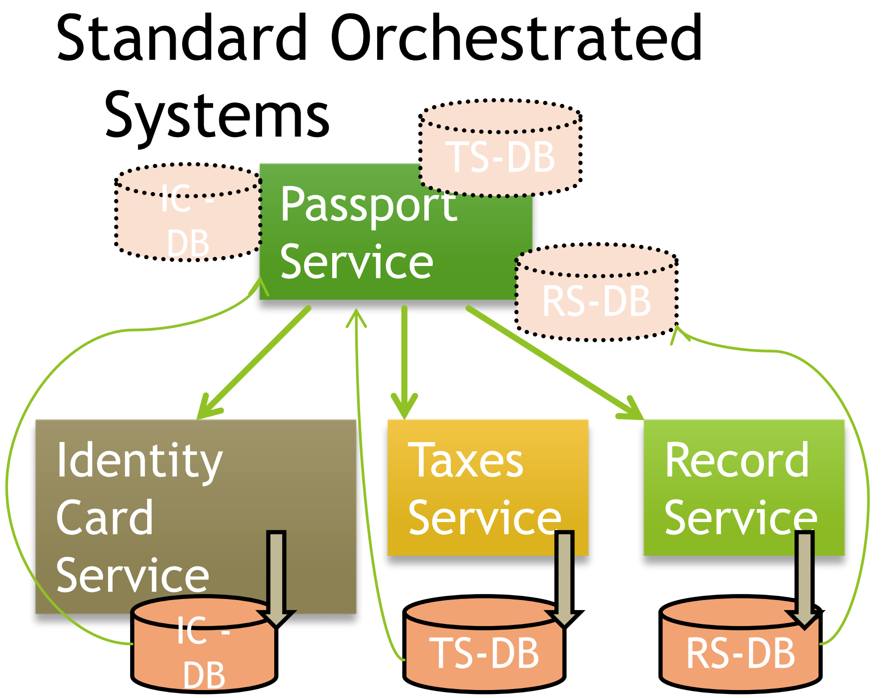
- H2020 Research Project

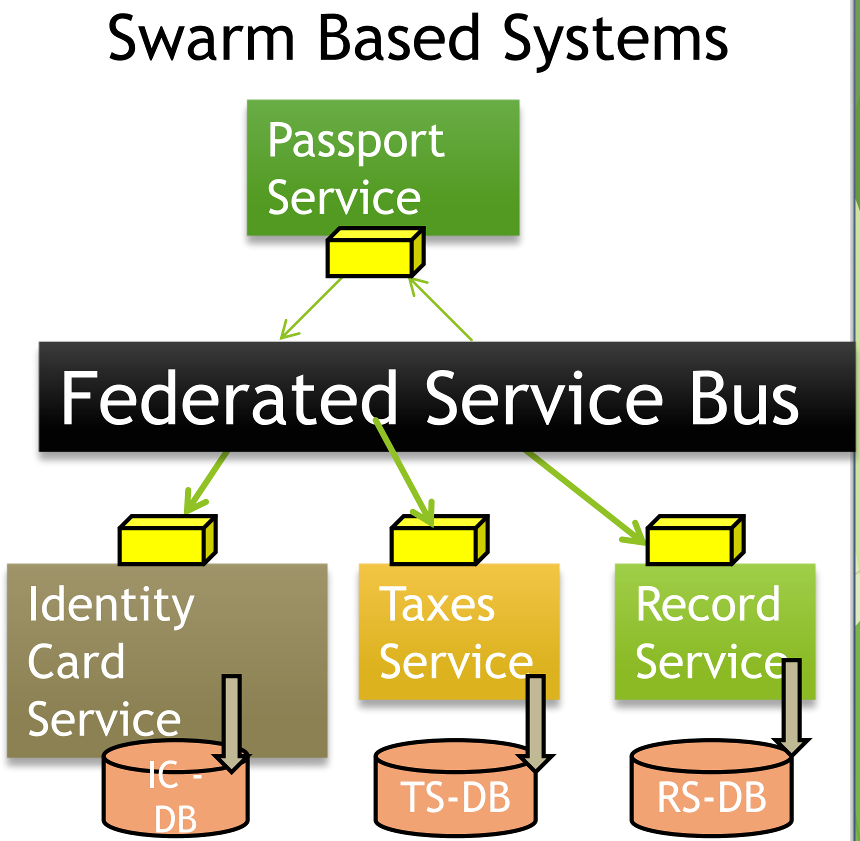
http://www.operando.eu/servizi/notizie/notizie_homepage.aspx

- OPERANDO - integrates and extends the existing *privacy techniques* to create a platform that will be used by independent organisations called Privacy Service Providers (PSPs) to ensure policies compliance regarding privacy laws and regulations
- OPERANDO platform supports flexible and viable business models: public administration, social networks and Internet of Things.

Integration and Privacy using Swarm Choreographies

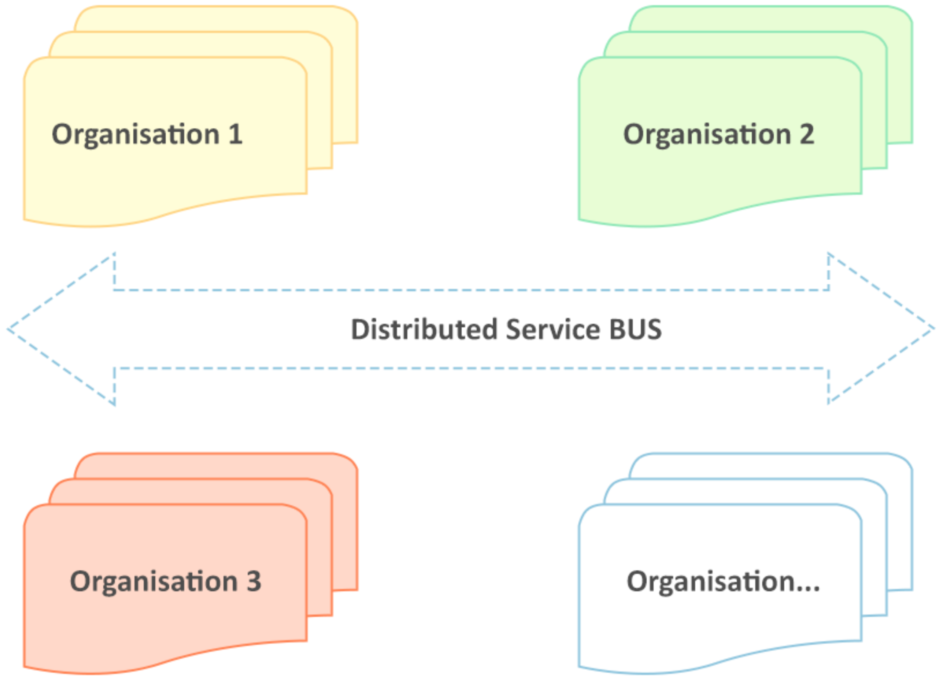
- **Privacy** – a collection of techniques and methods to share data, in respect to a set of rules, lows and polices
- **Executable Choreographies** is appropriate for **privacy** **(FSB – as a virtual level)**





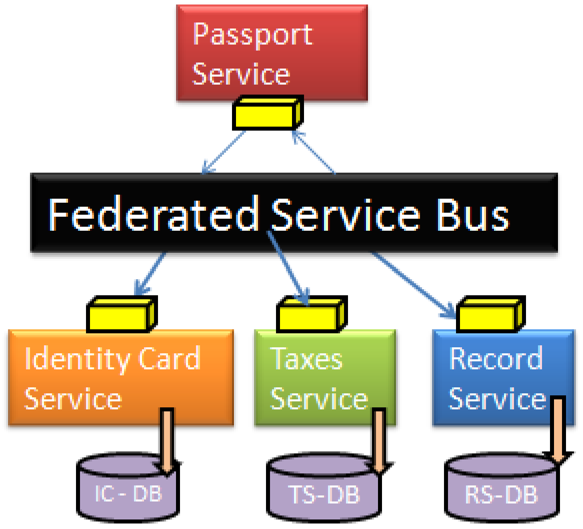
Integration and Privacy using Swarm Choreographies

- Federated Service Bus – general case.
- Components of *Distributed Service BUS* are at each Organisation and will be *thrown* among them

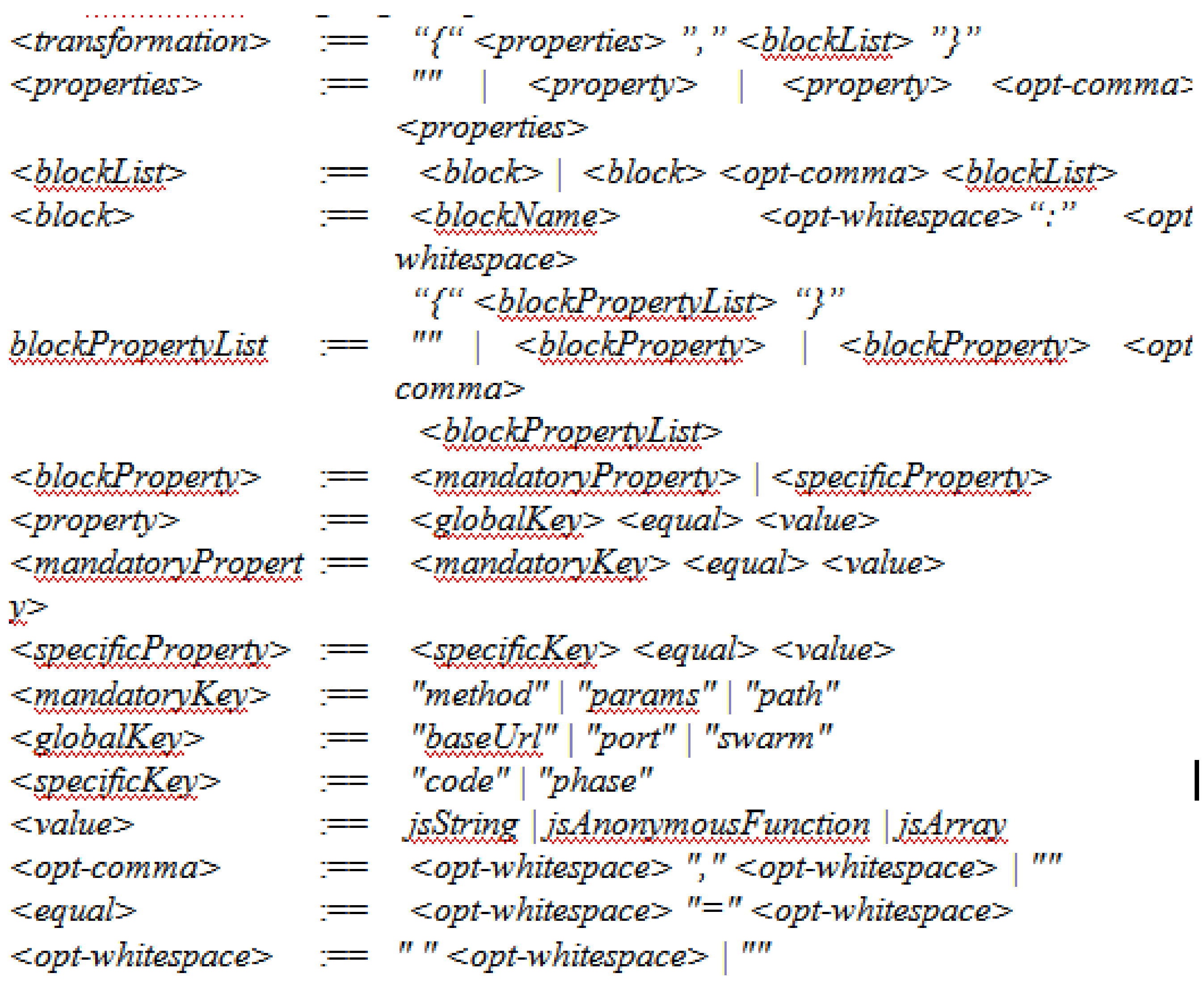


Web Service Transformation Language

- The problem: enable inbound and outbound usage of web services
- Solution: SwarmTL DSL allows *5 type of transformations*
  - *Service to Functions transformations (SF)*
  - *Choreography to Service transformations (CS)*
  - *Function to Service transformations (FS)*
  - *Service to Choreography transformations (SC)*
  - *Interceptor transformations (I)*



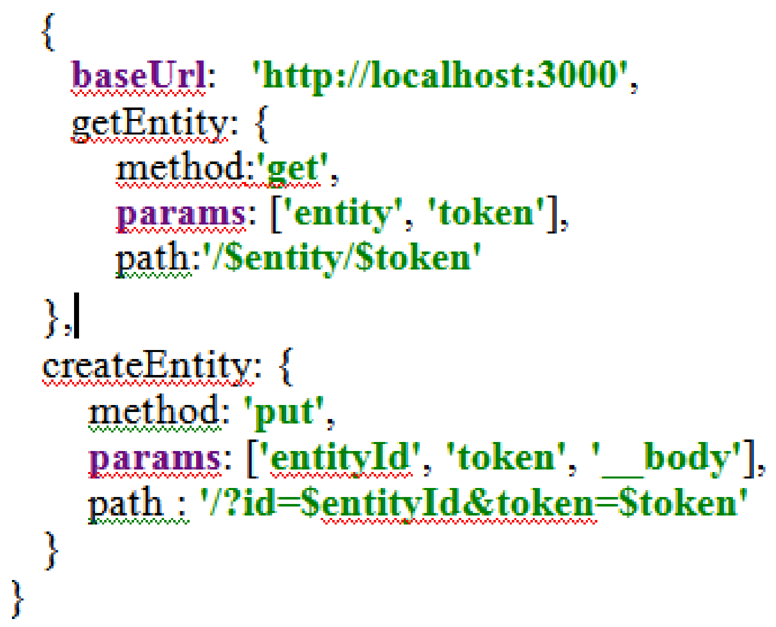
Web Service Transformation Language



```
<transformation>     ::=  "{" <properties> "," <blockList> "}"
<properties>         ::=  "" | <property> | <property> <opt-comma
                          <properties>
<blockList>          ::=  <block> | <block> <opt-comma> <blockList>
<block>              ::=  <blockName>        <opt-whitespace> ":"   <opt
                          whitespace>
                          "{" <blockPropertyList> "}"
blockPropertyList    ::=  "" | <blockProperty> | <blockProperty> <opt
                          comma>
                          <blockPropertyList>
<blockProperty>      ::=  <mandatoryProperty> | <specificProperty>
<property>           ::=  <globalKey> <equal> <value>
<mandatoryPropert    ::=  <mandatoryKey> <equal> <value>
y>
<specificProperty>   ::=  <specificKey> <equal> <value>
<mandatoryKey>       ::=  "method" | "params" | "path"
<globalKey>          ::=  "baseUrl" | "port" | "swarm"
<specificKey>        ::=  "code" | "phase"
<value>              ::=  jsString | jsAnonymousFunction | jsArray
<opt-comma>          ::=  <opt-whitespace> "," <opt-whitespace> | ""
<equal>              ::=  <opt-whitespace> "=" <opt-whitespace>
<opt-whitespace>     ::=  " " <opt-whitespace> | ""
```

Web Service Transformation Language

- Example: SF (Service to Functions) transformation that takes a remote REST web service from *http://localhost:3000* and exposes a set of functions



```
{
  baseUrl:   'http://localhost:3000',
  getEntity: {
     method:'get',
     params: ['entity', 'token'],
     path:'/$entity/$token'
 },
 createEntity: {
    method: 'put',
    params: ['entityId', 'token', '__body'],
    path : '/?id=$entityId&token=$token'
 }
}
```

Potentials Applications

Create alternative solutions for existing systems in order to preserve privacy properties and a better integration

- Better social networks
- Email services
- Document management applications
- Platforms to eliminate the middleman in two sided markets
- Better search engines (information, jobs, persons) without spying on the users

Create applications that are difficult to build with current technologies

- Document flows between organizations decreases both paper use and bureaucracy
- Real time data analysis for smart-city, smart society using anonymized data
- Personal Assistant that works for you and not spying on you

Research Areas and Conclusions

- **Take cloud automation of the configurations and deployment to the next level**
- Virtual databases and virtual servers **per user** not per application
- Easy to use programming models
- Error handling, distributed transactions and fault tolerance issues
- Privacy by design
- Automated verifications
- Development of business models to finance these applications

More information at:
-Alboaie Lenuta, Habilitation Thesis, 2019
Integration and Cloud Computing
-http://profs.info.uaic.ro/~adria/

https://profs.info.uaic.ro/~adria/teach/courses/CloudComputing/coursepractical-works.html

-ALBOAIE SÎNICĂ, Applications of executable choreographies (Summary of PhD thesis, 2018) :
http://doctorat.utcluj.ro:8080/Doctoranzi/?2

-PrivateSky 2017-2021:
https://profs.info.uaic.ro/~ads/PrivateSkyEn/