University of Bologna

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

Engineering Bologna Campus

Class of

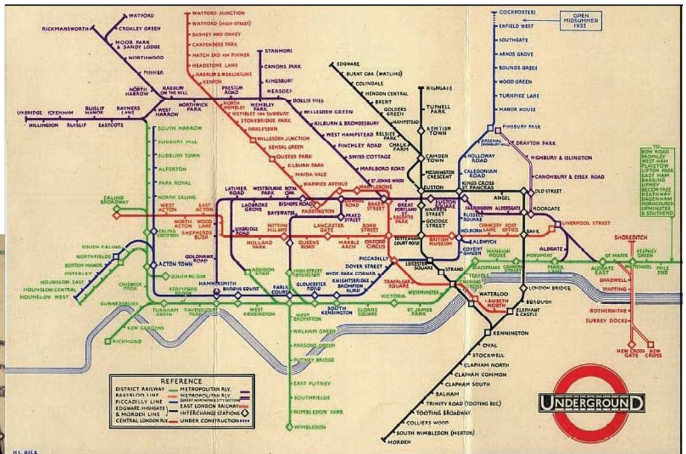
Computer Networks M

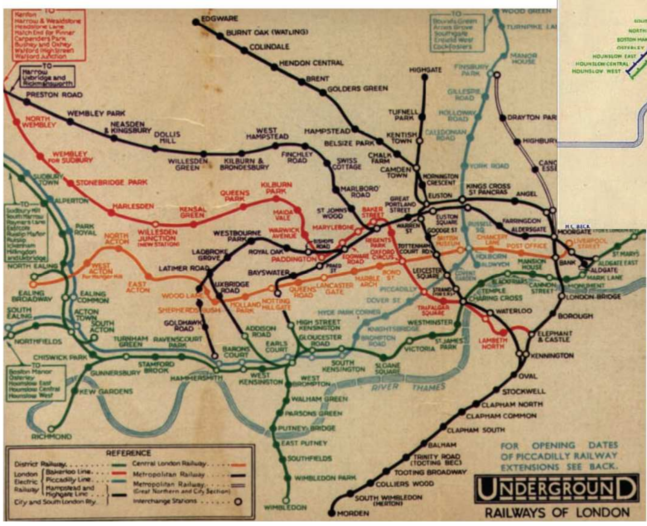
Goals, Basics, and Models

Antonio Corradi

Academic year 2015/2016

ABSTRACT …





Specially interesting in
complex systems
to **focus** on the **right target**

MODERN DISTRIBUTED SYSTEMS

They are complex but very well spread… but still there are unsolved issues; and that is why they are interesting ☺

We have to face still **many challenges** and **problems** to be solved toward a good design

As a few examples only of basic requirements

- **Scalability** and **good Answer and Service time**
- **Predictability** and **Performance**

But many difficulties

- **partial failure overcoming**
- **heterogeneity (at many levels)**
- **integration and standard**

…

SERVICES IN SYSTEMS and QUALITY

The first point in any system is to have a vision in terms of **services to be offered**. In that case, any situation of a relationship can be qualified by the **intended quality** to be provided for providers to requestors

We have to carefully define the Quality **of** the **S**ervice (**QoS**) to be granted in any situation and to operate on it

The QoS defines the whole context of the operation and how to quantify the operation results

Of course it is not easy to find a standard way to specify services and their properties in a clear way

Telco providers define service levels via indicators, such as throughput, jitter, and other measurable ones

QUALITY of SERVICE QoS

QoS description must take into account all the possible **aspects of a service, under many perspectives**

From the experience of telco, we may consider

- **Correctness**
- **Performance**
- **Reliability**
- **Security**
- **Scalability**

Some of the above aspects are mainly transport-related and tend to neglect **application and user experience (even if they have a larger meaning)**

Some areas are **more quantity-based and easy to quantify,** while others are more **subjective and descriptive**

QoS should take into account all cases

QUALITY of SERVICE INDICATORS

QoS must adapt to the different usage situations

QoS must be based on both kind of properties

- **Functional properties**
- **Non Functional properties**

The **functional ones** are easy to express and quantify

such as average packet delay (over a service), bandwidth, percentage of lost packet, … for one service

The **non functional ones** are hard to quantify

such as long term service availability, security level for the information, perceived user experience in video streaming, …

Sometimes we refer to **Quality of Experience** (**QoE**) in a provided service

AGREEMENT IN SYSTEMS: SLA

One important point is to understand how to **express the complexity** and to **rule the relationship between different involved subjects**

SLA **S**ervice **L**evel **A**greement

A typical indicator to reach a specific agreement between different parties on what you have to offer and why

Of course it is not easy to find a standard way to specify serve and its properties in a both formal and clear way

Communication providers define service levels via Mean Time Between Failures (MTBF), for reliability and other indicators for data rates, throughput and jitter...

Service providers must define service levels via more tailored indicators that relates and qualify the service for users and also some user experience

GOOD SUPPORT to ENTERPRISE

Several principle and systems to provide and give a scenario for business services

Middleware as a support to all operation phases in a company, also in terms of legacy systems

Service Oriented Architecture (SOA)

All the interactions among **programs and component are analyzed in terms of services**

Any service should have a very precise **interface**

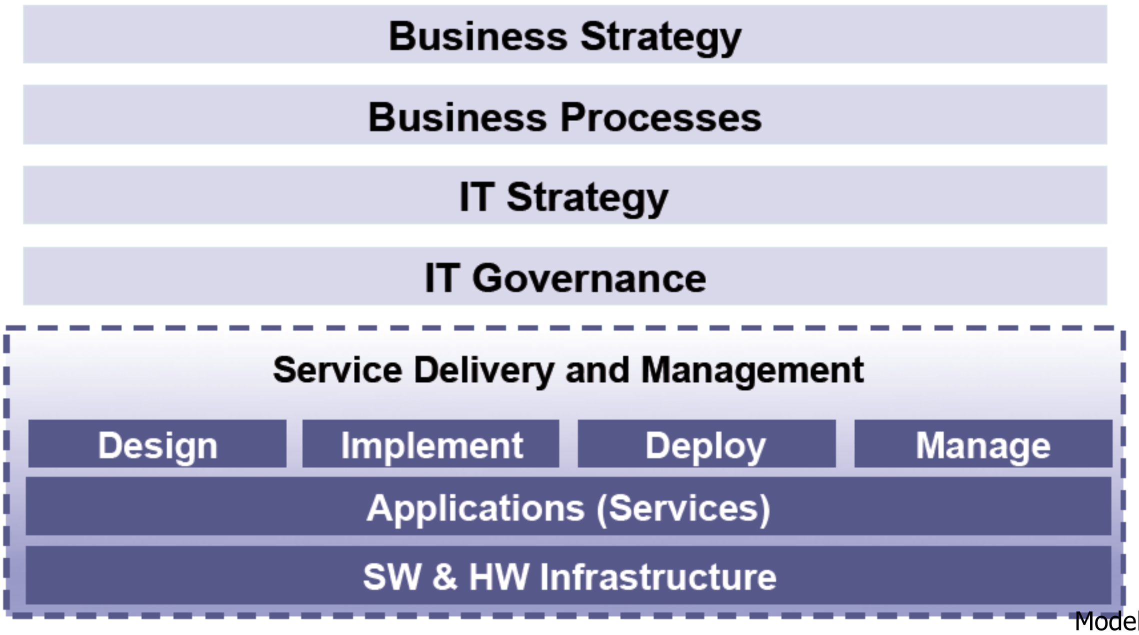
Enterprise Application Integration (EAI)

The need of **integrating the whole of the company IT resources** is the very core goal

That objective must be provided, while preserving Enterprise values

ENTERPRISE Information Technology

Modern Enterprise strategies require both existing and new **applications** to fast change with a critical impact on company assets



Typical different Applications in a Business

This list is only an idea, there may be many other components

- Enterprise Resource Planning (**ERP**)
- Customer relationship management (**CRM**)
- Supply chain management (**SCM**)
- Warehouse and stock management
- Finance and accounting
- Document Management Systems (**DMS**)
- Human Resource management (**HR**)
- Content Management Systems (**CMS**)
- Web site and company presentation
- Mail marketing
- Internal Cooperation tools

And many more….

Enterprise Application Integration

The idea of a complete Application integration or EAI is to have systems that produce a **unified integrated scenario** where all **typical Business applications programs and components** can be synergically provided

There are both:

- **Legacy components** to be reused
- **New components** to be designed and fast integrated

The easy and complete **integration** among **all business tools** has also another important side effect

The possible **control and monitor** of the **current performance** of any part of the whole business

- to have **fresh data** about performance
- to **rapidly change policies** and to **decide fast (re-)actions**

Service-Oriented Architecture

The basic interaction is via services defined as platform- and network-independent operations that must be cleanly available and clear in properties

Service-Oriented Architecture (SOA) is the enabling abstract architecture

A service must have an **interface to be called** and give back **some specific results**

The **format must be known** to all users and available to the support infrastructure

There are many ways to provide a SOA framework

SOA must offer basic capabilities for **description, discovery, and communication** of services

But it is not tied to any specified technical support

Service-Oriented Architecture or SOA

SOA is simply a model and it imposes some methodologies to obtain its goal of a fast and easy to discover service ecosystem

- Services are described by an **interface** that specifies the interaction abstract properties (API)
- The **interface** should not change and must be **clearly expressed** before any usage
- Servers should **register as the implementers** of the interface
- Client should **request the proper operations** by knowing the interface

Interaction is independent of any implementation detail, neither platform-, nor communication-, nor network-dependent

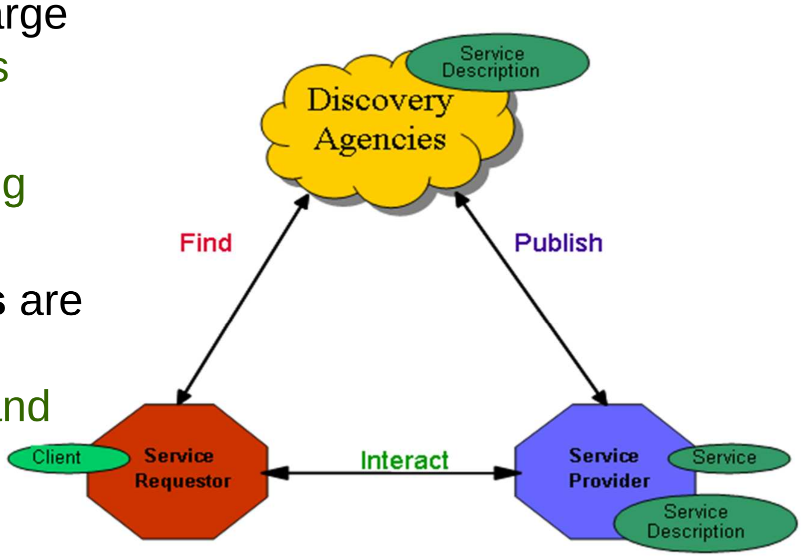
SOA actors or components

Service-Oriented Architecture SOA proposes a precise enabling architecture with three actors

Providers are in charge of furnishing services

Requestors are interested in obtaining services

Discovery agencies are responsible to give service information and full description of services



Service Conceptualization

One service is an **abstraction of any business process, resource, or application**, that can be described by a standard interface and that can be published and become widely known (discovery)

Services are:

- **reusable**, in the sense that they can be applied in several contexts (no limitation, in general anyone)
- **formal**, they are not ambiguous in defining the contract specifications (clear and clean interface)
- **loosely coupled**, they are not based on any assumptions on the context where they could be used
- **black box**, they are neither specifying the internal business logic nor tied to any implementation details of a specific solution

SOA Design Principles

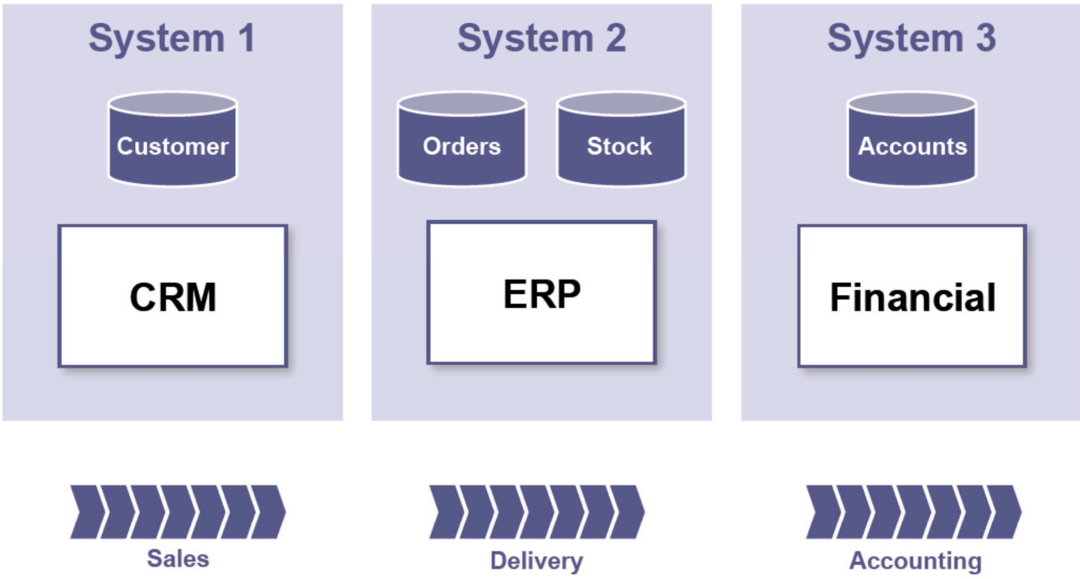
A service must be available by all **platforms that are publishing it** to all the **ones that are in need** of it, if the requestor **asks for the interface** in the right way

Interfaces should be widely spread and published in some discovery agencies

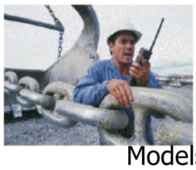
Services must be:

- **autonomous**, they must not depend on any context and should be capable of self managing
- **stateless**, the internal need of state should be minimized (eventually **stateless**); the client maintains the state
- **discovery-available**, all service must be found via opportune naming agents and must easy to retrieve and to use
- **composable**, existing services can be put together to produce a modular component to be invoked independently as a novel service (**composition to create new services**)

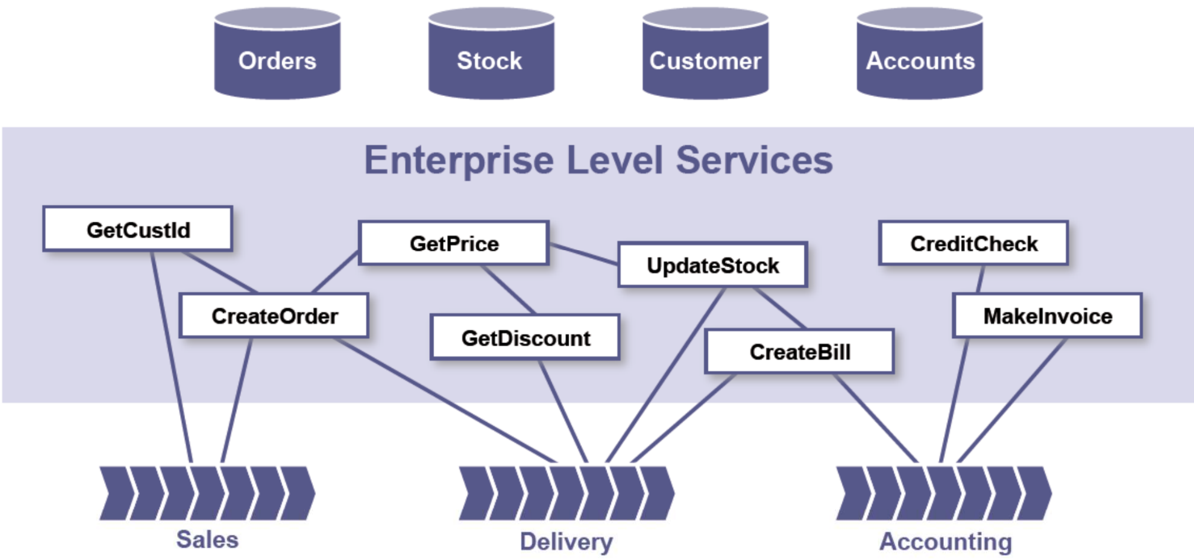
Traditional Business Architectures





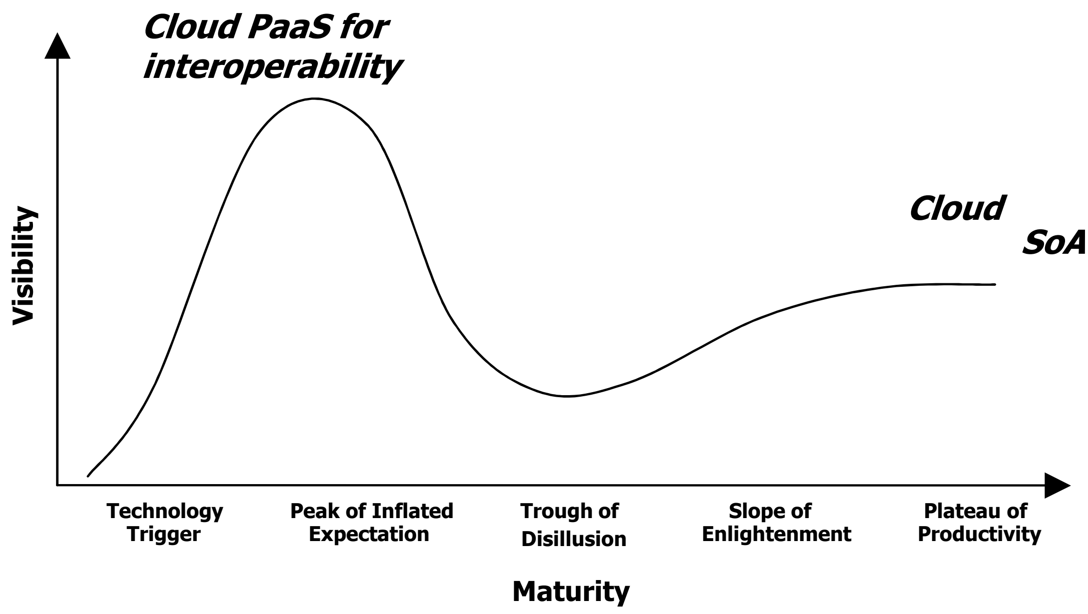



SOA-oriented ARCHITECTURES





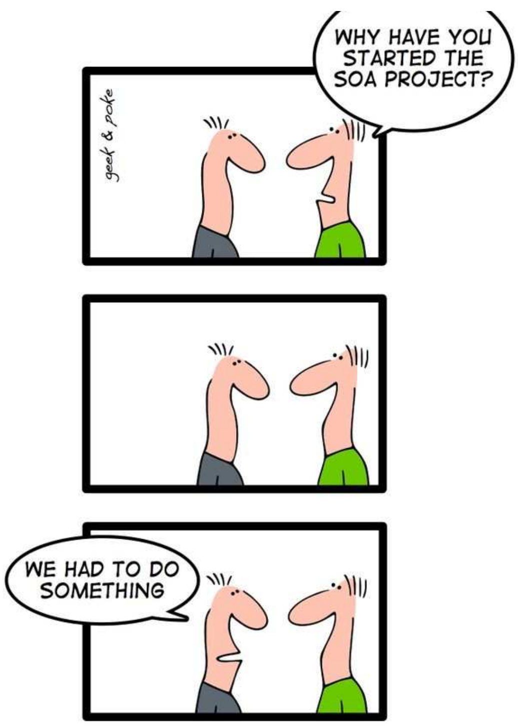

Evaluation and Evolution in Technologies

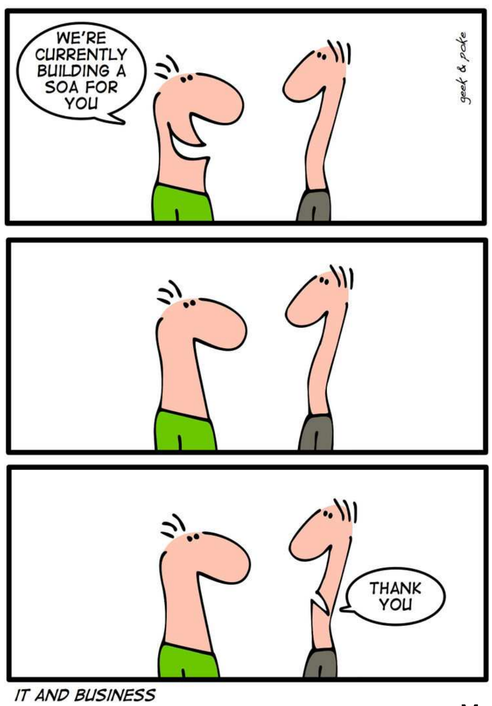


GARTNER technology life cycle

Any technology has its own life cycle, with hypes connected

SOA enthusiasm





DISTRIBUTED SYSTEMS DESCRIPTION

We can understand **distributed systems** and their operations only by conceptualizing a model

A distributed system consists of resources *(all the resources that may be requested during execution to grant any visible result)*

Resources can be, for instance, abstracting from our experience of one machine:

- **Physical memory (RAM),**
- **Disk (some levels of persistence)**
- **Computing (CPU, even many)**
- **I/O and communication support**
- **Other apparel and devices (sensors, actuators, etc. in a smartphone)**

We have to open up our perspective, and think to the whole system, …

A first step is about all **available applications and services**

ANOTHER SYSTEMS DESCRIPTION

A **distributed systems** can consist of several machines

> ***A distributed system consists of many resources***, in an organization that put together **several machines in a locality (more or less confined)**

Resources can be, abstracting from our experience of a system for an organization:

- **Several computing and memory resources (and other ones)**
- **Disks (for local and global persistency)**
- **Connecting support (network with some granted bandwidth)**
- **Other & Application services (OS, Web, Applications, ad hoc services, application define services and clients,…)**

Virtual resources and also corresponding physical resources (all the resources that may be requested during execution to grant any visible result)

MORE COMPLEXITY in SYSTEMS

A **distributed systems** must consider also a larger perspective, both at a **lower** and at a higher level

Resources can be at lower level

- **Operating systems and low level services**
- **Virtual resources insisting on physical ones (not only Virtual machines and Physical ones, but any kind: Virtualized connections and network)**

An optimized management of that environment is hard and to be carefully designed

Resources can be at higher level

- **Application system related services of any kind, from Web servers and services, Web containers, …**
- ***Real application, from management software, to final ad hoc software***

An optimized management of that application environment is even harder and must be carefully designed

SYSTEMS and OPERATIONS

In a business perspective, a **distributed system** can be **hosted on premises**, and in charge of the owning organization

Many companies have an **internal data center that must take charge of all aspects**, from the hosting of hardware, installation, maintenance, operation, and also of the whole software components and their operations

Also all human resources must be handled

Resources must be managed and handled along a business strategy

In a business perspective, a **distributed system** can be **outsourced**, and managed by some service provider

Many companies exploit an **external data center that must provide some business services,** as if they were internal, also in a **transparent way**

OUTSOURCING vs. CLOUD

Companies are used to **outsourcing** some parts since long ago (also maintaining other services as internal with the problem of their interconnection and integration)

The external data canter must be always accessible and capable of giving service with the negotiated SLA and the requested QoS

Some aspects are overcome, others to be solved

In recent years, **Cloud** has opened up more that perspective by providing any kind of service remotely, by producing a more **organized model of all the offered services**

Access is always **via web** and in some **agreed form**

Many private people and small companies have available many 'low cost' **external data centers to provide elastic, easy-to-use and pay-per-use services, *in a transparent way, as if it were internal***

RESOURCES

In a **DISTRIBUTED SYSTEM a central issue is Resource Management**

Definition of a resource

each component reusable or not, both hardware and software, needed for the application or system

Classifications (many different properties and aspects)

- physical resources vs. virtual resources
- physical resources vs. logical resources
- static resources vs. dynamic resources
- low-level resources vs. application resources

Resources have an external and an internal organization, based on **abstraction**

specification (visible interface) and **implementation** (not visible)

Different implementation of course, …

Concentrated & Distributed organization toward their best service

RESOURCE MANAGEMENT

Systems are very **differentiated in requirements and** there is no **magic recipe** for all cases
There are **many implementation models**
and many different ways of operating and serving results

The design of one interaction is split into two phases
- the **static** that plans the operations and precede the real operations (before running and out-of-band ☺)
- **the dynamic** that is the implementation of the operations and (while running services and in-band ☺)

The concurrency among **services and support actions** can produce **delays** and **overhead**

RESOURCE SERVICES

A resource can be available for providing its services with a typical interface (the simpler the better) as **SOA**
You become the client, and the service is provided to you by the server

The interface is deployed in two forms:
Service request
Distributed file system

Service Request
The client ask explicitly the server in a Client/Server approach

Distributed File System or a middleware approach
Unique service available in a transparent way (allocation transparent)

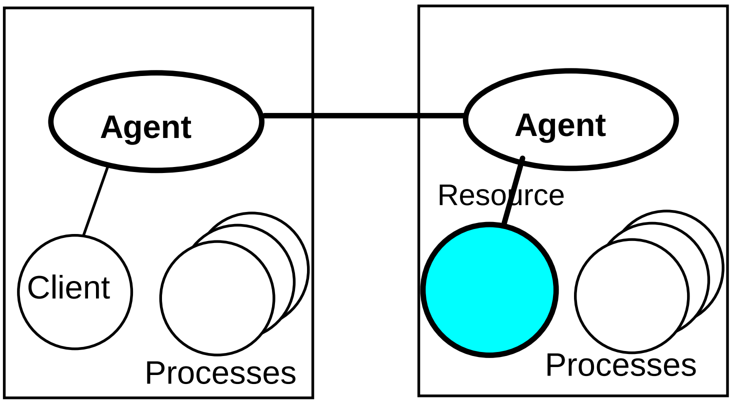
Transparency simplifies the interaction and users are freed of responsibility

MANAGEMENT by AGENT (DFS)

The deployment is a coordinate agent systems to provide a unique service

Agents must coordinate among themselves to operate and give the best result

Any kind of negotiation is possible among agents toward the final goal, also deciding to refuse the service



GENERAL MODELS

In **Distributed systems** maximum interest in **real operations, performance, distributed execution**

Models preventive vs. reactive ones

Preventive behaviors avoid a priori undesired events, but often introduce a fixed cost on the system (often computable)

Reactive Behaviors allow to introduce less support logic (and **may limit** operation costs) if specified undesired events do not occur

Models static vs. dynamic

Static behaviors do not allow to adjust the system to (even limited) **variations during execution**

Dynamic Behaviors allow you to let the system evolve along (limited) **variations in execution but can cause higher costs** (overhead)

STATIC and DYNAMIC MODELS

Dynamic models / Static Models

User number is predefined and fixed before run

Users can be added and deleted during the execution

Process number is predefined and fixed before run

Processes can be added and deleted during the execution

Node numbers is predefined and fixed before run

Processors can be added and deleted during the execution

Clients and their number is predefined and limited before run

Client traffic can be added and deleted during the execution

Services and support are predefined and fixed before run

Servers and services can be added during the execution

Services can vary during execution

TOWARD A RESOURCE MODEL

Some usual (logical) resources for execution

Processes as entities able of expressing execution via

- **local actions** on an internal and confines environment
- **communication actions** toward other processes by using *shared memory* and *message exchange*

Also data can exist *externally* to processes themselves (limited confinement and insufficient abstraction)

Objects as entities to express abstraction, as ability of

- enclose and hiding **internal resources** (data abstraction) with externally **visible interface** only of **operations**
- act on **internal resources** to complete externally requested operations

Passive Objects **data abstractions** with external executing entities

Active Objects **entities capable of both execution** and **data containment**

CLASSES vs. INTERFACES

A trend in software architectures puts together:

- **interfaces** as the **agreed contract of interaction, uniquely specified and not negotiable**
- **classes** that describe different **implementations** (many different can exist also different in QoS in the same system)

Distributed systems has spread since long ago the idea of having **interfaces as contracts** between different stakeholders - who also develop independent - and of keeping these separate from specific implementations (possibly multiple ones)

middleware are **usually based on interfaces** and less on classes (and other their separate implementations, as the components)

In OO languages, that separation **came later, but modern languages have incorporated quickly,** especially in **languages designed for distributed systems**

OBJECTS vs. COMPONENTS

We tend to refer to **Object models**, see Java and other usual languages

The Object model is not so confined and very dependent from the containing environment (fine-grained objects)

With the class relationship and subclassing

The distribution requires to **confine better objects boundaries and interactions** with the **containing environment**

The Component Model (coarser grain) succeeds

In defining more **self-contained** entities and more **transportable to different environment**

Definition of component: **static abstraction of a confined entity communication with the external world via ports**

COMPONENTS

A component is

- **Static** having its own life and being independent from application
- **Abstract** without any visibility of the component internal structure by showing externally only input output ports
- **Communicate only in a disciplined way by ports** as the only way to communicate to the external world (**IN** and **OUT**)

Effect of

better reusability, with easy transportability from one container to another (no hidden interactions, only visible and declared ones)
capacity of substitution, one implementation can replace another (dynamic replacement) without any container change

Toward SOA (**S**ervice **O**riented **A**rchitecture o **SOA**) ⇨ ports as tag for methods visibly accessible and easy to invoke

AGAIN COMPONENTS

Again a component definition

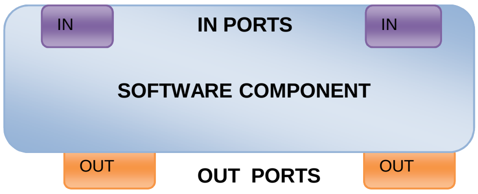
"A component is an object in a tuxedo. That is, a piece of software that is dressed to go out and interact with the world"
Michael Feathers

A **component** typically is one entity with **coarser grain** than one object, and it is typically more **self-contained** & capable of **operating** in very different **environment** …

Often it should work within a **container**, i.e., a **support server** capable of hosting the component to provide it **several needed functions; components focus only the business logic**

J2EE, **EJB** are containers that can host components and can provide most common support functions (initialization, finalization, …)

COMPONENT PROPRIETIES



A component has a **very disciplined interface** and must **declare the contract of interaction via ports** that regulate **accepted inbound requests (in ports)** and the **services you can ask outward (out ports)**

This interface rules precisely and statically the interaction with the outside world in an explicit (and not hidden) approach

A **component** is **self contained** but must handle only **some features** and delegate **other functions** to an **enclosing container that is able to reply and to manage**

Interaction container component is disciplined too and governed precisely; the container can operate autonomously

SYSTEMS with COMPONENTS

A system with components can provide several functions to hosted components

- **Life cycle;** the container can activate and deactivate components on need
- **Resource sharing;** resources are shared via container provisioning and encapsulation
- **Composition;** the container can help in forming newe components by putting together existing one
- **Activity support;** any interaction between components can be supported via container-offered activities
- **Control;** the container helps in monitoring, handling, and controlling components
- **Mobility;** the container has the capacity of extracting and moving components already executing

SYSTEMS for SERVICES

A natural evolutions of the above functions is the possibility of doing the same while hosting services

Several environments go in the sense of offering many functions toward operations, so easing the duties of the applications and clients

Let us think to a system that has the capacity of providing support for **service access, usage, and composition**

It can support management of services

- **Access** (via Web request or Web services)
- **Composition** (toward new services)
- **Life cycle support in many forms: Control, Elasticity, Resource sharing, Activity lifecycle, internal Optimization and Mobility, Accounting**

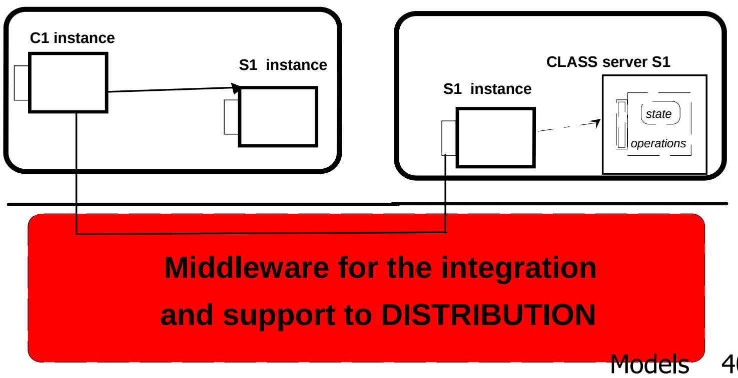
REMOTE REFERENCES

In **many local environments** (in **object-oriented** system), we need the capacity of **referring to some external resources**, in order to coordinate different machines (virtual or physical)

A C1 on one node must refer to a remote instance, the same as if they were local instances on the same node

To refer to a remote instance we need some **intermediary support that extends the visibility to remote nodes**

In some cases,
local and remote references are uniformed via
local intermediaries (**proxies**)
that play the enabling role and typically mask support transparently



RMI REMOTE REFERENCES

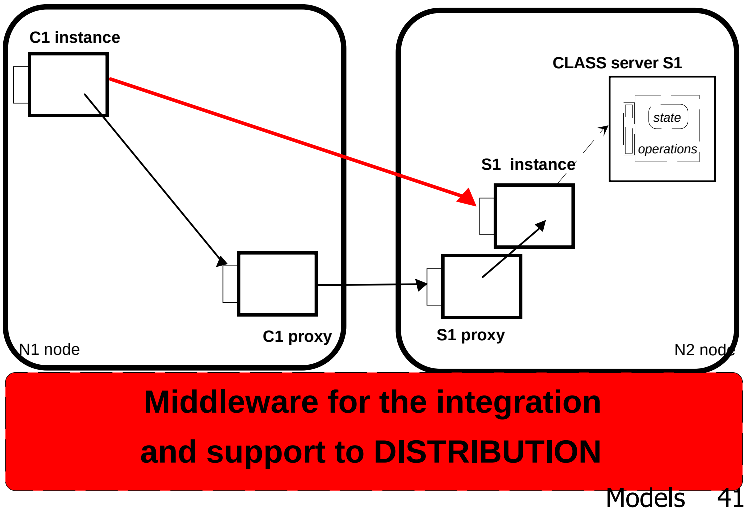
Between two JAVA JVM systems, we can use Java Remote Method Invocation (RMI) that build two proxies

-one from the customer (stub)

-one on the side of the servant (skeleton)

Such proxies are often **generated automatically** and make the user part reasoning regardless of the specific deployments

Similarly other environments (**CORBA**, **DCOM**, etc.) define their specific support for OO cases



REMOTE REFERENCES via PROXY

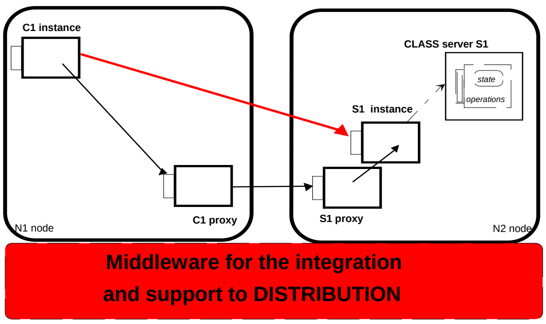
Two Java virtual machines can use **PROXIES to get remote visibility of object references**

RMI support many solutions but proposes problems:

- How do you get the reference to the server? (name system)
- Where are the ancillary classes?
- How to obtain them (while running)?
- And if there are any inconsistencies?
- And if the server is not active?
- And if you don't keep the status?

About **remote references**:

- two references to the same object?
- two references for the same service?



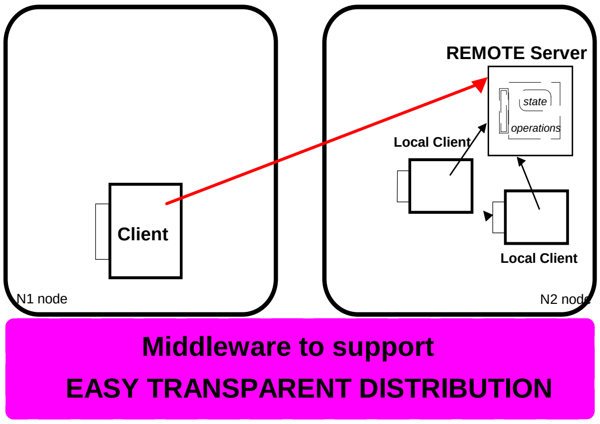
REMOTE REFERENCES & MIDDLEWARE

A central point in all middlewares that **abstract away and hide details from users for remote access** is how to enable and manage a **remote reference** in all its aspects

A remote reference allows access to non-local entity must surely be transparently

But costs must be considered and evaluated for each aspects of the support mechanism

- How does the remote reference cost?
- How is the cost of middleware to support organization?
- How to obtain remote references?
- Are inconsistencies possible?
- What are the responsibilities of the middleware? …
- …



INTERMEDIARIES & PROXIES

PROXY

In a communication we may have intermediaries placed and deployed either side, the client and the service provider

PROXY

from client or from server

proxy

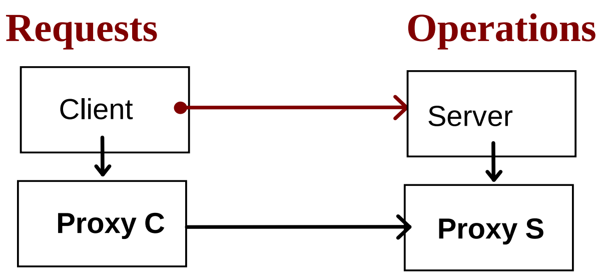
C/S stub & skeleton

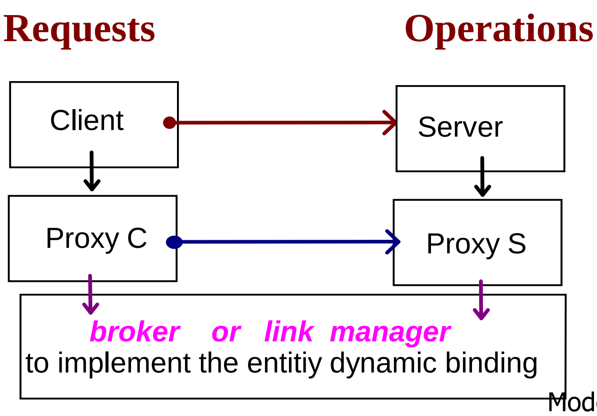
interceptor

to add functions

broker

something similar to a container





CONTAINER MODELS

CONTAINMENT

Often many features cannot be controlled directly from the application but left as **responsibilities to a delegated supervisor entity (container)** who deals with them,

- often introducing policies by default
- while avoiding typical user failures
- controlling external events

Containers (entities with many names, also called containers, **ENGINE, MIDDLEWARE**, ...) can take care of automatic actions that relieve the user responsibility from repetitive actions, that can be easily expressed

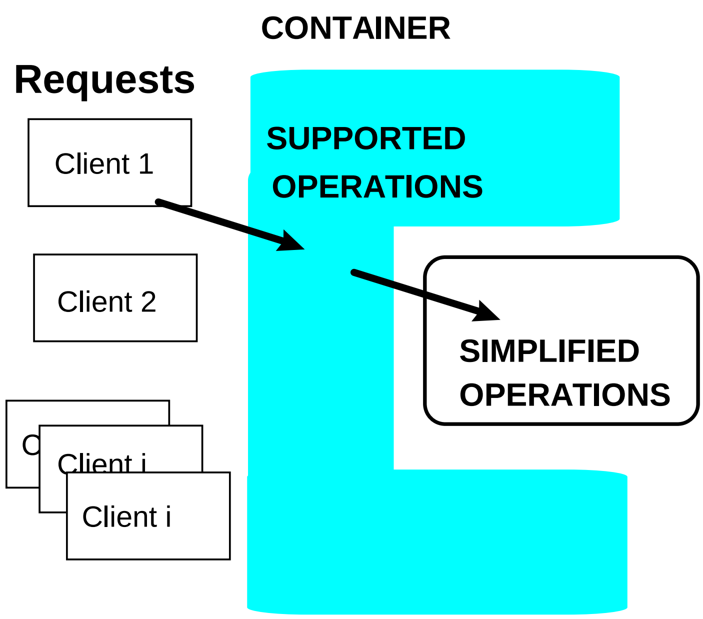
A user can then specify only the high-level part not repetitive, highly dependent from the application logic

MODELS FOR CONTAINMENT

CONTAINER

a service user may be integrated in an environment (middleware) that deals independently of many different aspects

See
CORBA all C/S aspects
Engine for GUI framework
Container for servlet
Support for components



Container can host **components more transportable & mobile**

One goal is also to **move around components between different containers** and allows that inter-container mobility

DELEGATION to CONTAINER (Middleware)

The container can provide "**automatically**" many features to support service

- Lifecycle Support

- **activating the servant/deactivate/**
- **maintaining state**
- **persistence and retrieval of information (interface with DB)**

- Support to the name system

- **the Discovery of servant/service**
- **Federation with other containers**

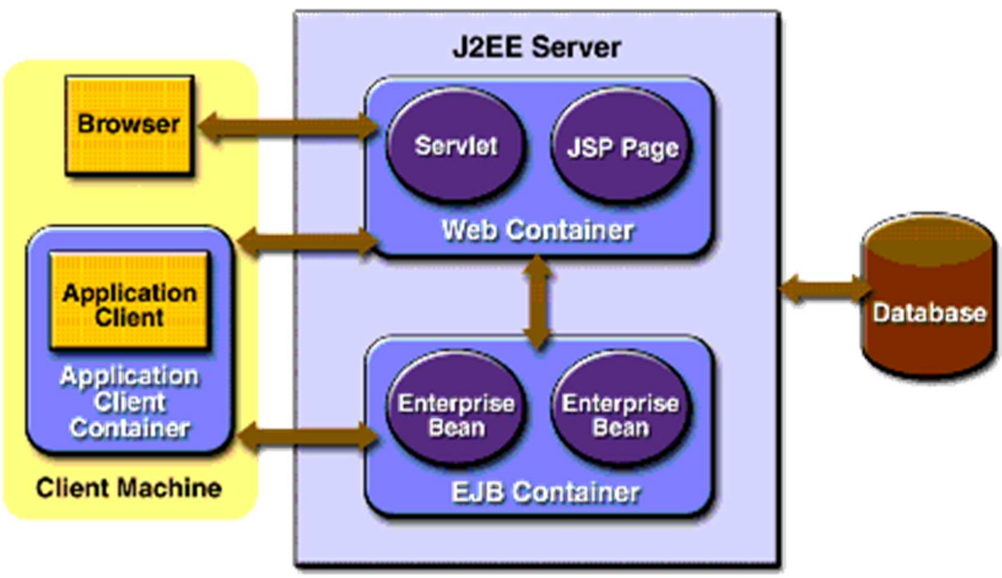
- Support to the QoS

- **fault tolerance, selection among possible deployment**
- **control of negotiated and obtained QoS**

…

J2EE – Java 2 Enterprise Edition

A container may **also be able to facilitate the execution of different components** such as servlets, JSPs, beans of various architectures and types



NEW MODELS FOR CONTAINMENT

More and more new forms of containment are available

There are several tools that can not only provide the hosting, but also allow the **management of the container** and the control of the **migration of components**

That feature is specifically more important when you have access to the container via **web functions** and describe your components as **microservices, easy to be installed and re-installed remotely**

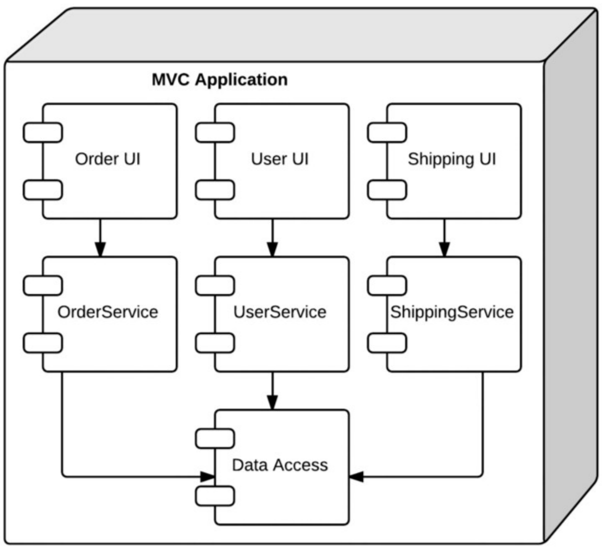
Microservices are small components capable of being hosted in different machines and easily managed

An OS container can host and control those components in easy way and also can suggest advices in designing autonomous components

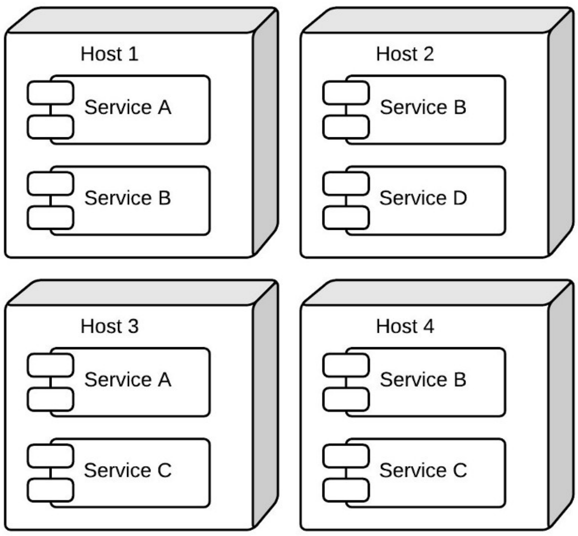
Docker is tool where you can specify what you need to have installed

DIFFERENT DESIGN MODELS

Microservices can be easily deployed and also moved from one container to another



Monolithic Architecture



Microservices Architecture

NEW MODELS FOR CONTAINMENT

Docker as a microservice language and tools for a Linux container that allows to design, host, control, and optimize services (both statically and dynamically)

Docker is tool with which you can specify an **entire application and its dependencies as a container** (so it becomes more portable and easy to be packaged)

Some requirements are crucial for microservice viability and operations:

- Possibility of **managing services from outside** (monitoring and handling of internal services)
- Easy **deployment and limited interference** (simplest interface possible)

DEPLOYMENT for an APPLICATION

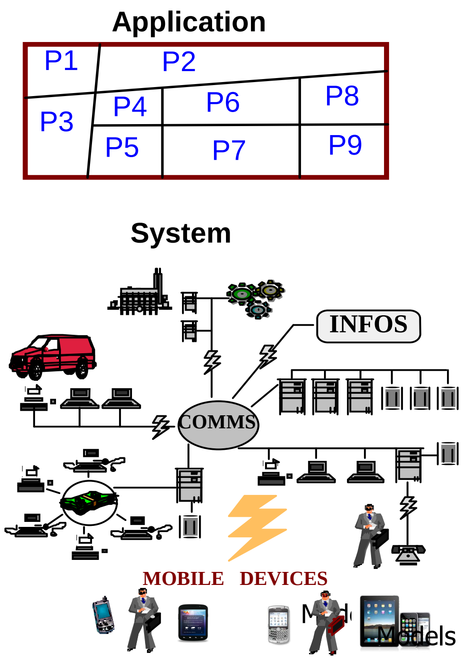
An application consists of very different logical and concrete resources: **processors, network, and also processes, objects, components**, ..., **up to service and request to them**

Application resources are many and differentiated too:

- **processes**
- **components**
- **objects and classes**

System resources are many and differentiated:

- **processors**
- **networks**
- **interconnected cluster**
- **cloud**



APPLICATION DEPLOYMENT

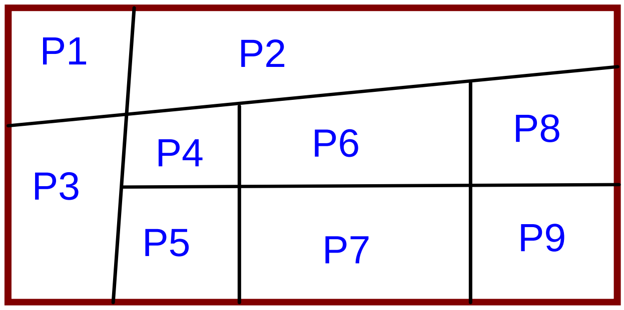
An application is developed as an organization of entities, **objects components**, and **classes**

if you are not working on a single machine, one must decide **a deployment on multiple machines** that must decide on how to

- partition the application **into constituent components**
- rely on a **support for remote references**

The application is divided into resources that represent partition (P1-P9) to be mapped on the specific deployment

Application



Possible partitioning of the resources

PARTITIONS in the APPLICATIONS

An application must be deployed on a number of processors and you have to decide how to group its components into partitions for processors themselves

The application involves both:

Static resources (represented in previous slide) easy to group as needed, so start executing with the components already allocated

Dynamic resources (previously not represented) that will be created during the execution or may not even be created at run time

For instance, the processes or the resources that depend on the execution and that only some runs can create, depending on the application state and the progress of applications

ALLOCATION STRATEGIES

Allocation

One application can use two different policies

either *static* or *dynamic ones* (**maybe *hybrid***)

Static allocation: specified a configuration (deployment), those resources are decided **before runtime**

Dynamic allocation: those resources are decided **at runtime**
⇨ **dynamic systems that can decide at run time**

Static allocation

Pros the allocation cost precede the execution

Cons the predefined allocation is inflexible

Dynamic allocation

Pros the allocation cost impact on the execution

Cons the allocation can adapt to the current situation and is only made by need (an on need)

MODELS for ALLOCATION

Allocation strategies

Static resources

always to be decided statically
and eventually optimized

Dynamic resources

either statically decided (with a policy to be actuated on need)
or decided at runtime

In dynamic systems, one can create **not forecast dynamic resources** and you can **think of to reallocate existing resources** (migration): resources can move around and setting can change during execution

Heavy moment of resources re-allocation

DEPLOYMENT SUPPORT

- *MANUAL*

→ **the user determines each individual object** and passes it **on the appropriate nodes** with the proper **sequence of commands**

- *FILE SCRIPT APPROACH*

→ you must write and run **some script files** (some **shell language,** bash, Perl, Python, etc.) with the command sequence to drive the configuration **by steps** and **in phases** that usually specifies **dependencies between objects**

- *APPROACH based on MODEL or MODEL-DRIVEN*

→ **automatic configuration** support through **declarative languages or working models** to obtain the configuration (e.g., SmartFrog and Radia)

ALLOCATION MODELS

- *EXPLICIT APPROACH* **(user driven)**

→ the user provides **before the execution the mapping for each resource** to be potentially created

- *IMPLICIT APPROACH* **(automatic)**

→ the system takes care of the **application resource mapping** **(both at deployment time and during execution)**

- *HYBRID APPROACH*

→ the system adopts a **default policy** applied to both **static and dynamic resources,** initially for the allocation of **new resources** and also to **migrate during run**

→ possible **user indications and advice** are taken into account to improve performance (please allocate together another resource: 2 VMs together on the same PM)

MODERN DEPLOYMENT

If an application is to be supported, it must typically be **deployed for a specific configuration**

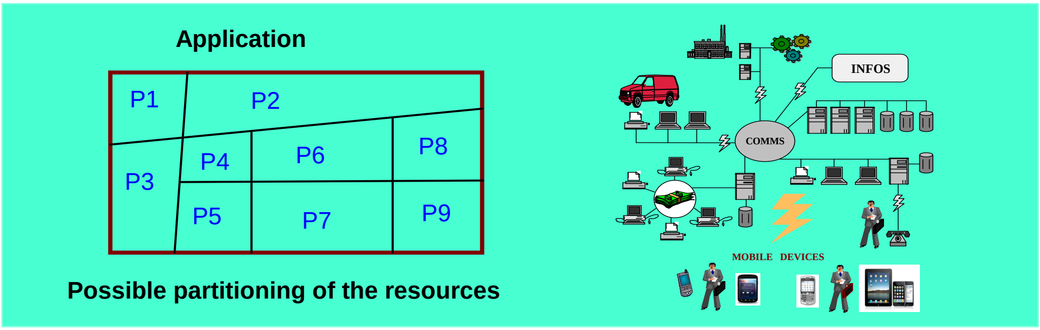
Traditionally the approach is:

We define how to *configure applications* taking into account the specific system resources available *(you specify for the environment)*

A novel approach is:

We ship together **the application with its required configuration** so they can be ported to different possible support environments

(microservices and **docker** approach, Cloud approach)



RESOURCE HANDLING → PROCESSES

Management with different costs and different goals
Allocation & (dynamic) re-allocation of processes

LOAD SHARING ⇨ a priori defined, before the run
(eventually actuated afterwards, at resource creation)
Resource allocation, *without* ***moving*** *any* ***resource once allocated (static allocation)***

LOAD BALANCING ⇨ done during the execution
After a specific allocation and a first execution, **already allocated and active resources** *can* **migrate** *to obtain a better global efficiency* ***(dynamic allocation)***

The static case can be studied in a more precise way, being out-of-band, while the dynamic must compete with the application execution

PROCESS ALLOCATION

Specifically, the cost considerations are crucial for:

Static evaluations
In that case, we work **'out of band'** (**before** the **deployment**)
and we can also use very precise (complex and long) **algorithms** to define the best allocation
Precise algorithm for allocation face the **NP-complete problem**
Heuristic algorithms ⇨ **Genetic, Tabu search**
Often these strategies are too expensive to be applied during the execution

Dynamic evaluations
goal ⇨ **overhead reduction**
Simple policies to respect the minimal intrusion
⇨ local policies and with the lowest implementation cost

MONITORING

MONITORING as an enabler for control & manage

To give fresh information on the system current load, observing the current situation

Picking up and collecting **load information** on
processors, resources & communication

* by using **events**
* by using **statistic** and **historical data**
* by observing on **limited intervals**

The monitoring get info on current load, by assuming continuity of application behavior and limited graceful gradients

collected information used to forecast next situations of resources in the future (**continuity assumption**)

There is an obvious need of limiting the cost of the information collection and maintenance to limit intrusion (minimal intrusion)

SUPPORT INTRUSION

Monitoring a component or an entire application is an example of an internal function very important to manage a system

In general purpose systems (so the ones we are interested in) the support does not have dedicated resources, but it has to use with the one exploited for the application

That competition suggest lo limit to the maximum the engagements of those resources so to limit the percentage of them taken out to the application

The general principle stemming from the above is

the **minimal intrusion principle**

Any support function must limit its operation cost to the minimum, compatibly with the achievement of its goal, so to intrude minimally with the application

COURSE OBJECTIVES

In **distributed systems** we focus on all the aspects related to **execution** and **operations**

Of course, you have to develop software, before execution

For instance, there may be classes and components that have no influence and correspondence during run

their importance for us is very limited, because we focus on the facets that impact during execution

We are interested in everything that has impact during at run time and that remains significant and vital by favoring, fostering, and enabling the distributed deployment (and makes us understand how they do)

for example, there are classes that then become **active processes and components** and will be distributed around, during the application lifecycle: those are the entities that interest us because they represent a part of the run-time system architecture

We focus the dynamic architecture, and in understanding how it is and how well it works

AGAIN for the CLASS PERSPECTIVE …

In distributed systems we seek for **performance** and **quality (QoS) and to grant them**

*For a specific architecture, we expect that there are involved **resources** and particularly **significant cases***

For example, RMI has a very strong impact on the cost and scalability of the overall system

the direct use of the socket and the lower level tools ensures less overhead and greater

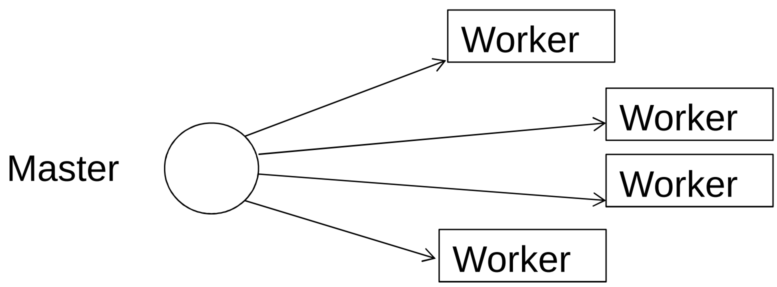
During execution, we are interested in bottlenecks, as the critical points and parts that may misbehave and are unsuitable toward a good system behavior

To adopt a tool as RMI (or an expensive remote request) instead of a message exchange in an occasional rare communication one off (maybe only once per run) tends to introduce a potential bottleneck to consider and to control in a project

the architecture should be checked and rested a priori and a posteriori on the field by quantifying execution

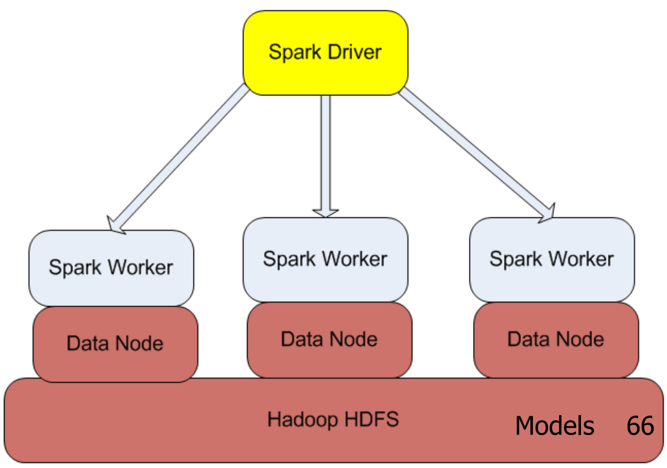
LOAD SHARING VIA FARM

Let us refer to a pattern called Farm, with a Master and several Workers, a pattern present extensively in many situations



Typically you have a master that can distribute the load to workers that execute in parallel and finally get results back

As in Spark where you have a front end that distributes load to other nodes
The Spark driver is the master and try to find the nodes that can work on specific searches in parallel



(STATIC) LOAD SHARING

If an application consists of *entities (processes)*

Sharing means to identify
the *processes and when and where to allocate them*

The static policy can apply only to process creation to find the available processors

Static decision does not allow any reallocation after the first decision

We may have many different allocation policies, either static or dynamic, o processors

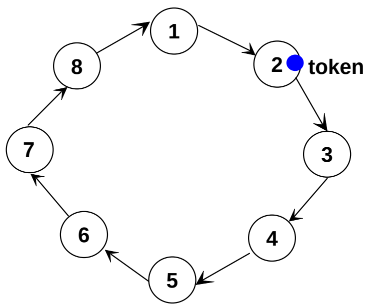
Processors in a **logical ring**	static one
Processors in **logical hierarchy**	static one
Processors with free links (**worm**)	dynamic one

LOAD SHARING

Logical Ring and token
We consider available processors in a logical ring

The **ring** represents the research space to find allocation for processes before creation
To identify a dynamic role, a token allows the current owner to become the current strategy maker: the ring must be passed around after a maximum permanence in a node
The current manager can initially broadcast to all processors a request for their load state and then the load is distributed via the ring

Static and proactive organization
easy to maintain and also
to restructure fast to recover
in case of fault

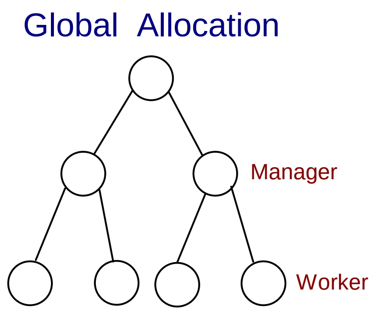


LOAD SHARING in MICROS

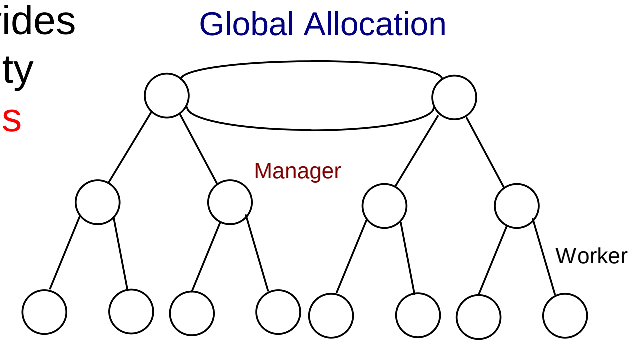
MICROS uses a logical hierarchy
The architecture is logical and the nodes are logically connected
Organization with roles in a farm

Worker → computing duties (**slave**)
Manager → handling and controlling role

The level number of depends on the workers



For **fault tolerance, MICROS** provides several managers and the possibility of introducing new nodes and levels by need
After the initial organization, the hierachy can shrink and expand



WORM LOAD SHARING

Some more dynamic approaches are novel and less statically planned

The work strategy of allocation is based the **cloning of worm segment on different close nodes**

A Worm is a set of multiple segments (each one executing a process) who can also communicate with each other for load sharing goal

A worm tend to colonize a node by installing a segment of the worm in the new node (one copy only)

The worm strategy is not planned in advance but expand in a dynamic discovery

the worm tries to expand by its segments that to find close free nodes to clone there, by using prompts and acceptance messages (called probes) sent by local decisions of segments that want to expand

LOAD BALANCING (DYNAMIC ONE)

GOALs of TRANSPARENT (to user) MIGRATION

- **Better, more efficient and more correct** resource usage
- **Balancing of computational and communication load**
- **Dynamic decisions and long term policies**

Requirements

Performance	to use resources at the best
Efficiency	limited overhead
Continuous operation	minimal intrusion

In general, the migration is part of the 'system functions' and it is not under user control but

Migrations can **interfere with normal application execution**

Transparency and automatic migration decisions toward a minimal cost and intrusion

MIGRATION – Some Considerations

The point is migrating or moving already established resources at run time with a minimal overhead

Any entity is in principle subjected to migration

DATA, OBJECTS, COMPONENTS, …PROCESSES MIGRATION

PROCESSES move from one node to another

the point for process is that we have an initial state and many updates when executing: which and how to move

Pre-emption

Priority to local usage

Multiple Migrations

To make in parallel many concurrent migrations

Avoid residual dependencies

The system must not have any trace of the moving of resources

Avoid thrashing

Avoid to move the same process without any execution of it

PROBLEMS in MIGRATION (INTERNAL)

In case of migration, the process must prepare the mobility phase and manage all resources previously available

→ **Environment change of the mobile resource**

- State identification

the process must identify which internal resources to carry on to the new location and begin to determine their internal state

- Block of the process itself before mobility

the process may have one part of state not transportable so to close before moving

Actions of closing local files or code to be managed (last wishes)

Actions of storing resources that can be moved and found in the new node to be enabled there again

- Block of activity to move

Completion of the activity on the old node and activation of mechanisms of movement on the new node

MIGRATION PROBLEMS (EXTERNAL)

In case of migration, during and after the migration

… there are messages to be forwarded and to be given back

→ **Change of name of mobile resources**

- Message redirection *pessimistic/proactive strategy*

The origin node keep track of the move and keep receiving messages and forwarding them to the new location

Chain of forwarding can grow for mobile processes

- Requalifying of allocation *pessimistic/proactive strategy*

The origin node keep track of the move and receive messages and forward them to the new location only during the transfer

Client nodes receiv the new location at reference

- Client Recovery *optimistic/reactive strategy*

The origin node does take any action.

Client messages can fail and it is client duty to find the new location

FIRST LESSON FROM MIGRATION

DETERMINE (for **processes**) **who, when, how, where** to migrate

Some criteria

- ***not all processes can migrate***

  Fixed are acyclic (short) ones and node dependent ones

- *It is opportune to have in any node a **migration handler***

Migration is based both on **policies**, and on **mechanisms**

MECHANISMS

Depend on the computational model and specificity of system

POLICIES

More general-purpose, independent from system

KEEP STRATEGIES AND MECHANISMS SEPARATED

The latter system tailored, the former can vary in system and can be under user control

MECHANISMS to ENABLE MIGRATION

Who migrates?

processes, passive objects (**file**), active objects, components, servers

RESOURCE composition and organization - discovery

Initial state: code + data (initial data)

Current state: data + visible resources (local and remote)

Computation block

Block of arriving messages: messages are either refused or forwarded

Transfer & Copy

There are two copies, an old and a new one: there is an activity of synchronization of the two data

Obsolete references

Requalification or other strategies

MIGRATION POLICIES

There are typically three phases

EVALUATION of load (**V**)

local load vs. global load

TRANSFER (**T**)

who to transfer and when to do it

LOCATION (**L**)

Where to migrate and re-insert the process

T & **L** are often intertwined and interdependent

NEED of integrating and interacting with local scheduling

There is an impact on the scheduling on both nodes of origin and arrival because of the competing with common resources

The planning can ease those steps

WHICH POLICIES of MIGRATION

STATIC **predefined** and a priori decided (**low cost**)

V *fixed threshold* as load (e.g., number of processes)

T moving of the *"newer" process*

L migration always from a *source node to a predefined sink node*

SEMIDYNAMIC predefined with **limited dependences** from **current state** – also using probabilistic policies (**limited cost**)

V *variable threshold* as load

T *cyclic identification* among processes

L *cyclic* allocation on sink node

DYNAMIC strictly **dependent on current state** (**even high cost**)

V comparison of load with neighbor (dynamic average load)

T information on process state

L discovery of sink nodes via messages in the neighborhood

MIGRATION POLICIES

POLICIES: SIMPLE vs. COMPLEX ONES

V T L for processes **acyclic** vs. **cyclic (normal duration vs daemon)**

V → fixed threshold vs. neighborhood comparison
T → process suitable for a specific neighbor or random choice
L → usage of **message called probe**

random, probabilistic, cyclic, shortest queue

unconditioned acceptance

probing, bidding

conditioned acceptance

probe: message to send to neighbor to ascertain possibility of moving
PROBING (**T & L together**)
to identify possible candidates to receive processes and pre-evaluate their reinsertion effect

DECISIONS in IMPLEMENTING MIGRATION

CENTRALIZED with a unique entity for controlling migration
DECENTRALIZED coordination of many different entities

implicit or explicit collection of information and distributed decision based on compared of state information (piggybacking)
favoring local movements in a neighborohood

RESPONSIBILITY couple SENDER-RECEIVER

SENDER initiative: the overloaded node must find the potential sink one (RECEIVER), asking for nodes receiving load

RECEIVER initiative: the underloaded node must find the potential source one (SENDER), asking in the neighborhood for load

MIXED solutions

SENDER initiative → more suitable for **low** system load
RECEIVER initiative → more suitable for **medium-high** system load

MIGRATION feasibility - LESSON

IMPORTANT RESULT

Migration has a cost, … but it may be effective

Even if **with simple policies** one can obtain **significant enhancement in a system (compared with no migration case)**

ANOTHER IMPORTANT RESULT

***More sophisticated policies does* NOT obtain significant enhancements** and cannot be **generally applied apart from specific (not so common) situations**

Some specific goals

- **STABILITY** avoid thrashing
- **EFFICIENCY** simple algorithm to compute and actuate
- **OPTIMALITY** not a real goal, but only sub optimality

OFF-THE-SHELF ALLOCATION

Data centers to make client life easier often offer ready-to-use standard allocations

In traditional on premises systems, resources are allocated exclusively and **for the whole time**, accounted also if **not used**

The **Cloud model allows a less traditional perspective:**
Resources are available pay-**per-use**
Also differentiating user type

- **Expert users** who have enough skill to which resources and how to manage them (in addition and subtraction)
- **Less expert users no so smart who** have available standard configurations **standard and ready-to-use**

Resources are available **by need** in an **elastic and flexible** way, by following closely the requirements with a continuous possibility of verifying current usage

CLOUD ARCHITECTURE

In case of Cloud, resources internally must be considered in a less traditional way

Not only you have the application mapping but you should consider that **very different execution environments and very different choices**

You can define and command:

- **logical resources** (already considered)
- **physical resources** (already considered)
- **virtual resources (**not only machines, but also **any kind)**

The degree of freedom you have are **many** and also **from different architectures** and **choices** can stem **very different final behavior**

So, you typically **decide**

how to put **your logical components over virtual resources**

and then also to **map the virtual over the physical one**

CLOUD CASE

We design an application thinking to **a client** that obtain **on-demand services** requested and obtained via **Web** and **the user must not worry (too) much about their management**

Their management is **Cloud-internally decided** and **provided**

Virtual and physical resources for Cloud are in **one data center** or in **different data centers (transparently)**

The user should definitely **use Resources-aaS** (**R**esources **as a S**ervice) and should expect a very **dynamic behavior** from the requested services

⇨ On need, the data center must prepare **new resources**, both physical and virtual ones, in a more or less automatic fashion

⇨ That makes the architecture perceived by user very **elastic, adaptable** and **flexible**

⇨ The problems are left to the **management of the data c**enter

CLOUD ARCHITECTURE

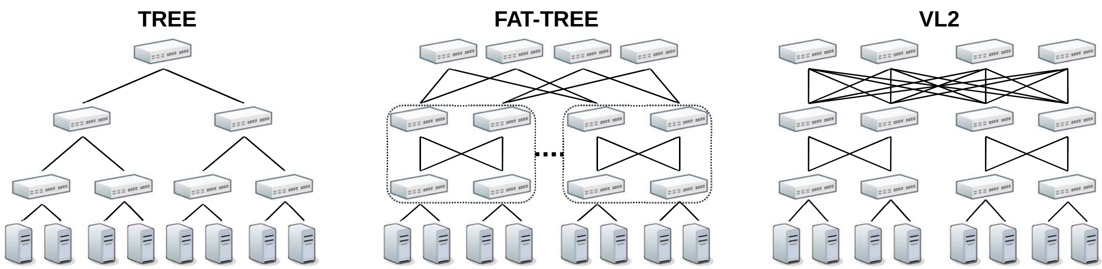
The Cloud makes an important step toward **transparency for users (PaaS, SaaS)**

But also makes available more low level details (IaaS)

In particular the data center complexity is visible inside

The data center has no **flat net** but typically **hierarchical ones that interconnect machines** and that can be **optimized by exploiting specific dynamic connections**

To reduce application time, the management can **allocate depending on internal data center interconnection**



INTEREST for DEPLOYMENT

Choosing a deployment instead of another

can have a big impact during the **specific execution** and must be carefully evaluated and decided

Let you assume you need communication resources,

- we must consider **internode communication tools available** whether resources will be allocated to different nodes
- we must choose the **most appropriate communication tools for allocation that we are determining** (in case of different and heterogeneous architectures support)
- we also need to **optimize communication tools** when resources are present on the same machine, **inter-and intra-node communications differentiating node** (as they often do the existing middleware)
- we need to verify that the deployment **is suitable with expressed communication tools** and does not **cause problems** (by identifying and eliminating bottlenecks and critical cases)

CLOUD DEPLOYMENT

Choosing a CLOUD deployment instead of another
can have a big impact during the **execution** and must be carefully evaluated and decided

Let you assume you need some resources and you do not have considered any policy,

- Typically you have several **setting** to decide among (some free, some are most expensive, …)
- You have to decide a suitable offering by considering the **average behavior and also its quality**: is it constant?, are there peaks?, are they regular?
- Your application has **specific requirements:** geographic allocation, reliability (multiple copies), QoS in terms of response time, specific persistency constraints, …
- Any specific **internal allocation constraints:** some parts must be close and heavily communicating
- **Last but most important: is your application compatible with the chosen Cloud?**

C/S Model as a SOA IMPLEMENTATION

Client/Server for any operation request
Intrinsically distributed as a model but
the model does not consider discovery agencies

Very high level communication rules where

client knows the server and interacts synchronously (result implied) and blocking (result awaited) by default

Model with tight coupling:

interacting parties must be **co-present for some time**

Obviously we are interested only in models inherently distributed and deployed, and leading to deployment really always distributed

There are many weaknesses and rigidities in C/S typically these usage difficulties are **overcome by small variations tailored** to specific needs

ADVANCED C/S MODELS - NOT ONLY C/S

Many variant of the Client/Server model

Novel variants

pull (synchronous non blocking)
(the client get afterwards the result, without waiting for it)

push (synchronous non blocking)
(the server gives the result afterwards to the client that do not wait for it)

delegation waiting for the result **(synchronous non blocking)**
(the delegate *waits for the client* and gives it the result)

notification for the result
(the delegate notifies the client that a result is arrived)

events (typically asynchronous, so non blocking)
(an event is generated from producer and advertised to consumers)

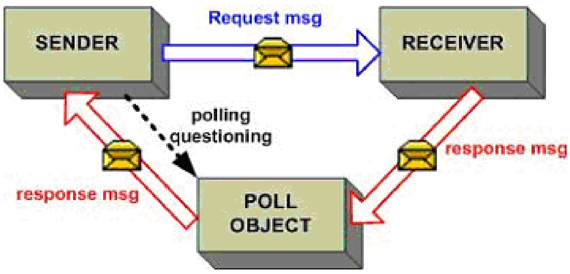
provisioning
(other parties can be interested in the call chain, apart from C/S)

DELEGATION – GET THE RESULT…

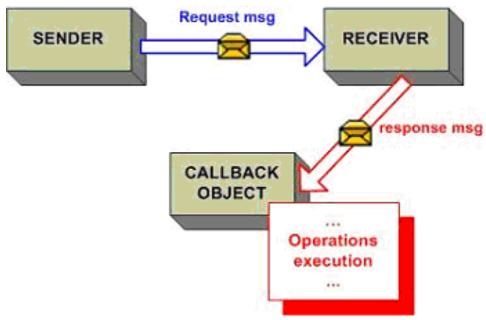
In a synchronous non blocking model, we may have a delegated entity fo the result

We add a new objects, typically called **Poll and Call-Back objects as intermediate entities**

Poll Object



Call-Back Object



Used for short operations and limited response time

Even long operations and indipendent from the client life cycle

We should define specifically the organizatn in any case

MESSAGE EXCHANGE

Model of MESSAGE exchange

very flexible but primitive, not user friendly

Sometimes the **message are only for the synchronization (signals) without any real data communication (carrying no information)**

Information exchange: **properties**

a/ synchronous	(**no** / result)
a/ symmetric	(the same knowledge of partner)
in/ direct	(intermediate entity or not)

Implementation

non/ blocking	(**un**/blocking of the sender)
un/ buffered	(non / message queuing)
un/ reliable	(with/without message loss)

Models with **multiple receivers** or **group messages**

multicast (MX) and broadcast (BX)

MODES of MESSAGE EXCHANGE

MESSAGE EXCHANGE varies a lot in different systems

Rendez-vous

One to one message exchange that is synchronous, blocking, symmetric, unbuffered, coupled (more than C/S)

With an intermediate entity (channel, …)

Message exchange typically asynchronous, non blocking, asymmetric, **decoupled** (less strict than C/S)

With intermediate entity & receivers group (events, …)

Message exchange typically asynchronous, non blocking, asymmetric, **decoupled and many to many**

C/S vs MESSAGE EXCHANGE

Client/Server

Model with strong coupling
implies **co-presence of interacting parties**

Mechanism suitable for high-level and simple communication

Very **high level** (very suitable for application usage)
but **not so flexible** for differentiated situations,
no Multicast (MX) and Broadcast (BX)

Sender/Receiver message exchange

Model with loose (minimal) coupling
imposes no **co-presence of interacting parties**

Very flexible, primitive, and expressive mechanism, maybe not so easy to use

Very **low level** (and suitable for any system potential usage):
many **differentiated modes of usage**, even easy support to any kind of needed communication, e.g., any form of **MX** and **BX**

DE / COUPLING

Communication tools can impose some ***constraints on the interacting entities*** ***(also no imposition)***

These constraints can even induce severe limitations on the interaction and force knowledge needs sometimes not required

Different ways of coupling

- **space**

The interacting entities must know each other and be colocated

- **time**

The interacting entities must be present at the same time (they should share some intervals of time)

- **synchronization**

The interacting entities must wait for each other and are subjected to reciprocal limitations and blocks

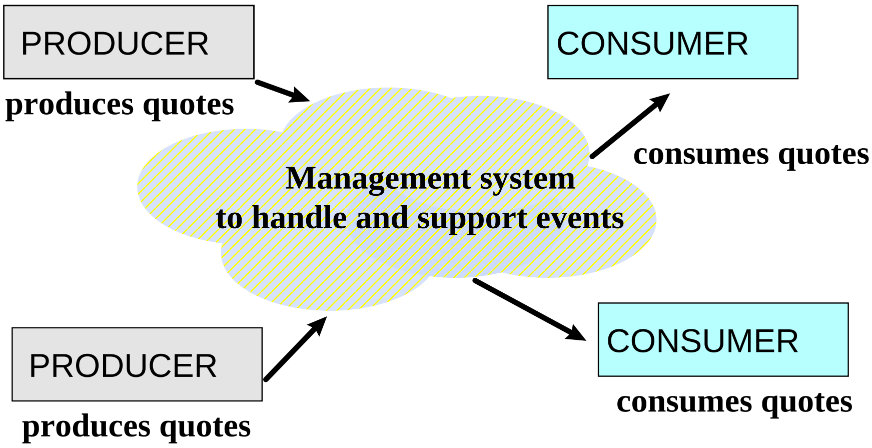
Decoupling becomes a factor to enable greater flexibility and to leverage the potential distribution of the load in a system

EVENT and PUBLISH-SUBSCRIBE

Decoupling between interacting entities

Events are generated by **producers,** free of doing it when they intend to generate events (**publish or PUB**) ***without worrying about delivery***

Consumers register their interest in specific events, topics, … (they have **subscribed SUB**) and the **event support** is in charge of the delivery



Producers and **consumers** are not required to be **present at the same time**

FROM LOCAL EVENTS

Different model than a synchronous requests of C/S t

The Framework tends to reverse the control for low level events

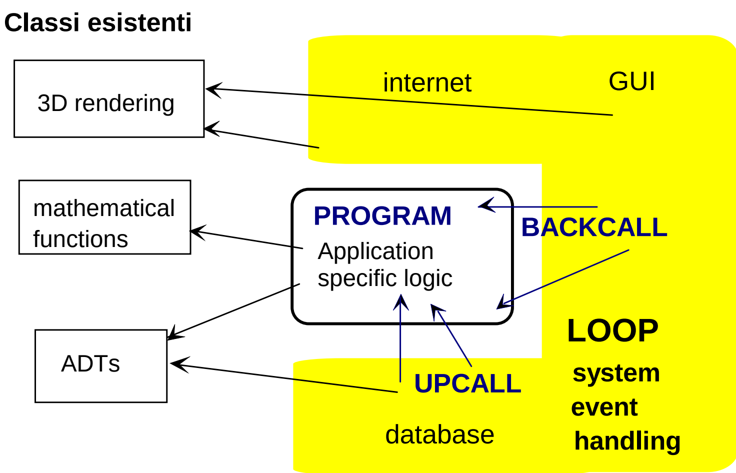
The user process does not wait for result but register with a handling action

Example: Windows asks all processes to provide a waiting loop to serve with the it is going to raise to them (and send to them)

When the result is produced the event is raised an the process can go on

Responses from the framework to the user are called **backcall** or **upcall**

They are similar to an asynchronous event generated by the framework and that application must manage through a **handler function** specified by the user



Available Services and Functions

EVENT SYSTEMS (DISTRIBUTED)

Event systems have been modeled and designed **without any locality constraints (no coupling)**

- **The model has its strength in the non-locality of interacting entities only local implementations**
- **Local implementations are not interesting** (*such as using the sharing on the same node, between producer and consumer),* ***arbitrary, and not meaningful downsizing*** *of the model*

Develop a system for events not taking into account the potential decoupling, ...

means to use badly the model properties, one of the worst things we can do to a technology

- If you constrain the events to the co-residence and co-presence of interacting entities, you produce a deployment that contrasts with the basic event model

EVENT SYSTEMS: INDICATORS

Event systems have been defined to model **large systems** and **scalable ones**

Some **indicators are core ones**

- **Cost in distributing events (to limit)**
- **Performance (to optimize)**
- **Scalability (to keep high)**
- **Latency (da limit in time)**
- **Pervasivity of provided services (to keep high)**
- **Independent develop and execution (high)**
- **Fault tolerance (maximal possible)**

When you implement event systems you start from **viability**, to mean that you grant that the **indicators are scalable**, in other words for all distributed implementation indicators keep **acceptable values**, possibly **'costant'**… at least **tested**

EVOLUTION of EVENTS

Primitive events

some **events** are **on/off signals without any content information**

interrupt events and signals triggered by low-level handling functions

Events that carry contents

some contents carry information and one can also **filters** events based **on interest** about **specific information**

RSS as an example, where there is interest only to specified topic and users can register to specific interests

Events with quality - Quality of Service

These **events** can provide **differentiated service for different users:** they can persist and be maintained for all or some users, the delivery can be different depending on receivers, …

Persistent events*: users not online do not lose any event, kept to be delivered a.s.a.p. when they are on*

Event priority*, e.g., depending on the number of resources devoted to users*

PUBLISH-SUBSCRIBE SYSTEMS

PUB-SUB systems are **advanced distributed systems** based on the **event model** and **message exchange** to take the best advantage of the flexibility and the decoupling of interaction

to increase **scalability and distribution**

The PUB-SUB model has also many other flexible aspects…

Message filtering based on

topic-based: based on a predefined topic (a specific interest between different channels: such as a specific RSS)

content-based: based on message contents (some keywords or also some more complex relationships)

type-based: based on message type (in case of different message types and a selection done on them)

Quality of Servizio (QoS) over messages

Persistency, Priority, Guarantee of maintenance and duration, …

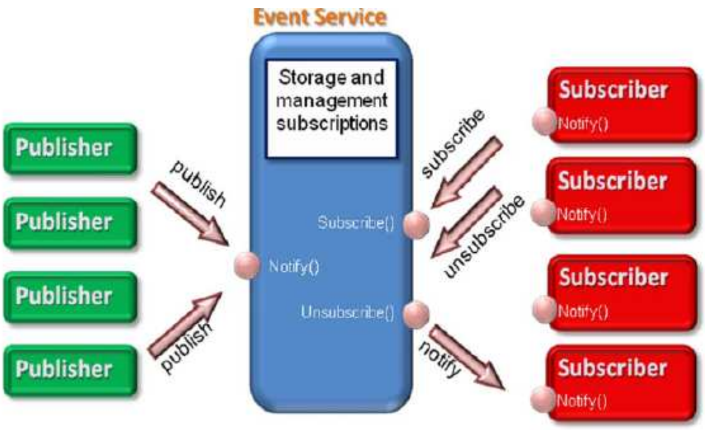
PUBLISH-SUBSCRIBE SYSTEMS

Real PUB-SUB systems support **operations for consumer subscription**

producers called also **publishers** provide events (they might ask which are current subscribers)

consumers or **subscriber** that have subscribed must receive events, via a notification

an **infrastructure** must ensure and grant the operations



MODELLI DISACCOPPIATI - TUPLE

TUPLE MODEL for decoupled interaction

A general model for **communication** and **synchronization**

designed as a *shared memory abstraction* + *communication*

A **tuple space** is a set of **structured relationships**, organized as a container for *attributes* and *values* for PUB-SUB

On a tuple space tuples can be deposited / extracted **high-level information** without **causing any interference** or incorrectness

A possible relationship: **message** (**from**, **to**, **body**)
The space is a container of **tuple values** according to the defined attributes
(the *attribute types,* here ASCII string)

Tuple values *message*: {**Antonio**, **Giovanni**, **msg1**}
{**Giovanni**, **Antonio**, **msg1**} {**Antonio**, **Giovanni**, **msg2**} …
There are no constraints on tuples that can be deposited and stay in the space forever (almost, it is a model) so **without time** or **space limits**

TUPLE - Linda (Gelernter)

Operations of **In** e **Out on the tuple space**

Tuple spaces offer operation always possible and correct for **readers (In consumer)** and writers (**Out producers**) competitors with access based on attribute contents

Out inserts one tuple in the space and **In** extracts one tuple from the space

The **Out** operation **emits a tuple** on the space available for a match with an In request and the tuple stays there until it is consumed by one corresponding In only

The **In** operation **extracts one matching tuple** from the space, if exists If it dose not exists, the In waits until one is received for the **match that is based on pattern** on the attribute values

In case of match with **multiple tuples**, only one is **non-deterministically extracted**

Out: message (P, Q, text1)

In: message (?from, Q, ?body)

The **In** may have name of attributes for larger matches

The **In** waits for one tuple with the second attribute the string Q, and give to the consumer the values **from**(=P) e **body**(=text1) of the matching tuple

DECOUPLING TUPLE

Tuple spaces

The communication is rather **decoupled** and **asynchronous**

In time

A producer can deposit tuples and go away, and only after a **long time**, the consumer can arrive and get the tuples

In (reciprocal) knowledge (space & synchronization)

The consumers do not know the producers in any way, but only the tuple contents they cannot interfere in any way with production (*one **in operation** extract one tuple, other **in-s** are queued and wait for their matching tuples and **outs operations***)

In quality - QoS

Tuple spaces are **persistent** and their requirement is to **maintain deposited tuples without limit (in memory and time)** without any preference for a specific requesting process

Tuple spaces (local implementation) are available to favor local communication **well formed** and **with high level operations**

Javaspaces, …

TRANSPARENCY vs. VISIBILITY

TRANSPARENCY (opposed to VISIBILITY)

Access	homogeneous access to local and remote resources
Allocation	allocation of resources independent from locality
Name	name independence form the node of allocation
Execution	same usage of both local and remote resources
Performance	no differences in usage perception in using services
Fault	capacity of providing services even in case of faults
Replication	capacity of providing servicing with a better QoS via transparent replication of resources

Is **transparency** always an optimal requirement to consider?
at **any cost**, at **any system level**, for **any application** and **tool**

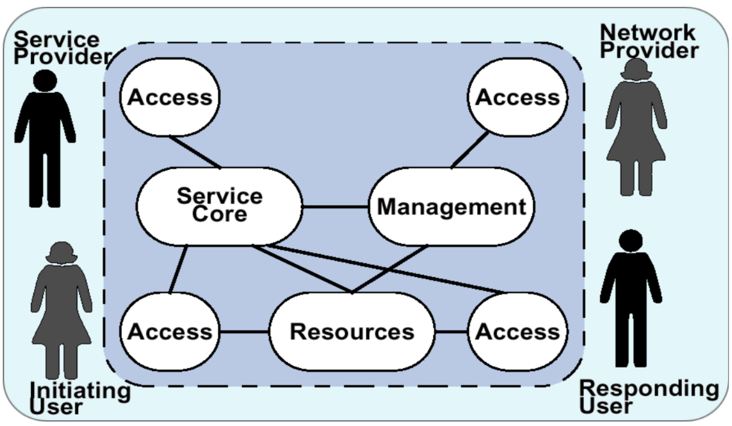
(??) **Location-awareness** to provide services that strictly depends on awareness and visibility of **current allocation**

TINA-Consortium – beyond a C/S

Telecommunications Information Networking Architecture
TINA-C - Consortium defines new service models and availability constraints

On the external site**, several service users** are considered (not only two parties, but several, a videoconference)
On the internal support, there **are several other parties**, in charge of some **aspects and their integration** makes available the whole service
All possible **providers** are included, both **network and service**

Another important aspect is the **management plan,** always crucial and core



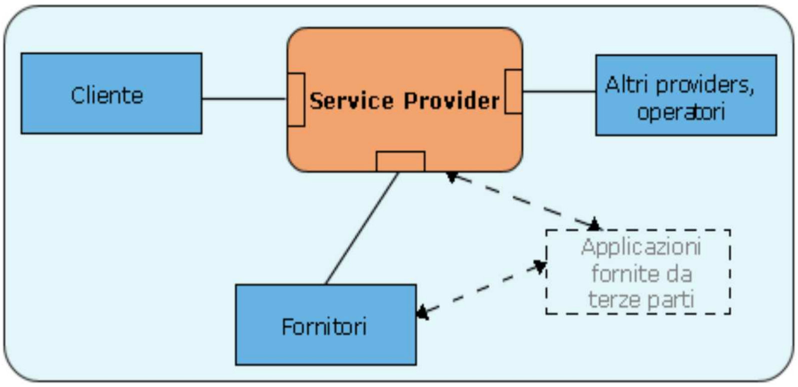
TINA-C – PROVISIONING & QoS

Agreement and negotiation

between the service parties involving communications resources must also take into account the need of **doing resource management** during the service life cycle

All parties must cooperate toward a respect of a SLA over QoS to maintain as a precise requirement

Only if the service is completely provided with the negotiated QoS is considered compliant, successful, and to be paid



CLOUD PROVISIONING & QoS

In a Cloud environment, we have a *similar setting*

On the **external site**, **several users are possible** and they may interact *among themselves* but also

- must **discover services** and interact with them
- can **pack some resources** inside the Cloud

On the **internal site**, there are several other aspects to be considered

- Many services may **be made available**, at different levels
- Services can be **temporary or persistent**
- User must be able to **control resource consumption**
- User can command not only available services, but **ship new ones** and control them and manage their lifecycle
- Any resource must be **available for access, inspection, maintenance**, and **changes** (even in case of sharing)
- Other constraints may be part of the SLA and internal management

REMOTE MANAGEMENT FOR QoS

In **remote environments**, such as in **outsourcing** and in **Cloud ones**, it is compulsory something to ascertain the current state of the remote installation, not only for accounting purposes

we have to offer a very rich **management interface**, to allow to:

- **Access to any user related resource** (processing, memory, persistent data, network, … any *-aaS)
- **Control of the consumption** of any user related resource (current state, history for some periods, peaks, trends, … user-defined indicators)
- Discovery of **new services and new available resources** (new service can offer off-the-shelf ready-to-use solutions)
- Installation of special **user settings and environment** (new service to be developed from composing available ones or in a more specifically client-tailored way)
- Enlargement to **federated environments** for resource integration

COMPUTATIONAL MODELS

INTRINSIC COMPLEXITY of the algorithms

dependence from problem dimension called **N**

complexity in time CT(n) (abbreviated as **T(N)**)

complexity in space CS(n)

Let us think to potentially parallel multiprocessor solutions (with **P** as **parallelism degree**), all to be considered for any specification and execution that can accommodate computation (i.e., as part of computing of the algorithm)

COMPLEXITY

T(1,N) **sequential** solution $T_1(N)$

T(P,N) **parallel** solution **with P** processors $T_P(N)$

SYNTHETIC INDICATORS

SPEED-UP *Improvement from sequential to parallel*

S(P,N) = T(1,N) / T(P,N) **$S_P(N) = T_1(N) / T_P(N)$**

EFFICIENCY in *resource usage*

E(P,N) = Speed-up / Number of Processor

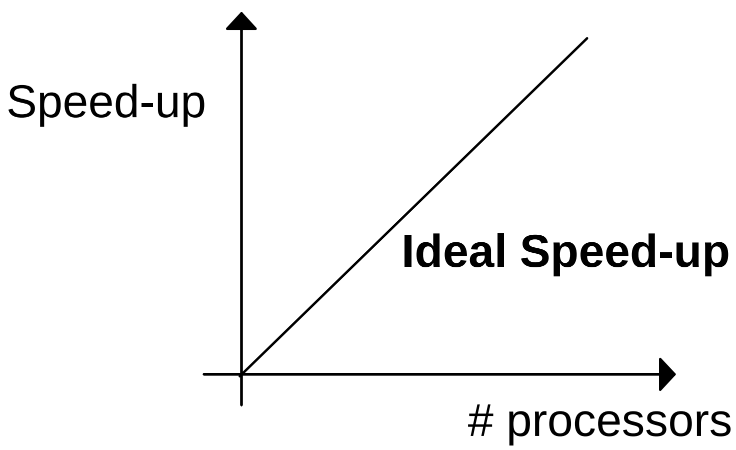
E(P,N) = $S_P(N)$ / P **$E_P(N) = T_1(N) / P\ T_P(N)$**

$S_P(N)$ up to ***P at most*** *and* *$E_P(N)$* ***1 at most***

The speed-up is the **potential improvement** when you introduce a **variation in processor numbers**, i.e., **real parallelism**

IDEAL INDICATORS

We assume and consider **average values**
ideal both SPEED-UP and EFFICIENZA



We are interested in the full range of results, so we average them bearing in mind that there may be specific cases of for only special cases depending on the algorithm

GROSCH LAW & LOADING FACTOR

Grosh law
The best deployment for a program is
a sequential execution by using a unique processor

N and P correlation:
We can assume N independent from P, or dependent from P
Loading factor or L = N / P

dependent size	(N function of P)
independent size	***(very interesting at N growing)***
identity size	(N == P)

GOAL
Which is the best choice and how to find the best approximation for any algorithm we want to explore in behaviour

SPEED-UP

Which is the best **speed-up** possible when passing from a sequential execution to parallel ones…
So how to get **optimal advantage from parallelism**

Amdhal law
the speed-up limit stems from the intrinsic sequential part

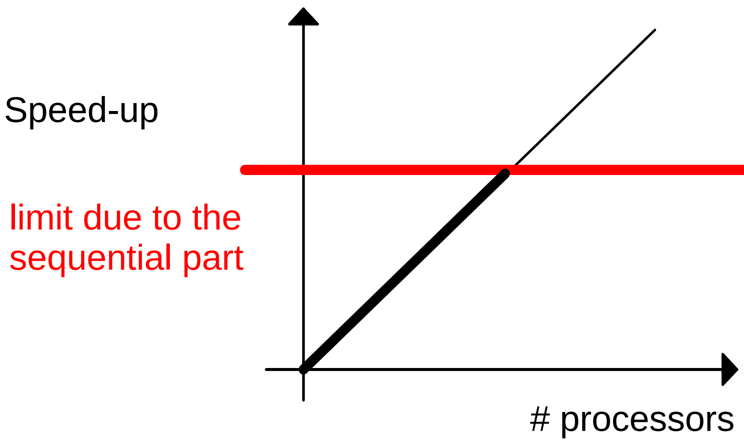
Any program can be split into two parts:
one **(potentially) *parallel* part** and ***sequential*** part
the latter is the limit to the speed-up

If a program consists of 100 operations with
80 ops can go parallel and
20 ops must be executed in sequence

With any number of processors, even 80 →
speed-up cannot be better than 5
Of course, it can be worse that that ….

MORE ON INDICATORS

Considering both **SPEED-UP** and **EFFICIENZA**



We cannot exceed the limit of speedup due to Amdhal's law

The speedup is limited, as well as the efficiency

We have first a linear zone at P growing (of growing in speed-up) then, we may have a **constant speed-up** but **lowering efficiency**

SPEED-UP (OPTIMAL?)

Is there any general low to get optimal indicators?

Heavily Loaded Limit $\quad \mathbf{T_{HL}(N)} = \inf_P \boldsymbol{T_P}(N)$

HL is for the ***P*** with which we get the least complexity of the algorithm (i.e., in our case the minimal T)

Typically, the optimum is when **N/P** is very **high**, i.e., if all processors are **very loaded,** anyone with a heavy **load to carry out** *(considering the limit of the limit of the sequential part)*

$T_P(N) = \mathbf{T_{CompP}} + \mathbf{T_{CommP}} \qquad \mathbf{T_{CompP}} = T_{CompPar} + \mathbf{T_{CompSeq}}$

$T_P(N) = T_{CompPar} + \mathbf{T_{CompSeq}} + \mathbf{T_{CommP}}$

Amdhal law bases on the ratio between the two parts of the algorithm (sequential and parallel) to identify the bottleneck

A small CASE STUDY (N==P)

Problem of dimension N by using P processors
The algorithm is the *sum of N given integers*
Complexity of sequential solution **O(N)**
Complexity of parallel model *identity size* (N == P)

We made available a number of processors P connected in a binary tree: any leave machine gets two integers and pass up the sum of them upwards; the root gets the final result by summing its two numbers and passes it to the final user

$N = 2^{H+1} \sim= P = \mathbf{2^{H+1}-1}$ (N values ~= P processors in the tree)

$H = O(\log_2 P) = O(\mathbf{\log_2 N})$ i.e., $H = \mathbf{\log_2 N} =\sim \mathbf{\log_2 P}$

$T_P(N) = O(H) = O(\mathbf{\log_2 N}) =\sim \mathbf{2 \log_2 N}$

Values flow from **leaves up to the root**, and any machine in the tree sum them up at **any step when they get data** (of course, we have to consider the time for the data communication)

Again for the CASE STUDY (N==P)

Efficiency goes to zero

$L = N / P = 1$

$\mathbf{S_P(N)} = T_1(N) / T_P(N) = O(N) / O(\log_2 N) = O(\mathbf{N/ \log_2 N})$
$\mathbf{S_P(N)} = O(P/ \log_2 P)$

$\mathbf{E_P(N)} = T_1(N) / P\ T_P(N) = O(1/ \log_2 P) = O(\mathbf{1/ \log_2 N})$

The larger the number of processors
(the speed-up increases) but the less is the efficiency

The processors work effectively for a fraction of the total time, much less of the entire solution time (EP(N) decreases with increasing P)

The CASE STUDY (independent size)

Problem of size N using P processors

If we can divide the problem, by putting together a **local work** and the **communication part**, where the **local computation can engage all processors** in any phase**,** we can obtain **better indicators**

Any processor has **some local work load factor** (to compute the sum locally) and a phase of **exchange of information** (Comm) to combine the results

$L = N/P$

$T(P,N) = O(N/P + \log_2 P) = O(L + \log_2 P)$ ossia $T_{Comm} + T_{Comm}$

$\mathbf{S_P(N)} = T_1(N) / T_P(N) = O(\mathbf{N / ((N/P) + \log_2 P)}) =$

$O(\mathbf{P / (1 + P/N \log_2 P)})$

$\mathbf{E_P(N)} = T_1(N) / P\, T_P(N) = O(\mathbf{1/(1 + P/N \log_2 P)})$

N>>P **speed-up goes to P and efficiency goes to 1**

MORE on the CASE STUDY

A more precise computation of indicators in the case of the sum of N integers with P processors with both local load and communications of data

Let us consider the same unit cost for any sum and communication

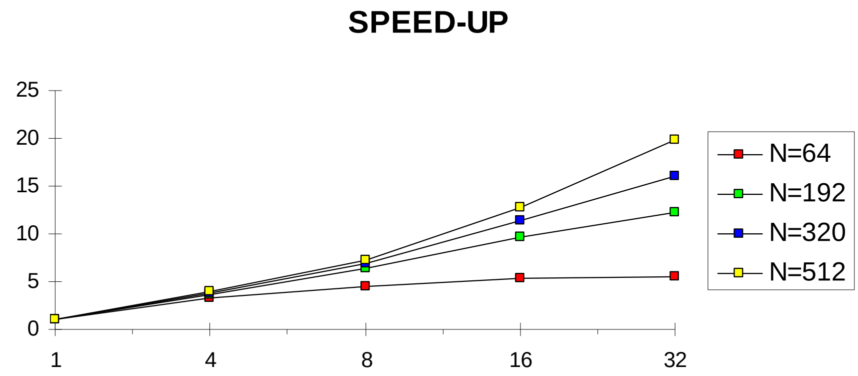
$\mathbf{T_P(N)} =\sim N/P + 2 \log_2 P$ total number of nodes $P = 2^{H+1}-1$

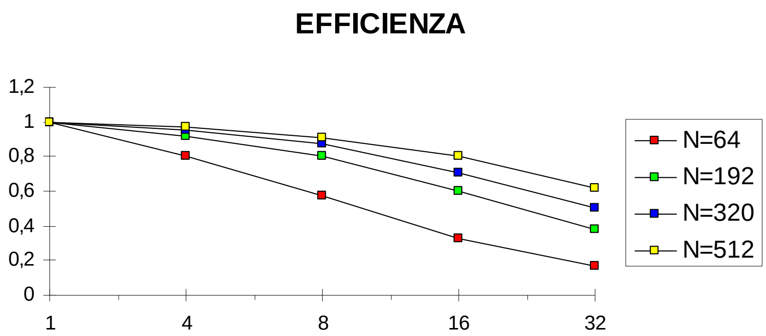
$\mathbf{S_P(N)} = N / (N/P + 2 \log_2 P) = \mathbf{N P / (N + 2 P \log_2 P)}$

$\mathbf{E_P(N)} = \mathbf{N / (N + 2 P \log_2 P)}$

Both indicators depends both on **P** and **N**

In graphical terms





SPEED-UP and EFFICIENCY INDICATORS

PROBLEMS

- we consider the O() so with a constant factors
- the worst case is not considered (it can be important)
- we neglect several issues outside

We also neglect
Moving of I/O data &
mapping (specific deployment)

In the real world →
We need also consider other communications for the application (also before and after the application run)

Initial transfer of data values
Print & manage of intermediate values
Harvesting and handling of final results

MORE on the CASE STUDY

Complexity of the parallel model *heavily loaded limit*
At L growth **$T_{P\,HL}$ (P,N)** $= O(L + \log_2 P) \Rightarrow O_{HL}(L)$

$S_{P\,HL}$ (N) $= O(LP) / O(L + \log_2 P) \Rightarrow$ **O_{HL} (P)**

$E_{P\,HL}$(N) $= O(LP) / O(LP + P\log_2 P) \Rightarrow$ **O_{HL} (1)**

If intuitively we overload all node
Then, the loading factor L is very high ⇨
We can also reach both
an ideal speed-up and an ideal efficiency
by loading at the best all processors, without leaving any node with a low level of load, and the risk of becoming idle

MAPPING

Let us assume to have made a mapping in an optimal way (**configuration** and **deployment**)
Too often we cannot decide the best allocation
Typically we have dynamic problems in communications in the run
We can consider a new function the **Total Overhead, or T_0**

To keep into account the time and resources spent in other actions, such as ***communication***
T_1(N) sequential execution time
T_p(N) parallel execution time
T_0(N) $= T_0(T_1, P) = P * T_P(N) - T_1(N) = |P * T_P(N) - T_1(N)|$
When you work at the optimal efficiency, you have no overhead
T_0(N) = 0 $\Rightarrow P * T_P(N) = T_1(N)$

OVERHEAD TIME

$\mathbf{T_0(N) >= 0}$ ⇨ $T_1(N) <= P * T_P(N)$ i.e.,

$$P * T_P(N) = T_0(N) + T_1(N)$$

T_0 indicates the lost work

$\mathbf{T_P(N)} = (T_0(N) + T_1(N)) / P$

$\mathbf{S_P(N)} = T_1(N) / T_P(N) = \mathbf{P * T_1(N) / (T_0(N) + T_1(N))}$

$\mathbf{E_P(N)} = S / P = \mathbf{T_1(N) / (T_0(N) + T_1(N))}$

$\mathbf{E_P(N)} = 1 / (T_0(N)/T_1(N) + 1) = \mathbf{1 / (1 + T_0(N)/T_1(N))}$

We should make very extensive campaigns of data collections to find out the **real dependencies** of $T_0(N)$ from N and from P

AGAIN for the CASE STUDY

More, in the case of the addition of N numbers with P processors

Let us consider unitary the cost of a sum and any communication

$\mathbf{T_P(N)} =\sim N/P + 2 \log_2 P$ total number of nodes $\mathbf{P = 2^{H+1}-1}$

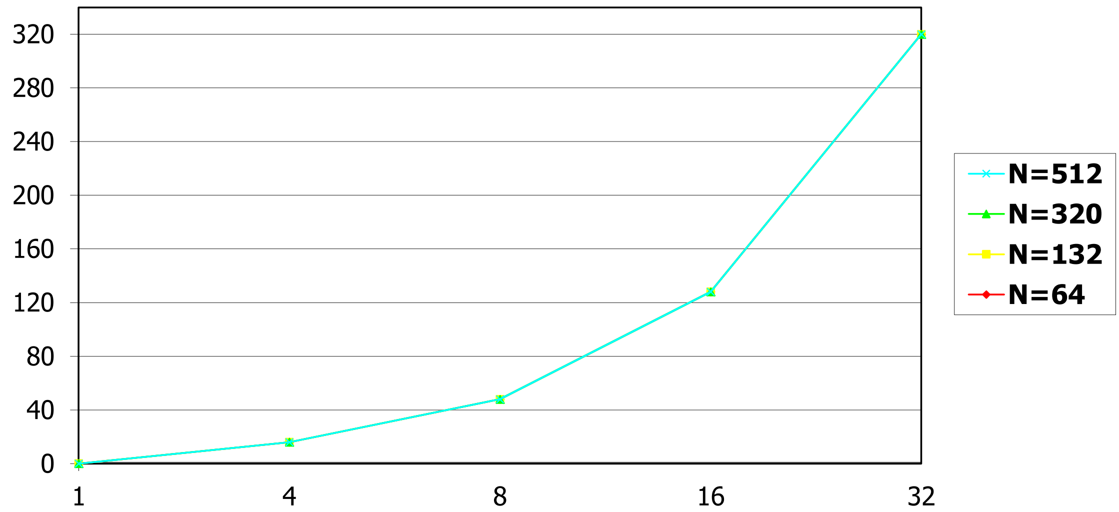
$\mathbf{T_0(N,P)} = P\, T_P(N) - T_1(N) =\sim P\,(N/P + 2 \log_2 P) – N$

$\mathbf{T_0(N,P)} =\sim \mathbf{2\, P \log_2 P}$

The T_0 overhead depends mostly on the **number of engaged processors**

The growth stems from the necessity of coordinating the application workflow, bot for the initial phases, during main execution, and after for results collecting

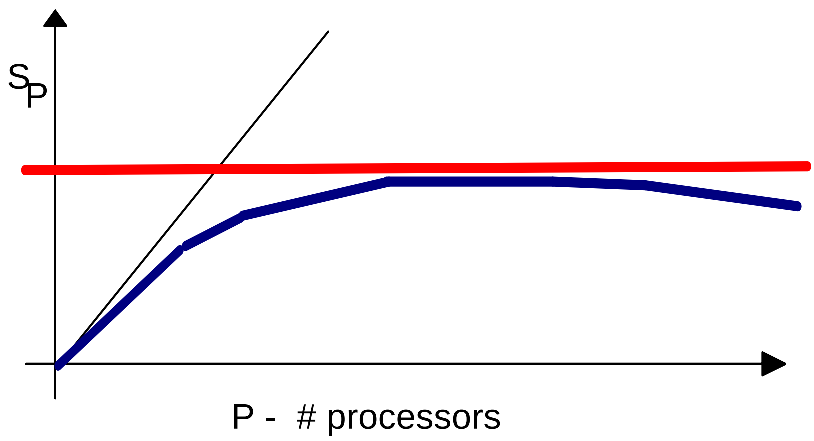
Graphically for an example T_o



The curves are the same

MORE REAL INDICATORS

Considering the real **SPEED-UP in a less ideal scenario**



We cannot get for long an ideal speed-up

The speed-up is usually constrained because of the overhead

Typically, we have an initial linear **behavior**, the a **constant growth**, then a **slow diminishing due to the overhead**

ISOEFFICIENCY

$\mathbf{E_P(N,P)} = \mathbf{1/}\ (T_0(N)/T_1(N) + \mathbf{1})$ $\mathbf{T_1(N)}$ as the useful work

Goal ⇨ to **keep costant** the **efficiency**
$T_0(N)/T_1(N) = \boldsymbol{(1 - E) / E}$ $T_0(N) = (1 - E) / E \ \ T_1(N)$

$\mathbf{T0(N,P) = (\ (1 - E)/\ E)}\ T_1(N,P)\ \mathbf{=\ K\ T1(N)}$

$\mathbf{T_0(N,P) = K\ \ T_1(N)}$ by using a constant (?) **K factor**

The costant K (?) is an indicator of system behavior

In the example (1 node /1 value) K non costant al all
For the tree case, K depends both on P & N
and it is approximately ($\mathbf{2\ P\ log_2\ P\ /\ N}$)

ISOEFFICIENCY FACTOR

Isoefficiency function
If we keep N constant and vary P, **K can indicate whether a parallelizable system can maintain a constant efficiency**
→ i.e., potentially **an ideal speed-up**

if K is small ⇨ ***high scalability is possible***
Se K è elevata ⇨ ***less scalable system***
K non constant ⇨ **non scalable systems (mostly all)**

In the tree case, K is $2\ P\ \log_2 P\ /\ N$
so the system is scarcely scalable (if any)

In general, all reals ystems are all non scalable (sic ☹)

A MEDITATION CASE

Let us assume that we are a system manager of a data center and have a **general application (proposed by a user)** and we know it **consists of Q processes**
We have a **very large number of processors available**
HOW TO manage the processor allocation?

To state a policy on the processor number to be used, you may consider (if relevant and it is feasible):

How are the processes?
how they interact?
How to load any single node?
Application need QoS, replication, objects, classes?

the Grosh law says that the best way is to use one processor, if it is possible

NEVER POSSIBLE!

A REFLECTION CASE

*Tyr to consider the experience of a data center where many applications arrive to be run fast and resources must be kept into account, and **always be used at best***

heavily loaded limit is a good target
good efficiency can steam from high loaded processors

Keep in mind your experience of PC and personal users.
The Grosh law
The detail of the applications are important for efficiency?
How approximate the loading factor in terms of processes and processors? Define an expression in term of them

But try to discuss **how many processes** are reasonable and effective