TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Distributed System Architectures

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2021/22

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

inizialmente sarà il nostro campo di interesse!

Software vs. system architecture of DS

- **Software architecture**: logical organization and interaction of software components (each other and with the surrounding environment) constituting the DS
  - ie deployment del sw sul sistema.
- **System architecture**: final instantiation (including deployment) of a software architecture
  - Software components need to be placed on system resources
  - E.g., a container containing a microservice needs to be instantiated on a machine

- Let's first focus on software architectures for DS

Architectural styles (*patterns*) for DS

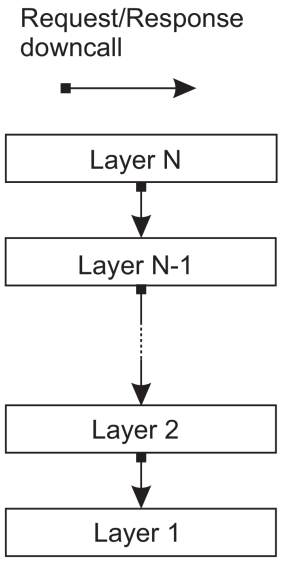
- Pattern: a commonly applied solution for a class of problems
- Architectural style: a coherent set of design decisions concerning the software architecture
  - Formulated in terms of definition and usage of components and connectors
- Component (*building block*)
  - Modular unit with well-defined required and provided interfaces
  - Fully replaceable within its environment
- Connector
  - Mechanism that connects components among each other, mediating communication, coordination or cooperation among components
  - Example: mechanisms for (remote) procedure call, messaging

Main architectural styles for DS

- Layered style
- Object-based style
- RESTful style
- Event-driven style
- Data-oriented style
- Publish/subscribe style

Layered style (componenti disposti in livelli differenti)

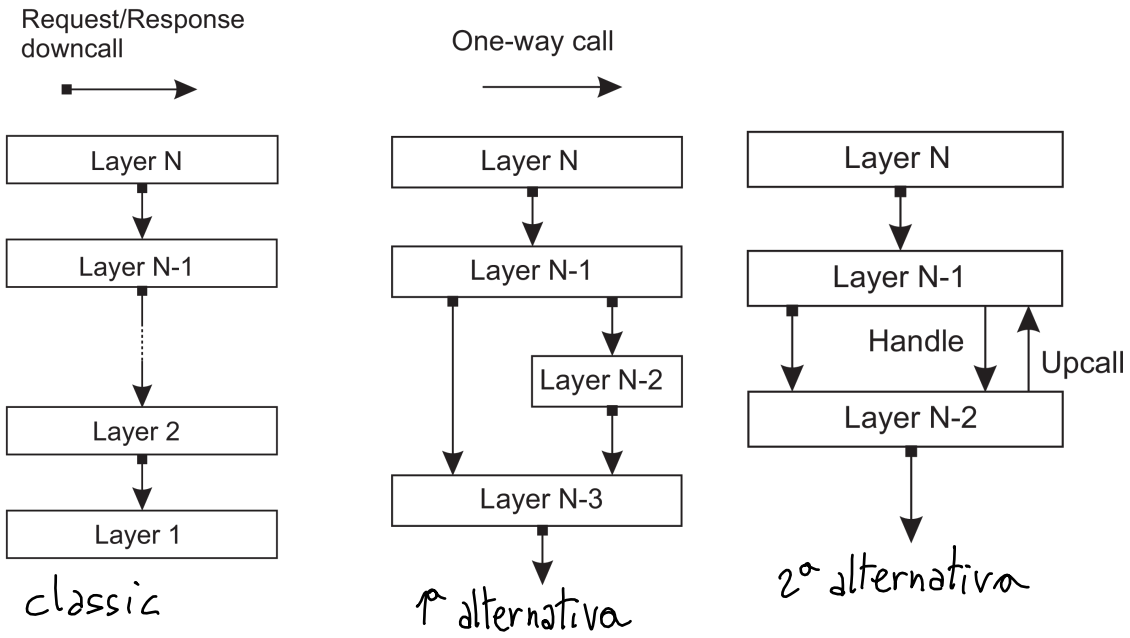
- Components organized in *layers*
- Component at layer *i* invokes component at layer *j (with j<i)*
- Components communicate by exchanging messages
  - Request/response downcall
- Pros: separation of concerns (funzionalità) among components (software)
  - E.g., web app based on MVC design



(3 schemi "particolari")

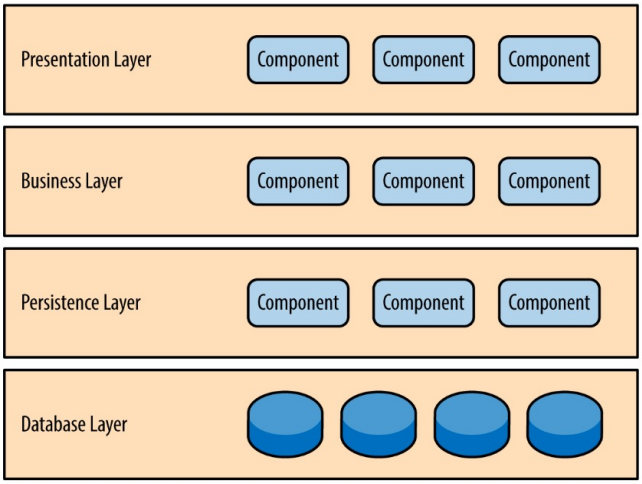
Layered style

- Different layered organizations, poca flessibilità



Application layering

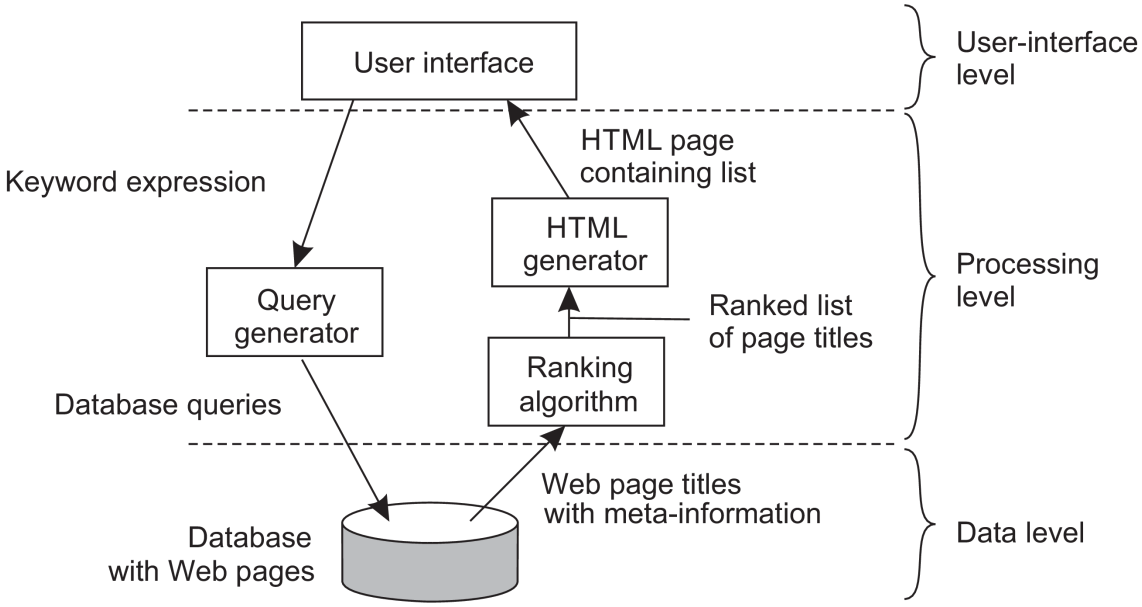
- Typical layers in a distributed app with layered architecture: presentation, business, persistence, database
  - In some cases, business and persistence layers are combined into a single business layer



14/10/22

Application layering: example

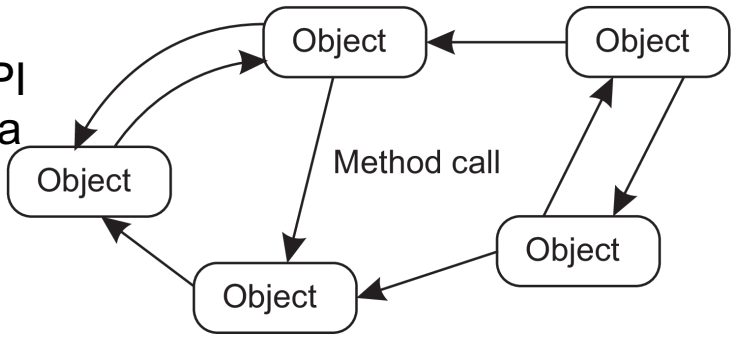
- Simple Web search engine



un componente == oggetto, incapsula le sue caratteristiche

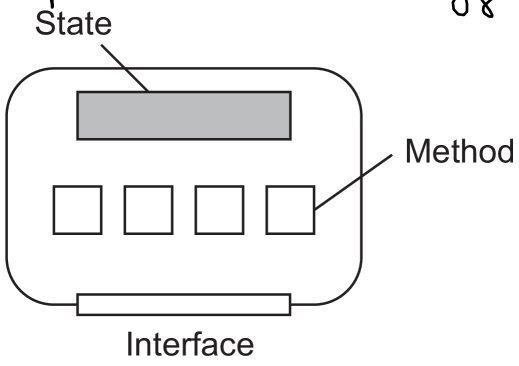
Object-based style (evoluzione del precedente)

- Component=object: encapsulates a data structure and provides API to access and modify data
  - Encapsulation and information hiding reduce management complexity
  - Object reusability among different apps
  - Wrapping of legacy components (posso includere componenti LEGACY come oggetti)



- Communication between components: through remote procedure call (RPC) or remote method invocation (RMI)

(OK anche per stile "a livelli")



- No disaccoppiamento temporale

RESTful style (applicazione è un insieme di risorse gestite manualmente)

- DS as collection of resources, individually managed by components
- **Representational State Transfer (REST)**: proposed by Roy Fielding, co-author of HTTP/1.1
  - Resources may be added, removed, retrieved, and modified by (remote) applications (HTTP methods) uso HTTP per interfaccia servizi.
  - Resources are identified through a single naming scheme (Uniform Resource Identifier, URI)

```
URI = scheme:[//authority]path[?query][#fragment]
authority = [userinfo@]host[:port]
```

(https, name server)

- Components expose a uniform interface
- Messages sent to/from component are self-described
- Interactions are stateless (leggero, se si rompe riparto e via)
  - State must be transferred from clients to servers

REST operations

- Basic operations
  - Use HTTP methods: GET, PUT, POST and DELETE

Operation	Description
PUT	Create a new resource
GET	Retrieve the state of a resource in some representation
DELETE	Delete a resource
POST	Modify a resource by transferring a new state

Example: REST API in AWS S3

- S3: Cloud storage service offered by AWS
- Objects are stored into buckets ("contenitori")
  - Flat structure: no directory hierarchy
    - However, logical hierarchy can be created by using object names that imply a folder structure, e.g., `photos/2021/September/sample.jpg`
      non c'è il concetto di cartelle (NESTING) ma posso nominare il file come un Percorso
  - Operations on object *objectname* stored in bucket *bucketname* require to specify them in the URI together with AWS region, e.g., `https://bucketname.s3.Region.amazonaws.com/objectname`

- All operations are carried out by sending HTTP requests:
  - Create a bucket/object: `PUT`, along with the URI
  - List objects: `GET` on a bucket name
  - Read an object: `GET` on a full URI

REST API in AWS S3

- To retrieve object from bucket (`GetObject`)

```
GET /puppy.jpg HTTP/1.1
Host: examplebucket.s3.us-west-1.amazonaws.com
Date: Mon, 11 Apr 2016 12:00:00 GMT
x-amz-date: Mon, 11 Apr 2016 12:00:00 GMT
Authorization: authorization string
```

- To add an object to a bucket (`PutObject`)

```
PUT /ObjectName HTTP/1.1
Host: examplebucket.s3.us-west-1.amazonaws.com
Date: date
Authorization: authorization string
```

See https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html

REST API in AWS S3

Notes:

- in the example we use virtual hosted-style URL, i.e., bucket name (`examplebucket`) is part of hostname in the URL, e.g.,

```
https://examplebucket.s3.us-west-1.amazonaws.com/puppy.jpg
```

- Alternative is path-style URL, e.g.,

```
https://s3.us-west-1.amazonaws.com/examplebucket/puppy.jpg
```

- HTTP Authorization header to authenticate S3 request

- You need the relevant permission for operations, e.g., you must have WRITE permissions on a bucket to add an object to it
  - You will see how to set IAM permission

I tre stili hanno forte dipendenza (accoppiamento), che ci sta stretto.

Decoupling

- Strong dependencies between components may introduce unneeded limitations
- Solution: let components indirectly communicate through some intermediary
  - Clean separation between computation and coordination

> "***All problems in computer science can be solved by another level of indirection***"
>
> (David Wheeler, Titan project)

(disaccoppiamento)

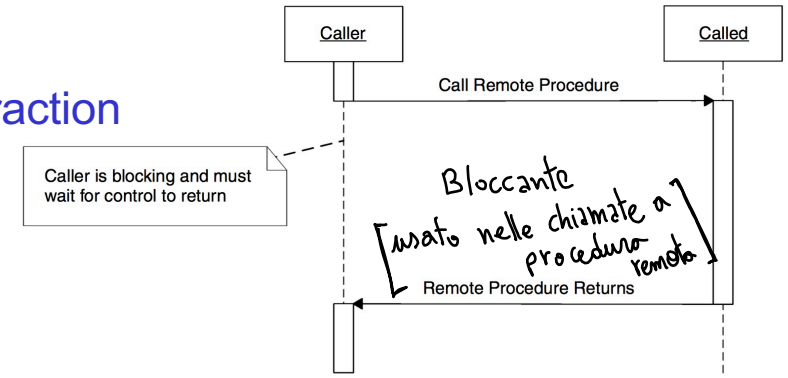
- ***Decoupling***: enabling factor to
  - Achieve greater flexibility
  - Define architectural styles that allow to better exploit distribution, scalability, and elasticity (e.g., microservices architecture and serverless computing)

Decoupling properties

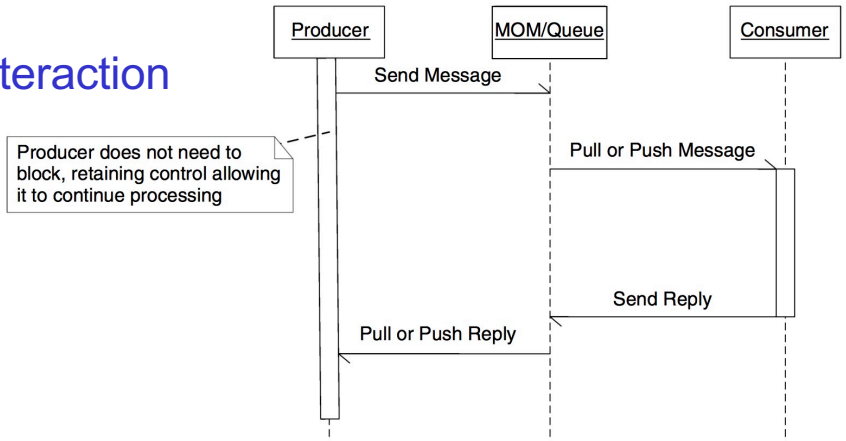
- ***Space*** (or referential) decoupling
  - Components do not need to know each other in order to communicate and cooperate; they are anonymous
- ***Time*** (or temporal) decoupling
  - Interacting components do not need to be present at the same time when communication occurs
- ***Synchronization*** decoupling
  - Interacting components do not need to wait each other and are not reciprocally blocked

Synchronous vs. asynchronous interaction

Synchronous interaction



Asynchronous interaction



Decoupling: pros and cons

- Thanks to decoupling, DS can be flexible while dealing with changes and can provide more dependable and elastic services
  - Space decoupling: components can be replaced, updated, replicated or migrated, se componente rotto, lo rimpiazzo! Cliente vuole solo risorsa, non importa DA CHI.
  - Time decoupling: allows to manage volatility (senders and receivers can come and go)
  - Synchronization decoupling: no blocking

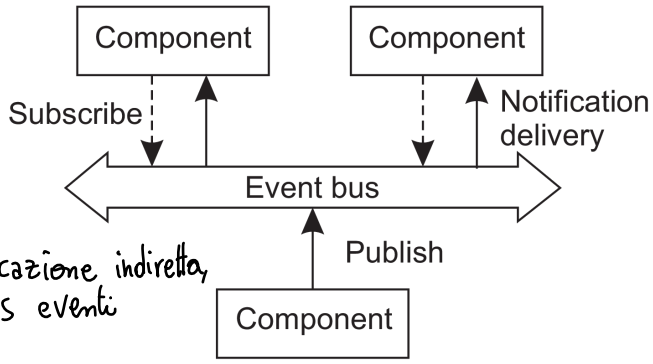
↗ flessibilità
↘ prestazioni
perché ∃ un intermediario

- Main disadvantage:
  - Performance overhead introduced by indirection

Architectural style evolution

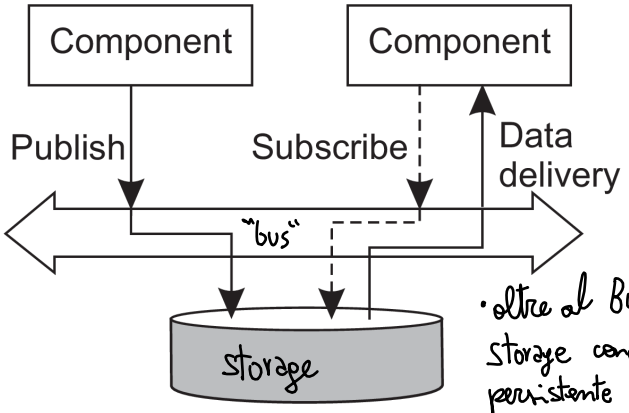
- Introducing decoupling, alternative architectural styles where components communicate indirectly

1. **Event-driven architecture**



• comunicazione indiretta, c'è BUS eventi

2. **Data-oriented architecture**

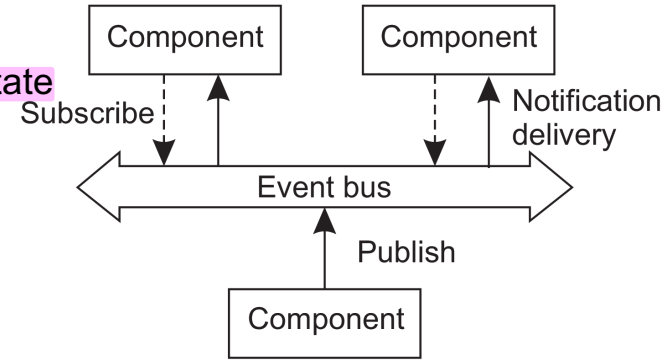


• oltre al BUS ho storage condiviso e persistente

3. **Publish-subscribe architecture**

Event-driven style

- Components communicate by means of event propagation
  - **Event**: a significant change in state (e.g., change in temperature, opening a door)
- Components
  - Subscribe to events they are interested in being notified
  - Receive notifications about events
- Communication
  - Based on message exchange
  - Asynchronous
  - Multicast ("uno" invia, "molti" ricevono)
  - Anonymous (componente che si iscrive ad un evento NON SA chi invia le notifiche)

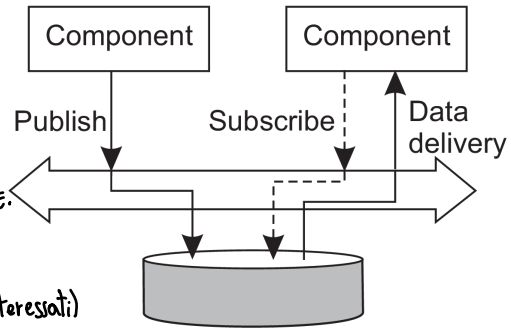


- Example: Java Swing

Which type of decoupling?

Data-oriented style

- Communication among components: through *shared data space* (usually passive, sometimes active) providing *persistent storage*
  - Data added to or removed from shared space, aka blackboard model
- API for shared space
  - `write`, `take`, `read` and their variants (`takeIfExists`, `readIfExists`)
  - In case of active space: also `notify` or `push`, so to avoid polling
  - Mutual exclusion to access data
- Examples:
  - Linda and tuple space
  - JavaSpaces, GigaSpaces XAP, TIBCO ActiveSpaces, Fly Object Space



Which type of decoupling?

How can we implement shared space?

Example: Linda tuple space

- Data is contained in ordered sequences of typed fields (tuples)
- Tuples are stored in a persistent, global shared space (tuple space)
- Three simple operations:
  - `in(t)`: atomically read and remove tuple matching template `t`
  - `rd(t)`: return copy of tuple matching template `t`
  - `out(t)`: write tuple `t` to tuple space
- Some details
  - Calling `out(t)` twice leads to storing two copies of tuple `t`: a tuple space is modeled as multiset
  - Both `in` and `rd` are blocking operations: the caller will be blocked until a matching tuple is found, or has become available

Example: Linda tuple space

Bob

```
1 blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]
2
3 blog._out(("bob","distsys","I am studying chap 2"))
4 blog._out(("bob","distsys","The linda example's pretty simple"))
5 blog._out(("bob","gtcn","Cool book!"))
```

Alice

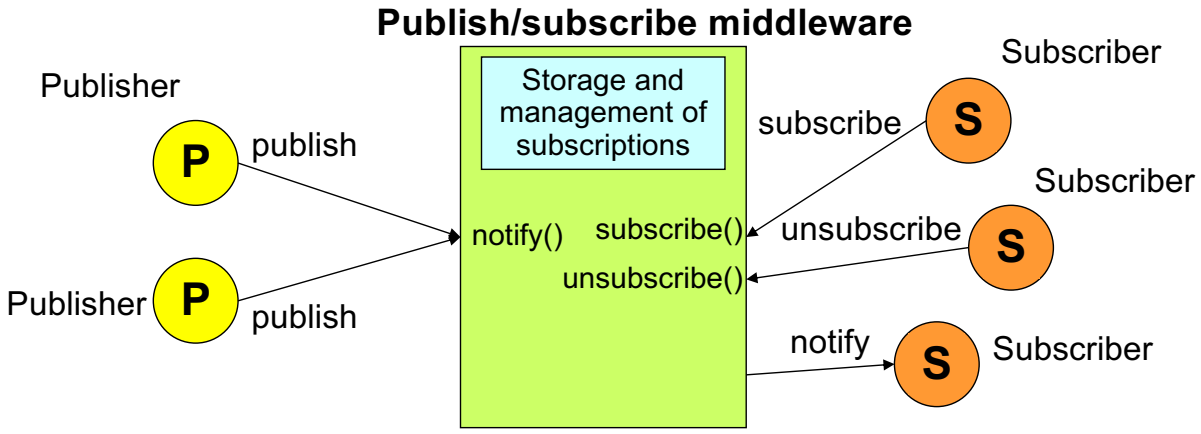
```
1 blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]
2
3 blog._out(("alice","gtcn","This graph theory stuff is not easy"))
4 blog._out(("alice","distsys","I like systems more than graphs"))
```

Chuck

```
1 blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]
2
3 t1 = blog._rd(("bob","distsys",str))
4 t2 = blog._rd(("alice","gtcn",str))
5 t3 = blog._rd(("bob","gtcn",str))
```

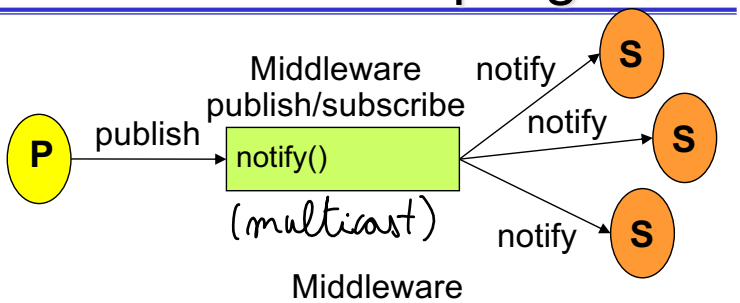

3° stile : Publish/subscribe style

- Producers generate events (publish) and are not interested in their delivery to subscribers (aka consumers)
- Consumers register as interested to events (subscribe) and are notified (notify) of their occurrence
- Full decoupling between interacting entities

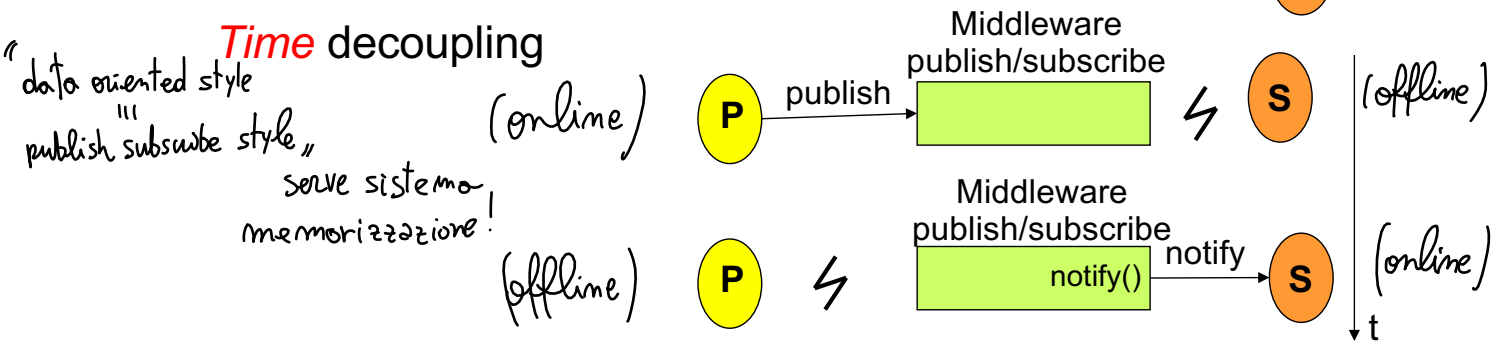


Publish/subscribe and decoupling

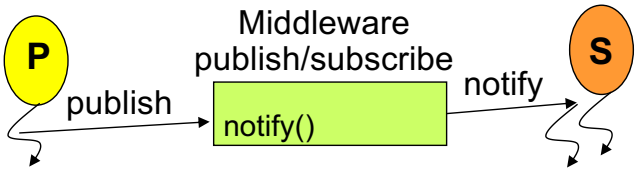
Space decoupling



Time decoupling



Synchronization decoupling



P.T. Eugster *et al.*, "The many faces of publish/subscribe" *ACM Comput. Surv.* 35(2):114-131, June 2003.

Publish/subscribe variants

- Most widely used subscription schemes in publish/subscribe systems
  - Topic-based (argomento)
    - Participants can publish events and subscribe to individual topics, which are identified by keywords (semplice)
    - Cons: limited expressiveness (uso solo parole chiave)
  - Content-based (contenuto, non argomento. Uso meta-dati)
    - Events are not classified according to some predefined external criterion, but according to their actual content, e.g., meta-data associated to events
    - Consumers subscribe to events by specifying filters
    - Cons: implementation challenges

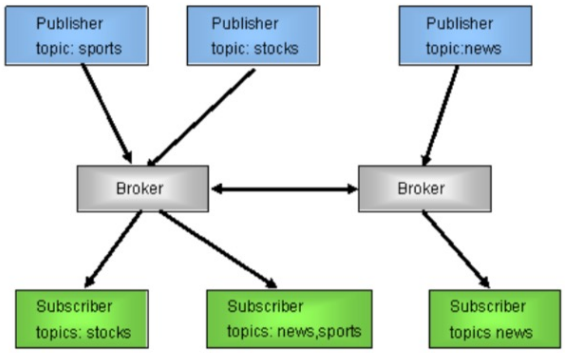
Publish/subscribe implementation issues

- Tasks of publish/subscribe system
  - Ensure that events are delivered efficiently to subscribers
  - In a secure, scalable, dependable, concurrent way

- Centralized vs. distributed implementations
  - ***Centralized***: single node acting as event broker that maintains a repository of subscriptions and matches event notifications against subscriptions
    - Pro: simple
    - Cons: lacks resiliency and scalability

Publish/subscribe implementation issues

- Centralized vs. distributed implementations
  - ***Distributed***: alternative architectures, such as cluster of cooperative brokers based on master/worker architecture as well as fully decentralized implementations (e.g., P2P based on DHT or gossiping)
    - Example: Apache Kafka (implementazione effettiva)
    - Distributed implementation of content-based scheme is more complex



17/10/22

Choosing an architectural style

- No single solution: you can tackle the same problem with many different designs and architectural styles

- Choice depends often on extra-functional requirements:
  - Costs (resource usage, development effort needed)
  - Scalability and elasticity (effects of scaling work complexity and available resources)
  - Performance (e.g., execution time, response time, latency)
  - Reliability
  - Fault tolerance
  - Maintainability (extending system with new components)
  - Usability (ease of configuration and usage)
  - Reusability

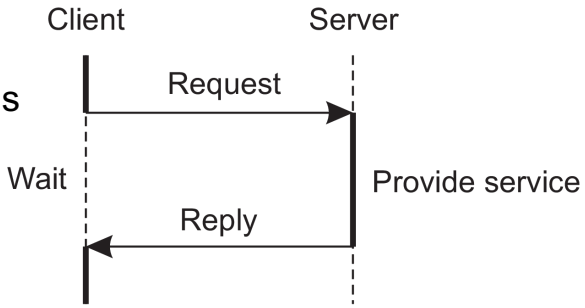
System architecture of DS

- Runtime instantiation of DS software architecture
  - Which components?
  - How do they interact with each other?
  - Where to place them? dove fare deployment?

- Types of system architectures
  - Centralized architectures
  - Decentralized architectures
  - Hybrid architectures (alcune più indirizzate verso il centralizzato, altre sul decentralizzato!)

Centralized system architectures

- Basic client-server model
  - Two groups: servers offer services and clients use services
  - Clients and servers can be on different machines
  - E.g., Web clients and servers



- Clients follow request/reply model with respect to using services
- Communication based on message exchange
  - Limited performance
  - *Idempotent operation*: operation can be repeated multiple times without harm (ritorna sempre stesso risultato! facilmente replicabile!)
- Communication is often *synchronous* and *blocking*
- Strong coupling: e.g., coexistence of interacting entities

client-server presenti, si conoscono, chiamate bloccanti.

Multi-tiered client-server architectures

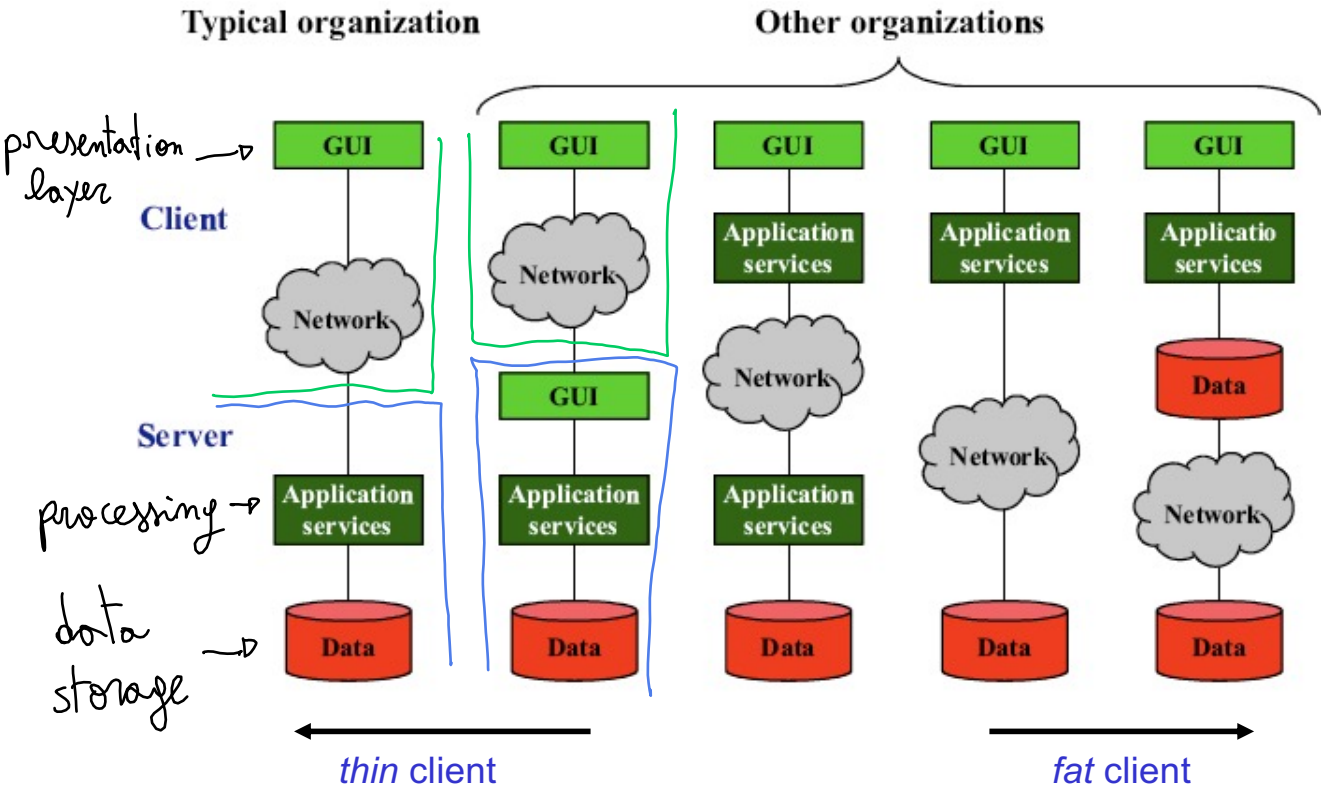
come istanziare i componenti in questa architettura?

- How to map logical levels (***layers***) into physical levels (***tiers***)?
- Two-tiered architectures
  - 2 physical levels
- Three-tiered architectures
  - 3 physical levels

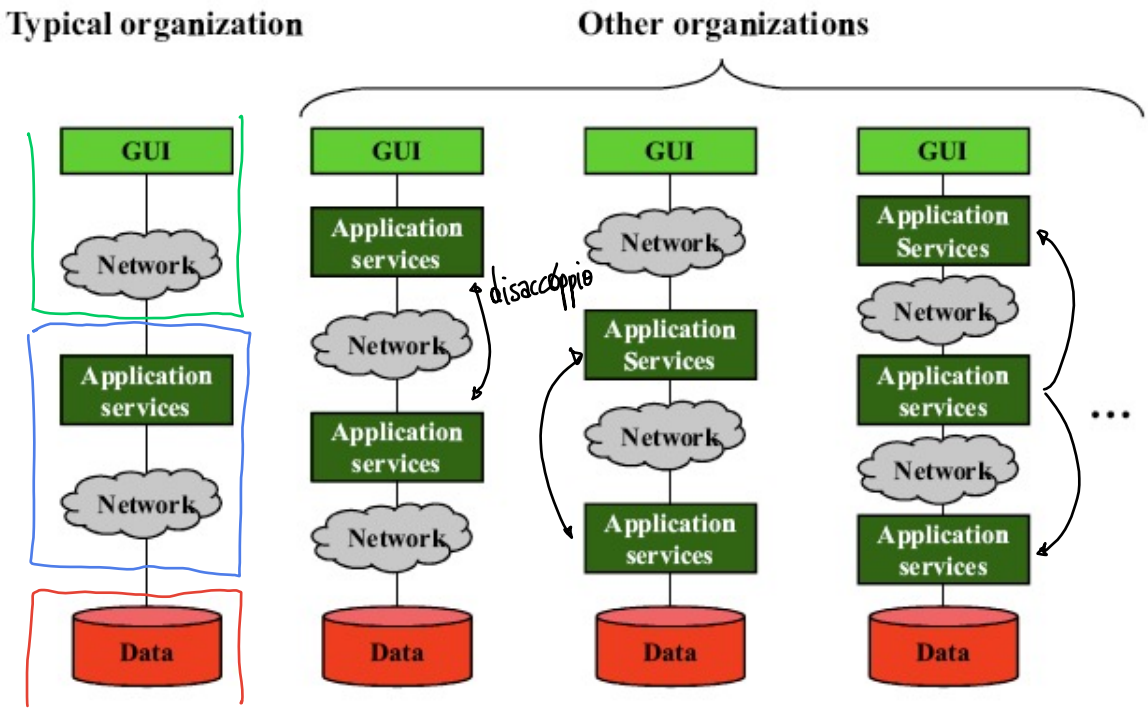
due alternative

- Different organizations
  - Depending on the distribution of:
    1. presentation layer
    2. processing (or business logic) layer
    3. and data storage layer

2-tiered client-server architectures



3-tiered client-server architectures

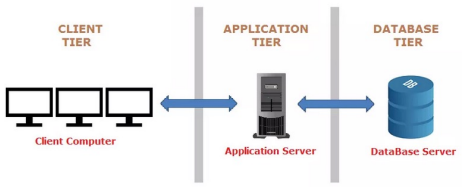


Multi-tiered client-server architectures

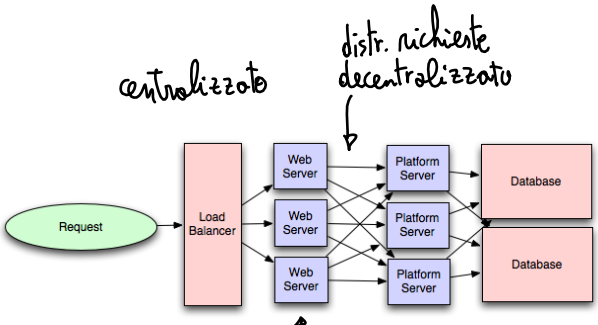
- From 2 to N tiers (N >=3)
- Adding a new physical level
  - Improve flexibility, functionality and distribution possibilities
  - But possible performance degradation when increasing the number of physical levels
    - Communication cost increases
    - Complexity increases in terms of management and optimization

From multi-tiered architectures to…

- Vertical distribution
  - Divide distributed applications into logical layers and run components in each layer on a different physical server

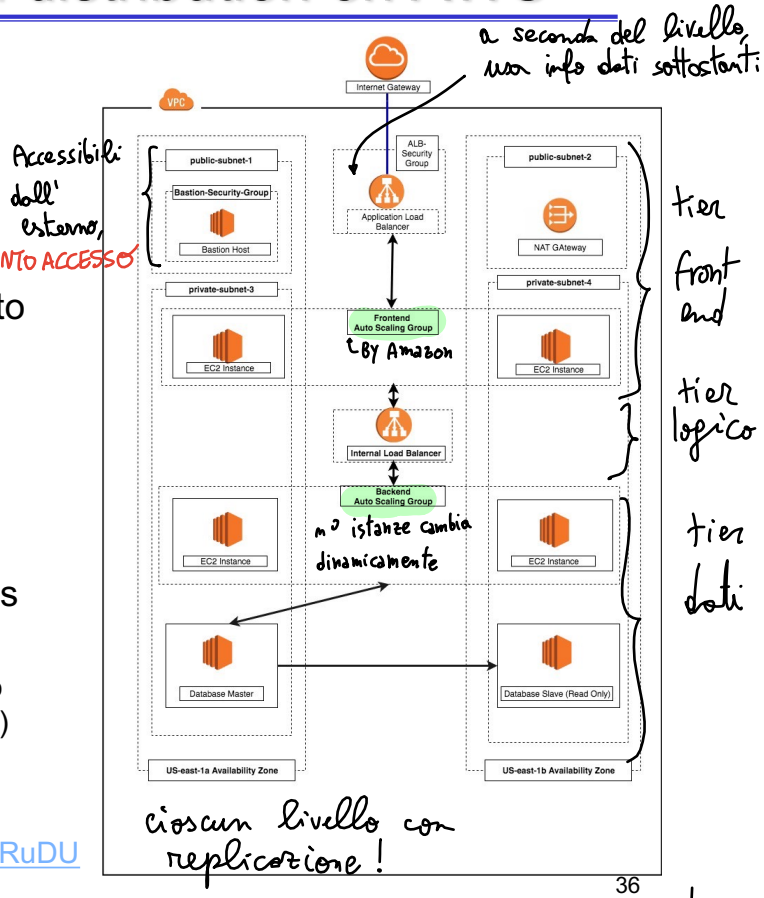


- Horizontal distribution
  - Distribute each single layer on different physical server
  - Load sharing among multiple servers, where load distributing is managed by a load balancer
  - E.g.,: distributed Web cluster



Example: horizontal distribution on AWS

- 3-tiered architecture with horizontal distribution on AWS
  - Elasticity: each tier can dynamically scale horizontally to support increasing or decreasing request load
  - High availability: replication in different availability zones
  - Security: application tiers communicate within themselves with a private IP
    - All the tiers in a private subnet (no public IP assigned to its instances) within the VPC



Source: https://bit.ly/2SSRuDU

Il 1° tier è pubblico, ha un NAT che converte IP in privato, per accedere ai tier sottostanti (vi accedo Indirittamente). Accedo dal "Load Balancer".

Decentralized system architectures

- **Peer-to-peer** (P2P) systems
- P2P: class of systems and applications that use distributed resources to perform a function (even critical) in a decentralized way, ✗ controllore centralizzato.
  - "P2P is a class of applications that takes advantage of resources available at the edges of the Internet" (Clay Shirky, 2000) → distribuite ai bordi della rete, by user, non by cloud.
- Peer: entity with similar capabilities to other entities in the system
- Shared resources: files, storage space, computing power, bandwidth
  - Give and receive resources from a community of peers

Features of P2P systems

- Peer: system nodes with equal roles, privileges and responsibilities, nodi "simmetrici"*
  - Autonomous nodes located at the network edges (distribuiti ai bordi della rete)
  - *Hybrid P2P* system: differently from *pure* P2P system, some nodes are super-peer (have different functionalities than the others)
- No centralized control
  - A peer behaves as client and server and shares resources and services (symmetric functionality: *servent* = server + client)
- Highly distributed systems
  - Hundreds of thousands of nodes
  - Highly dynamic and autonomous nodes
    - A node can enter or exit the P2P network at any time (join/leave operations) (churn elevato)
  - Redundancy of information

* Nei P2P Ibridi non sono simmetrici, ∃ superpeer!

file composti da chunk ("pezzettini") posseduti da gli utenti che hanno il File.

P2P applications

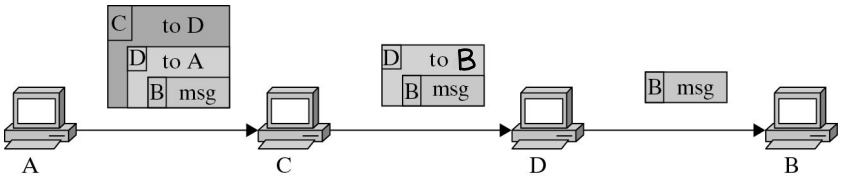
- Content distribution and storage
  - Content: file, video streaming, …
  - Networks, protocols and clients for *file sharing* (P2P "killer application"): Gnutella, FastTrack, eDonkey, BitTorrent, Kademlia, eMule, uTorrent, …
  - *P2P TV* (video streaming of TV channels in real time): PPLive, …
  - *File storage*: Freenet, OceanStore, …
- Computing resource sharing (distributed computation)
  - SETI@home (search for extraterrestrial intelligence), Folding@home (protein folding)
- Collaboration and communication
  - e.g., Chat/IRC, Instant Messaging, XMPP, Skype, …
- Content Delivery Networks, distributed file systems
  - e.g., CoralCDN, Internet Planetary File System
- Blockchains

Challenges in P2P computing

- Heterogeneity in peer resources
  - Hardware, software and network heterogeneity
- Scalability
  - System scaling related to performance and bandwidth
- Location
  - Data location, data locality, network proximity, and interoperability
- Fault tolerance
  - Failure management
- Performance
  - Routing efficiency, load balancing, self-organization

Challenges in P2P computing

- Free-riding avoidance
  - *Free-riders*: peers that are selfish and unwilling to contribute anything
- Anonymity and privacy
  - *Onion routing* for anonymous communications



- Trust and reputation management (non si conoscono)
  - Lack of trust among peers who are strangers to each other
- Network threats and defense against attacks
- *Churn* resilience (nodi vanno, vengono, crashano ... con loro le risorse!)
  - Peers come, leave and even fail at random
  - Resources are dynamically added or removed

Principali operazioni

Main tasks of a P2P system node

- Let's consider file sharing as app
- A P2P system node performs the following basic operations:

1) • Bootstrap: entry phase into the P2P network (entro nella rete)
   - Possible solutions: static configuration, pre-existing caches, well-known nodes, deve trovare peer e rete!

2) • Resource lookup: how to locate resources in a P2P network? chi ha risorsa che cerco?

3) • Retrieval of the localized resource, recupero la risorsa!

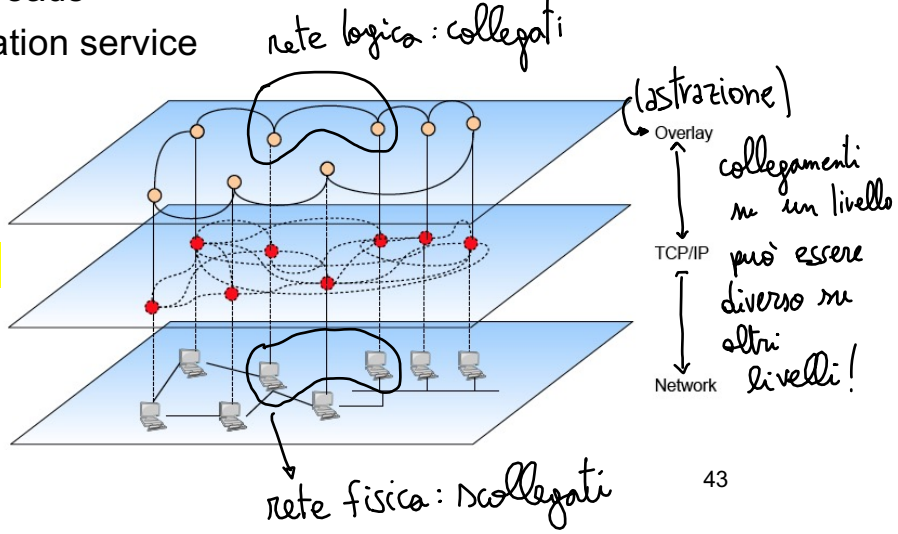
- We focus on resource lookup

P2P overlay

- P2P networks are commonly called overlays
- **Overlay network**: logic network connecting peers laid over the IP infrastructure
  - Based on an underlying physical network
  - Set of logic links between the peers, not corresponding to physical connections
  - Nodes run processes/threads
  - Provides a resource location service
  - Application-level routing

Application-level abstraction

non c'è corrispondenza tra i layer!



Overlay routing

- Basic idea:
  - The P2P system finds the path to reach a resource
- Compared to traditional routing
  - Resource: no network node address, but files, available CPU, free disk space, ...
- We focus on routing, not on the interaction between peers to retrieve a resource
  - Retrieval typically occurs with a direct interaction between peers, eg. using HTTP / HTTPS
- Routing su risorsa/metadati, non su IP.

Tasks of overlay network

- Besides routing of requests to resources, an overlay network also allows to:
  - Insert and delete resources
  - Add and remove nodes
  - Identify resources and nodes

↑ importante la gestione

- How to identify resources?
  - Globally Unique IDentifier (GUID): it is usually derived as a secure hash function from some (or all) of the resource's state

→ univoco nella rete P2P

↳ MD5, SHA-1 su nome o data o ultima modifica ... stringa 128 bit o 256 bit

- We also need to identity peers through a unique identifier
  - Again, generally computed through a hash function (la stessa di prima)

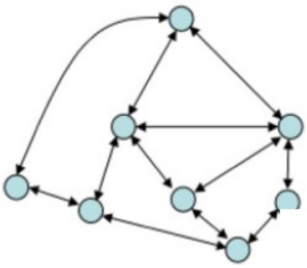
Hash (IP/ora accesso/MAC ...)

P2P overlay classification

- How to manage resources and nodes?
  - Depends on the overlay network type

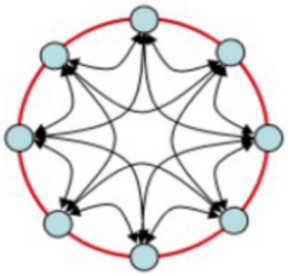
Unstructured overlay networks

2 famiglie



Structured overlay networks

struttura precisa (es anello, ipercubo) e ben definita.



Unstructured overlay networks

- Overlay network built on **random graphs**
  - No particular structure on the overlay network *by design*
  - Peers are arbitrarily connected: each peer joins the network following some simple and local rule
  - A joining node contacts a set of neighbors, somehow selected
  - No control over network topology or placement of resources on nodes
- Goal: to manage nodes with highly dynamic behavior (i.e., high rates of churn)
- Examples: Gnutella, Bitcoin

✓ non c'è struttura da mantenere, inserimenti/cancellazioni di Nodi easy.

✓ resiliente (non mantengo struttura)

✗ # lookup: senza struttura, localizzazione risorse complessa.

Unstructured overlay networks

- Pros:
  - Easy maintenance because insertion and deletion of nodes and resources are easily managed
  - Highly resilient
- Cons
  - High lookup cost: resource location is complicated by the lack of structure

O(N) vs O(log n)

$ lookup senza struttura

$ lookup con struttura (che implica vincoli su nodi IN/OUT)

Break: random graphs models

- 3 *main models* of random graphs
  1. Erdos-Renyi
  2. Small-world networks (dist. max 6 HOPS)
  3. Scale-free networks (le più complesse)

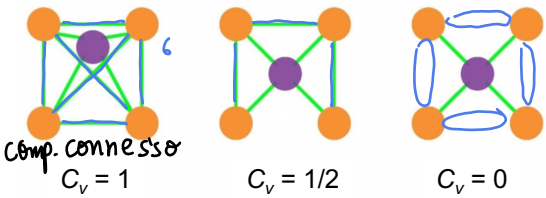
modelli per creazione di grafi random, alcuni validi anche per le reti sociali.

Random graphs models: properties dei grafi

- Main properties of random graphs we consider
  - *Clustering coefficient* (C_v) *of vertex v:* number of edges (L_v) between the m_v neighbors of v divided by maximum number of possible edges between them

senza pensare per "v" ! conto UNA VOLTA

$$C_v = \frac{2L_v}{m_v(m_v - 1)}$$



- se nodo connesso (totalmente) ho $\frac{n(n-1)}{2}$, però io escludo "v" cioè n=4 (vicini) e $\frac{n(n-1)}{2} = \frac{4 \cdot 3}{2} = 6$
  - $0 \le C_v \le 1$
  - $C_v = 0$: no links between v's neighbors
  - $C_v = 1$: each of v's neighbors links to each other
  - ***Clustering coefficient of a graph***: average clustering coefficient over all vertices
  - ***Average shortest path length***: shortest path between two vertices averaged over all pairs of vertices
  - ***Graph diameter***: the longest shortest path

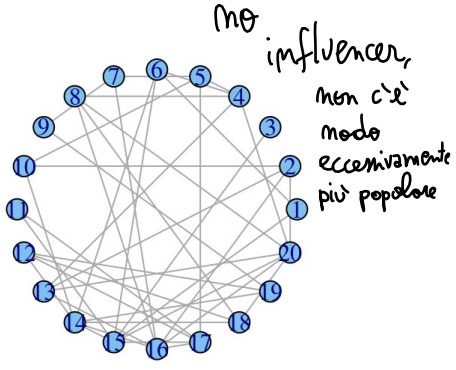
Random graph models: Erdos-Renyi

- **Erdos-Renyi** random graph: classical random network
  - N: number of vertices (fixed), a priori
  - p: probability that two vertices have an edge
  - Vertex degree follows binomial distribution

    p_k: probability that vertex v has degree k

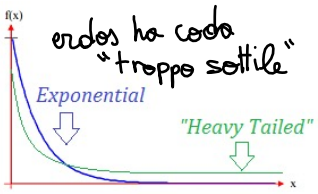
    N, k in input

$$p_k = \binom{N-1}{k} p^k (1-p)^{N-1-k}$$



N=20, p=0.2

  - Average clustering coefficient: p
  - Average shortest path: log(N) / log((N-1)p)
  - Graph diameter: ~ log(N)
- Wrapping up:
  - Small average shortest path length
  - Low clustering coefficient
    - Not observed in many real-world networks!
  - Homogeneous network with an exponential tail



Random graph models: Watts-Strogatz

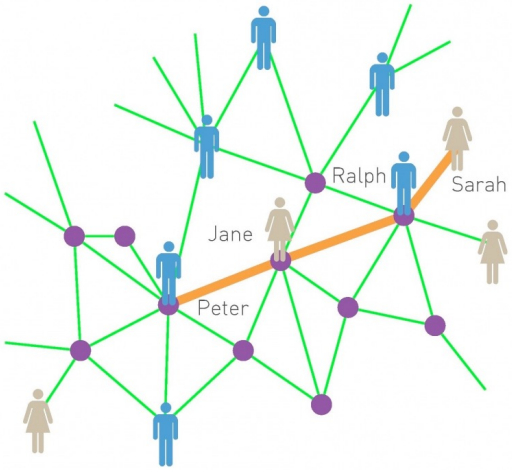
- Properties we can observe in real-world networks
  - High clustering coefficient
  - Small-world property: average distance between two nodes depends logarithmically on *N*
  - Presence of hubs (high-degree nodes), i.e., non-homogeneous network, gli "influencer"
- Erdos-Renyi model generates a small-world network but does not satisfy the other two properties

- **Watts-Strogatz** random graph: small-world network with high clustering coefficient
- What is the small-world property?

19/10/22

Small world

- The *small-world phenomenon*, also known as *6 degrees of separation*, states that if you choose any two individuals anywhere on Earth, you will find a path of at most 6 acquaintances between them
  - The distance between any 2 nodes in a network is unexpectedly small

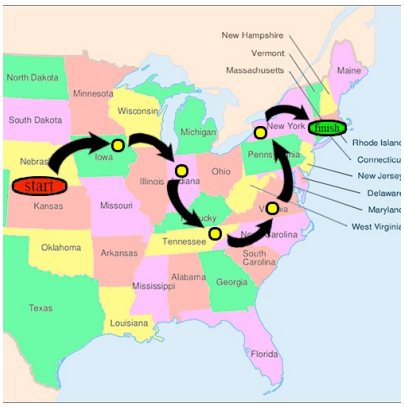


Small world: experimental confirmation

- The first small-world study comprised several experiments conducted in 1967 by Stanley Milgram to measure the average path length for social networks of people in the United States
  - Letters sent from randomly selected residents of Wichita (Kansas) and Omaha (Nebraska) to two target persons in Boston
  - Senders were asked to forward the letter to a friend, relative or acquaintance who is most likely to know the target person
- The research was groundbreaking in that it suggested that human society is a small-world network characterized by short path lengths
- The experiments are often associated with the phrase “six degrees of separation”, although Milgram did not use this term himself

Small world: experimental confirmation

One possible path of a message in the “Small World” experiment by Stanley Milgram



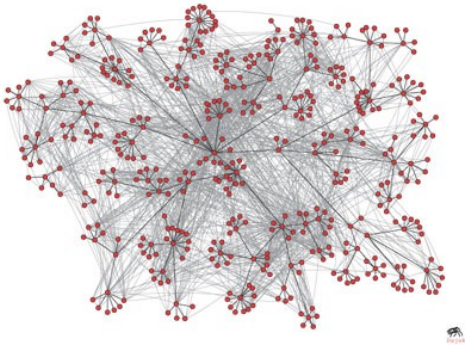
Random graph models: Watts-Strogatz

- **Watts-Strogatz** random graph: *small-world* network
- Properties:
  - Small average shortest path length
  - High clustering coefficient, independent of the number of vertices
- Most nodes are not neighbors of one another, but most nodes can be reached from every other by a small number of hops
  - Example: social network
- Limitation: does not account for the formation of hubs

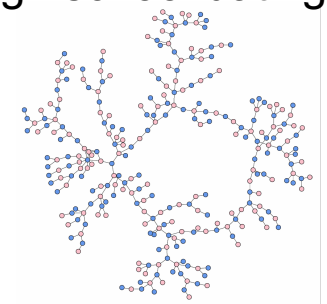
(modi "alto livello")

Examples of social networks

Corporate e-mail communication



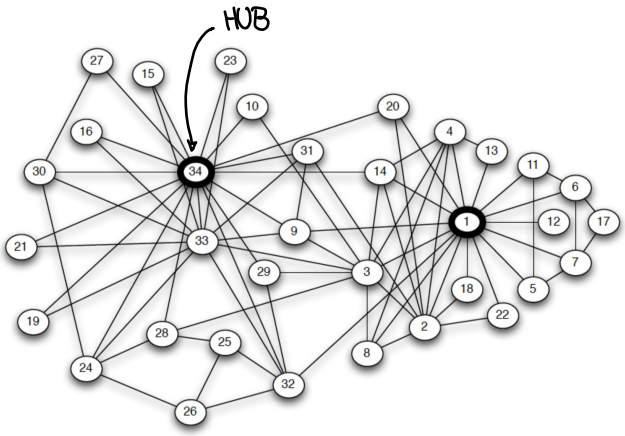
High school dating



Trails of Flickr users in Manhattan

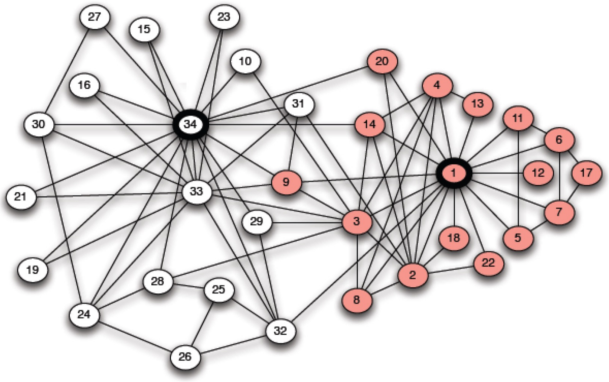
Examples of social networks

Friendships within a 34-person karate club



Clues to fault lines that eventually split the club apart

("1" e "34" litigano)



Random graph models: Albert-Barabasi

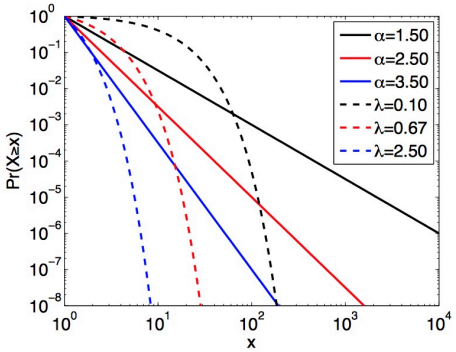
- **Albert-Barabasi** random graph: *scale-free* network
  - Vertex degree follows power law distribution (Legge distribuzione di potenza)
    grado nodo $K = p_k \sim ck^{-\alpha}$ where $2 < \alpha < 3$ (Legge Pareto si basa su lei)
    80% effetto dip. dal 20% cause.
- **Power law**: event frequency varies as power of some attribute of that event
  - Example: number of cities having a certain population size, allocation of wealth among individuals
  - Some special types of power law
    - Pareto law: "80% of effects come from 20% of causes"
    - Zipf's law: word frequency in a text
  - Property: *scale invariance*, see https://bit.ly/3iuGN8T
    - f(x) is scale-invariant, if on multiplying the argument x by some constant scaling factor (λ), one obtains $f(\lambda x)=\lambda^{-\beta} f(x)$, i.e., same shape is retained but with different scale

      (es: Fiocco di neve: la sua struttura è composta da tanti fiocchi di neve)

Random graph models: Albert-Barabasi

implica mediana > media

- **Power law** *(continued)*
  - Power law probability distributions are also *heavy-tailed* (coda lunga)
    - Distribution tail contains a great deal of probability (i.e., heavier tail than exponential distribution): events that are effectively "impossible" (negligible probability under an exponential distribution) become practically commonplace under a power-law distribution, see http://bit.ly/2hXOc48



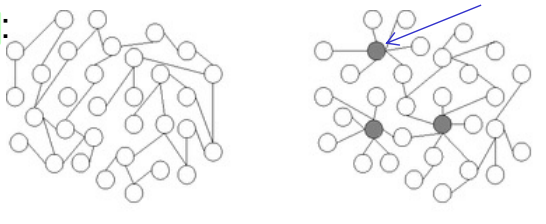
se α ↗ ≈ esponenziale
se α ↘ peso coda ↗

Power-law (α) and exponential distributions (λ)

Random graph models: Albert-Barabasi

- How do new nodes attach themselves to existing nodes? On the basis of preferential attachment
  - The more connected a node is, the more likely it is to receive new links, e.g. think about social networks connecting people!
  - There are a few hubs, but their number decreases exponentially (power law)
- Scale-free: graph diameter ~ ln (ln N)
  - Many networks are conjectured to be scale-free (Internet, Web, social networks, power grids), but still controversial hypothesis in some cases
  - Scale-free networks are highly resilient against faulty constituents: ideal interconnect also for cloud computing

(nodi hub più "delicati")



(a) Random network (b) Scale-free network

Random graph models: references

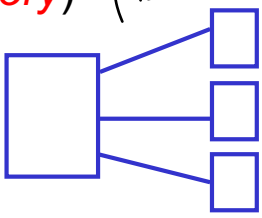
- A.-L. Barabási, "Network Science", chapters 1, 2 and 3, Cambridge University Press, 2016. http://networksciencebook.com
- D. Easley and J. Kleinberg, "Networks, Crowds, and Markets: Reasoning About a Highly Connected World", chapters 1 and 2, Cambridge University Press, 2010. http://www.cs.cornell.edu/home/kleinber/networks-book/

Take-away: topologies of many real-life P2P overlays are power-law random graphs

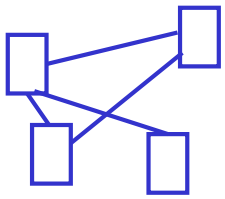
Let's go back to unstructured overlay networks

Unstructured overlay networks: routing

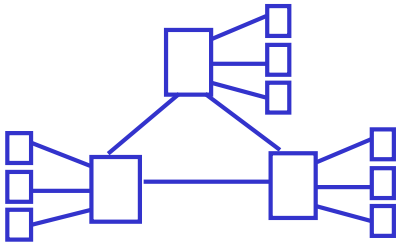
- Let's classify unstructured overlays according to the distribution of peer-resource index (*directory*) (dove metto gli indici dei peer?)
- Centralized unstructured overlay: central directory (e.g., Napster)



- Fully decentralized unstructured overlay: distributed directory (e.g., Gnutella)



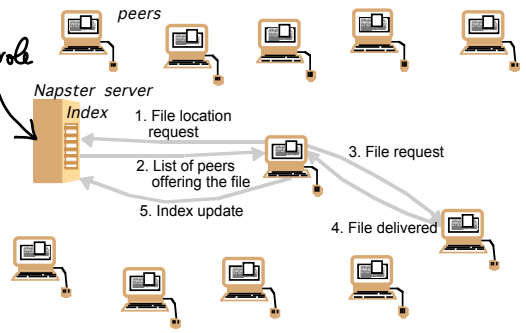
- Hybrid unstructured overlay: *semi-centralized directory*
  - Routing *limited to super-peers*



Centralized unstructured overlay

- A directory server is responsible for resource-peer index: lookup(resource name) → {list of peers}
- Pros:
  - Very simple
  - Search is handled by a single directory server
  - The directory server is a single point of control: provides definitive answer to query

se trovi server centrale lo butti giù



- Cons
  - Expensive management of central directory
  - Single directory server: performance bottleneck (limited scalability) and single point of failure (technical and legal reasons, e.g. Napster)

Fully decentralized unstructured overlays

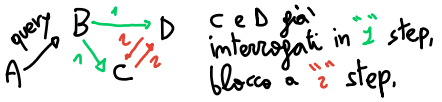
- Adopt a fully decentralized approach to find resources
- How to solve the lookup problem in these overlays?
  - Query flooding
  - Random walk
  - Gossiping (in upcoming lesson)

Se non c'è server centrale, come faccio a trovare nodi peer che hanno risorse di mio interesse? ("lookup problem")

Query flooding

- Resource lookup managed according to query **flooding**
  - Originator sends the lookup query to its neighboring peers
  - Each peer either responds if it contains the resource or forwards the query to its neighbors (excluding the neighbor from whom it received the query), possibile inoltro ∞
- Optimization #1: avoid indefinite query forwarding
  - Use TTL to limit the search range → in "hop", no in secondi
  - At each forwarding, decrease TTL by 1; when TTL=0, the query is no longer forwarded

può scadere prima di trovare una risorsa esistente



- Optimization #2: avoid cyclic paths
  - Assign a unique query ID so to not process the query again
- Lookup cost: O(N), being N the number of nodes in the network

Query flooding

- Alternatives for sending back the response to the originator:
  - Direct: from the peer having the resource to the query originator (ID nodo richiedente insieme alla query)
  - Backward routing

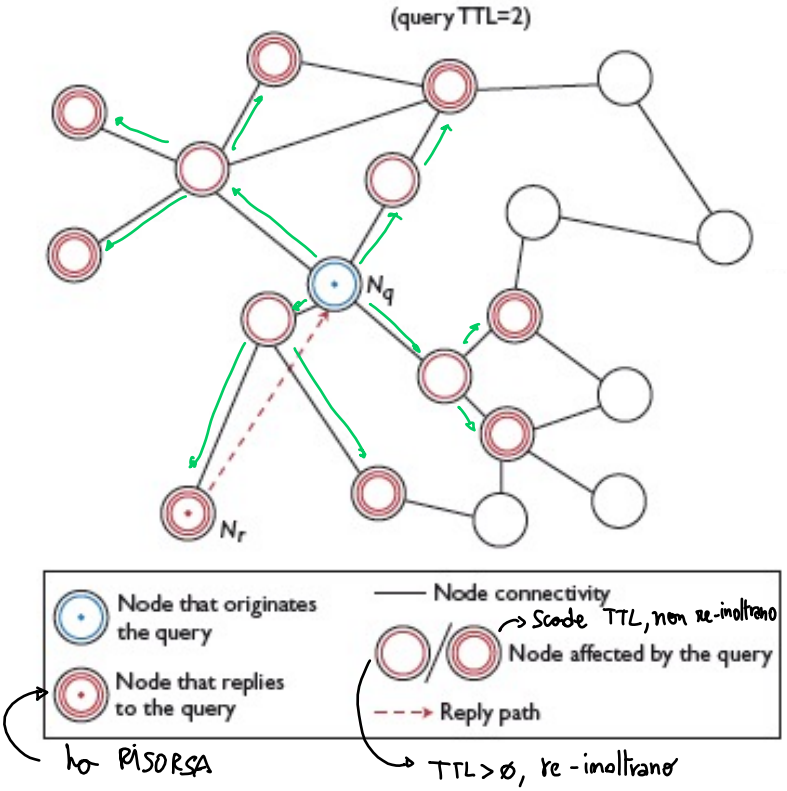
✗ solo chi ha fa la query scopre chi ha la risorsa, gli altri no, non condivide le info!

- Backward routing
  - Response is forwarded back along the same path followed by the query until it reaches its originator
  - Query ID can be used to locate the backward path
  - Which are the advantages over sending directly the response?

✗ info dinamiche, possono cambiare
✗ tali info possono saturare la cache!

✗ devo "ricordare" il percorso → associo query a ID mittente
→ metto query ma size ?

Query flooding: example



Query flooding: cons

- Communication overhead
  - Exponential explosion in the number of messages
  - Unsuccessful messages use network bandwidth
- High lookup cost
  - How to determine the TTL value?
- Denial-of-service attacks are possible
  - Black-hole nodes in case of congestion
- False negatives
  - No guarantee that (all) nodes that own the resource will be queried
- Lack of relationship between overlay and physical network topology
  - How far apart are “neighbor” peers?

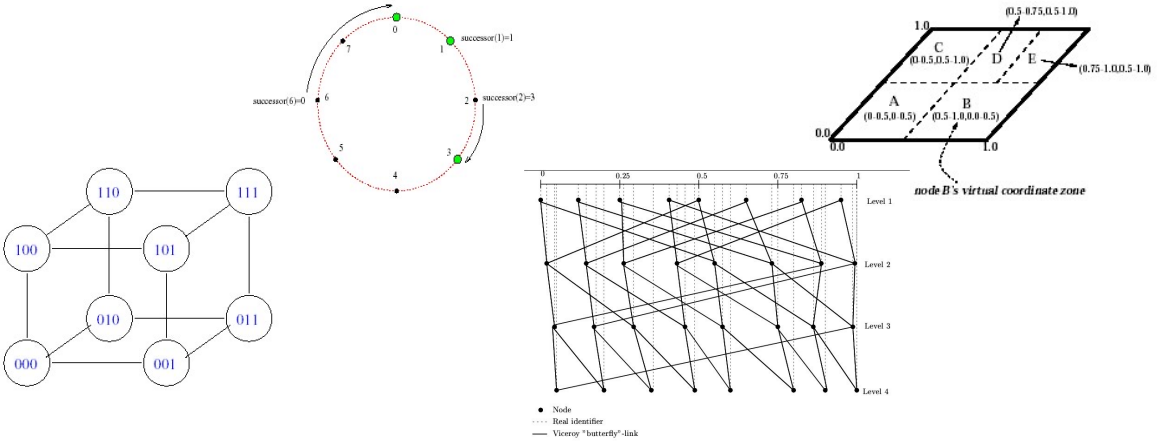
Random walk

- In standard random walk, the originator forwards the lookup query to one randomly chosen neighbor
  - This neighbor randomly chooses one of its neighbors and forwards the request to that neighbor
  - This procedure continues until the resource is found
- Message traffic is cut down with respect to flooding, but lookup time increases, non è detto trovi risorsa!
- To decrease lookup time, the querying peer can start *k* random walks simultaneously
  - With *k* random walks the originator forwards *k* copies of the query to *k* randomly selected neighbors
  - Then, each request takes its own random walk

topologia ben definita, ∃ regole

Structured overlay networks

- The forwarding of the lookup query is based on a well-defined set of information about other peer in the system
- The resulting overlay network is structured
  - Constraints on how resources and peers are positioned on the network
  - Overlay network topology: ring, tree, hypercube, grid, ...

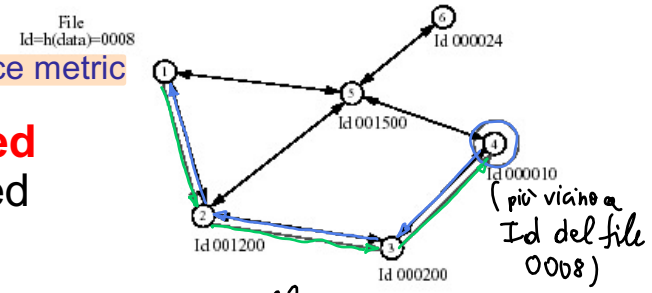


Structured overlay networks

- Goals: improve scalability by lowering lookup cost and reduce communication overhead with respect to unstructured overlays
  - Efficient key-based resource lookup
    - Overlay structure guarantees that lookup has a given complexity (e.g., O(log N)) vs O(N) di quelli senza struttura!
  - Complexity guarantees also for peer join and leave
    (devo collocare i nodi, nei senza struttura è O(1), qui no!)
- Cons: peer join and leave become more expensive operations
  - Topology structure must be maintained

Routing in structured overlays

- Basic ideas
  - Each peer is responsible for some resources and knows some peers according to the overlay structure
  - Each resource is assigned a GUID
  - Each peer is assigned a GUID
  - GUIDs are computed using a hash function
  (sono identificate tramite HASH, di grandi dimensioni)
  - Same large identifier space used for peers and resources GUIDs
  - Lookup query is routed to the peer whose GUID is the "closest" to the resource GUID
    - Closest: according to some distance metric
- Routing is based on **Distributed Hash Table (DHT)**: a distributed key-value data store



in Chord, mappo risorse e peer sullo stesso spazio di indirizzamento!

Distributed Hash Table

- Distributed abstraction of conventional hash table (HT) that maps keys to values
- Recall conventional HT
  - Table of *(key, value)* tuples of size *M* (bucket)
  - Key lookup: hash function maps *keys* to range *0 ... M-1*, mappo in bucket
  - Searching is very efficient: O(1)
  - Need to handle collisions because multiple keys may hash to the same value

V O(1)
X collisioni

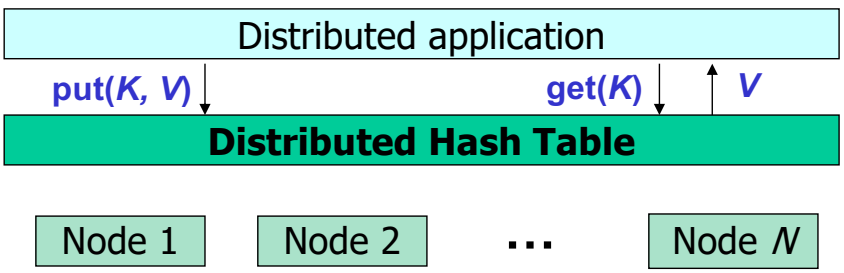
- DHT
  - Lookup similar to conventional HT: map the resource key to find the bucket (or slot) containing that resource
  - But DHT buckets are spread across multiple nodes (peers): how to map the resource key to find the peer responsible of the bucket?

sono distribuite su diversi peer, cioè i bucket (peer = bucket)

Distributed Hash Table: API

(come HT, ma Bucket distribuiti)

- Key-value pairs (key *K*, value *V*) stored in DHT
  - *K* is the key that identifies the resource (contained in *V*) and corresponds to the resource GUID
- API for accessing DHT (common to many DHT-based systems)
  - *V* = **get**(*K*): retrieve *V* associated with *K* from the node that stores it
  - **put**(*K*, *V*): store the resource *V* in the node responsible for the resource identified by *K*
  - **remove**(*K*): delete the reference to *K* and the associated *V*



Hard perché: decentralizzate, scalabili, efficienti, dinamiche (tempi ridotti) (nodi IN/out/crash)

21/10/22

Designing a DHT

- Resources and nodes are mapped onto the same identifier space using a hash function
  - GUID composed of *m* bits (usually *m* = 128 or 160)
  - E.g., SHA-1 cryptographic hash function
  - Hash function applied on metadata and/or data of resources (name, creation date, content, ...) and nodes
- Each node manages a portion of the resources stored in the DHT
  - Each node is assigned a contiguous portion of the keys and stores information about the resources mapped to its own portion of keys
- Routing in DHT: given *K*, map it into the GUID of the node "closest" to *K*
- Data replication can be exploited to improve availability by mapping each key to at least one node

ID nodo e risorse mappate su stesso spazio di indirizzamento

(quindi porzione contigua delle chiavi)

Issues related to DHTs

- Avoid hotspots by evenly distributing key responsibility among peers
- Avoid remapping all keys if the DHT size changes (i.e., when peers join or leave)
  - **Consistent hashing** to address these issues
- Only directly support *exact-match* search
  - Since each resource is identified only by its key, to search for a resource we need to know the key
  - Easy to make exact-match search queries, e.g. based on resource name
  - Difficult and expensive to support more complex queries
    - E.g., wildcard query, range query
  - We will consider only exact-match

(hotspot gestisce molte chiavi, o le più richieste, voglio evitarlo!)

→ devo riassegnare le chiavi in modo limitato, cioè impattando il meno possibile.

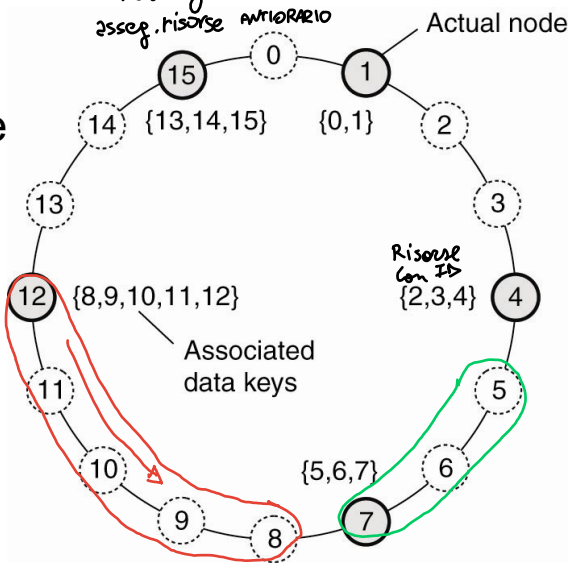
P2P systems based on DHTs

- Characterized by high scalability with respect to system size (i.e., *N*)
- Several proposals for DHT-based P2P systems
  - How do they differ?
    1. Definition of identifier space (and therefore network topology)
    2. Selection of peers to communicate with (i.e., distance metric)
  - More than 20 protocols and implementations for structured P2P networks, including:
    - **Chord** (MIT)
    - **Pastry** (Rice Univ., Microsoft)
    - Tapestry (Berkeley Univ.)
    - CAN (Berkeley Univ.)
    - Kademlia (NY Univ.)

Chord

https://github.com/sit/dht/wiki

- Elegant resource lookup algorithm for P2P networks
- Nodes (peers) and resources are mapped onto a **ring** using **consistent hashing**
- Each node is responsible for the keys placed between itself and the preceding node in counter-clockwise direction
  - Resource with key *k* is managed by the ring node whose identifier is the smallest id ≥ *k*
  - This node is called *succ*(*k*), ***successor*** of key *k*
    - E.g., *succ*(1)=1, *succ*(10)=12
- Distance metric: based on linear difference between identifiers (senso orario)



Stoica et al., Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications, IEEE/ACM TON, 2003

Consistent hashing

Nato per caching distribuito

- A special hashing technique
  - Both items (resources) and buckets (nodes) are uniformly mapped on the same identifier space (the ring) using a standard hash function (e.g., SHA-1, MD5)
  - Each node manages an interval of consecutive hash keys, not a set of sparse keys
- Original devised by Karger et al. at MIT for distributed caching (riadatta tecniche esistenti, non ne inventa nuove)

  "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web", STOC 1997.
- Some details and Java implementation

  www.tom-e-white.com/2007/11/consistent-hashing.html
- Largely used in real systems, e.g., Amazon Dynamo and memcached

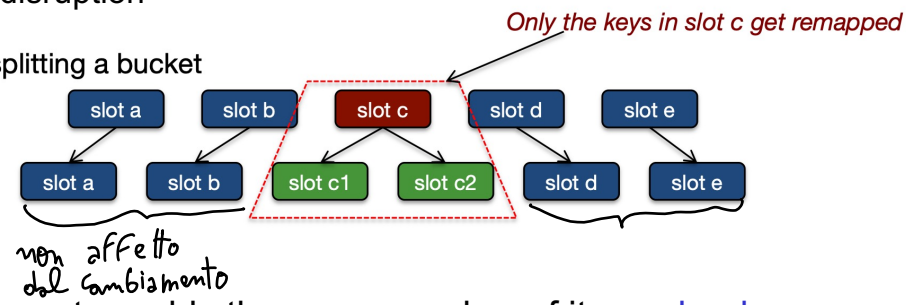
  → sist. dist. storage in memory (più veloce di ssd)

Chord: consistent hashing

(fondamentale)

- Consistent hashing is integral to Chord robustness and performance
  1. In case of hash table resizing (adding or removing a bucket): most keys will hash to the same bucket as before (voglio minimo impatto)
     - Practical impact: peers can join and leave the network with minimal disruption

Example: splitting a bucket



  2. All buckets get roughly the same number of items: load balancing

Chord: towards routing

- The simplest approach: lookup can be performed by traversing the ring, going one node at a time
- Can we do better than O(*N*) lookup? (nodo chiede a nodo successivo se ha la risorsa x)

- Simple approach for great performance
  - Have all nodes know about each other
    - When a peer gets a query, it searches its table of nodes for the node that owns those values
    - Gives us O(1) performance
  - Join/leave node operations must inform everyone
  - Maybe not a good solution if we have lots of peers (large tables) troppe info, con leave/join deve aggiornare tutte le tabelle

- Chord uses a compromise to avoid large tables at each node: **finger table** info su nodi adiacenti in senso "orario".
  - A partial list of nodes, progressively more distant

es: $FT_{[1]} = 4$, $FT_{[2]} = 15$, $FT_{[3]} = 2$ dell'esempio di Chord

Chord: finger table

- Finger table (FT)
  - Routing table of a node used to lookup a key
  - FT has *m* rows, with *m* = # GUID bits (perché siamo su un anello)
  - If FT_p is the FT of node *p*, then $FT_p[i] = succ(p + 2^{i-1}) \bmod 2^m$, $1 \le i \le m$ (non da 0) [riga i-esima
    - $succ(p+1)$, $succ(p+2)$, $succ(p+4)$, $succ(p+8)$, $succ(p+16)$, …
- Basic idea
  - Each node knows "well" closer nodes and has a rough knowledge of farther nodes

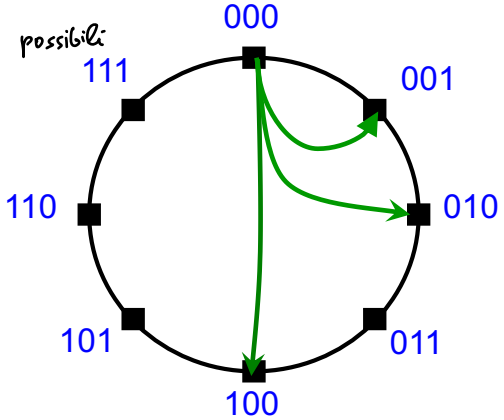
(2^3 ID possibili)

In the example m=3

$FT_0[1]=0+2^0=1$

$FT_0[2]=0+2^1=2$

$FT_0[3]=0+2^2=4$



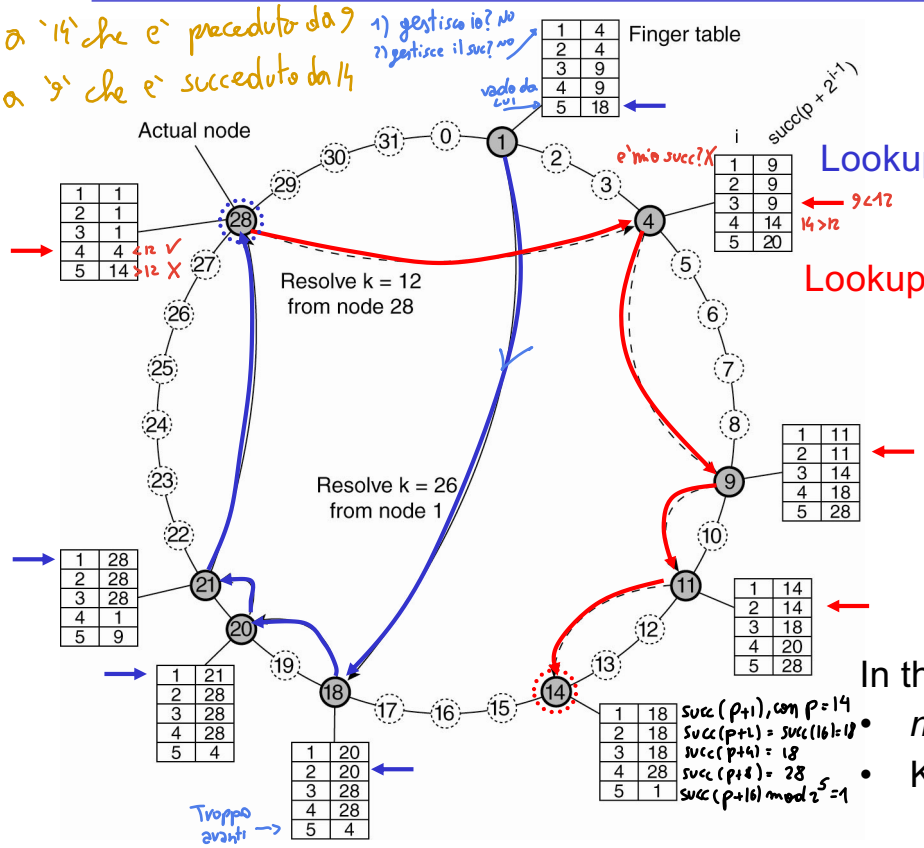
Chord: routing algorithm

- How to map key *k* into *succ(k)* starting from node *p* (routing algorithm):
  - If *k* belongs to the ring portion managed by *p*, lookup ends
  - If $p < k \leq FT_p[1]$, p forwards the request to its successor
  - Otherwise, *p* forwards the request to node *q* with index *j* in FT_p by considering the clockwise ordering

    $FT_p[j] \leq k < FT_p[j+1]$

    q is the farthest node from *p* whose key is less than or equal to *k*
- Features
  - It quickly reaches the vicinity of the searched point, and then proceeds with gradually smaller jumps
  - Lookup cost: O(log *N*), being *N* the number of nodes … not as cool as O(1) but way better than O(*N*)

Chord: routing algorithm example



Lookup for key 26 from node 1

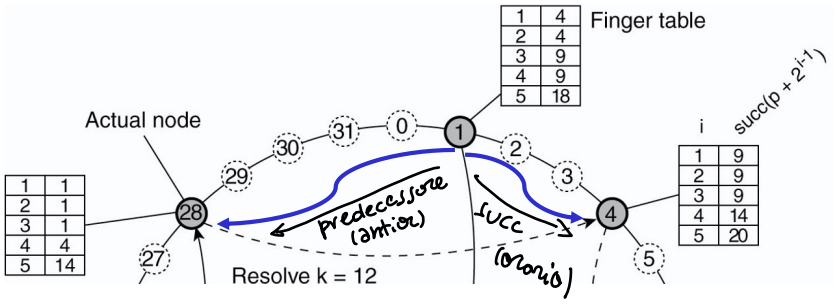
Lookup for key 12 from node 28

In the example:

- *m*=5 righe ∀ nodo
- Keys from 0 to 2^5-1 (32 tot)

Chord: node join and leave

- In addition to successor pointer, each node also keeps the pointer to its predecessor so to simplify ring maintenance operations
  - Predecessor of node *p* is the first node met in counter-clockwise direction starting at *p*-1
  - When a node joins or leaves, successor and predecessor pointers should be up to date



Se "1" esce → tocca 28 e 4.
Se "30" entra → tocca 28 e 1.

Chord: node join and leave

- When node *p* joins the overlay network, it has to find its place in the Chord ring:
  - Asks to a node to find its successor *succ*(*p*+1) on the ring
  - Joins the ring linking to its successor and informs its successor of its presence
  - Initialize its FT looking for $succ(p + 2^{i-1})$, $2 \le i \le m$
  - Informs its predecessor to update the FT
  - Transfers from its successor to itself the keys for which it becomes responsible
- Example: node 7 joins (see slide 84)
  - Node 7 successor is node 9
  - Node 9 predecessor changes to node 7
  - Node 4 successor changes to node 7
  - Keys 5, 6 and 7 are transferred to node 7

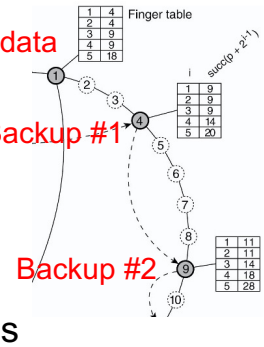
Chord: node join and leave

(NO CRASH, avverte)

- When node *p* voluntary leaves the overlay network:
  - Transfers the keys it is responsible for to its successor
  - Updates the predecessor pointer held by its successor to the node that precedes *p*
  - Updates the successor pointer of its predecessor to its successor
- Example: node 11 leaves (see slide 84)
  - Keys 10 and 11 are transferred to node 14
  - Node 14 predecessor changes to node 9
  - Node 14 successor changes to node 14
- Join/leave operations require $O(\log^2 N)$
- To keep the finger tables updated, each node periodically executes a ring stabilization procedure
  - Nodes can also leave the network abruptly because of failure

Chord: fault tolerance

- Nodes might crash IMPROVVISO
  - *(K, V)* data should be replicated
  - Create *R* replicas, storing each one at *R-1* successor nodes in the ring (R configurabile)
- Need to know multiple successors
  - A node needs to know how to find its successor's successor (or more)
    - Easy only if it knows all nodes!
  - When a node is back up, it needs to:
    - Check with successors for updates of data it owns
    - Check with predecessors for updates of data it stores as backups



Chord: summing up

- Pros
  - Simple and elegant
  - Load balancing
    - Keys are evenly distributed among nodes
  - Scalability
    - Efficient lookup operations: O (log (N))
  - Robustness
    - Periodically update of nodes finger tables to reflect changes in the network
- Cons
  - Physical proximity is not considered (nodo "vicino" a liv. risorsa ≠ nodo vicino fisicam.
  - Expensive support for searches without exact matching
  - Original Chord ring-maintenance protocol is not correct
    - "Reasoning about Identifier Spaces: How to Make Chord Correct" https://arxiv.org/pdf/1610.01140.pdf

~ query "esatte" (ID specifico).

DHT: √ ricerca dati in P2P estesi
√ decentralizzate
X aspetti sicurezza difficile da gestire (Bizantino)
X nodi IN/out e log(N) è alto
X un po' deludente

√ hashing consistente + elegante
√ scalabilità incrementale
√ replicazione, fault-tolerance
√ conf. minima
√ No single-server

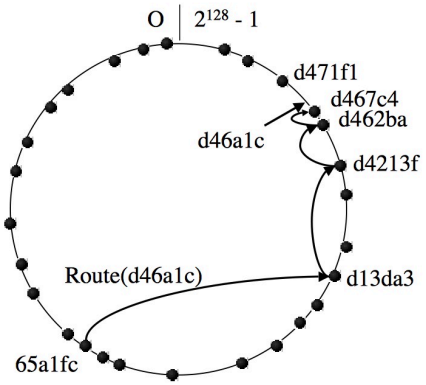
Pastry http://www.freepastry.org

- Provides a substrate for P2P applications
  - Developed by Microsoft Research and Rice University
  - Applications on top of Pastry: Scribe (multicast), SQUIRREL (cooperative caching), PAST (storage)
- Based on ***Plaxton routing***
  - Mechanism for efficient diffusion of objects over a network
  - Published in 1997, before P2P systems came about!
  - Basic idea: resource A is stored by the node whose ID has the longest prefix matching with A's ID
  - *Prefix matching routing*: based on the comparison of node prefix and resource prefix
- Pastry uses a more complex solution: each node also keeps a *leaf set* (set of closest peers by node ID)

Skip per ora

Pastry

- Like in Chord, GUIDs of nodes and resources are mapped onto a ring
- Each key (GUID) is represented with *d* digits in a given base (*b* bits for each digit, usually *b*=4)
- Routing is based on ***longest prefix matching***
  - At each step, the lookup request is forwarded to the node whose ID is gradually closer to that of the searched resource
- Lookup is $O(\log_2{}^b N)$
- Each node has:
  - Routing table
  - Leaf set: L/2 closest peers in each direction around the ring



Looking for resource d46a1c from node 65a1fc

Plaxton routing

- Example: key with *b*=2 and *d*=4 → key of 8 bits
  - Keys are usually longer (128 bits)
- Rules to compose the routing table
  - The node IDs on the *n*-th row share the first *n* digits with the current node ID
  - The (*n*+1)-th digit of the IDs on the *n*-th row is the column number
- More than one node can correspond to each item in the table
  - The closest node in the leaf set is chosen according to some proximity metric (e.g., RTT)

Routing table of node 1220

	0	1	2	3
0	0212	-	2311	3121
1	1031	1123	-	1312
2	1200	1211	-	1231
3	-	1221	1222	1223

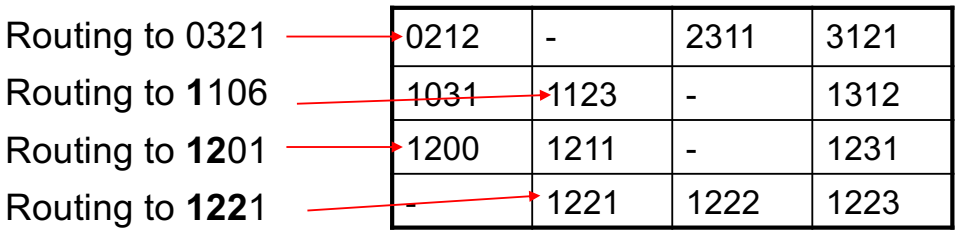
$\lceil \log_2^b N \rceil$ rows in the table

2^b-1 items in each table row

Plaxton routing

- The lookup request is forwarded according to longest prefix matching
  - To the node that shares with the destination node at least one more digit than the current node
  - If it does not exist, to the numerically closest node

Routing table of node 1220



0212	-	2311	3121
1031	1123	-	1312
1200	1211	-	1231
-	1221	1222	1223

DHTs: summing up

- DHTs in retrospective
  - Seem promising for finding data in large P2P systems
  - Decentralization seems good for load, fault tolerance
  - But: security problems are difficult
  - But: churn is a problem, particularly if log(*N*) is big
  - DHTs have not had the hoped-for impact

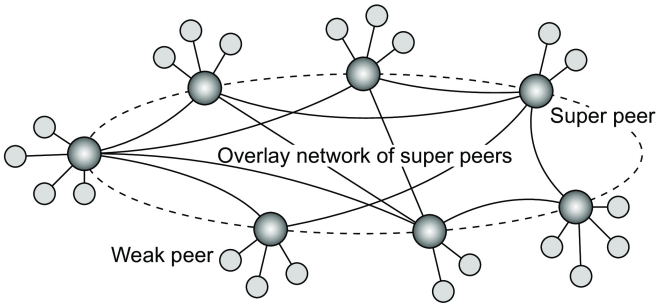
- However, DHTs got right for
  - Consistent hashing: elegant way to spread load across machines
  - Incremental scalability
  - Replication for high availability, efficient recovery
  - Self-management: minimal configuration
  - No single server to shut down/monitor

24/10/22

Hybrid architectures

- So far we have considered centralized and decentralized architectures
- Hybrid architectures aim to combine the benefits of both
- Two examples of hybrid architectures

a) P2P systems with super peers (è ibrida)



- strutturato o meno, i superpeer sono "nodi di riferimento", perché ricevono query dai peer sottostanti e le inoltrano agli altri superpeer.
- ✓ meno nodi per il routing, flooding impatta meno.
- ✗ rottura superpeer
- superpeer può essere collo di bottiglia se troppo carico. Superpeer scelto dopo "ELEZIONE", è il più performante.

Hybrid architectures

b) Classic BitTorrent (BT)

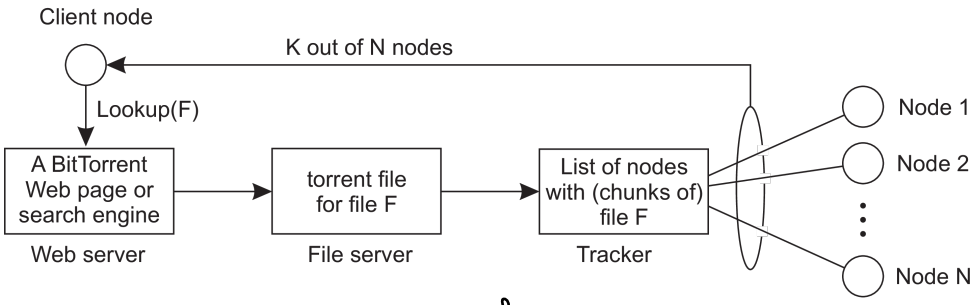
1. User clicks on download link
   - Gets *torrent* file that points to *trackers*
   - Trackers are servers knowing active, collaborating peers that can provide chunks of the file
2. User's BT client talks to tracker (semplifica LOOKUP)
   - Tracker tells it list of peers who have chunks of file
3. User's BT client downloads chunks of file from peers, joining a *swarm* of downloaders, who in parallel get file chunks from source but also distribute these chunks amongst each other

File diviso in chunks che possono essere scaricati in // da più peer (P1 scarica c1, P2 scarica c2 in //)

Forse "centralizzata"

Forse "decentralizzata"

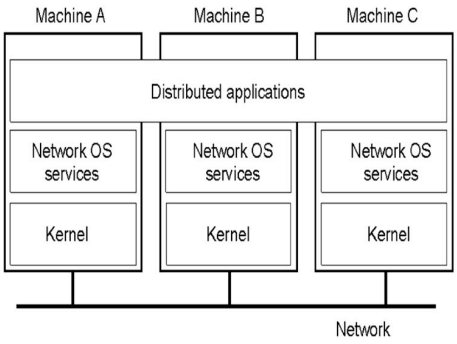
ecco perché è ibrida



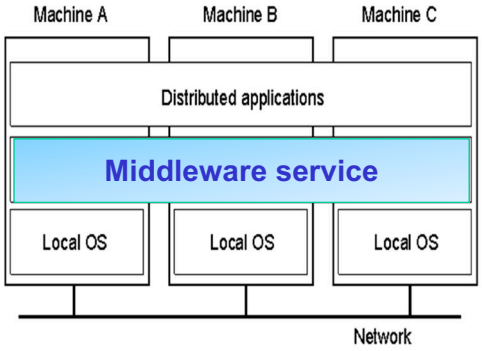
scaricando chunks, creo altra copia dei chunks per mantenerlo nella rete P2P.

Where middleware fits

- Network/OS based DS
  - Communication services
  - But the developer of distributed applications has to mask the differences among the DS nodes

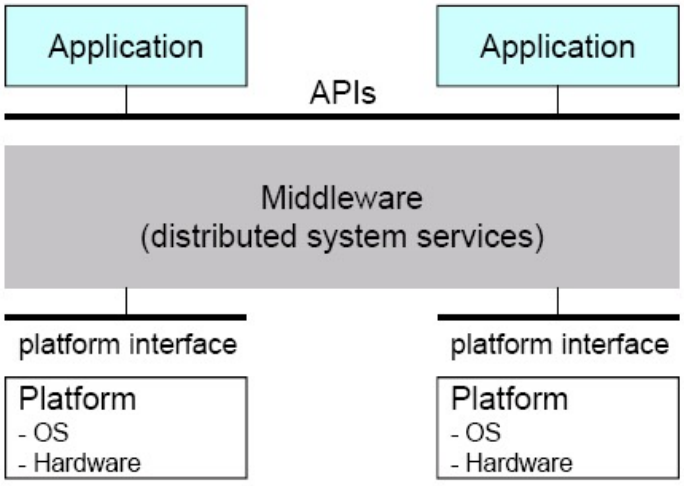


- **Middleware** based DS
  - Middleware provides services to help building distributed applications
  - Masking differences in the underlying platform



Middleware: definition

- Set of tools that sit between applications and the low-level support (hardware, OS, …) (P. Bernstein)



Middleware: other definitions

- Software layer that provides a programming abstraction as well as masks the heterogeneity of the underlying networks, hardware, operating systems and programming languages (Coulouris & Dollimore)
- "Middle" virtual layer between applications and the underlying platforms they run on … that provides significant transparency (R. J. Anthony)

NB: VIRTUALE perché è distribuito. chiamate a procedura remota.

- Examples of middleware: RPC, RMI, CORBA, Java EE, .NET, Web Services, Enterprise Service Bus (ESB), Kafka

Types of middleware

- Many types of middleware
  - We consider a non-exhaustive classification related to the architectural styles we examined
  - Recent middleware platforms offer hybrid solutions
  - Other types: database middleware, transaction-processing middleware, …

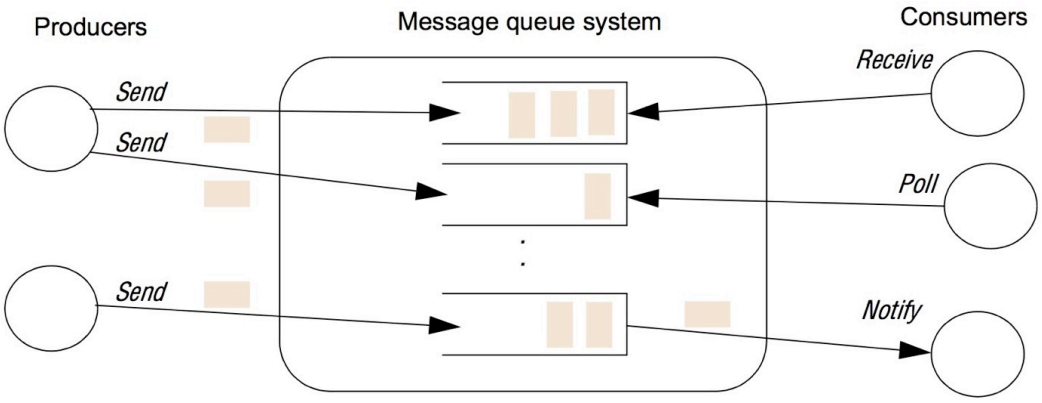
- **RPC/RMI middleware**
  - Client/server (objects) communicate via remote procedure call (remote method invocation) sul proxy server, che risponde al proxy client
  - Interface Definition Language (IDL) to define the contract
  - Communication can be synchronous (older) or asynchronous (newer) → in crescita ↳ più diffusa.
  - Stubs to achieve access transparency (stub = proxy)
  - Examples: SUN RPC, Java RMI, gRPC

Non c'è persistenza. Client/Server devono essere presenti.

Types of middleware

- **Message-oriented middleware (MOM)**
  - Asynchronous and persistent communication (non devono essere entrambi presenti)
  - Offers high flexibility and reliability
  - Many implementations based on **message queue systems**
  - Examples: ActiveMQ, Kafka, IBM MQSeries, RabbitMQ

- La persistenza permette di realizzare i 3 tipi di disaccoppiamento.



Types of middleware

- **Component-based middleware**
  - Evolution of RPC/RMI middleware
  - Both synchronous and asynchronous communication
  - Components live in containers (application servers) (→ non quelli di Docker), which are able to manage the configuration and distribution of components and provide support functionalities
  - Examples: .NET, Java EE (e.g., WildFly, Apache Geronimo, Spring),
- **Service-oriented middleware** (diversi linguaggi)
  - Emphasis on interoperability between heterogeneous components, based on open and universally accepted standard protocols
  - Both synchronous and asynchronous communication
  - Flexibility in the organization of elements (services)
  - Web services, microservices-based architecture
  - Examples: Apache CFX, WSO2

Middleware and architectural style

- In many cases the middleware follows a specific architectural style (rapporto 1-1)
  - E.g., object-based style and RMI middleware
  - This architectural approach to middleware offers design simplicity
  - But lacks of adaptability and flexibility at run-time
- Preferable to have a middleware that can adapt its behavior and properties at run-time with respect to the specific application and environment
  - Reflective middleware: reconfigurable middleware platform that uses *reflection* as a principled mechanism to dynamically adapt middleware behavior to changing environmental context
  - **Self-adaptive middleware** è invece quello che proveremo a fare noi!

Self-adaptive software systems

"Intelligence is the ability to adapt to changes"
S. Hawking

- We prefer a software system capable of adapting its operations at run-time with respect to itself and the environment: a **self-adaptive** (or **autonomic**) software system
  - Applications of autonomic systems in new computing environments
    - Fog computing
    - Mobile edge computing
    - Internet of Things
    - Cyber-physical systems

↳ es. Kubernetis (riporziona container e modo rotto tra quelli non rotti)

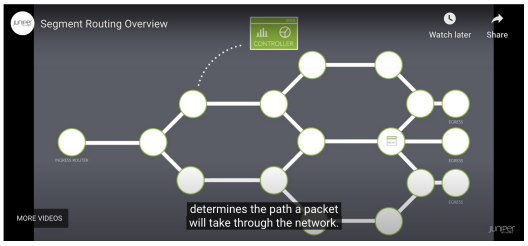
Self-adaptive software systems

- **Autonomic computing**: paradigm capable of responding to the need of managing IT systems complexity and heterogeneity through automatic adaptations
  - Inspired by human autonomic nervous system, able to control some vital functions (heart rate, digestion, temperature, ...) masking their complexity to humans

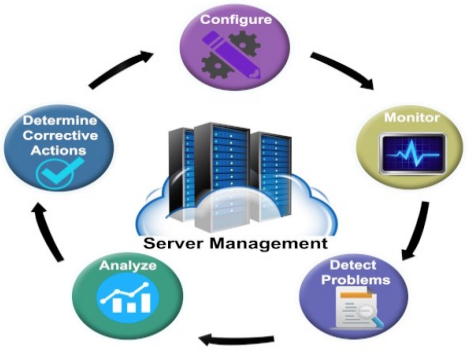
- A self-adaptive (or autonomic) software system can:
  - Manage its functionalities and goals autonomously (i.e., without or with minimal required human intervention)
  - Handle changes and uncertainty in its environment and the system itself

J.O. Kephart, D.M. Chess, The vision of Autonomic Computing, *IEEE Computer*, 2003.

Self-adaptation is everywhere

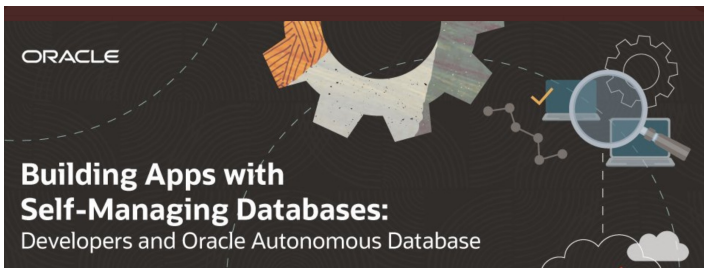


https://www.juniper.net/



http://www.jayatech.in/

https://aws.amazon.com/it/ec2



https://twitter.com/OracleDevs/

Goals of self-adaptive systems

detto self-star

- A self-adaptive (*self-**) software system is able to self-manage, pursuing the following objectives: (non è detto li rispetti tutti)
- Self-optimizing
  - Capability of system to optimize its resource usage or performance (e.g., by changing the deployment) while providing its required quality goals
  - E.g., change placement of application components onto system nodes to satisfy application response time
- Self-healing (auto-guarirsi)
  - Capability of system to discover, diagnose and recover from faults to provide its required quality goals or degrade gracefully otherwise
  - E.g., detect a crashed node and exclude it from serving requests (con Elastic Load Balancer)
  - identificazione di intrusioni/comportamenti anomali in contesti di Cyber Security

a runtime

oppure creare repliche su nodi più vicini ad una certa area.

senza richiamare l'amministratore del sistema

"IMPLICA" / sono correlati

Goals of self-adaptive systems

- Self-protecting
  - Capability of system to defend against security threats and anticipate problems they may cause, to provide its required quality goals
  - E.g., in a network of IoT devices detect a jamming attack that corrupts network traffic and adapt the packets schedule
- Self-configuring
  - Capability of system to automatically integrate new elements, without interrupting the system's normal operation, or tune some configuration parameters
  - E.g., discover a new node and add it to serve requests, non è l'amministratore del sistema che lo fa!

Come raggiunge questi obiettivi?

How to achieve the goals of a self-* system?

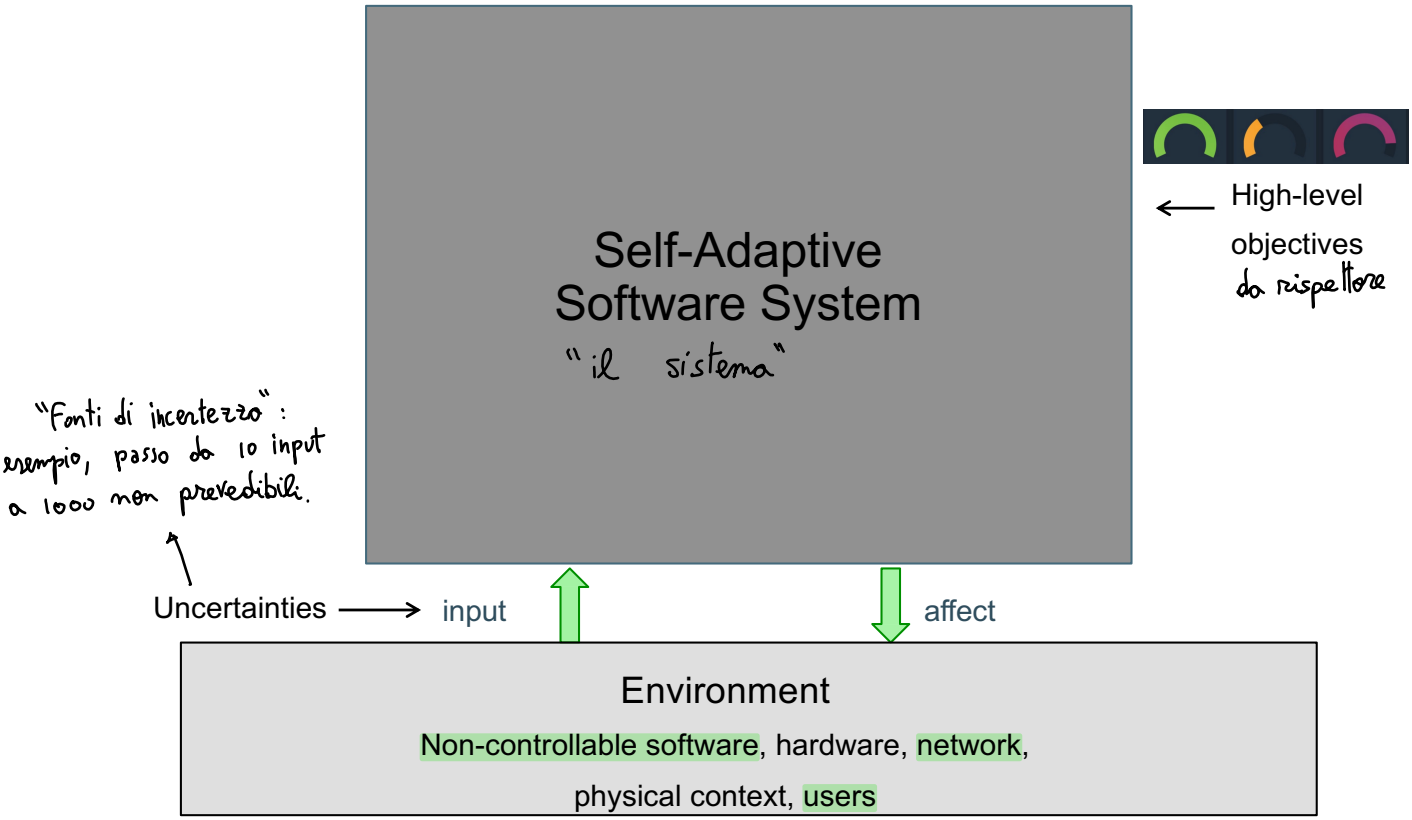
- The system should know its internal state (self-awareness) and the current external operating conditions (self-situation)
- Should identify changes regarding its state and the surrounding environment (self-monitoring)
- And should adapt consequently (self-adjustment)

serve una componente di monitoraggio

si adatta

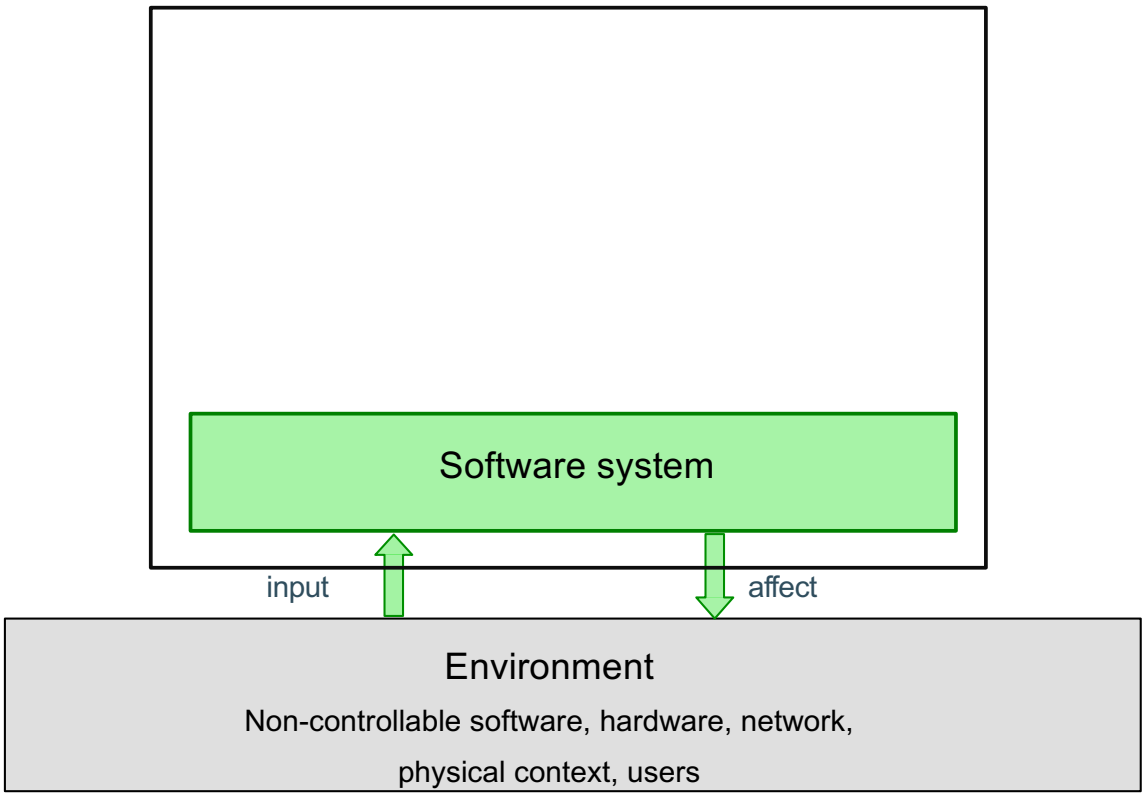
- These attributes are the implementation mechanisms

Conceptual model of self-adaptive system

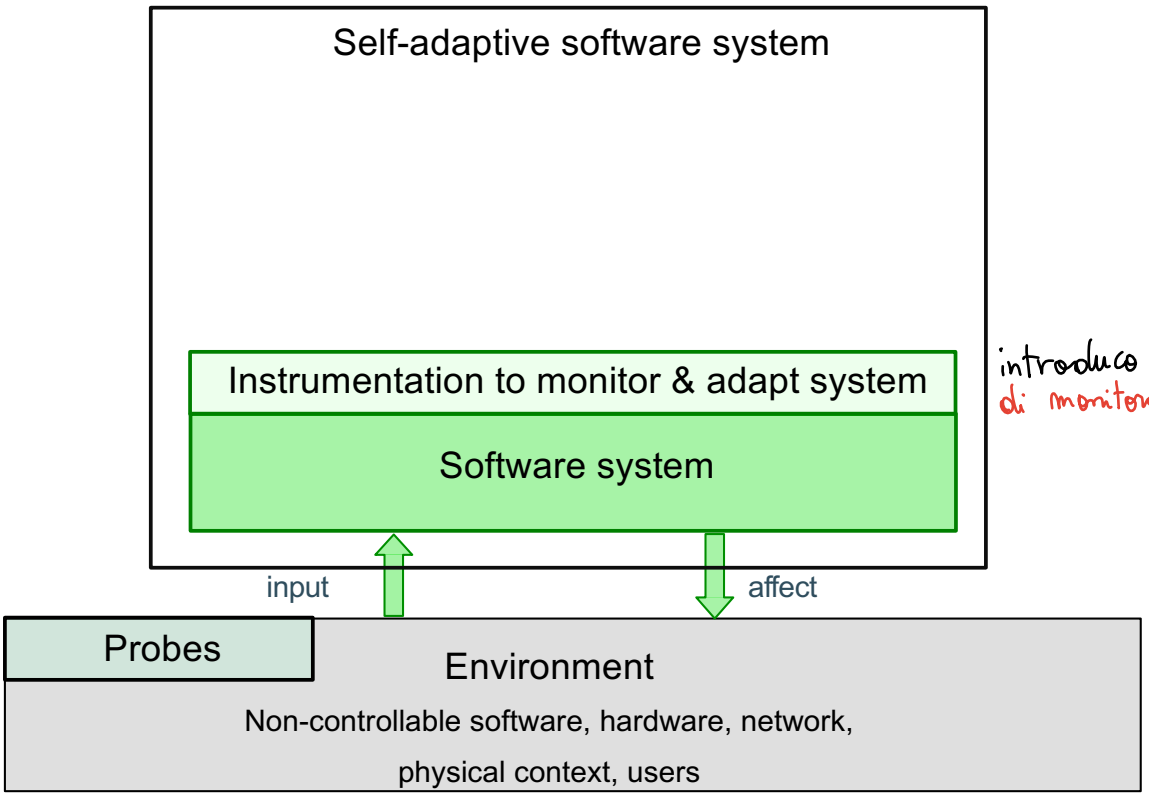


Quali sono i componenti?

Conceptual model of self-adaptive system

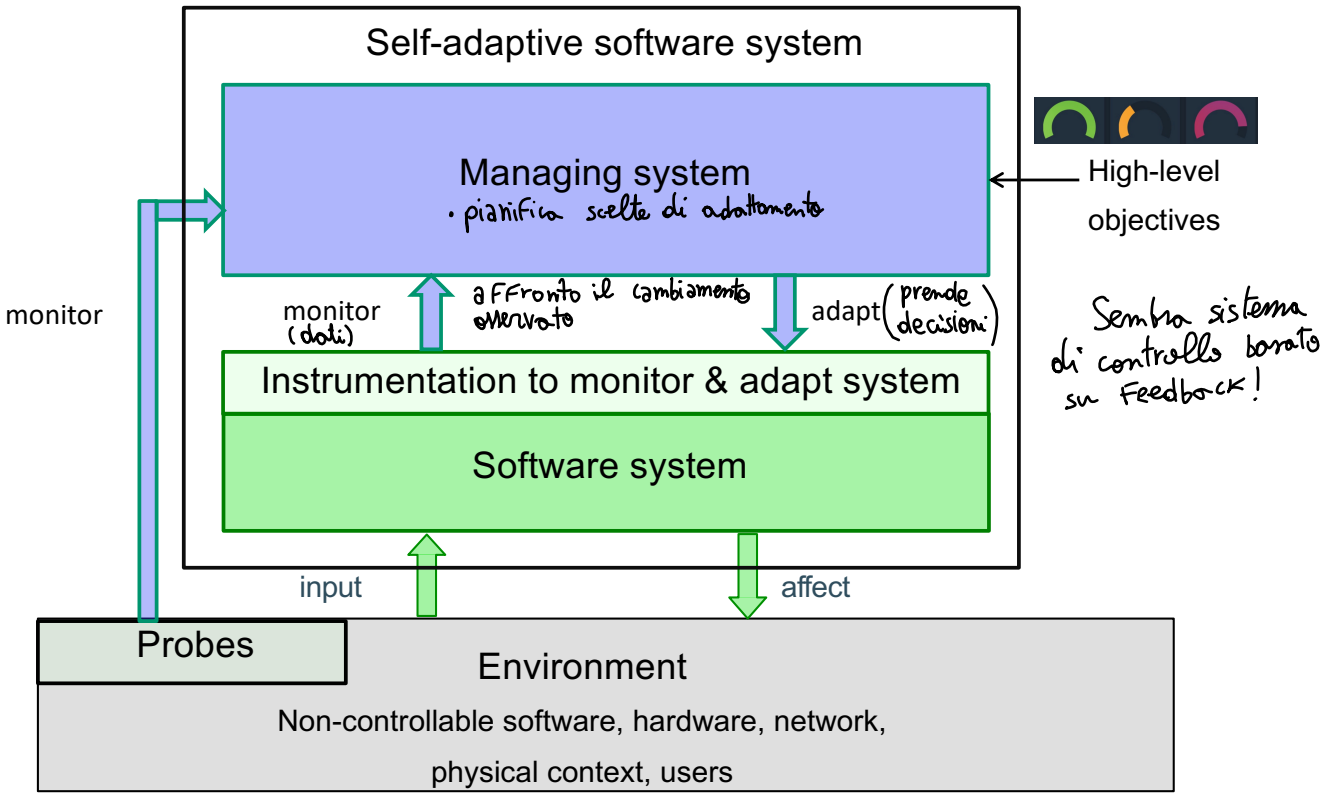


Conceptual model of self-adaptive system



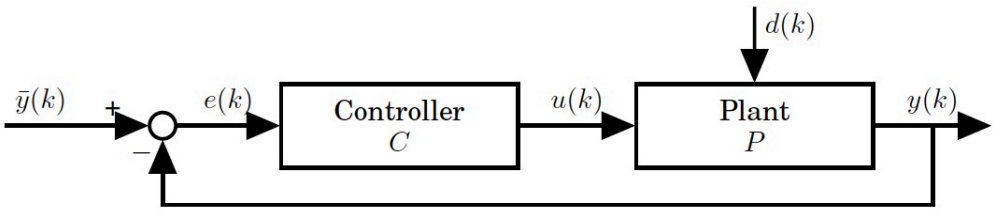
introduco strumenti di monitoraggio

Conceptual model of self-adaptive system



You are already familiar with this model

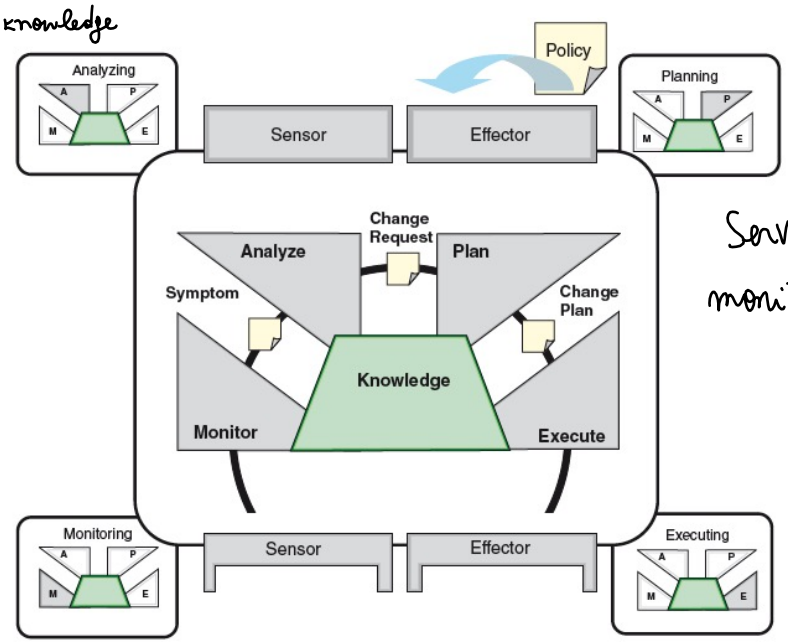
- The control-theory perspective of a self-adaptive system



Nell'Ing. del SW l'architettura software di riferimento per sistemi auto-adattativi

MAPE: reference architecture for self-adaptive systems

- **MAPE** (Monitor, Analyze, Plan, Execute) loop
  - MAPE-K ← knowledge



Serve per realizzare monitor + Managing System!

MAPE: building blocks (or phases)

- **M**onitor
  - Collects data from the managed system and execution environment through sensors; aggregates, filters and correlates these data into symptoms that can be analyzed → cerca "trend" o caratteristiche (mem usata, throughput, ...)
- **A**nalyze
  - Observes and analyzes situations to determine need for adaptation
  - If adaptation is required, it triggers Plan (es: uso risorse = 80%, monitor ha regola: risorse > 70% scala → il Plan riconfigura adattamento.)
- **P**lan
  - Determines which mitigation actions need to be performed so to enact a desired alteration in the managed system
- **E**xecute
  - Enacts the change plan by carrying out the actions determined by Plan through effectors so to adapt the managed system
- **K**nowledge
  - Stores shared knowledge regarding relevant aspects of the managed system, environment, and the administrator's goals

MAPE: Monitor

- Main design options for Monitor:
  - When to monitor: continuously, on demand
  - What to monitor: resources, workload, performance, …
  - How to monitor: architecture (centralized vs. decentralized), methodology (active vs. passive)
  - Where to store monitored data and how (e.g., some pre-processing)

MAPE: Analyze

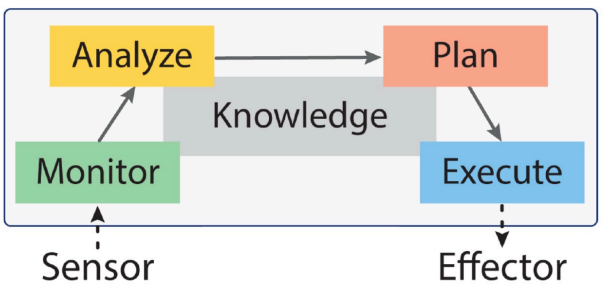
- Main design options for Analyze:
  - When to analyze: event- or time-triggered
  - Reactive vs proactive adaptation
    - Reactive: *in reaction* to events that have already occurred (e.g., increase number of resources to react to workload increase)
    - Proactive: based on *prediction* so to plan adaptation actions in advance (e.g., increase number of resources before workload increase occurs)

MAPE: Plan

- Perhaps the most challenging and studied MAPE phase, ∃ tante metodologie di approccio.
- A variety of methodologies and techniques can be used to plan the adaptation
  - Optimization theory (non sempre va bene, a volte è "troppo lento" rispetto l'evento, può portare problemi NP-hard)
  - Heuristics
  - Machine learning, including reinforcement learning (tipo di alg. ML) → supervised (reg. Lin.) / unsupervised (tec, alg K-means); → cerco di imparare, non usano DATI ma sono lenti ad apprendere.
  - Control theory
  - Queueing theory
  - …
- Example: elastic provisioning of Cloud resources (e.g., virtual machines or containers)

How to control adaptation in a decentralized fashion?

- Alternatives to design the control architecture of a self-adaptive system:
  - Centralized MAPE: all MAPE components on same node, simpler but lack of scalability in geo-distributed environments
  - **Decentralized MAPE**: MAPE components are distributed; many patterns, each one with pros and cons
    - No clear winner, it depends on system and application features and requirements



How to decentralize the adaptation control

- Architectural patterns for decentralized MAPE

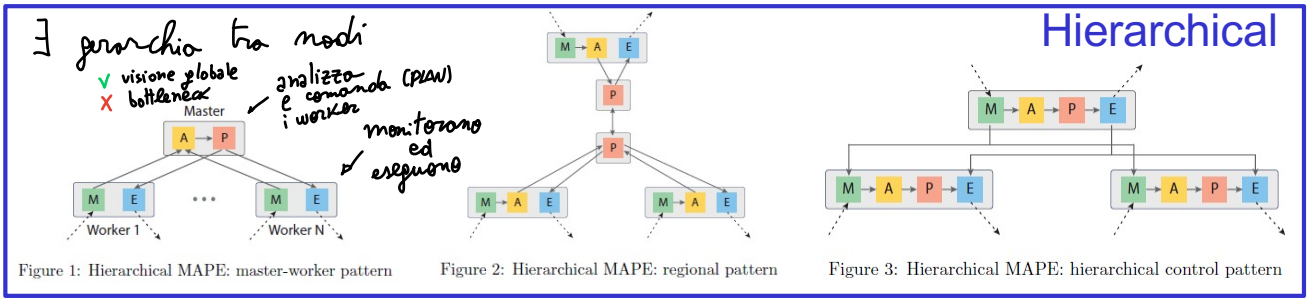


Figure 1: Hierarchical MAPE: master-worker pattern

Figure 2: Hierarchical MAPE: regional pattern

Figure 3: Hierarchical MAPE: hierarchical control pattern

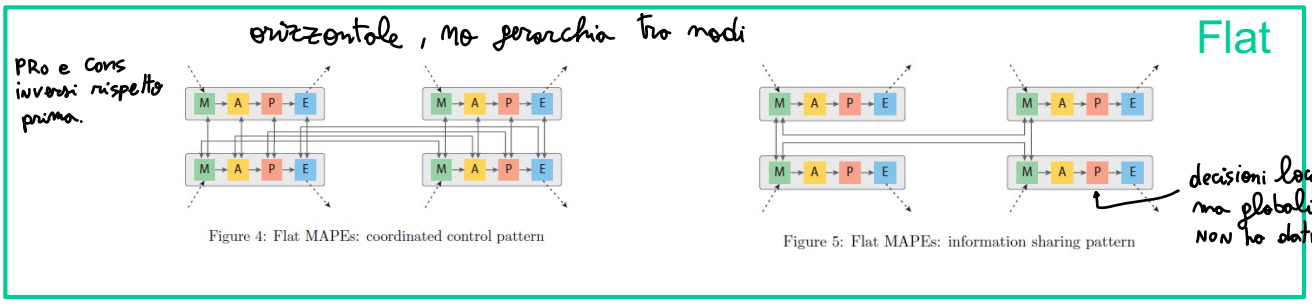


Figure 4: Flat MAPEs: coordinated control pattern

Figure 5: Flat MAPEs: information sharing pattern

D. Weyns et al., "On patterns for decentralized control in self-adaptive systems. In *Software Engineering for Self-Adaptive Systems II*, 2013.

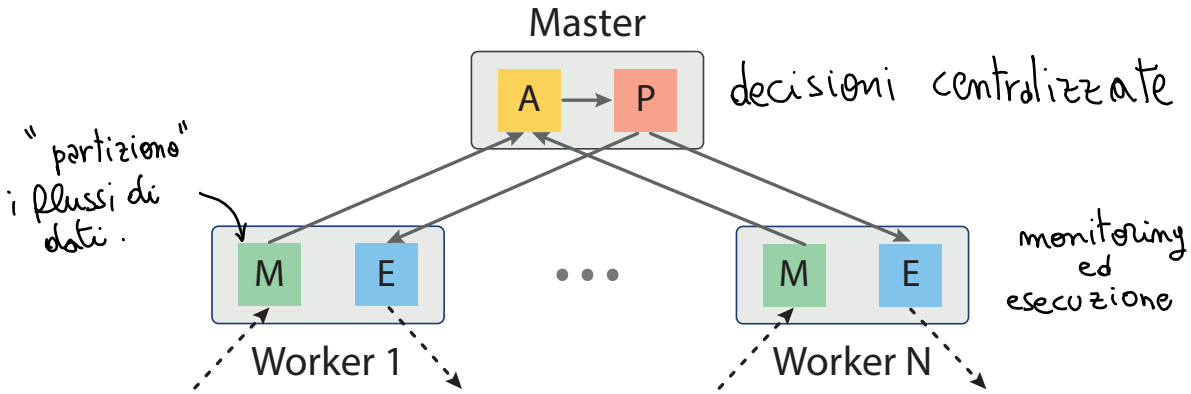
26/10/22

How to decentralize the adaptation control

- First design choice: ***hierarchical*** vs. ***flat***
  - Hierarchical: easier but higher risk of bottlenecks in top levels of hierarchy
  - Flat: more difficult to coordinate, but can scale better
- **Hierarchical MAPE patterns**: multiple MAPE loops organized in a hierarchy, where a higher-level control loop manages subordinated control loops
  - Master-worker
  - Regional
  - Hierarchical control
- **Flat MAPE patterns**: multiple MAPE loops cooperate as peers
  - Coordinated control
  - Information sharing

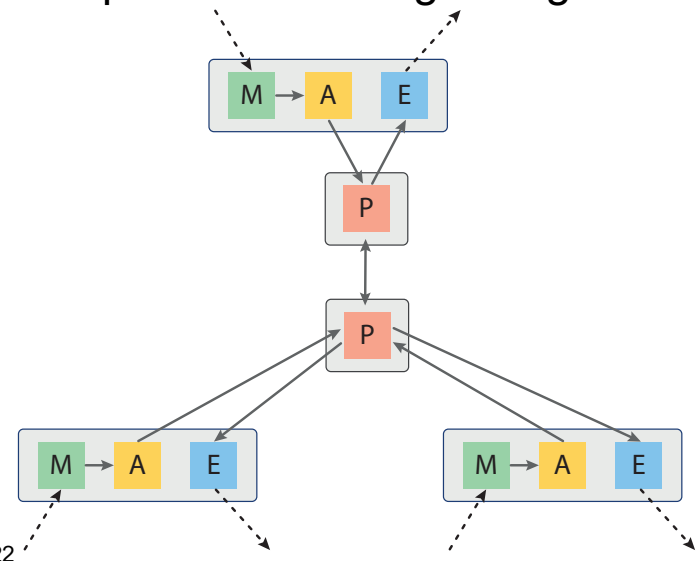
Hierarchical MAPE: master-worker pattern

- Decentralize M and E on workers, keep A and P centralized on master
- √ Pro: global view on master that can achieve global adaptation goals
- ✗ Cons: communication overhead and risk of performance bottleneck and SPOF on master



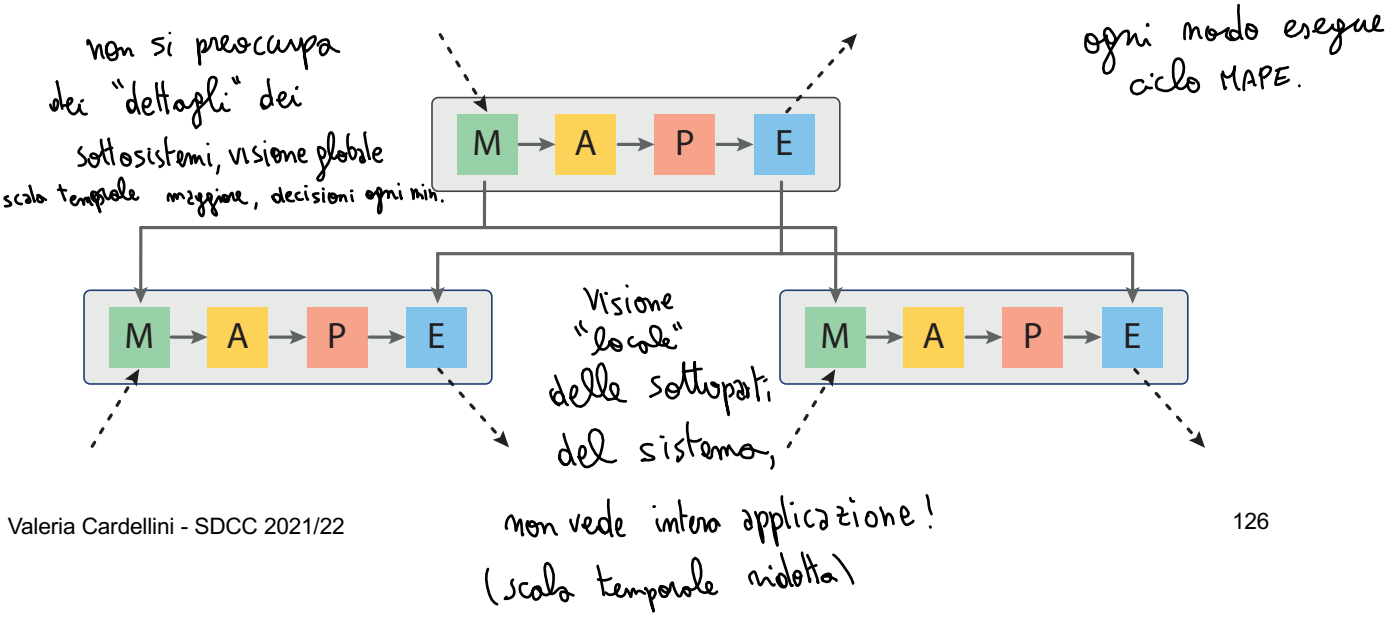
Hierarchical MAPE: regional pattern SKIP

- Multiple and loosely coupled regions, each having a two-level hierarchical structure
- Pro: better flexibility (regions can be under different ownership or administrations)
- Con: more complex to achieve global goals



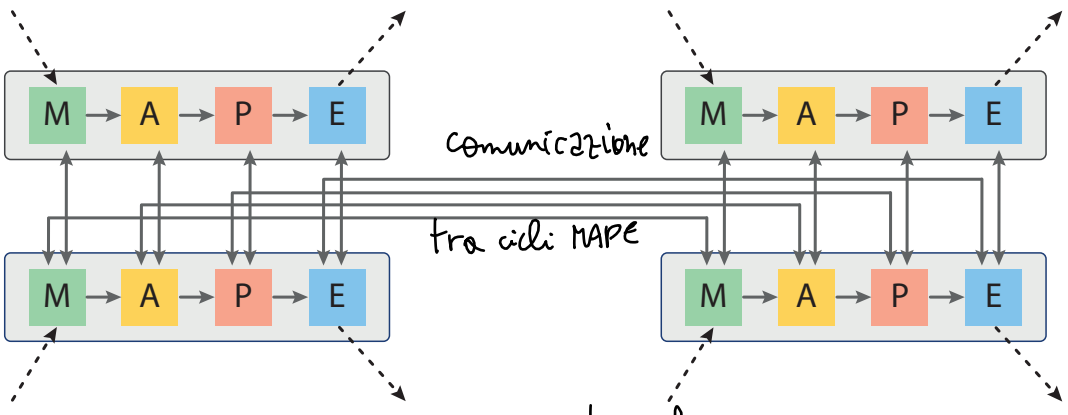
Hierarchical MAPE: hierarchical control pattern

- Multiple MAPE loops which can operate at different time scales and with separation of concerns
- Pro: top-level MAPE can achieve global goals
- Con: can be non-trivial to identify different levels of control, depends on controlled system characteristics



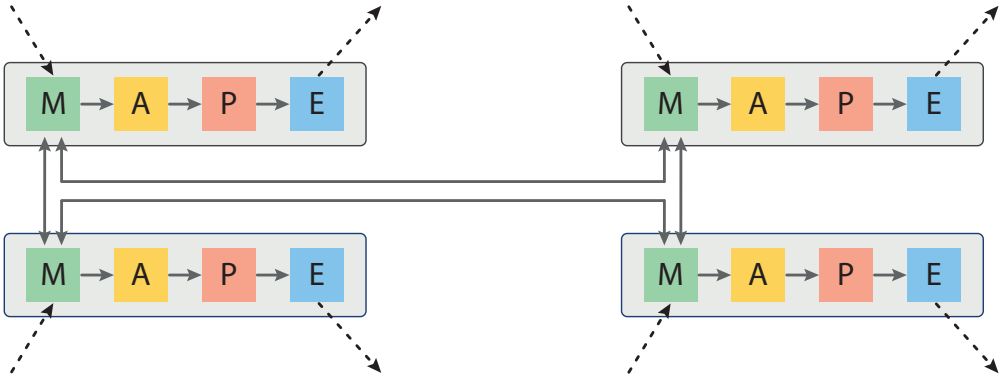
Flat MAPE: coordinated control pattern

- Multiple control loops, each one in charge of some part of the controlled system but coordinated through interaction
- Pro: better scalability, No Point OF failure
- Con: more difficult to take joint adaptation decisions, latenza tra msg.



Flat MAPE: information sharing pattern

- Special case of coordinated control pattern: interaction only among M components, comunico solo Monitoring
- Pro: better scalability
- Con: lack of planning coordination, conflicting or sub-optimal adaptation actions can be enacted

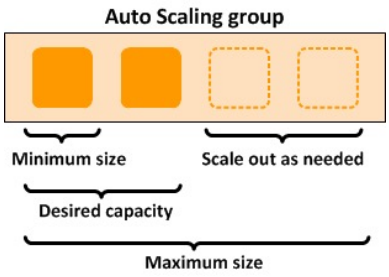


Examples of self-adaptive systems

- Let's analyze 3 examples of self-adaptive systems for resource management
  1. Auto-scaling EC2 instances
  2. Selecting services of composite applications
  3. Auto-scaling microservice-based applications
- Common ground
  - Applications facing unexpected events (e.g., workload surge, node crash)
  - Adaptation goal: satisfy some QoS requirement (e.g., application response time, application availability)
  - Examples differ in planning methodologies and control architectures

Example 1: Amazon EC2 Auto Scaling

- AWS service to automatically add or remove EC2 instances according to user-defined conditions and health checks https://aws.amazon.com/ec2/autoscaling/
  - Monitor load on EC2 instances using CloudWatch
- **Dynamic scaling**: user-defined scaling plan tells when and how to scale (*reactive* scaling)
  - Based on threshold-based heuristic policy
    - Set high and low thresholds on some metric(s) for alarms
    - Example of scale-out rule: if average CPU utilization of all instances > 70% in last 1 minute, add 1 new instance
    - Example of scale-in rule: if average CPU utilization of all instances is <35% in last 5 min, remove 1 instance



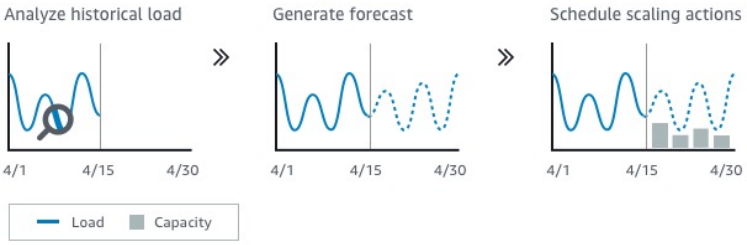
Example 1: Amazon EC2 Auto Scaling

- Dynamic scaling based on threshold-based heuristic policy
  - Pro
    - Simple and intuitive policy: select metrics and thresholds on some metric(s) for alarms
  - Cons
    - How to set thresholds values? Can be application-dependent (application tasks can be CPU-intensive, memory-intensive, IO-intensive or a mix) and not robust against changing load
    - Metrics are not application-specific (e.g., response time)

Example 1: Amazon EC2 Auto Scaling

Modalità 2:

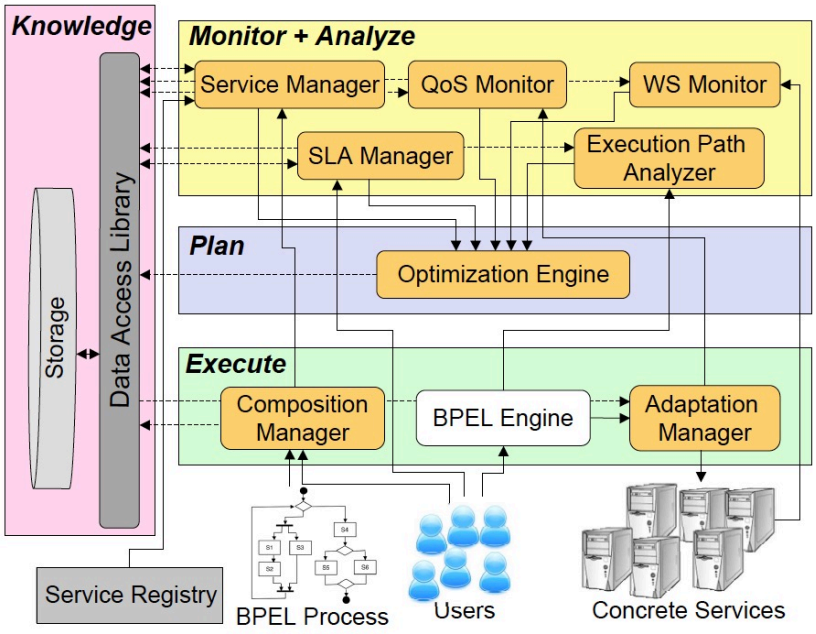
- **Predictive scaling**: based on machine learning (*proactive* scaling)
  - Trained ML model to predict application expected traffic and EC2 usage, including daily and weekly patterns
  - Historical data collected from CloudWatch
  - Model needs at least one day's of historical data to start making predictions
    - Re-evaluated every 24 hours to forecast for the next 48 hours



- Pro: proactive
- Cons:
  - Requires training (the more the historical data, the more accurate the forecast)
  - Load metric choice is core (deve essere scelta)

Example 2: Service selection

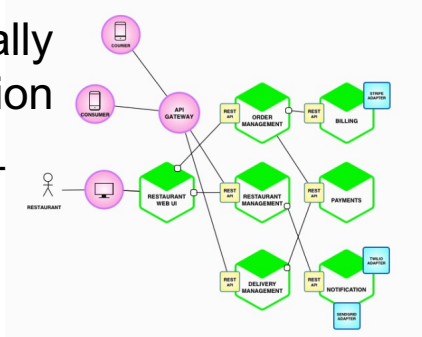
- QoS-driven self-adaptation of SOA applications
  - Centralized Plan: select *optimal set of concrete services* (and their coordination) by means of linear programming optimization

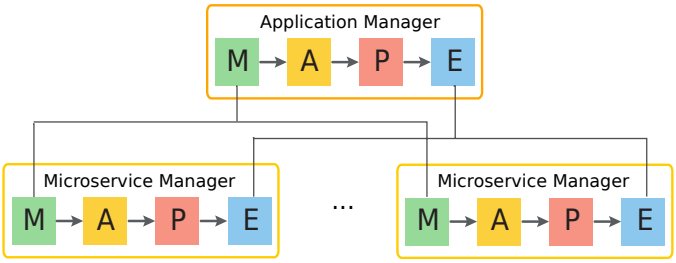


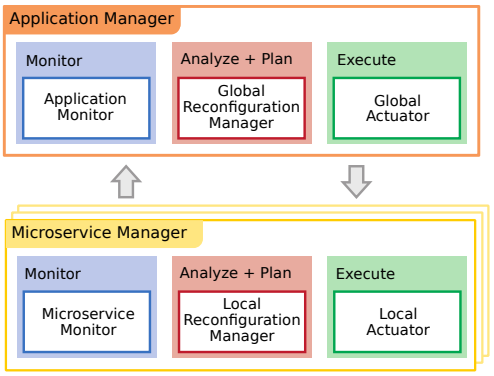
V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. Lo Presti, R. Mirandola, "MOSES: a framework for QoS driven runtime adaptation of service-oriented systems", *IEEE Trans. Soft. Eng* 2012

Example 3: hierarchical scaling of microservices in Kubernetes

- Hierarchical control pattern to elastically scale a microservices-based application
  - Local Plan at microservice level to scale-out/in each microservice uses a policy based on reinforcement learning
  - Global Plan at application level uses a policy based on queueing theory







F. Rossi, V. Cardellini, F. Lo Presti, "Hierarchical scaling of microservices in Kubernetes", *Proc. IEEE ACSOS 2020*.