



**Università degli Studi di Bologna  
Facoltà di Ingegneria**

# **Corso di Reti di Calcolatori L-A**

***Generalità, obiettivi e modelli di base***

**Antonio Corradi**

**Anno accademico 2009/2010**

# OGGETTO del CORSO

---

## Reti di calcolatori e Sistemi distribuiti

Definizione delle architetture di interesse

**Insieme di sistemi distinti in località diverse  
che usano la comunicazione e la cooperazione  
per ottenere risultati coordinati**

**Sistemi più complessi** ma anche  
**motivazioni forti** all'uso per la possibilità di

- **accesso a risorse remote**
- **condivisione** risorse remote come locali

# DIMENSIONI dei SISTEMI d'INTERESSE

---

Studiamo sistemi di **piccola** (pochi nodi), **media** (decine), **grande** dimensione (globali tipo Internet)

Requisiti

- **TRASPARENZA della ALLOCAZIONE**
- **DINAMICITÀ del SISTEMA**
- **QUALITÀ dei SERVIZI (QoS)**

ma seri problemi teorici (**COMPLESSITÀ**)

**Concorrenza**: moltissimi processi possono eseguire

**Nessun tempo globale**: non sincronizzazione degli orologi

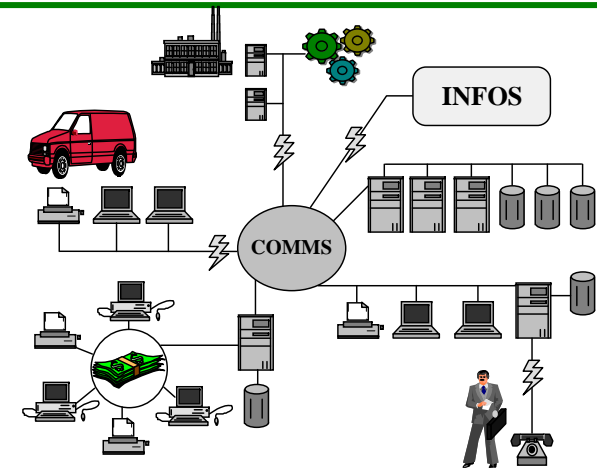
**Fallimenti indipendenti**: molte cause di fallimento, crash di macchine e possibili problemi di rete

# SISTEMI DISTRIBUITI

## MOTIVAZIONI tecniche ed economiche

### Network locali:

- wide WAN,
- locali LAN, e anche
- reti wireless ...



## Richieste distribuite per domande distribuite con accessi eterogenei - **prenotazioni aeree**

- **accessibilità** e **condivisione** delle risorse
- **affidabilità** (dependability) per tollerare guasti
- **uniformità** in **crescita** e **scalabilità**  
(indipendenza dal numero dei nodi del sistema)
- **apertura** del sistema o openness (capacità di evolvere e operare seguendo le evoluzioni di specifiche)

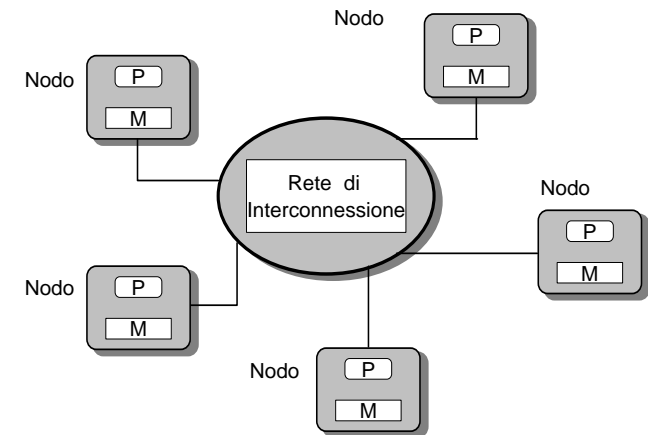
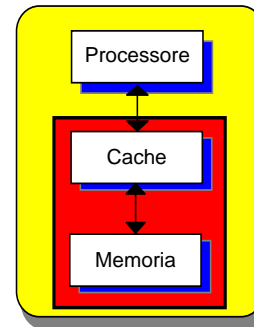
# TIPICA ARCHITETTURA di INTERESSE

## Architetture MIMD

(Multiple Instruction Multiple Data)

fatte di **nodi diversi**

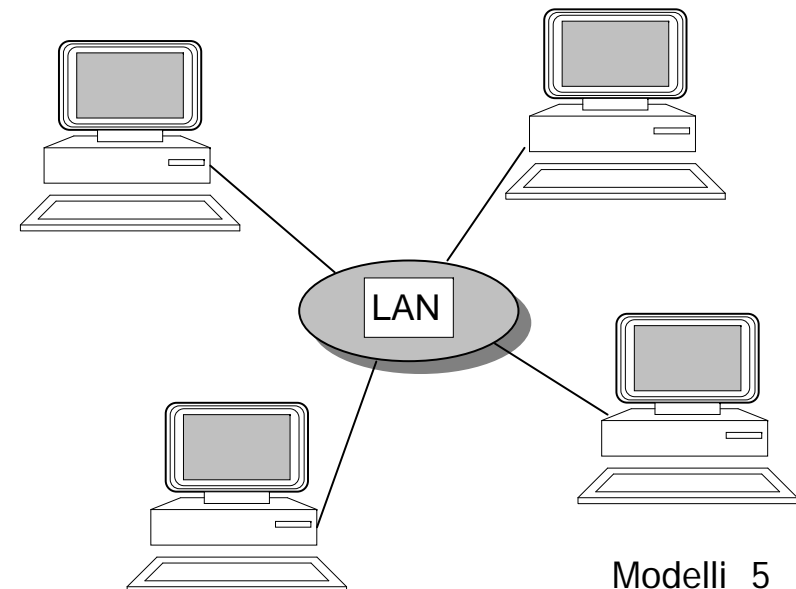
**Nodo** processore collegato  
alla memoria privata organizzata a livelli



## Reti di workstation

**Calcolatori indipendenti** connessi  
da una o più reti locali

Il corso si occupa non di  
**architetture hardware**  
ma di **architetture software**



# ARCHITETTURA SOFTWARE (?)

## Per una applicazione distribuita

Analisi, sviluppo, tramite algoritmo e sua codifica in linguaggi di programmazione

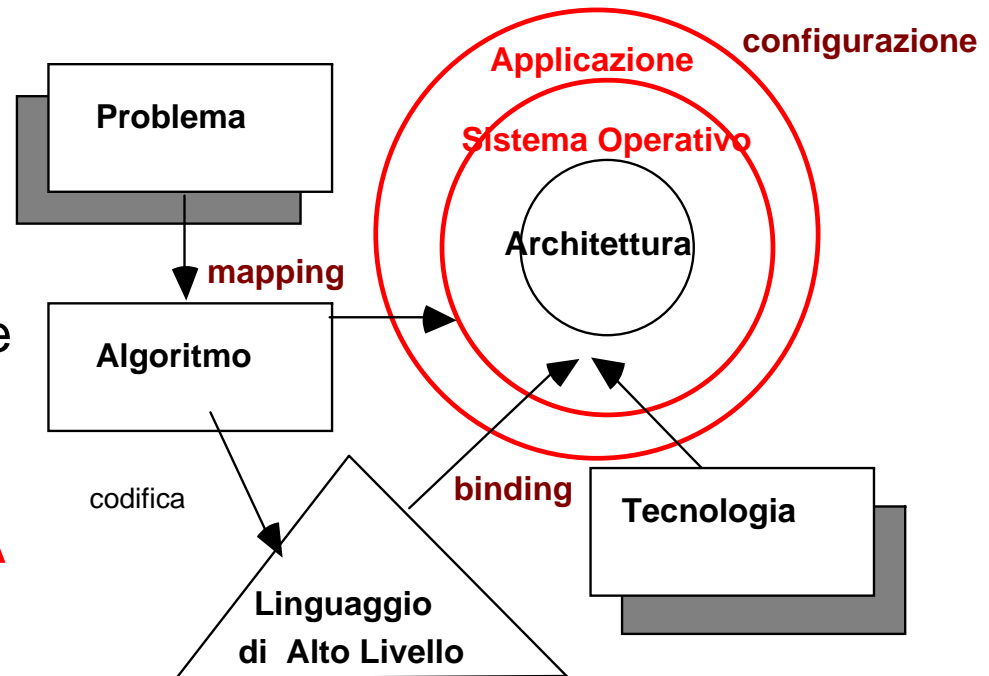
### MAPPING

**configurazione** per l'architettura e  
**allocazione** a risorse e località

### BINDING di risorse

come ogni entità della applicazione  
**si lega** alle risorse del sistema

Gestione **STATICA** vs. **DINAMICA**  
del legame di BINDING



Massimo interesse per l'**esecuzione** ed il **supporto**

# STATO dell'ARTE

---

Per i **sistemi operativi** esiste uno **standard** per **funzioni di accesso (API)** e modello architetturale

**UNIX**

**Sistema operativo concentrato (e relativa macchina virtuale)**

**modello di conformità**

- per caratteristiche di **accesso ai file**
- per possibilità di **concorrenza**
- per possibilità di **comunicazione**

**con primitive di sistema (kernel) invocabili da diversi ambienti**

Soluzioni diffuse tutte ispirate:

- **Linux** ed altre evoluzioni che fanno ancora i conti con UNIX
- Microsoft **Windows NT** che introduce alcuni altri elementi rinforzato da OPEN SOURCE e FREE SOFTWARE

**Importanza degli STANDARD, anche di fatto**

# UNIX come STANDARD

---

Per i **processi** esiste condiviso il modello a **processi pesanti**, meno accettata la specifica di processi leggeri

Alcuni **sistemi** si discostano poco per ottenere **migliori prestazioni** nella gestione introducendo processi leggeri

Mantenendo le **funzioni di accesso (API)** e il modello architetturale

**Esempio: non si più usano kernel monolitici (come primo UNIX)**

che si accettano in blocco (o meno) **che possono produrre overhead per modifiche anche minime**

**Uso di microkernel (vedi UNIX, WINxx, ... ecc. ecc.)**

realizzazioni minimali del supporto (**meccanismi**) di S.O. nel kernel con le **politiche** realizzate a livello applicativo sopra al kernel

- apertura a nuove strategie (**generalità e flessibilità**)
- **costi superiori** delle strategie (rispetto a soluzione nel kernel)

I microkernel contengono supporto ai **processi** e **comunicazioni** tra loro, politiche realizzate in spazio utente



# MODELLO dei PROCESSI

---

**Processi differenziati**  $\Rightarrow$  processi **pesanti** / processi **leggeri**

## IMPLEMENTATIVAMENTE

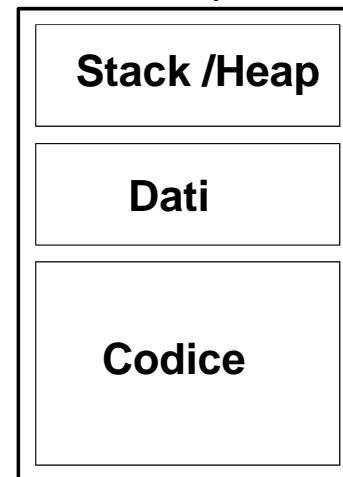
Processo come aggregazione di parecchi componenti

Uno **SPAZIO di indirizzamento** e uno **SPAZIO di esecuzione**  
insieme di **codice**, **dati** (statici e dinamici),

**parte di supporto, cioè di interazione con il sistema operativo (file system, shell, socket, etc.) e per la comunicazione**

I **processi pesanti** richiedono  
molte risorse ad esempio in UNIX  
cambiamento di contesto operazione  
molto pesante con overhead  
soprattutto per la parte di sistema

Processo pesante



# PROCESSI LEGGERI

**Processi leggeri**  $\Rightarrow$  per ovviare ai limiti dei processi **pesanti** si creano **entità più leggere con limiti precisi di visibilità e barriere di condivisione**

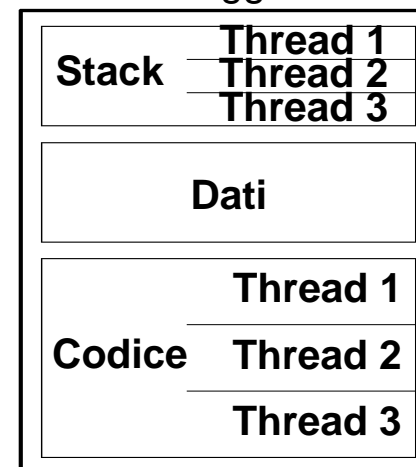
**Processi leggeri** sono attività che **condividono visibilità tra di loro** caratterizzate da uno **stato limitato** e producono **overhead limitato**

ad esempio in UNIX, le librerie di thread o in Java i thread

Quale **contenitore unico** si considera, in genere, un processo pesante per fornire la visibilità comune a tutti i thread

Tutti i sistemi vanno nel senso di offrire granularità differenziate per ottenere un servizio migliore e più adatto ai requisiti dell'utente

Processi leggeri



# STANDARD nel Distribuito

---

## *Problema fondamentale*

**Sistema distribuito** fatto di **nodi** anche **molto diversi** ed **eterogenei** con **esigenze difficili** da prevedere **tutte prima** della **esecuzione**

Oltre a **UNIX** come macchina virtuale standard, altri approcci ...

## **APPROCCI AD AMBIENTI APERTI**

uso di un **ambiente aperto unificante** per superare le differenze delle diverse piattaforme anche **durante la esecuzione** senza bloccare il sistema e pensare ad un riprogetto

## **Modelli di sistemi di supporto Object-Oriented** (uso di compilatori)

Si definisce un ambiente standard, CORBA, .NET, ...

o un linguaggio unificante come **Java**

## **Modelli ad ambiente interpretati** (uso di interpreti e script)

Si introducono ed usano linguaggi di script e interpreti corrispondenti per fare fronte alle esigenze dinamiche durante la esecuzione

tipo Shell (bash, o altre), TCL/Tk, Perl, Python ...

Si noti che Java nasce anche come linguaggio interpretato

# JAVA come STANDARD

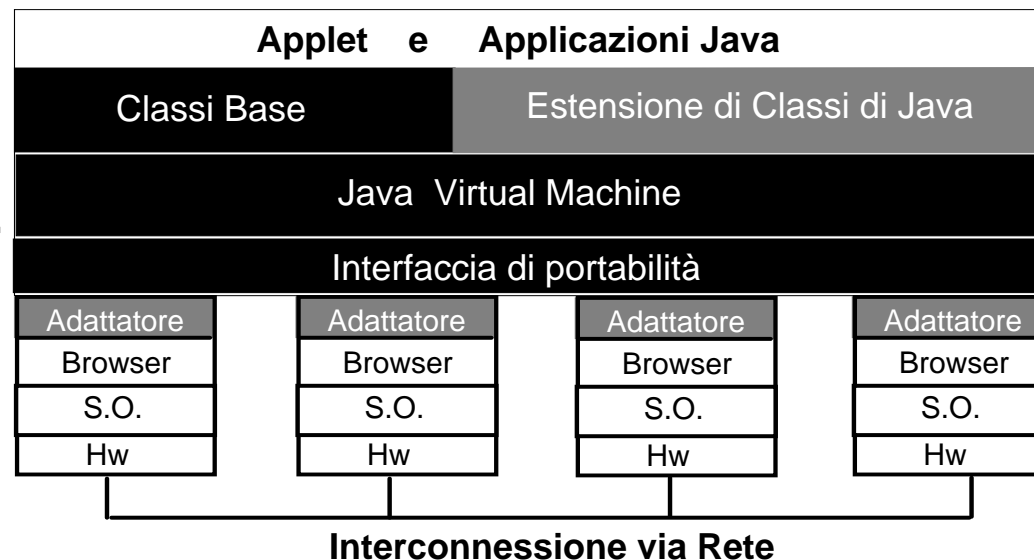
## Java (JDK1.5)

Linguaggio Object-Oriented legato ad un insieme di librerie per legarsi e richiedere le funzioni del sistema operativo nativo sottostante

- **processi leggeri** (thread)
- **file system** e risorse di sistema
- **risorse di sistema e di comunicazione** (**Web**, **sicurezza**, ...)

Le versioni di Java vanno oltre in questa direzione di integrazione

- gestione, monitoraggio, e accounting di risorse
- modello a processi da migliorare ...



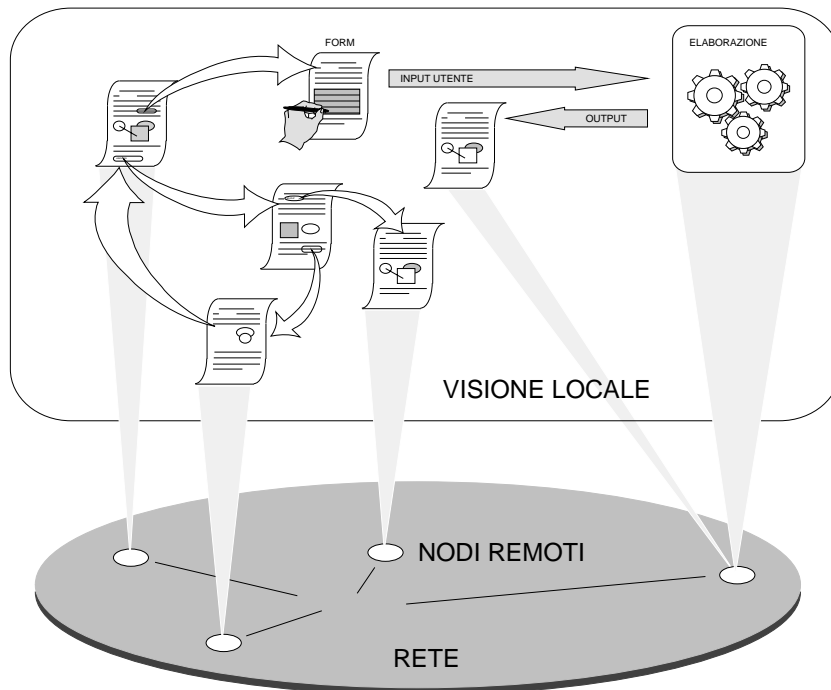
# Un caso usuale distribuito

Il primo caso è quello di un **accesso a pagine Web**

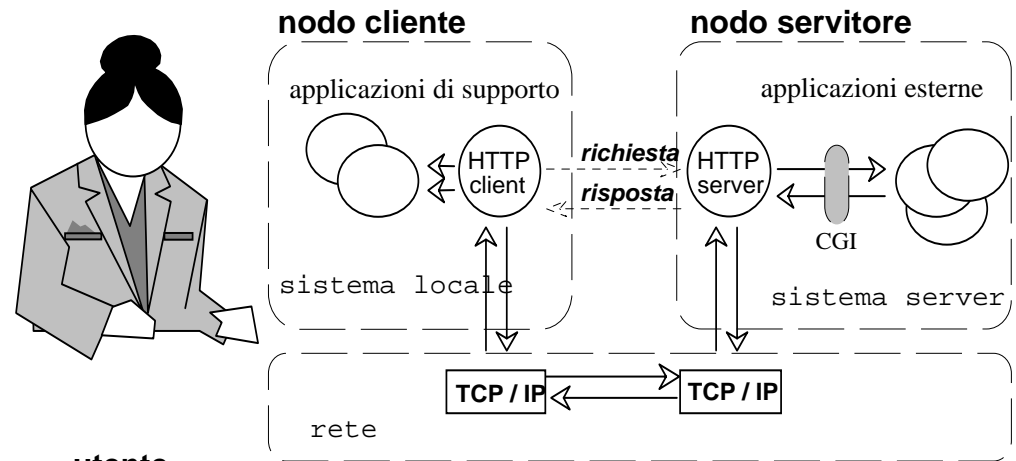
*Il sistema non è tanto distribuito ma piuttosto semplicemente in rete ...*

Un utente accede alle pagine Web che sono depositate e mantenute da vari server (in modo trasparente alla allocazione)

**Visione utente**



**Visione tecnica (architettura?)**



utente

**interazione cliente /servitore**

# ARCHITETTURA in GIOCO

## Il sistema prevede molti rapporti cliente/servitore

L'utente che è cliente del browser

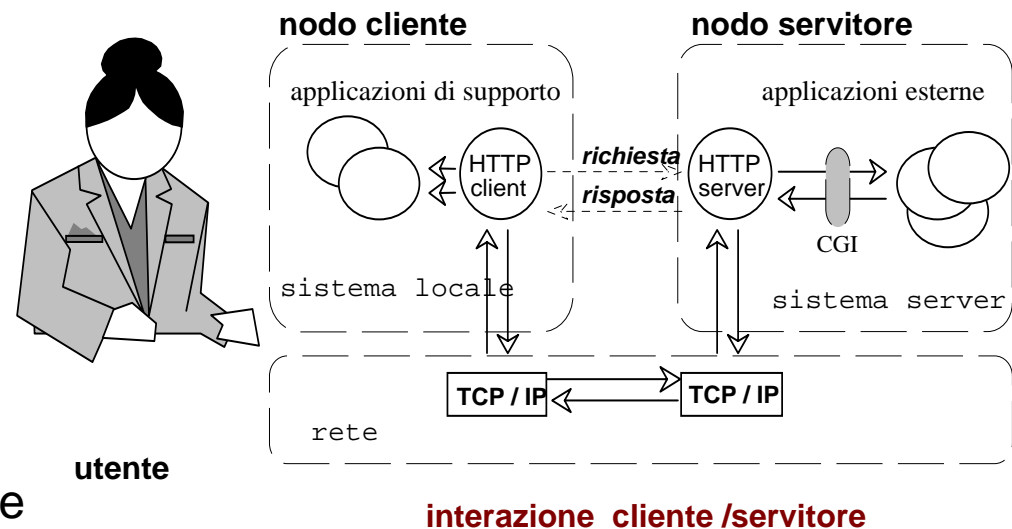
Il browser sulla macchina cliente che è cliente del nodo server

Il cliente TCP che è cliente della driver TCP del nodo server

Il cliente IP che invia le cose al successivo (mittente /destinatario)

Ma anche si mettono in gioco molte **altre relazioni C/S** con **consistenza e replicazione**

- **Nodi estremi** che fanno cache
- **Nodi intermedi o proxy** che possono fare cache per molti nodi server
- **Altri intermediari** di organizzazione e verifiche di freschezza dei dati



# CLIENTE / SERVITORE

Un modello a due entità: il **CLIENTE** chiede, il **SERVER** risponde

**Client** richiede il servizio e **Server** offre il servizio

*il cliente che invoca il servizio e aspetta il completamento del servizio*

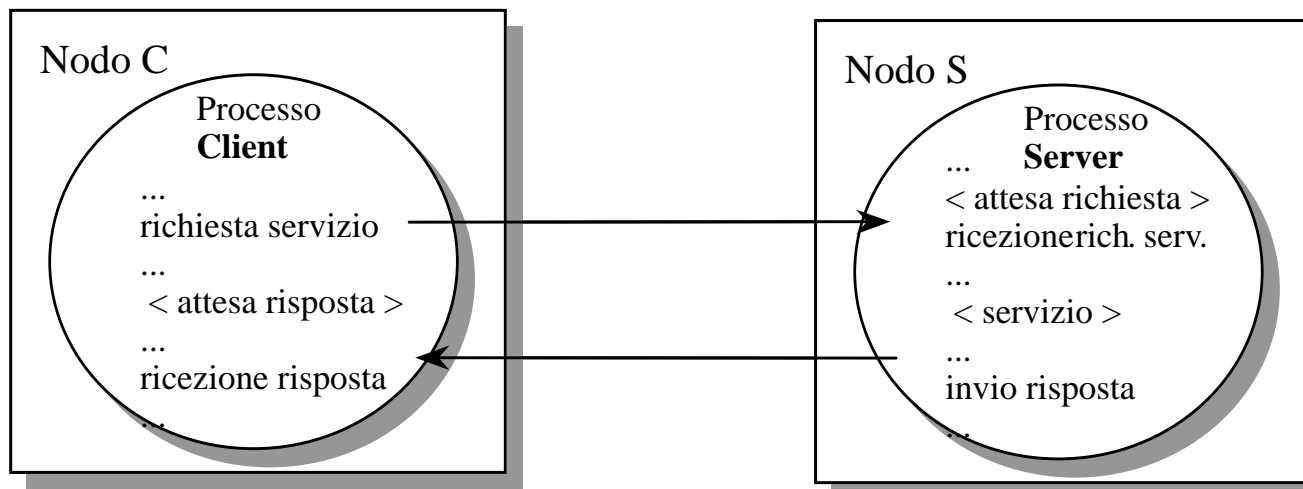
*il servitore attende richieste e le riceve, le realizza e le attua, poi risponde*

**MODELLO SINCRONO** c'è risposta (**semantica**)

**BLOCCANTE** attesa della risposta (**decisione locale**)

il modello C/S risolve il problema del rendez-vous (per **sincronizzare i processi comunicanti**) con Server come processo sempre in attesa di richieste di servizio

Il supporto non deve attivare il processo S alla ricezione di un messaggio



# MODELLO CLIENTE / SERVITORE

Il modello di comunicazione è **1 a molti (1 SERVER e N CLIENTI)**

**Modello molti:1, sincrono, dinamico, asimmetrico**

**ASIMMETRICO e Dinamico**

*il cliente conosce il servitore, solo al momento della invocazione*

*Il servitore non conosce a priori i clienti possibili*

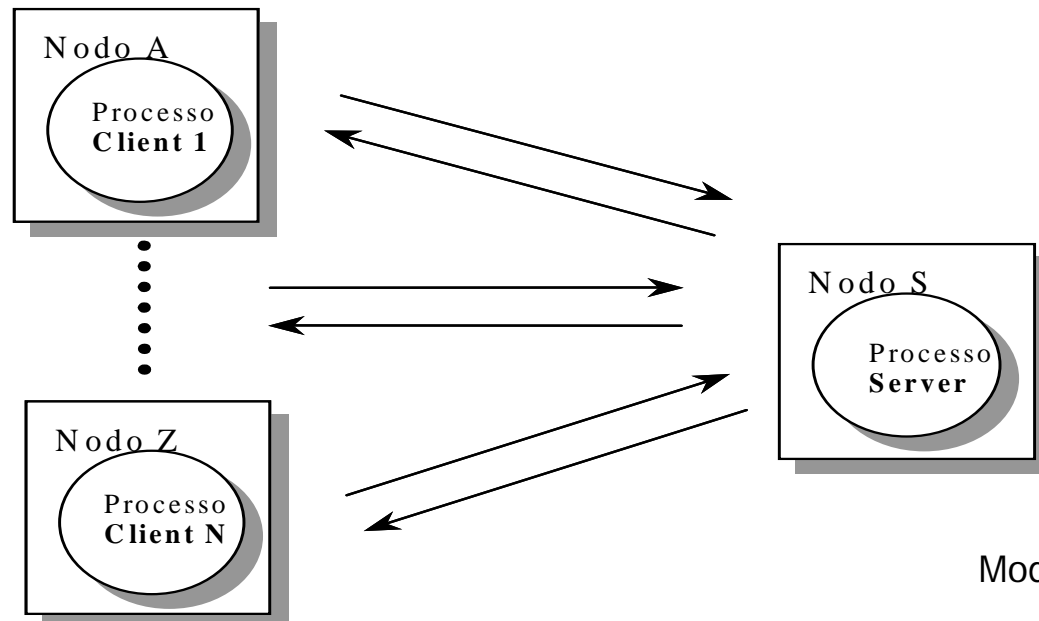
**DINAMICO** legame (binding) tra cliente e servitore è dinamico

**SINCRONO** **si prevede risposta dal servitore al cliente**

*Il supporto non fa azioni particolari, solo favorisce la comunicazione*

In caso il server non sia attivo, il supporto invia una indicazione di errore al cliente

*In tutti gli altri casi  
il cliente aspetta  
la risposta*





# PROGETTO CLIENTE / SERVITORE

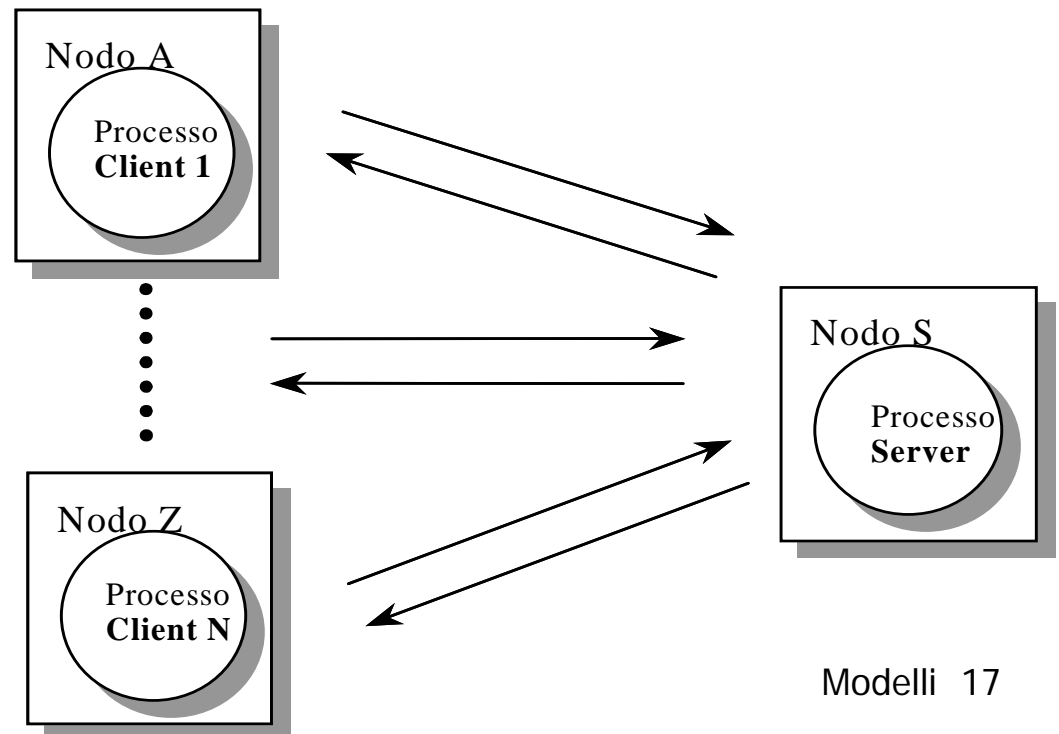
Il progetto del **server** è più **complesso** rispetto al progetto del **cliente**

il **Server**, oltre alla **comunicazione**, deve accedere alle **risorse del sistema** considerando **molteplici clienti** e anche problemi di:

- integrità dei dati
- accessi concorrenti
- autenticazione utenti
- autorizzazione all'accesso
- privacy delle informazioni

Inoltre

**il servizio deve essere sempre pronto alle eventuali richieste**



# C / S CLIENTE e OLTRE

---

Ragioniamo per una generica **interazione C / S**

**il Client** deve fare la richiesta e aspettare

Se arriva risposta ok

se eccezione, azione compensativa

se non arriva risposta?

- Non si aspetta per sempre: **timeout**, poi eccezione locale
- Richiesta ad un altro server ...
- Ripetizione delle richiesta: dopo quanto tempo, quante volte?

**In caso non si riesca, si rinuncia ⇒ il server è guasto (?)**

Il server potrebbe anche essere **lento e congestionato** nel servizio da richieste precedenti

**Il client aspetta fino ad una risposta (se ne ha bisogno)**

Ripetutamente chiede lo stesso servizio ma non attende per ogni richiesta, fino ad una risposta in tempi accettabili (**polling di ripetizioni**)

## C / S: IL SERVITORE ...

---

Per la stessa **interazione C / S**

**il Server** deve aspettare le richieste che sono messe in coda  
prende una richiesta e la serve, dà risposta  
passa alla richiesta successiva ...

**Ciclo** di lavoro **sequenziale** molto semplificante

Se si sono stabiliti altri protocolli, come nel caso precedente

- **Invio di risultato successivo**: dopo il timeout, il supporto assorbe
- **Ripetizione delle richieste**: il server deve riconoscere le richieste

**Il server deve riconoscere una stessa richiesta ⇒ e fare un solo servizio e fornire una sola risposta**

Coordinamento facilitato dalla coda di richieste e da un supporto dello stato delle richieste

*Il client deve identificare in modo unico le richieste*

Il server dopo avere fatto la operazione e prodotto il risultato deve mantenerlo fino alla consegna richiesta dal cliente specifico

## C / S: IL CLIENTE ...

---

Per questo tipo di interazione C / S oltre il default sincrono  
**Modelli verso la asincronia (senza risposta)**

Si parla di **modello di interazione pull**

***Il cliente ha sempre la iniziativa***

**Si semplifica il progetto server e il cliente decide**

Si può anche pensare ad un modello opposto per la consegna del risultato **modello di interazione push**

**Il cliente fa la richiesta, una volta, si sblocca e può fare altro**

**Il server arriva a fare il servizio e ha la responsabilità di consegna del risultato al cliente**

**Il modello Push** fa diventare il server cliente di ogni cliente  
scaricando il cliente (*senza cicli attivi di richieste*)

**Ma** carica di ulteriori compiti il servitore

# FORME CLIENTE / SERVITORE

---

Sono possibili molte forme di **interazione C / S** appena cominciamo ad esaminare sistemi reali

## **Modello di interazione pull o push**

**per arrivare ad una interazione flessibile e adatta ai casi che possono servire nei diversi possibili usi**

Il **polling** è una sequenza di richieste cliente/servitore

Il **modello push** comporta un rapporto C/S dal cliente al servitore e un successivo rapporto C/S dal servitore al cliente

**Modello di interazione push** è molto usato per ampliare la fascia di utenza

Un utente

- registra una richiesta di interesse per un **feed RSS** (RDF Site Summary) che si incarica della emissione di notizie ai registrati e
- poi riceve ogni nuova notizia su iniziativa del servitore

# PUBLISH / SUBSCRIBE

## Modelli molti a molti

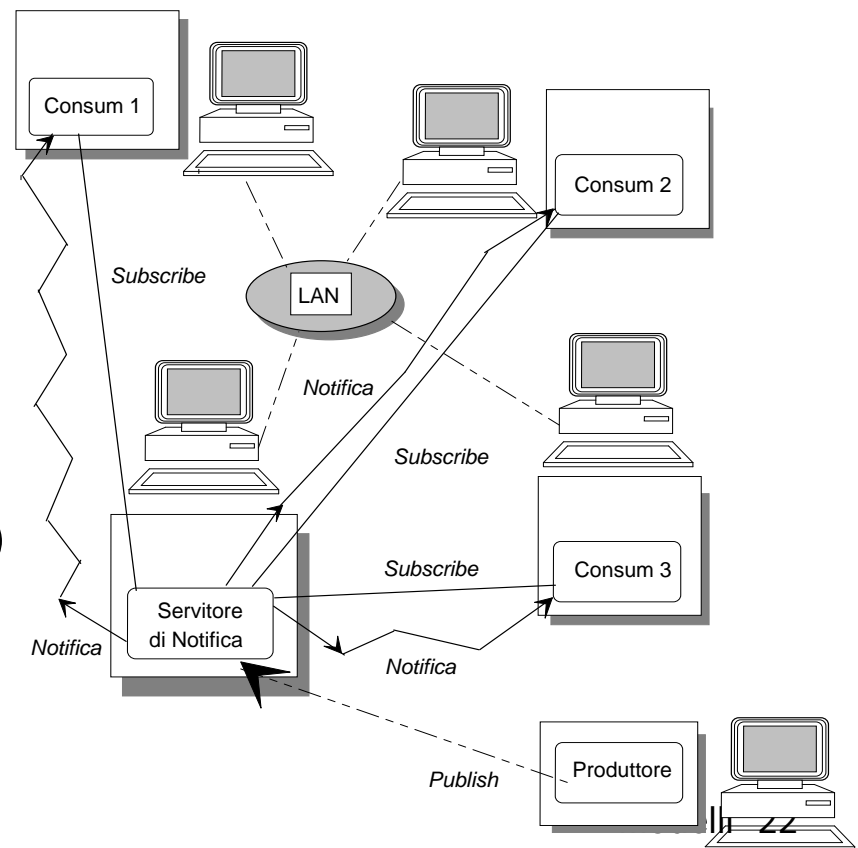
in caso di iniziativa mista, è necessario che gli interessati manifestino interesse alle informazioni attraverso **una registrazione (subscribe)** e poi si ricevano tutte le informazioni generate (**publish**)

**Modello publish / subscribe**  
per il **push con tre entità**

Da una parte gli **utenti**  
(**consumatori**)

che si registrano come interesse  
(**subscribe**)

Dall'altra, un **gestore** (*servitore di notifica*)  
registra interessi, riceve  
gli eventi generati (da un **produttore**)  
che fa **publish** e **notifica** gli **eventi**  
ai consumatori sottoscrittori



# MODELLI a DELEGAZIONE

Possibilità di **delegare** una funzionalità ad una entità che **opera al posto** del responsabile e lo libera di un compito

Entità **PROXY, DELEGATE, AGENTI, ATTORI**

che svolgono una funzione al posto di qualcun altro

*Un cliente lascia una altra entità ad aspettare una risposta ad una operazione fatta ad un server lento*

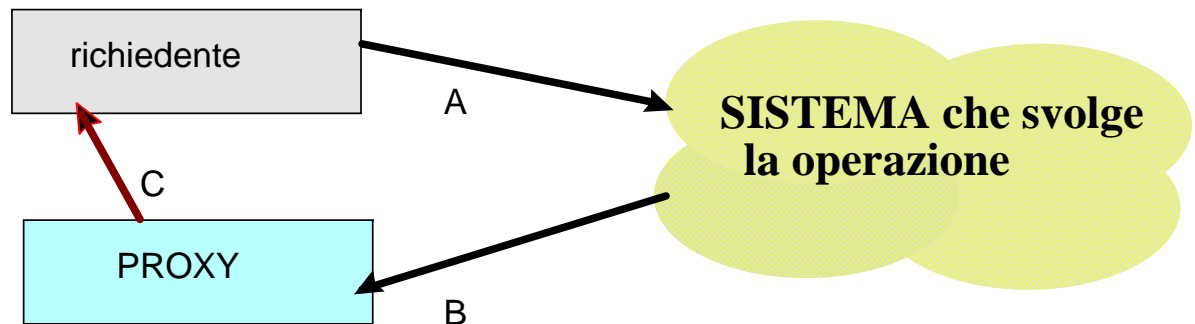
*Il proxy lavora in modo push per fornire la risposta al cliente stesso*

La notifica avviene usando **eventi** come nel framework

Gli **eventi** si interpongono tra utenti e basso livello e permettono una gestione applicativa del sistema isolando i dettagli

**Vedi molti sistemi a finestre grafiche**

invia richiesta, con delega a un proxy



il proxy invia la risposta al richiedente ad esempio usando eventi locali

# MODELLO a FRAMEWORK

**Modello diverso rispetto a C/S di richieste sincrone a kernel**

Il Framework **tende a rovesciare il controllo** (*per eventi di sistema*)

*Il processo utente non aspetta, ma registra con una propria funzione*

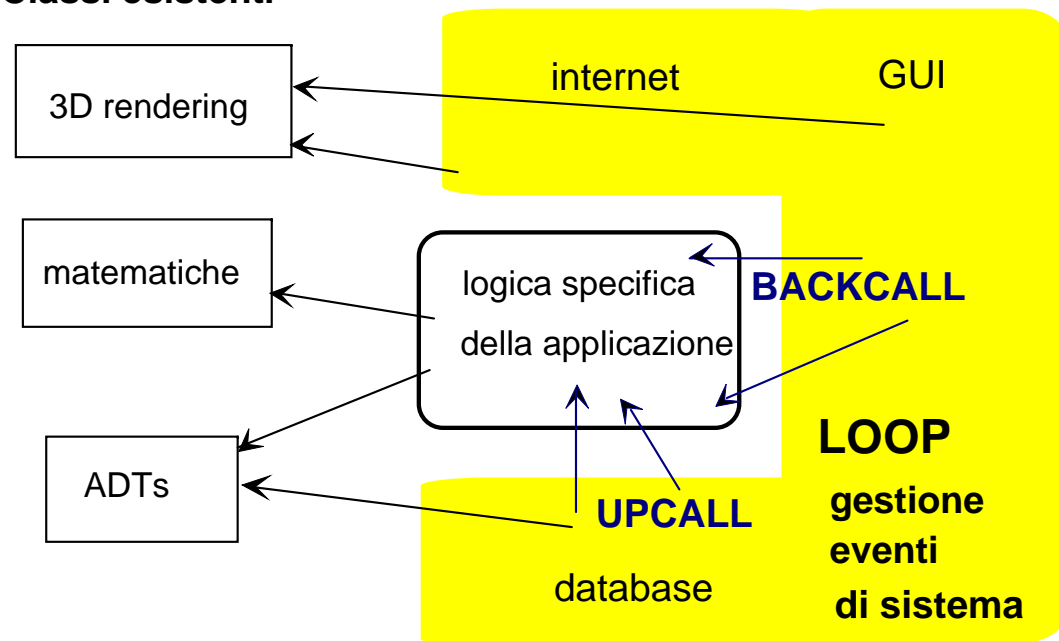
Esempio: **Windows che prevede per i processi un loop di attesa di eventi da smistare ai richiedenti**

*All'arrivo di un risultato questo viene portato al processo significativo*

Le risposte dal framework al processo utente sono dette **backcall** o **upcall** (push del framework)

Sono assimilabili a **eventi asincroni** generati dal supporto che le applicazioni devono gestire

Classi esistenti



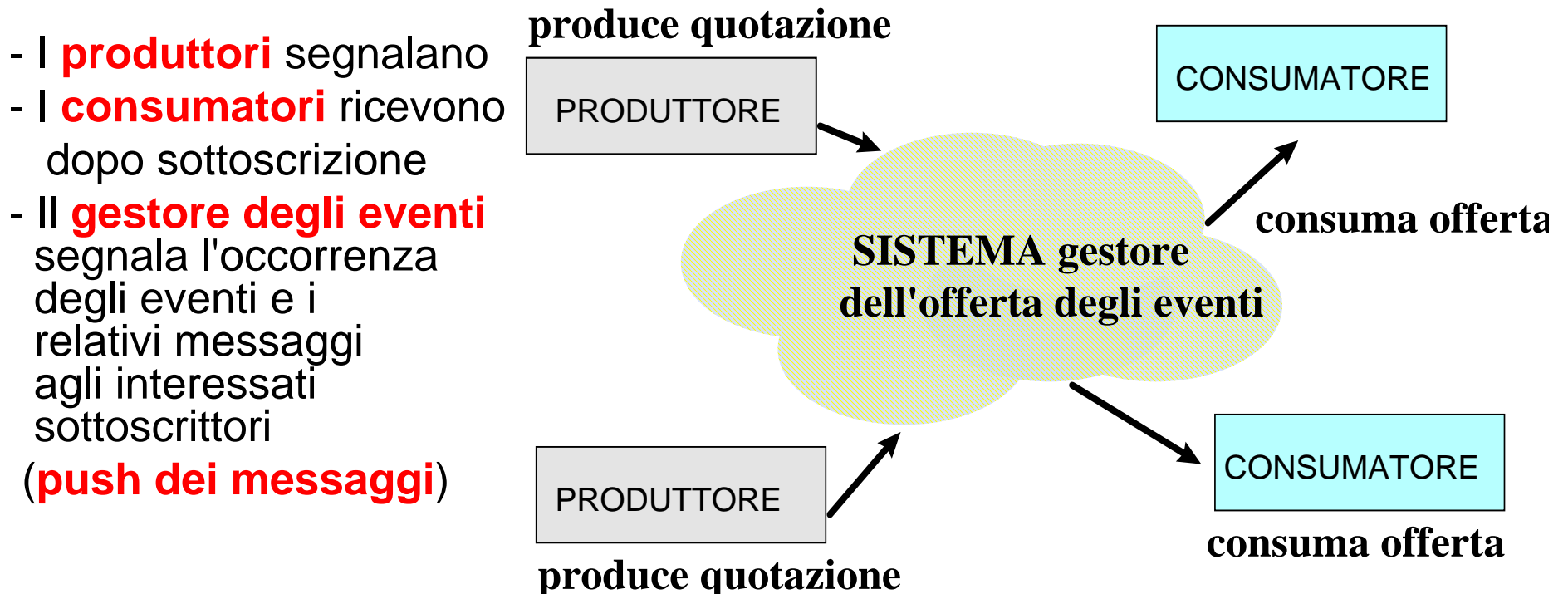
Funzioni e servizi



# MODELLO ad EVENTI

**Modello diverso dal C/S per molti aspetti (MOLTI A MOLTI)**

Il modello ad eventi è piuttosto **asincrono** e prevede di avere **molti** produttori, **molti** consumatori, e ad assumere un **sistema di supporto** **si gestisce l'invio di messaggi** disaccoppiando gli interessati



## ANCORA C / S: CONNESSIONE

---

Nella **interazione C / S**, si considerano due tipi principali riguardo all'insieme delle richieste

- **interazione connection-oriented (con connessione)**  
si stabilisce un **canale di comunicazione virtuale** prima di iniziare lo scambio dei dati (es. connessione telefonica)
- **interazione connectionless (senza connessione)**  
**senza connessione virtuale**, ma semplice scambio di messaggi isolati tra loro (es. il sistema postale)

La scelta tra le due forme dipende dal **tipo di applicazione** e anche da vincoli imposti dal **livello di comunicazione** sottostante

Per esempio, in Internet il **livello di trasporto** prevede i protocolli TCP e UDP basati su IP (tipicamente *connectionless* e *best effort*)

- **UDP senza connessione**, non reliable e non preserva ordine messaggi
- **TCP con connessione**, reliable (affidabile) e preserva l'ordine di invio dei messaggi e a maggiore affidabilità

# INTERCONNESSIONE

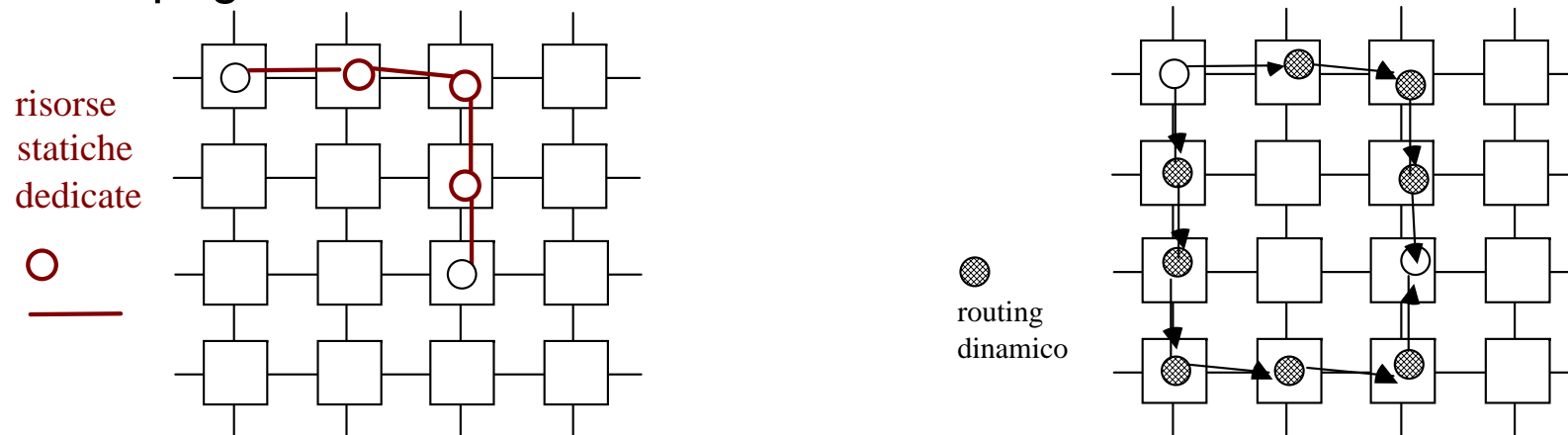
Nella **interazione C / S**, è anche importante come si trasferiscono i messaggi parte della sequenza di comunicazione

- **connessione**

Tutti i messaggi seguono la stessa strada (route) per la coppia mittente destinatario **decise staticamente** e **impegnano risorse intermedie**

- **senza connessione**

I messaggi possono seguire strada diverse **decise dinamicamente** e **non impegnano staticamente risorse intermedie**



Alcuni modelli a connessione (**TCP basato su IP**), non impegnano risorse intermedie ma solo sul mittente / destinatario

# COMUNICAZIONE: VISIBILITÀ

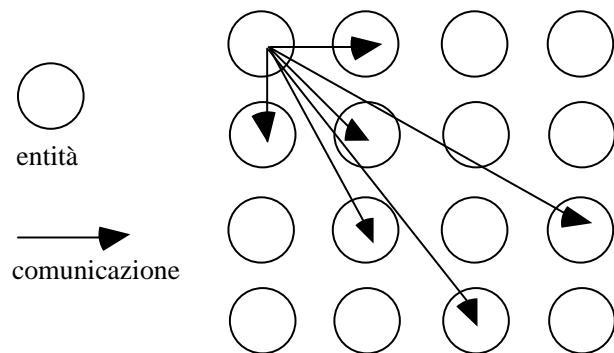
Nella **comunicazione**, è anche importante se si possa essere in visibilità di tutti i potenziali partecipanti (**scalabilità**)

**concetto di località** (limiti alla comunicazione)  
**vs.** **globalità** (nessun vincolo)

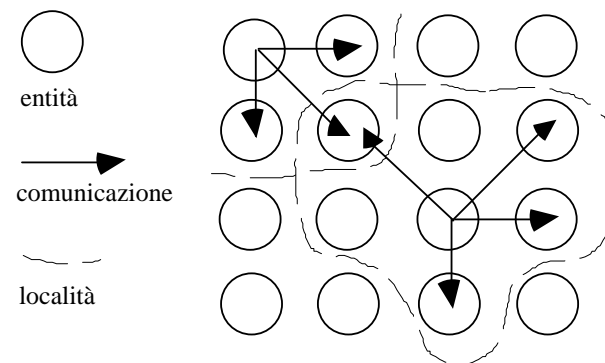
**modelli globali** non impongono restrizioni alle interazioni ⇒  
operazioni non scalabili dipendenti dal diametro del sistema

**modelli locali** (o ristretti) prevedono limiti alla interazione ⇒  
operazioni (forse) scalabili poco dipendenti dal diametro del sistema

## Modelli globali



## Modelli locali



**Si va verso la località, i vincoli, i domini, per ottenere scalabilità**

# CLIENTE / SERVITORE: STATO

---

Nella interazione C / S, un aspetto centrale è lo stato della interazione (o meno) vs. auto contenimento delle richieste

**stateless** non si tiene traccia dello stato, ogni messaggio è completamente indipendente dagli altri e auto contenuto

**stateful**, si mantiene lo stato dell'interazione tra i due interagenti e un messaggio con la operazione conseguente può dipendere da quelli precedenti

**Lo stato dell'interazione usualmente memorizzato nel Server**  
(che può essere **stateless** o **stateful**)

Un **Server stateful** garantisce **efficienza** (dimensioni messaggi più contenute e migliore velocità di risposta del Server)

Un **Server stateless** è **più leggero** e **più affidabile** in presenza di malfunzionamenti (soprattutto causati dalla rete)

# STATO sul SERVITORE

---

In caso di **server stateful**, il Server deve potere identificare il Client e tenerne traccia per interazioni future

Per esempio, ci sono notevoli differenze tra un file server **stateful** e **stateless**

```
key=open(filename, intentions);
```

```
rc=read(key, buffer, howmany);
```

```
rc=write(key, buffer, howmany);
```

La ripetizione di operazione fa avanzare l'I/O pointer

```
rc=read(filename, from, buffer, howmany);
```

```
rc=write(filename, from, buffer, howmany);
```

Ogni operazione è autocontenuta e specifica tutto, in questo caso il primo byte oggetto dell'operazione a partire dall'inizio del file (**from**) (NOTA: il file system NFS di SUN è stateless)

Le operazioni devono essere **controllate dal cliente** che mantiene lo **stato di ogni file** a cui accedere e **la storia**

# SERVITORE STATEFUL

---

Un **server stateful** deve sicuramente presentare un **impegno di risorse ulteriore**

il Server deve potere identificare la **sessione del Client** e tenerne **traccia per interazioni future**: ad esempio può più facilmente riconoscere ed autorizzare utenti e operazioni

L'impegno può anche durare per molto tempo e potrebbe crescere con l'accumulo di molte (troppe) altre richieste

**key=open(filename, intentions);**

L'impegno inizia alla open e registra la sessione

**ok=close(key);**

Fino ad una chiusura che potrebbe non arrivare

Il vantaggio è di avere **minore costo delle operazioni in termini di banda impegnata, sicurezza, e garanzie di ripristino**, in caso di problemi dei clienti

# STATO INTERAZIONE e IDEMPOTENZA

---

I **modelli stateless** portano a un progetto del **cliente più complesso**, ma semplificano il **progetto del server**

I modelli di **interazione stateful** tendono a richiedere **al server di mantenere lo stato della interazione**

*Si pensi allo stato mantenuto sul server per ogni file aperto*

La scelta tra server **stateless** o **stateful** deriva dall'applicazione

Un'interazione **stateless** è sensata e possibile (viabile) SOLO se il protocollo è progettato per operazioni **idempotenti**

Ogni richiesta potrebbe non arrivare, o arrivare fuori ordine o arrivare ripetuta

**Operazioni idempotenti tendono a produrre lo stesso risultato, anche se ripetute**

Per esempio, un Server fornisce sempre la stessa risposta ad un messaggio M indipendentemente da altri messaggi (anche M) ricevuti dal Server stesso

Ovviamente, a meno che lo stato delle risorse corrispondenti non sia variato



# STATO INTERAZIONE

---

I **modelli con stato** hanno il **server** che può / deve mantenere traccia della interazione

*Per quanto tempo e con che costi?*

Si distingue lo stato in base alla durata massima:

- **Stato permanente** mantenuto per sempre
- **Stato soft o a tempo** che rimane per un tempo massimo

Si pensi ad un server web che deve **mantenere le risorse** per reggere tutte le richieste dei clienti che hanno acceduto (sessioni in atto)

Si pensi ad un server che deve riconoscere tutti i clienti che sono **autorizzati** ad accedere (username e password) tramite tabelle di riconoscimento

**I modelli con stato (sul server) hanno un costo in risorse richieste ed una complessità (di progetto del server) superiore rispetto ai modelli senza stato**

La complessità altrimenti è ripartita sui clienti e sulla interazione che deve specificare anche tutte le informazioni relative allo stato stesso

```
rc=read(filename, from, buffer, howmany);
```

Ogni operazione deve dare tutte le informazioni

# PROGETTO SERVITORE

---

Una proprietà che caratterizza il **server** è la **concorrenza** delle azioni, cioè la *possibilità di portare avanti operazioni in parallelo*

Il server è tipicamente un processo ma la concorrenza può migliorare le prestazioni

si pensi ad un Web server che deve rispondere a moltissime richieste contemporaneamente

Un **Server iterativo** o **sequenziale** processa le richieste di servizio una alla volta, e mette in coda di attesa le altre

possibile basso utilizzo delle risorse, in quanto non c'è sovrapposizione tra elaborazione ed I/O

Un **Server concorrente** può gestire molte richieste di servizio insieme (concorrentemente anche se non in parallelo), cioè accettare una richiesta anche prima del termine di quella in corso di servizio  
migliori prestazioni ottenute da sovrapposizione elaborazione ed I/O ma maggiore complessità progettuale

# PROGETTO SERVITORE

Possiamo avere molti diversi schemi di cooperazione per servizio che possiamo classificare in base a **diverse proprietà**

## SERVIZIO

**sequenziale/iterativo  
concorrente**

**con stato interazione  
senza stato sul server**

**con connessione  
senza connessione**

		Tipo di comunicazione	
		connessione	senza connessione
S E R V E R	sequenziale iterativo		
	concorrente singolo processo		
	concorrente multi processo		

La scelta del tipo di Server dipende dalle caratteristiche del servizio da fornire

**Lo stato ha effetto sul protocollo e sulle entità**

# PROPRIETÀ del SERVITORE

---

Nel progetto di una interazione, scegliamo in base a caratteristiche tecnologiche, **ad esempio il sistema operativo di supporto**, e anche del **protocollo** che vogliamo realizzare e dei **vincoli di costo**

*In un ambiente Unix*, la generazione di un **processo pesante** (fork) è facile e semplificata per la condivisione, si possono utilizzare **servitori multiprocesso** per avere la **concorrenza**

Un server Web per Unix tende a generare un processo per ogni richiesta di servizio e deve farlo in modo efficiente

*In un ambiente Java*, la generazione di un **processo leggero** (thread) è facile e semplificata per la condivisione, si possono utilizzare **servitori multiprocesso** per avere la **concorrenza**

Un server Web per ambienti Java tende a generare un thread per ogni richiesta di servizio e lo fa in modo efficiente per il basso costo del supporto dei thread (pure a livello applicativo)

# MODELLI di SERVIZIO

---

## Servitore sequenziale o iterativo

si possono introdurre ritardi se la coda di richieste cresce

## Servitore concorrente

capacità di servire richieste insieme sfruttando tempi morti

## Servitore senza stato sul server

il servitore dimentica le richieste appena le ha eseguite

## Servitore con stato interazione

il servitore deve tenere traccia della interazione con i clienti

## Servitore senza connessione

ogni richiesta arriva in modo indipendente (fuori ordine)

## Servitore con connessione

le richieste arrivano in ordine di emissione del cliente

# SERVITORE SEQUENZIALE

---

## Servitore iterativo

che serve una richiesta alla volta e **itera** il servizio

Dal punto di vista cliente abbiamo **due indicatori distinti**

- **tempo di elaborazione** (di servizio) di una richiesta

$T_S$  tempo per servizio di una richiesta isolata

- **tempo di risposta** osservato dal cliente  $T_R$

$T_R$  ritardo totale tra la spedizione della richiesta e l'arrivo della risposta dal server

$$T_R = T_S + 2 T_C + T_Q$$

$T_C$  tempo di comunicazione medio

$T_Q$  tempo di accodamento medio

Con lunghe code di richieste, il tempo di risposta può diventare anche molto maggiore del tempo di elaborazione della richiesta

# SERVITORE ITERATIVO

## Servitore iterativo

che serve una richiesta alla volta e **accoda la altre in attesa**

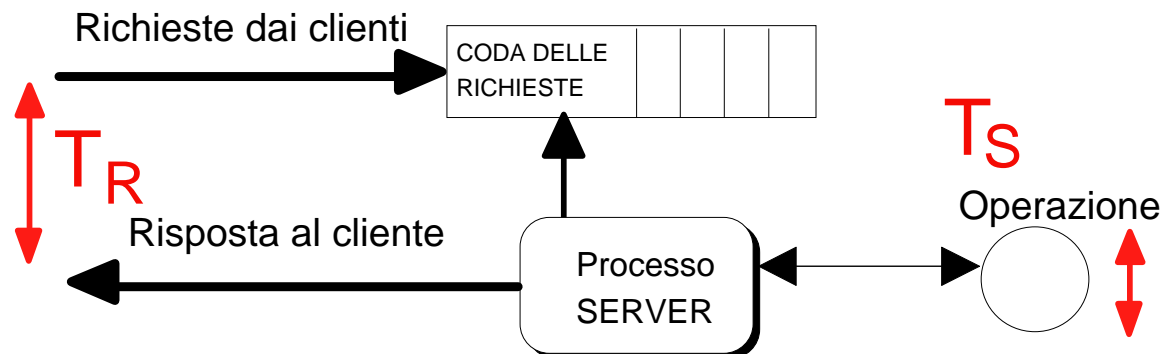
Tralasciando il tempo di comunicazione, se  $N$  è la lunghezza media della coda, l'attesa media è  $N/2 * T_s$  e

$$T_R (\text{medio}) = (N/2+1) * T_s$$

Soluzioni per limitare l'overhead

**limitare la lunghezza della coda (sveltendo il servizio)** e

**rifiutare le richieste a coda piena (rifiutando servizio)**



# SERVITORE CONCORRENTE

---

## Servitore con più servizi attivi insieme

che serve molte richieste insieme e può ottimizzare l'uso della risorsa processore eliminando i tempi di idle

$$T_R = T_S + 2 T_C + T_Q + T_I + T_g$$

$T_C$  comunicazione e  $T_Q$  accodamento (*trascurabili*)

$T_I$  tempo di *interleaving* (può *sottrarre tempo*)

$T_g$  tempo di *generazione del processo* (*dipende dalla tecnologia*)

La **concorrenza** può produrre significative riduzioni del  $T_R$

- se la risposta richiede un tempo di attesa significativo di I/O o di sospensione del servizio con possibilità di interleaving
- se le richieste richiedono tempi di elaborazione molto variabili
- se il server è eseguito in un multiprocessore, cioè con servizi in reale parallelismo

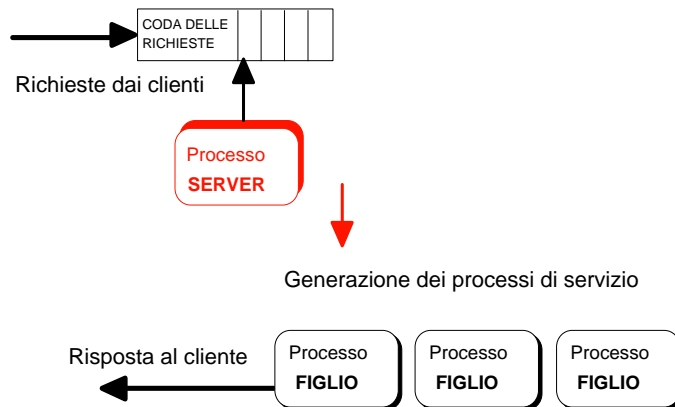


# SERVITORE CONCORRENTE

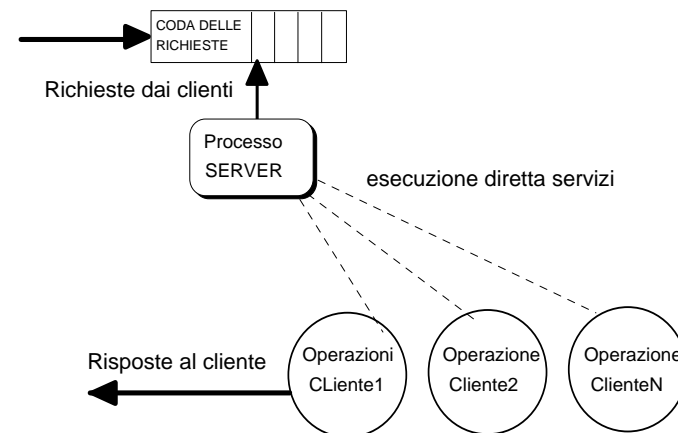
Il progetto di un servitore concorrente può seguire diversi schemi

**Servitore concorrente multiprocesso:** un processo server si occupa della coda delle richieste e genera processi figli, uno per ogni servizio

**Servitore concorrente monoprocesso:** un unico processo server si divide tra il servizio della coda delle richieste e le operazioni vere e proprie



**servitore concorrente  
multiprocesso**



**servitore concorrente  
singolo processo**

# PROCESSI e OGGETTI

---

**Processi e oggetti** ⇔ vanno d'accordo?

**sono entità spesso separabili e ortogonali (a volte)**

**Esistono entrambi durante la esecuzione di una applicazione?**

**Considerare i modelli di esecuzione noti**

## UNIX

i **processi** sono attività esistono durante la esecuzione (barriera di visibilità) e che possono accedere ad oggetti privati solamente

## Java

gli **oggetti** sono presenti come risorse di memoria da riferire per il loro tempo di vita (come le classi)

i **processi (thread)** sono attività ed oggetti con vincoli specifici e capaci di eseguire sugli oggetti caricati in memoria

# PROGETTO di C/S

---

**Progetto dei Clienti** ⇒ più semplice

Spesso sono sequenziali, potrebbero essere anche concorrenti

**Progetto dei Servitori** ⇒ più complesso per la operazione svolta dal server stesso e dai molteplici clienti (**molti a 1**)

*Ma anche...* Un server deve essere presente al momento delle richieste dei clienti ⇒ **deve essere sempre presente**

La garanzia attraverso server come processi eterni, **processi demoni** sulle macchine server che sopravvivono alla durata delle **singole applicazioni** per vivere per tutta la durata del sistema

Daemon tipici processi server che eseguono un **ciclo infinito di attesa** di richieste ed esecuzione delle stesse a volte **sequenziali**, ma spesso **concorrenti**

Java riconosce **thread daemon** rispetto a thread user e li esegue per tutta la durata di una sessione della virtual machine (JVM), terminandone l'esecuzione solo quando termina l'ultimo user thread

# PROGETTO dei COMPONENTI C/S

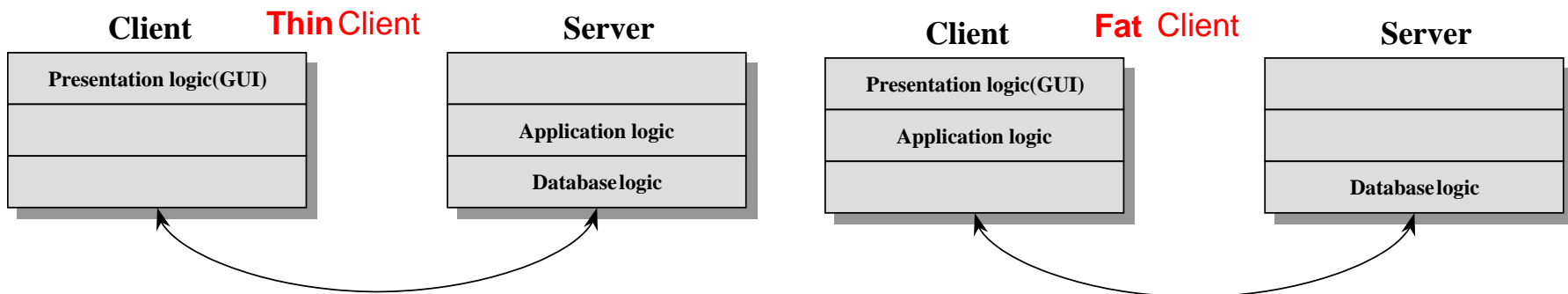
## Separazione delle parti di un servizio tra C /S

la logica applicativa si distribuisce tra i Client e i Server in base a molte considerazioni di progetto

In genere il **server prevede un progetto intrinsecamente più complesso** ma il **client** deve spesso essere più snello per ragioni di vincoli tecnologici e di limiti di costo di accesso

Molte considerazioni guidano le scelte: configurazione del sistema, carico sul server, traffico di rete

**Fat Client** (più pesante) vs. **Thin Client** (più leggero)



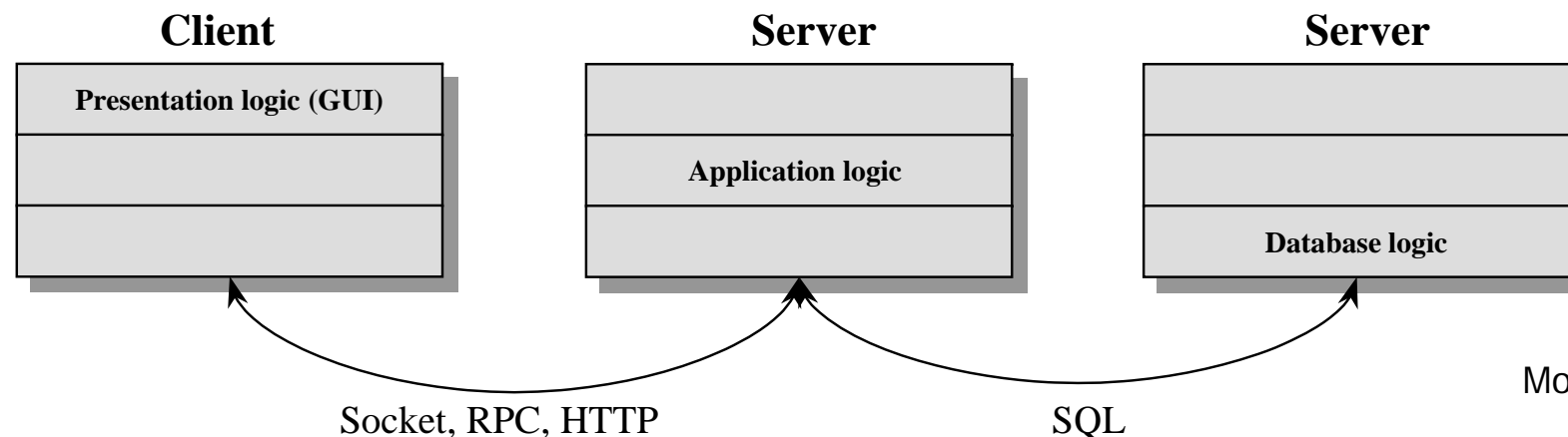
# COMPONENTI LOGICI in C/S

**Fat Client** carica la parte cliente di alcuni doveri

**Thin Client** il cliente rimane snello e a risorse limitate

Si può anche andare oltre con la decomposizione, giocando su più livelli a parte gli estremi o su più tier (**Sistemi multi-tier**)

Un sistema C/S può essere anche decomposto in parti (**3-tier C/S**), per distribuire il carico di lavoro di servizio su diverse macchine che si incaricano di svolgere funzioni molto diverse, e sono separate per superare problemi di *sbilanciamento di carico*, di *limiti di esecuzione*, di *disponibilità e tolleranza ai guasti*, di *vicinanza geografica*



# MODELLO di COMUNICAZIONE in C/S

---

## Schema di base C/S: asimmetrico, sincrono, bloccante

I servizi forniti da un servitore che risponde a diversi clienti (molti:1) che conoscono il servitore e che non sono noti al servitore

**MODELLO**      interazione sincrona (default)

ma anche      **asincrona / non bloccante**

**asincrona**       $\Rightarrow$       nessun (interesse per il) risultato

**non bloccante**  $\Rightarrow$       non interessa aspettare risultato

**Molte variazioni sul MODELLO oltre il default**

anche      a time out, con push da parte del server

e anche considerando eventualmente più server possibili per portare a termine il servizio in modo da avere più possibilità

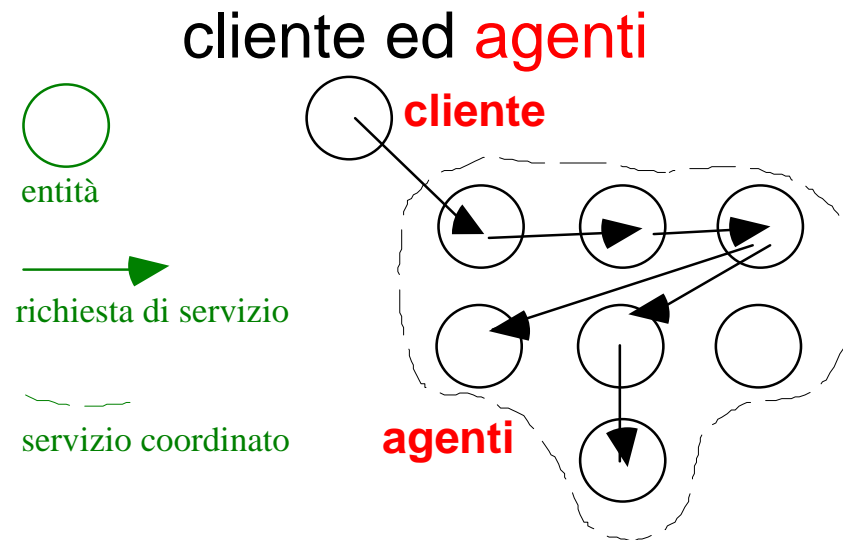
a default  $\Rightarrow$       interessa avere un'unica operazione e unico servizio

# MODELLO ad AGENTI MULTIPLI

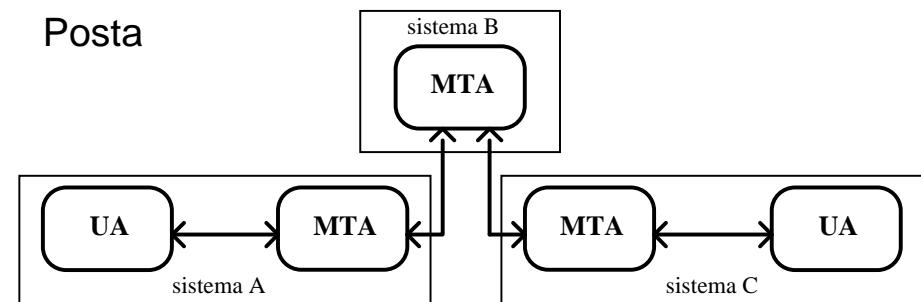
**Schema** in cui i servizi sono forniti dal coordinamento di più servitori detti **agenti che forniscono un servizio globale unico (modello ad almeno due tier o livelli)**

Gli **agenti** forniscono il servizio coordinato e possono:

- **partizionare le capacità di servizio**
  - **replicare le funzionalità di servizio**
- in modo **trasparente al cliente**



**agenti di posta (mail)**



# ARCHITETTURE A PIÙ LIVELLI

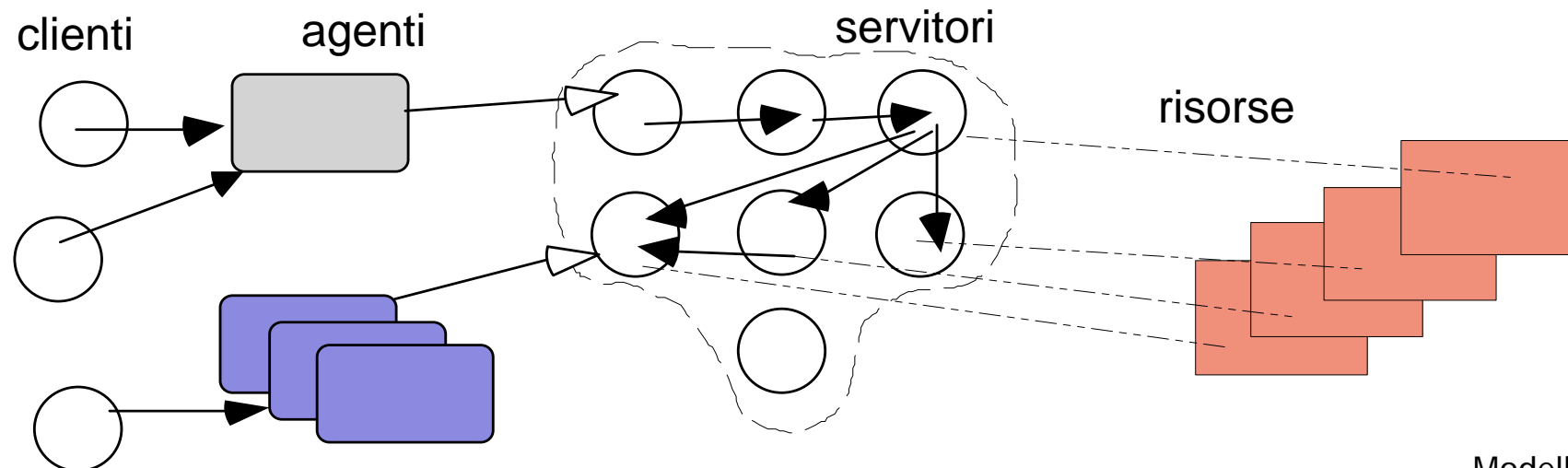
**MODELLI a LIVELLI MULTIPLI per la divisione dei compiti**

**Clienti** che interagiscono con Agenti

**Agenti** anche paralleli e capaci di coordinarsi

**Server** di operazione paralleli, replicati, e coordinati

Con necessità di **protocolli** di coordinamento, sincronizzazione, rilevazione e tolleranza ai guasti





# MODELLI E SISTEMI DI NOMI

---

## Conoscenza reciproche delle entità / servizi

nella relazione C/S, il **cliente** deve riferire il **servitore**  
Questo è reso possibile dal **nome del servitore** nel cliente

I clienti possono usare molte forme di nomi diversi

- indirizzoServizio 123456
- nomeGestore.nomeServitore 123:123456
- nomeNodo.nomeServizio 123:stampa
- nomeServitore prcs#1234
- nomeServizio stampa

## Nomi TRASPARENTI e NON TRASPARENTI alla allocazione

**Visibilità della allocazione** nel nome dei servizi ai nodi e degli indirizzi all'interno del nodo

*La trasparenza non lega il nome a dettagli di basso livello ☺*

# MODELLI DI NOMI

---

**I sistemi di nomi sono il primo supporto architetturale**

i **nomi**, ossia i **riferimenti ad altre entità**, sono distribuiti nel codice dei clienti, degli utilizzatori, delle librerie, ecc. e se ne deve garantire la consistenza

Come si qualificano i **nomi** e **quando** si risolvono i **riferimenti**?

## **BINDING STATICO vs. DINAMICO**

**statico:** i riferimenti risolti prima della esecuzione

**dinamico:** i riferimenti risolti solo al momento del bisogno

In caso di **sistemi concentrati**, **binding tipicamente statico** ma esistono e sono diffuse per riutilizzo anche librerie dinamiche

si risolve il tutto staticamente e non si necessita di un servizio di nomi vista la **invarianza dei nomi** e della **allocazione delle entità**

I nomi sono risolti **prima della esecuzione** e non è il caso di **cambiare alcuna allocazione** (altrimenti insorgono problemi)

# SISTEMI DI NOMI DISTRIBUITI

---

**In sistemi distribuiti, le risorse sono dinamiche e non prevedibili staticamente, i sistemi di nomi sono presenti durante l'esecuzione**

In caso dinamico, le entità **non sono staticamente** fissate

In caso dinamico, nasce la necessità di un **servizio di nomi (name server)** che mantiene e risolve i nomi e che fornisce il servizio durante la esecuzione coordinandosi con i gestori della allocazione

Tipicamente si usano delle **tabelle di allocazione** controllate da opportuni GESTORI dei NOMI

**Nomi non trasparenti**      dipendenti dalla locazione

**Nomi trasparenti**      **non dipendenti dalla locazione**

Se le entità cambiano allocazione, non cambiano i nomi

entità mobili - nomi invarianti ⇒ **Indipendenza dalla allocazione**

# SISTEMI DI NOMI in SISTEMI APERTI

Deve essere possibile **durante la esecuzione** inserire **nuove entità** del tutto compatibili con quelle già esistenti

**Tipicamente si devono introdurre molteplici gestori di nomi distinti e coordinarne la esecuzione (agenti)**

## Partizionamento dei gestori

ciascuno responsabile di una sola parte (partizione) dei riferimenti  
**località** (in generale i riferimenti più richiesti)

## Replicazione dei gestori

ciascuno responsabile con altri di una parte (partizione) dei riferimenti  
coordinamento

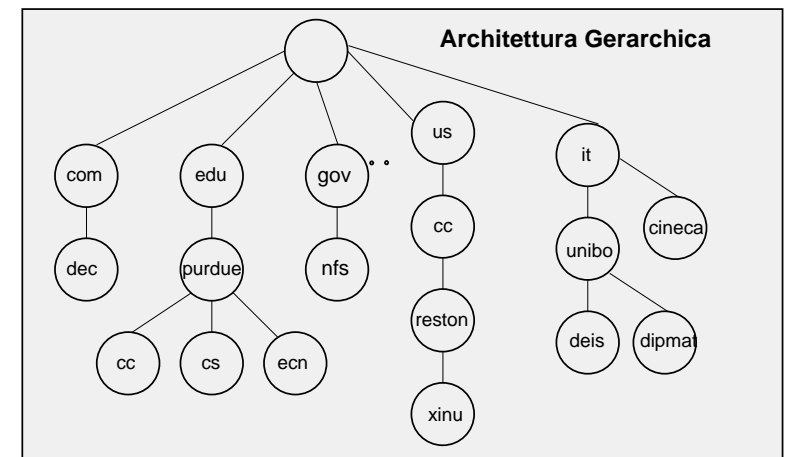
I gestori sono spesso organizzati a livelli

*gestore generale che coordina*

*gestori di più basso livello*

*in molti livelli con località informazioni*

(vedi DNS)



# DOMAIN NAME SYSTEM (DNS)

**DNS** come insieme di gestori di tabella di nomi logici e di indirizzi IP  
obiettivo  $\Rightarrow$  attuare corrispondenze tra **nomi logici host** e **indirizzi IP**

Primo passo – primi anni 80: uso di un file locale, /etc/hosts

Non sufficiente su scala globale (quante repliche?)

**DNS introduce nomi logici in una gerarchia (albero di DSN)**

intesa come gerarchia di domini logici e centri di servizio

la corrispondenza tra nomi logici e indirizzi fisici avviene dinamicamente  
tramite un servizio di nomi che risponde (dinamicamente) alle richieste

La traslazione è:

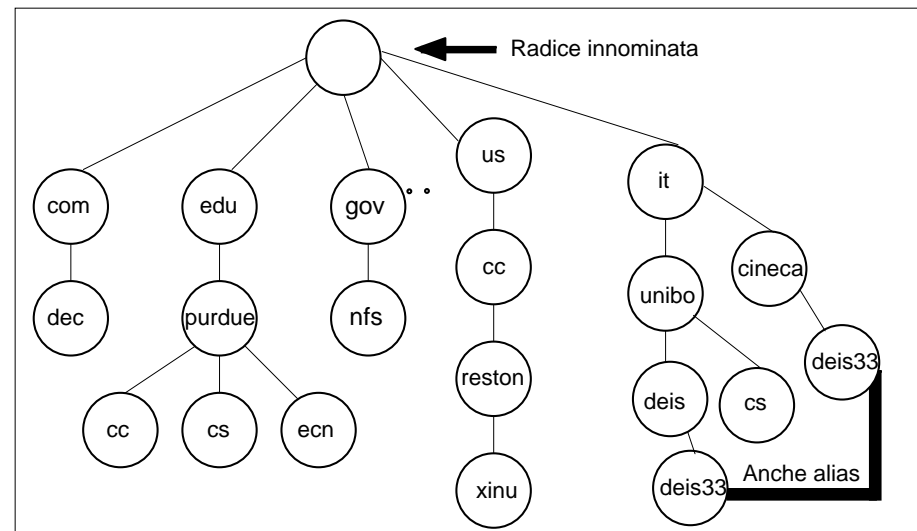
statica vs. **dinamica**

locale vs. **globale**

in una gestione globale

non centralizzata ma favorendo  
la località

Esempio di divisione dei compiti  
e coordinamento



# NOMI di DNS

---

**Ogni nome** è diviso in parti che rappresentano domini diversi (server)  
I domini di base (di primo livello) sono i seguenti:

Nome dominio	Significato
<b>COM</b>	Organizzazioni commerciali
<b>EDU</b>	Istituzioni per l'istruzione
<b>GOV</b>	Istituzioni governative
<b>MIL</b>	Gruppi militari
<b>NET</b>	Maggiori centri di supporto alla rete
<b>ORG</b>	Organizzazioni diverse dalle precedenti
<b>ARPA</b>	Dominio temporaneo dell'ARPANET (obsoleto)
<b>INT</b>	Organizzazioni internazionali (schema geografico)
<i>codice nazionale</i>	Ciascuna nazione (schema geografico)

Un nome logico: `deis33.deis.unibo.it`  
riferisce un *dominio nazionale italiano*

Questo nome è a quattro livelli, come numero di domini

Potremmo avere alias: `deis33.cineca.it`  
con un nome a tre livelli, come numero di domini

**Il numero dei livelli è dinamico**

# LIVELLI di DNS

---

Ogni nome permette dei mappaggi logici propri

`deis33.deis.unibo.it`

**country**

it = Italia,

**organisation**

unibo = Università di Bologna,

**department**

deis = Nome/Sigla Organizzazione locale,

**machine**

deis33 = nome della macchina,

Livello	Descrizione	Nome dominio	Sigle
minimo	locale	deis33.deis.unibo.it	deis33 = Host
intermedio2	sottogruppo	deis.unibo.it	deis = Department
intermedio1	gruppo	unibo.it	unibo = Organisation
massimo	postazione	it	it = Italy

Livello	Descrizione	Nome dominio	Sigle
minimo	locale	deis33.cineca.it	deis33 = macchina
intermedio	gruppo	cineca.it	cineca = gruppo
massimo	organizzazione	it	it = Italia

# NOMI di DNS

I singoli nomi sono case insensitive e al max 63 char per dominio

Il nome completo al max 255 char

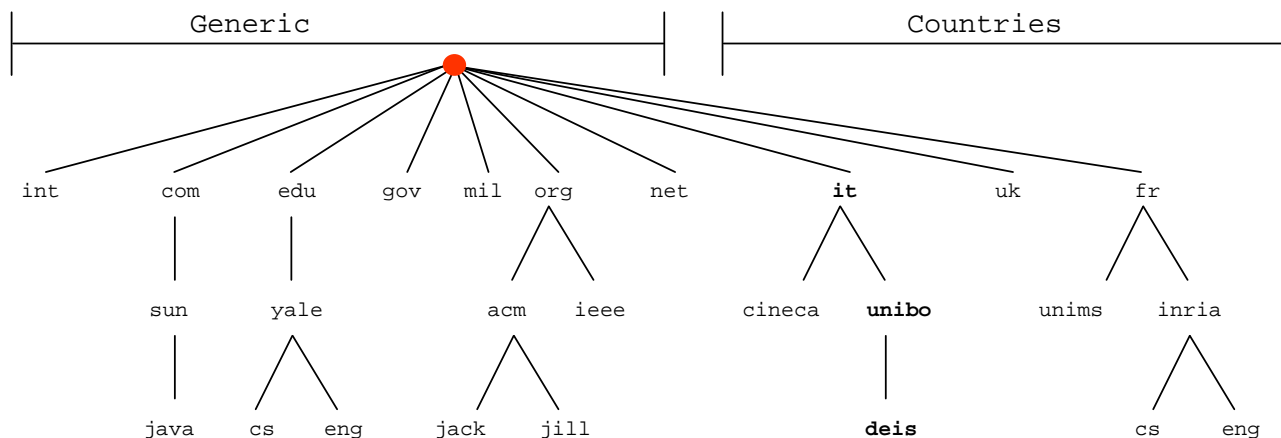
I domini sono del **tutto logici** e non sono collegati in nessun modo alle **reti fisiche** o alle **organizzazioni di rete** (logico vs. fisico)

Ogni dominio può essere usato in modo **relativo** o **assoluto**

Ogni dominio relativo fa riferimento al dominio che lo contiene  
deis.unibo.it

deis è interno a unibo, interno a it, che è interno alla root

## Possibile gerarchia





# GERARCHIA di SERVITORI di DNS

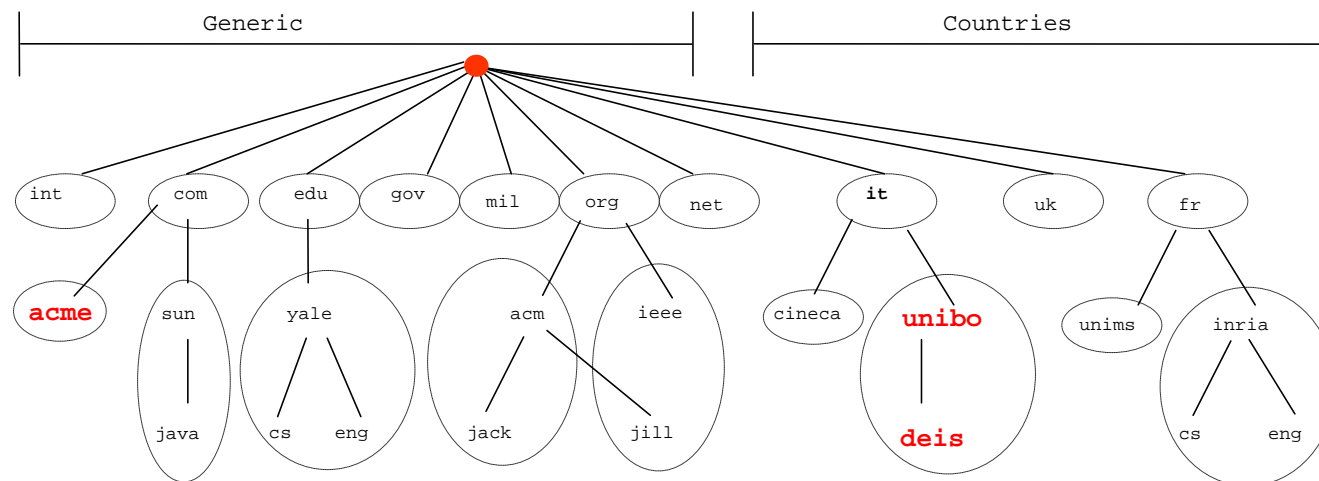
Ogni nome di dominio corrisponde ad un **server di responsabilità**

I domini sono organizzati su responsabilità primaria di un server (detto di ZONA)

La suddivisione in zone per ragioni geografiche o di organizzazione

Ogni zona riconosce una autorità ossia un server che fornisce le corrette corrispondenze

Si pensi ad una azienda **acme.com** che si colloca nella **gerarchia** e che gestisce una **zona di responsabilità**



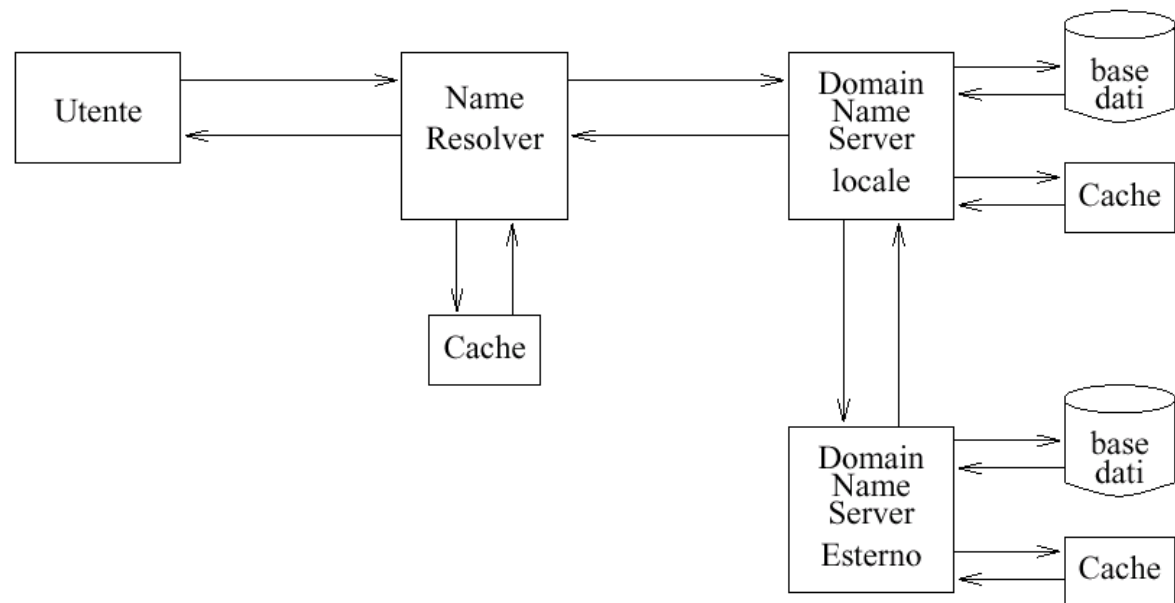
# DELEGA di DNS

I diversi servitori che gestiscono i domini suddivisi in zone d'autorità possono a loro volta delegare la gestione ad altri server

Un server di dominio può **delegare sottodomini** a **servitori sottostanti** (che se ne assumono responsabilità e autorità)

**Le richieste sono smistate dal sistema in modo opportuno**

Ogni richiesta di utente viene fatta al servizio di nomi in modo indiretto tramite un **agente specifico (name resolver)** per la gestione dei nomi della località



**Si noti il caching di corrispondenze**

# ARCHITETTURA del DNS

---

I diversi server DNS sono organizzati per ottenere

Diversi requisiti  $\Rightarrow$  **affidabilità, efficienza, località**

Ogni dominio ha **name resolver** e **domain name server** locali (*uno o più*) e usa estensivamente **cache** per conoscenze pregresse

Un cliente chiede un mappaggio al name resolver...

- Il **resolver** fornisce la risposta o perché conosce già la corrispondenza (**cache**) o la trova attraverso una richiesta C/S ad un name server
- I **DNS name server** sono organizzati **come agenti** per ottenere informazioni reciprocamente (dalla più corretta autorità) e potere fornire a loro volta risposte consultando le rispettive tabelle DNS
- La **organizzazione ad albero** consente di muoversi tra i DNS con le richieste necessarie fino a raggiungere il dominio di autorità
- I diversi server DNS **sono replicati** per fornire risposta anche in caso di problemi

# REPLICAZIONE del DNS

---

Ogni dominio corrisponde a più **Domain Name Server**, di cui uno ha autorità sulla traslazione degli indirizzi. Ogni **Domain Name Server** non ha una visione completa, ma solo locale (*la tabella locale di corrispondenza*)

## Necessità di replicazione

- I diversi server DNS **sono anche replicati** per fornire servizio anche a fronte di guasti di server della gerarchia
- In genere, ogni zona ha un ***primary master*** responsabile per i dati della intera zona, in più una serie di ***secondary master*** copie del primary, con consistenza garantita dal protocollo DNS (non massima)
- Il primario di una zona può diventare il backup (master secondario) di un'altra zona
- allo start up il secondario chiede i dati al primario e può fare fronte al traffico in caso di guasto
- Ad intervalli prefissati, i secondari chiedono le informazioni al primario (*modello pull*)

# PROTOCOLLI di DNS

---

## Efficienza su località

Per favorire le richieste a maggiore località, si mantengono i dati ottenuti in previsione di nuove richieste

Tutti i resolver ed i DNS server mantengono caching informazioni per ottimizzare i tempi di risposta al cliente

## Protocolli di richiesta e risposta per il name server

I clienti e resolver fanno richieste usando di protocollo UDP  
(comunicazione usando le porte 53)

Lo stesso tra diversi DNS tra di loro

In caso di messaggi troppo lunghi si usa TCP, eventualmente dopo una risposta di eccezione: uso di TCP per l'aggiornamento dei diversi secondari che devono leggere delle tabelle di responsabilità altrui

# RISOLUZIONE QUERY DNS

---

Il resolver conosce un server di dominio e attua le query locali

Il protocollo DNS prevede due tipi di query: **ricorsiva** e **iterativa**

La **ricorsiva** richiede che al cliente

- o si fornisca risposta (anche chiedendo ad altri)

- o si segnali errore (dominio non esistente, etc.)

**si prevede una catena di server request/reply**

La **iterativa** richiede che al cliente si fornisca

- o la risposta

- o il migliore suggerimento come riferimento al migliore DNS server

**si prevede una sequenza di richieste request/reply a server**

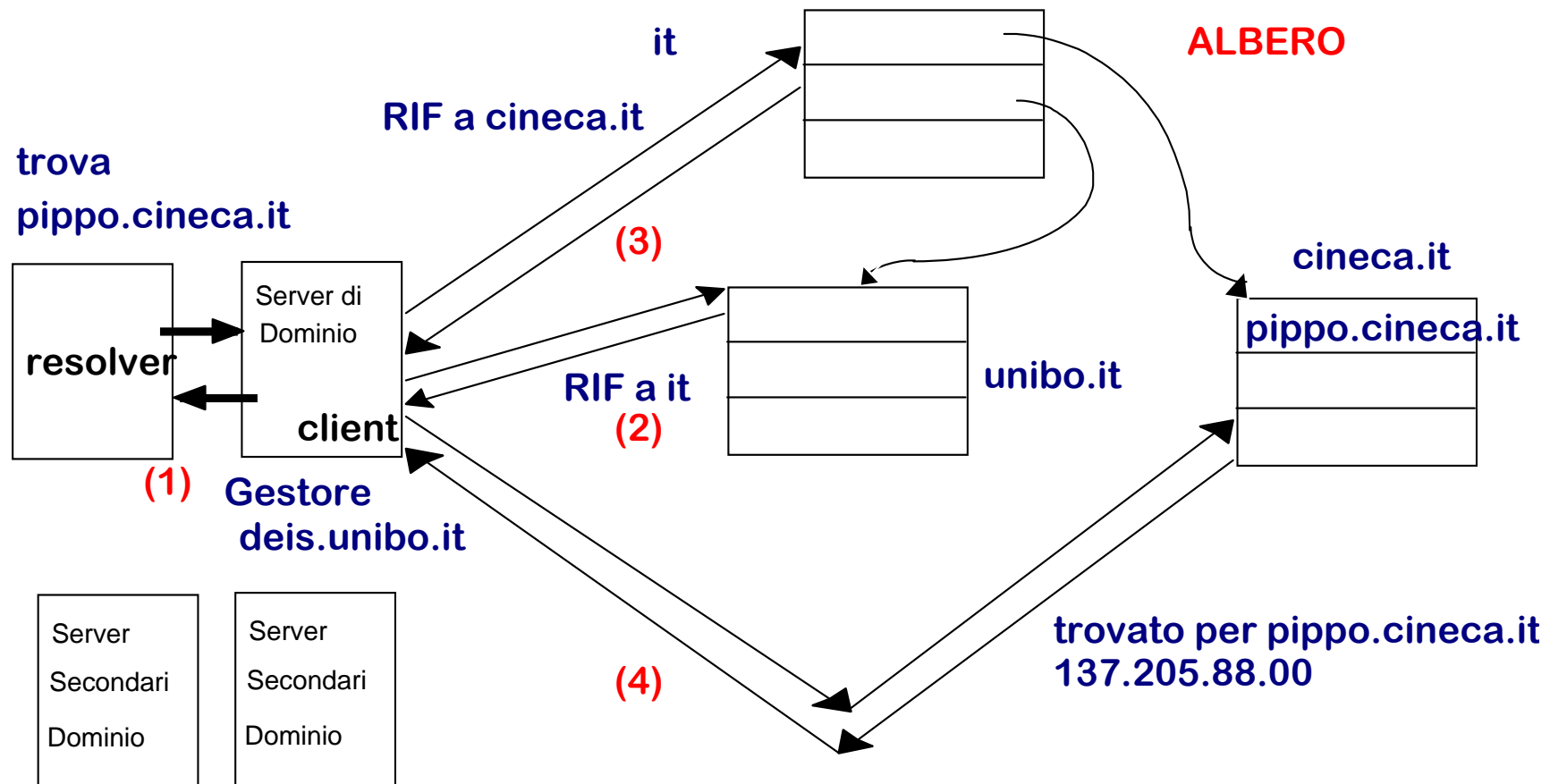
Il resolver query tipicamente ricorsiva

il server di dominio si incarica di rispondere coordinandosi con altri  
(query iterativa o ricorsiva)

# RISOLUZIONE QUERY DNS

## Ricorsiva e Iterativa

query: `pippo.cineca.it` a server DNS `deis.unibo.it`



# RISOLUZIONE NOMI

---

Se il server possiede il **nome** (in tabella o cache), **risponde**

Se **query ricorsiva**, **cerca altre risposte** e rimane impegnato fino a risposta o time-out

Se **query iterativa**, il server che non possiede il nome, risponde con **un riferimento al gestore più vicino che possa ragionevolmente rispondere** (considerando l'albero dei servitori ed il nome fornito)

**Ogni name server decide come rispondere**

TIPICAMENTE, il name server locale fa query iterativa, senza conoscenza a priori della gerarchia ma solo del DNS locale

Il name server root e altri a livelli alti non query ricorsive

Si usano **timeout** ed eventualmente il server ne consulta altri e, se le zone hanno secondari, ci si rivolge anche a quelli

TIPICAMENTE, si provano **diversi server** noti mandando a ciascuno almeno due o più ritrasmissioni con time-out crescenti (4 volte)

se non c'è risposta si assume che il server sia fallito

(timeout  $\Rightarrow$  server crash)



# TABELLA NOMI DNS

---

Un server crea una tabella con un record sorgente per ogni risorsa e lo carica dinamicamente da file di configurazione, e lo aggiorna

Le query consultano l'insieme dei **record**, con **molte attributi** come:

Nome dominio  
Time to live (tempo di validità in secondi)  
Classe (Internet IN)  
Tipo (descrizione del tipo)  
Valore (valore dell'attributo)

Tipo	Significato	Valore
SOA	Start of Authority	parametri della zona
A	IP host address	intero a 32 bit (dot notation)
MX	Mail exchange	server di domino di mail
NS	Name server	server per dominio corrente
CNAME	Canonical name	alias di nome in un dominio
PTR	Pointer	per corrispondenza inversa
HINFO	Host description	descrizione di host e SO
TXT	Text	testo qualunque

# ESEMPIO FILE NOMI DNS

---

@ **IN SOA** promet1.deis.unibo.it. postmaster.deis.unibo.it.

(644 10800 1800 604800 86400)

; serial number, refresh, retry, expiration, TTL in sec

**IN NS** promet1.deis.unibo.it.

**IN NS** almadns.unibo.it.

**MX** 10 deis.unibo.it.

**MX** 40 mail.ing.unibo.it.

lab2 **IN NS** lab2fw.deis.unibo.it.

lab2fw **IN A** 137.204.57.136

**IN HINFO** HW:PC IBM SW:WINDOWS 95

**IN WKS** 137.204.57.136 TCP FTP TELNET SMTP

**IN MX** 40 lab2fw.deis.unibo.it.

deis18 **IN TXT** "Qualunque testo non significativo"

deis18 **IN RP** root.deis.unibo.it luca\ghedini.mail.ing.unibo.it

; record per responsabile

146 **IN PTR** deiscorradi.deis.unibo.it.

; record per la corrispondenza inversa

# QUERY NOMI DNS

A server DNS si fanno generalmente query dirette, cioè

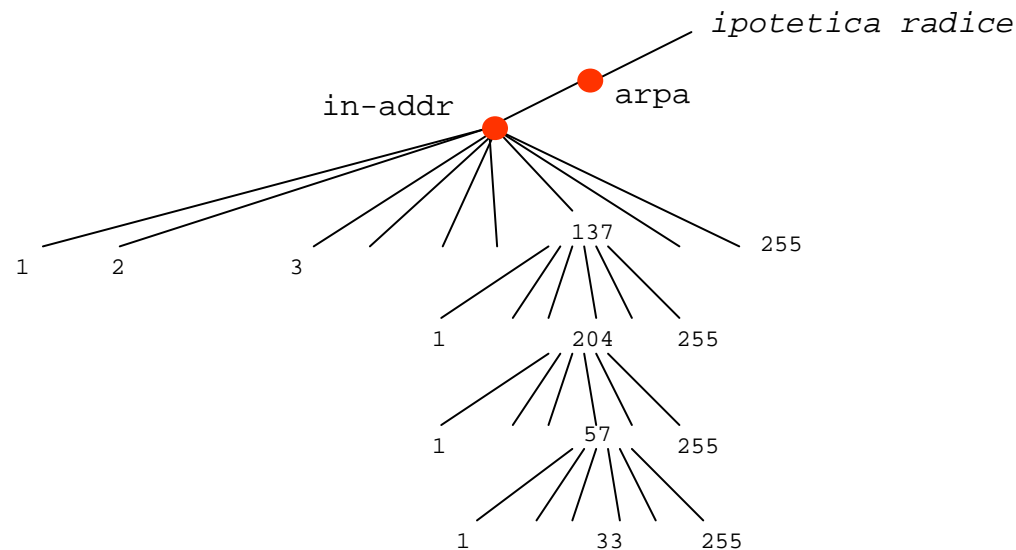
*si entra con un nome logico e ci si aspetta un numero IP*

Uso di `nslookup` come tool

Ma anche possibili query inverse

*si entra con un numero IP e ci si aspetta un nome logico*

Queste risoluzioni richiedono un albero di corrispondenza inverso



# QUERY DIRETTA DNS

---

> **unibo.it.**

Server: **promet1.deis.unibo.it** Address: 137.204.59.1, res\_mkqry(0, unibo.it, 1, 1)

**Got answer:** HEADER:

opcode = QUERY, id = 15, rcode = NOERROR header flags: response, want recursion, recursion avail. questions =1, answers =1, authority rec. = 4, additional =4

QUESTIONS: unibo.it, type = A, class = IN

ANSWERS:

-> unibo.it internet address = 137.204.1.15 ttl = 58196 (16 hours 9 mins 56 secs)

AUTHORITY RECORDS:

-> unibo.it nameserver = dns2.cineca.it ttl = 155488 (1 day 19 hours 11 mins 28 s)

-> unibo.it nameserver = dns.nis.garr.it ttl = 155488 (1 day 19 hours 11 mins 28 s)

-> unibo.it nameserver = dns.cineca.it ttl = 155488 (1 day 19 hours 11 mins 28 s)

-> unibo.it nameserver = almadns.unibo.it ttl = 155488 (1 day 19 hours 11 ms 28 s)

ADDITIONAL RECORDS:

-> dns2.cineca.it internet address = 130.186.1.1 ttl = 258410 (2 days 23 hours 46 mins 50 secs) .... -----

Non-authoritative answer:

**Name: unibo.it Address: 137.204.1.15**

# QUERY DIRETTA DNS

---

> **cineca.it.**

Server: **promet1.deis.unibo.it** res\_mkquery(0, cineca.it, 1, 1)

Got answer: HEADER:

opcode = QUERY, id = 28, rcode = NOERROR header flags: response, want recursion, recursion avail. questions =1, answers =0, authority records =1, addit. = 0

**QUESTIONS: cineca.it, type = A, class = IN**

AUTHORITY RECORDS:

-> cineca.it

ttl = 10784 (2 hours 59 mins 44 secs)

origin = dns.cineca.it

mail addr = hostmaster.cineca.it

serial = 1999052501

refresh = 86400 (1 day)

retry = 7200 (2 hours)

expire = 2592000 (30 days)

minimum ttl = 259200 (3 days)

**Name: cineca.it Address: 130.186.1.1**

# QUERY INVERSA DNS

---

Per una query inversa, si deve lavorare sull'albero pointer e con l'indirizzo girato in byte: qui si vuole 137.204.57.86...

> set q=ptr

> 86.57.204.137.in-addr.arpa.

Server: promet1.deis.unibo.it Address: 137.204.59.1

86.57.204.137.in-addr.arpa      **name = deis86.deis.unibo.it**

57.204.137.in-addr.arpa      nameserver = admii.arl.army.mil

57.204.137.in-addr.arpa      nameserver = almadns.unibo.it

57.204.137.in-addr.arpa      nameserver = promet4.deis.unibo.it

57.204.137.in-addr.arpa      nameserver = promet1.deis.unibo.it

admii.arl.army.mil      internet address = 128.63.31.4

admii.arl.army.mil      internet address = 128.63.5.4

almadns.unibo.it      internet address = 137.204.1.15

...

Non tutti i server consentono queste query! A chi?

# NOMI di INTERNET e OSI

## STANDARD di comunicazione ISO – OSI

**OSI Open System Interconnection** con obiettivo di comunicazione aperta tra reti e tecnologie diverse proprietarie

ESEMPI di sistemi APERTI: UNIX che non lega ad un produttore, con software free (open source) e con driver TCP/IP ossia Internet

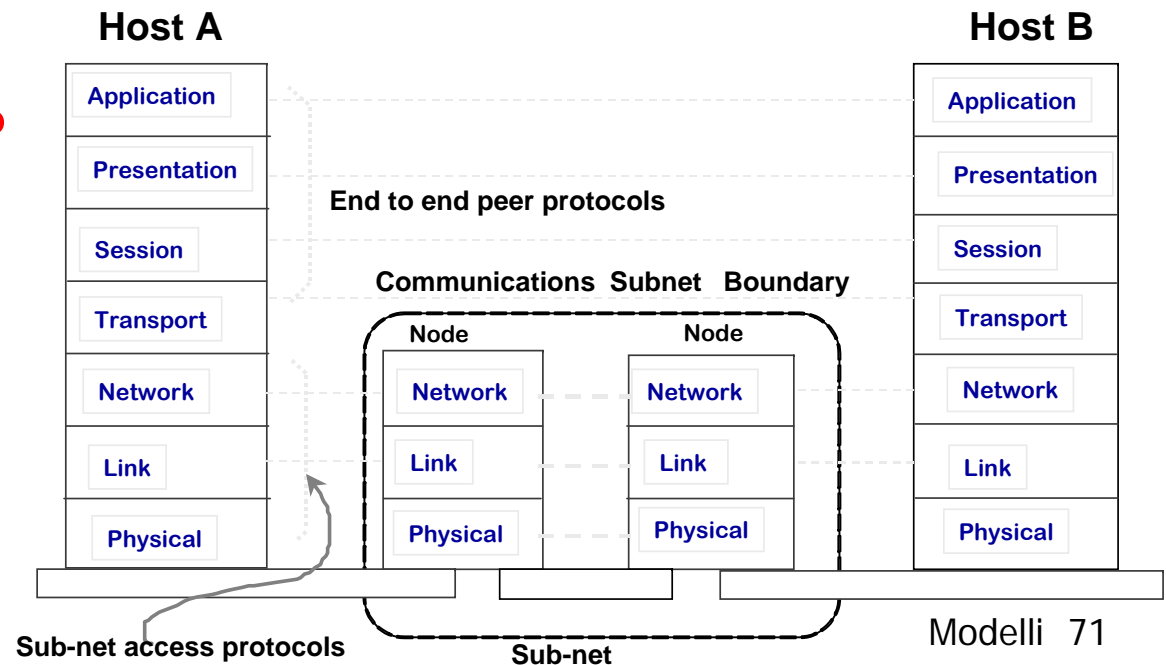
Vantaggi degli standard aperti ⇒ **INTEROPERABILITÀ**

In TCP/IP

Livello 3: **porte UDP e TCP**

Livello 2: **nomi IP**

4	Processo
3	Trasporto
2	Rete
1	Data Link



# NOMI di INTERNET o TCP/IP

Livello Trasporto definisce le **porte** per i servizi

Livello IP, definisce i **nomi IP** per i diversi nodi nella comunicazione

NOMI IP: IP individua connessioni nella rete virtuale, definendo per ogni connessione un indirizzo Internet unico a 32 bit

**IP-ADDRESS** ⇒ suddiviso nella coppia (NETID, HOSTID)

STANDARD IETF prescrive nomi dati di autorità

Network Information Center (NIC) assegna il numero di rete, cioè l'informazione usata nei gateway per routing e poi la autorità locale definisce i nomi di host

