

## Architetture di sistemi distribuiti

Antonio Lioy  
< lioy@polito.it >

*Politecnico di Torino*  
*Dip. Automatica e Informatica*

## Modello tipico di un'applicazione

- **interfaccia utente (UI)**
  - gestione di tutto l'I/O con l'utente
- **logica applicativa**
  - elaborazioni da fare per fornire il servizio all'utente
- **dati (grezzi)**
  - informazioni necessarie all'applicazione



## Esempio (applicaz. "classica")

```

#include <stdio.h>
int main ( )
{
    double percentuale_iva = 20;
    double prezzo;

    char buf[100];
    printf ("costo? ");
    gets (buf);
    sscanf (buf, "%lf", &costo);
    prezzo = costo * (1 + percentuale_iva / 100);
    printf ("prezzo di vendita = %.2lf\n", prezzo);
    return 0;
}
    
```

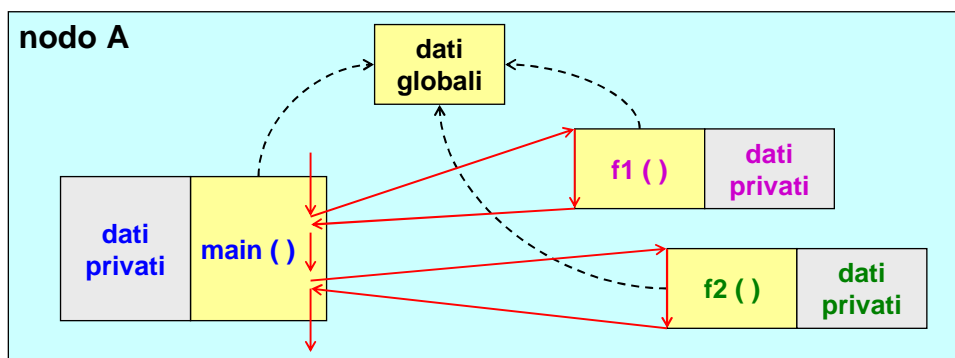
interfaccia utente

dati applicativi

logica applicativa

## Elaborazione "classica"

- dati locali (condivisi / privati)
- unico spazio di indirizzamento
- elaborazione sequenziale su unica CPU
- flusso elaborazione univoco (eccezione: interrupt)



## Elaborazione "classica": vantaggi

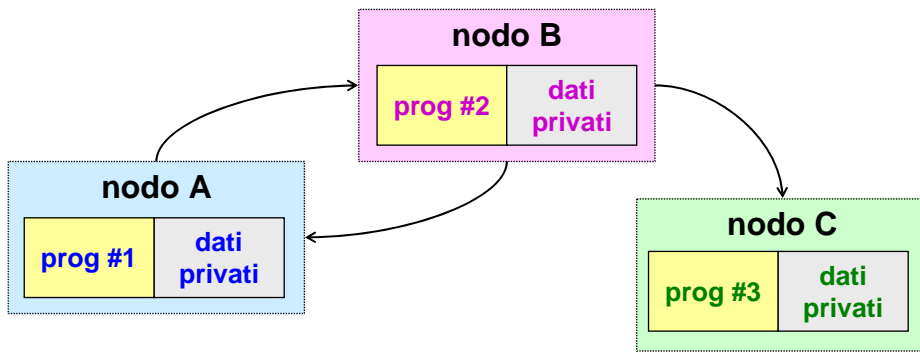
- **semplicità di programmazione**
- **robustezza**
- **buona possibilità di ottimizzazione**

## Elaborazione "classica": problemi

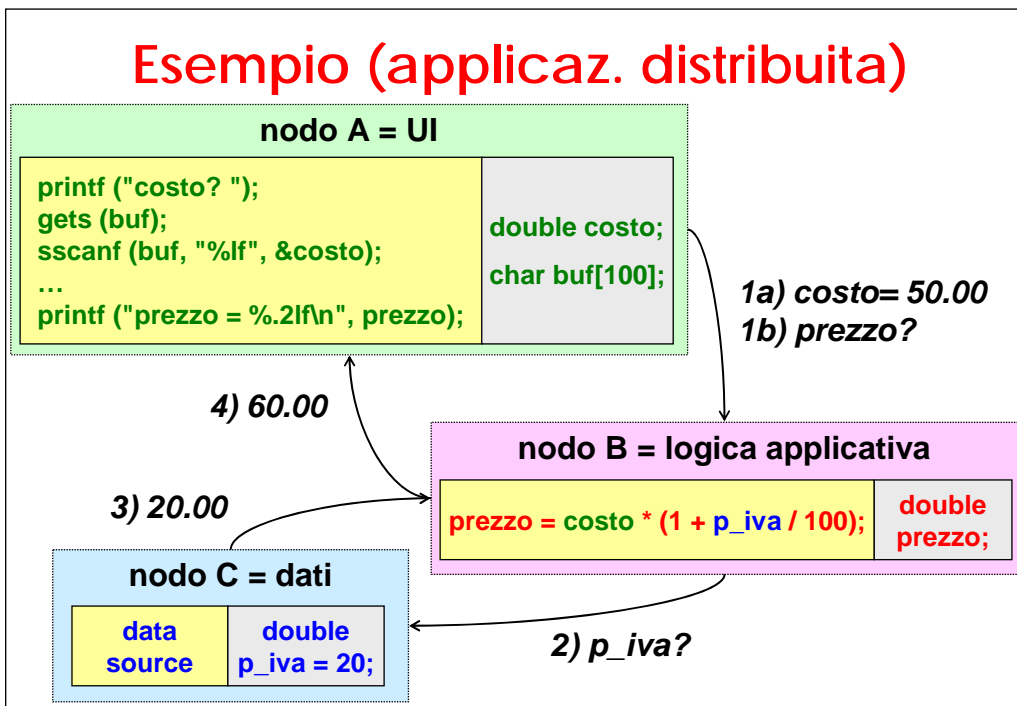
- **protezione dei dati da operazioni illegali**
  - operazioni compiute sui dati globali
  - sono accessibili anche i dati privati (!)
  - parzialmente migliorabile con OOP
- **basse prestazioni**
  - unica CPU, elaborazione sequenziale
  - migliorabile con sistemi multi-CPU e programmazione concorrente (thread, processi)
- **uso solo tramite accesso fisico al sistema**
  - terminali o "console"
  - migliorabile con collegamenti via modem / rete

## Elaborazione distribuita

- dati solo locali (privati)
- molti spazi di indirizzamento
- elaborazione concorrente su CPU diverse
- molti flussi di elaborazione



## Esempio (applicaz. distribuita)



## Elaborazione distribuita: vantaggi

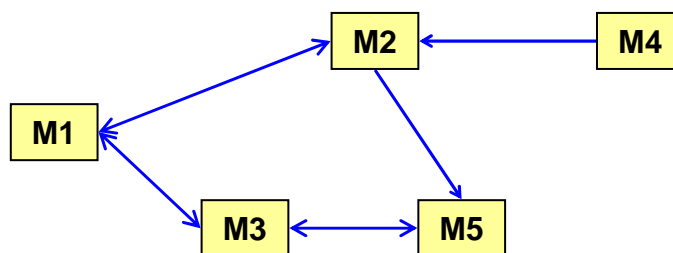
- **elevate prestazioni**
  - molte CPU
- **buona scalabilità**
  - più facile aumentare il n. di CPU che la potenza di una singola CPU
- **protezione dei dati da operazioni illegali**
  - spazi di memoria separati, accessibili solo tramite i rispettivi programmi
- **accesso tramite rete**
  - non necessaria presenza fisica dell'utente

## Elaborazione distribuita: problemi

- **complessità di programmazione:**
  - come comunicano i vari programmi?
  - formato dei dati sui vari nodi di rete?
  - necessità di definire **protocolli (applicativi)**
  - sincronizzazione delle operazioni può portare ad attese e rallentamenti
- **scarsa robustezza**
  - maggiori possibilità di errore / malfunzionamenti
- **difficile ottimizzazione**
  - mancanza di una visione globale

## Architettura software

- collezione di moduli software (o componenti)
- ... interagenti tramite un ben definito paradigma di comunicazione (o connettori)
- nota: non è detto che la comunicazione sia effettuata via rete (es. IPC sullo stesso nodo)



## Modello client-server

- metodo più diffuso per creare applicativi distribuiti
- client e server sono due processi separati:
  - il server fornisce un generico servizio
  - il client richiede il servizio
- anche sul medesimo sistema

Attenzione alla differenza tra client e server:

- come **elementi hw** di un sistema di elaborazione
- come **processi** di un'architettura distribuita

## Il server

- **idealmente è in esecuzione “da sempre”:**
  - attivato al boot
  - attivato esplicitamente dal sistemista
- **accetta richieste da uno o più punti:**
  - porta TCP o UDP (analogo al concetto di SAP OSI)
  - porte fisse e solitamente predeterminate
- **manda risposte relative al servizio**
- **idealmente non termina mai:**
  - allo shutdown
  - azione esplicita del sistemista

## Il client

- **attivato su richiesta di un “utente”**
- **invia richiesta verso un server**
- **attende la risposta su una porta allocata dinamicamente (non può essere una porta fissa perché ci possono essere molti “utenti” che operano simultaneamente, es. due finestre di un browser web)**
- **esegue un numero finito di richieste e poi termina**

## Architetture

- usando i concetti di client e server si possono costruire svariate architetture
- architettura client-server (C/S)
  - architettura asimmetrica
  - il posizionamento del server è determinato a priori
- architettura peer-to-peer (P2P)
  - architettura simmetrica
  - ogni nodo può ricoprire il ruolo di client e di server (simultaneamente o in tempi diversi)

## Architettura client-server (C/S)

- architettura in cui processi client richiedono i servizi offerti da processi server
- vantaggi:
  - semplicità di realizzazione
  - semplificazione del client
- svantaggi:
  - sovraccarico del server
  - sovraccarico del canale di comunicazione



## Architettura C/S 2-tier

- è il C/S classico, originale (es. NFS)
- il client interagisce direttamente con il server senza passaggi intermedi
- architettura tipicamente distribuita su scala sia locale sia geografica
- usata in ambienti di piccole dimensioni (50-100 client simultanei)
- svantaggi
  - bassa scalabilità (es. al crescere del numero di utenti, decrescono le prestazioni del server)

## C/S 2-tier: client pesante o leggero?

- tre componenti (UI, logica applicativa, dati) ... da distribuire su due soli elementi (client e server)
- soluzione 1 = fat client / thin server
  - client = UI + logica applicativa
  - server = dati
  - schema tradizionale, difficoltà di sviluppo (sw ad-hoc) e gestione (installazione, aggiornamento), minor sicurezza
- soluzione 2 = thin client / fat server:
  - client = UI
  - server = logica applicativa + dati
  - (es. il web) pesante sui server, maggior sicurezza

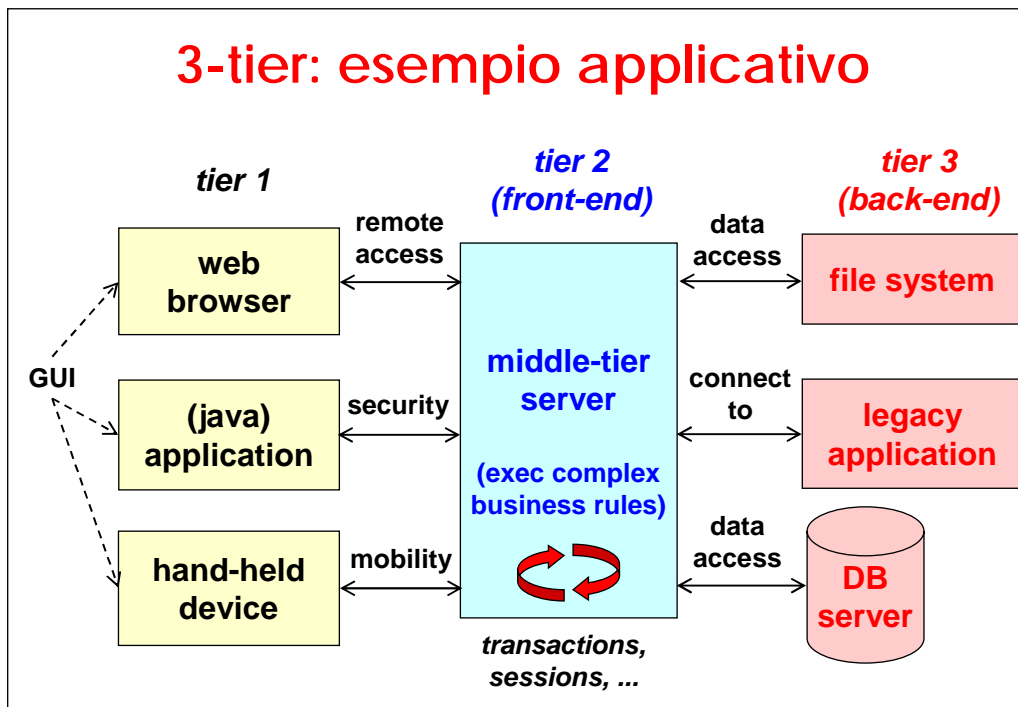
## Architettura C/S 3-tier

- **un componente (o agente) è inserito tra client e server, per svolgere vari ruoli:**
  - filtro (es. adatta un sistema legacy su mainframe a un ambiente C-S)
  - bilanciamento del carico di lavoro sul/i server (es. load balancer con più server equivalenti)
  - servizi intelligenti (es. distribuire una richiesta su più server, collezionare i risultati e restituirli al client come risposta singola)



## Esempi di sistemi C/S 3-tier


- **per migliorare le prestazioni (di calcolo):**
  - agente = load balancer
  - server = server farm di server omogenei o equivalenti
- **per adattare il server alle capacità del client:**
  - agente = mediatore / broker
  - server = insieme di server eterogenei / non equivalenti




### La UI ed il web

- UI “tradizionale” custom:
  - difficile sviluppo, deployment e manutenzione
  - difficile addestramento degli utenti
- UI “moderna” web è divisa in due:
  - UI client-side standard (=browser)
  - UI server-side standard (=server web) e programmabile facilmente

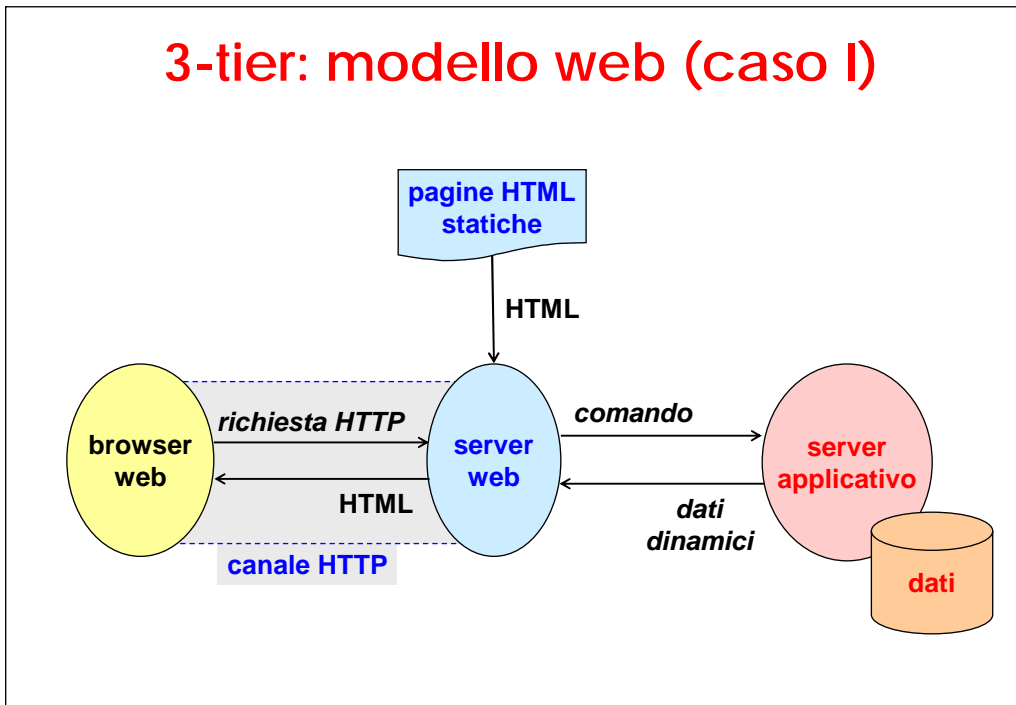
client-side  
(interazione con l'utente)



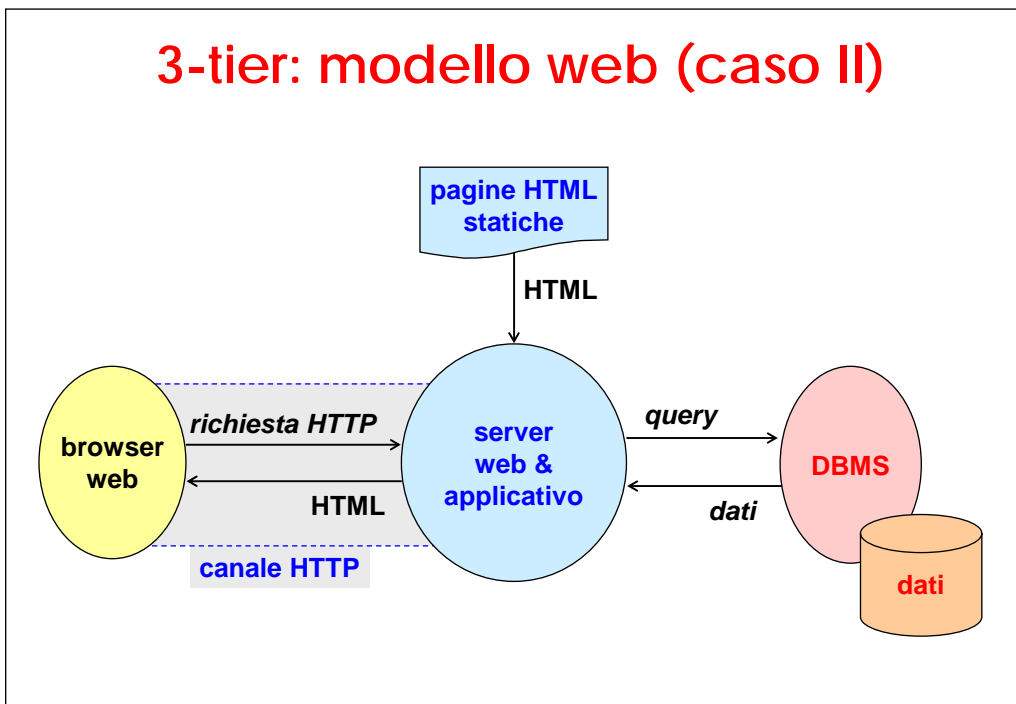


server-side  
(interfaccia applicativa)

### 3-tier: modello web (caso I)

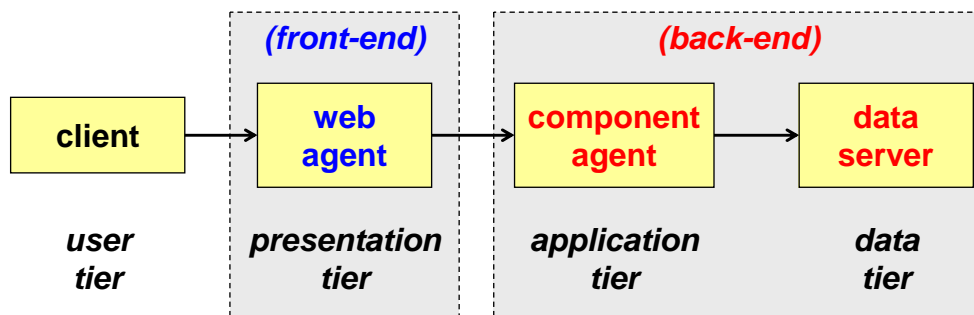


### 3-tier: modello web (caso II)



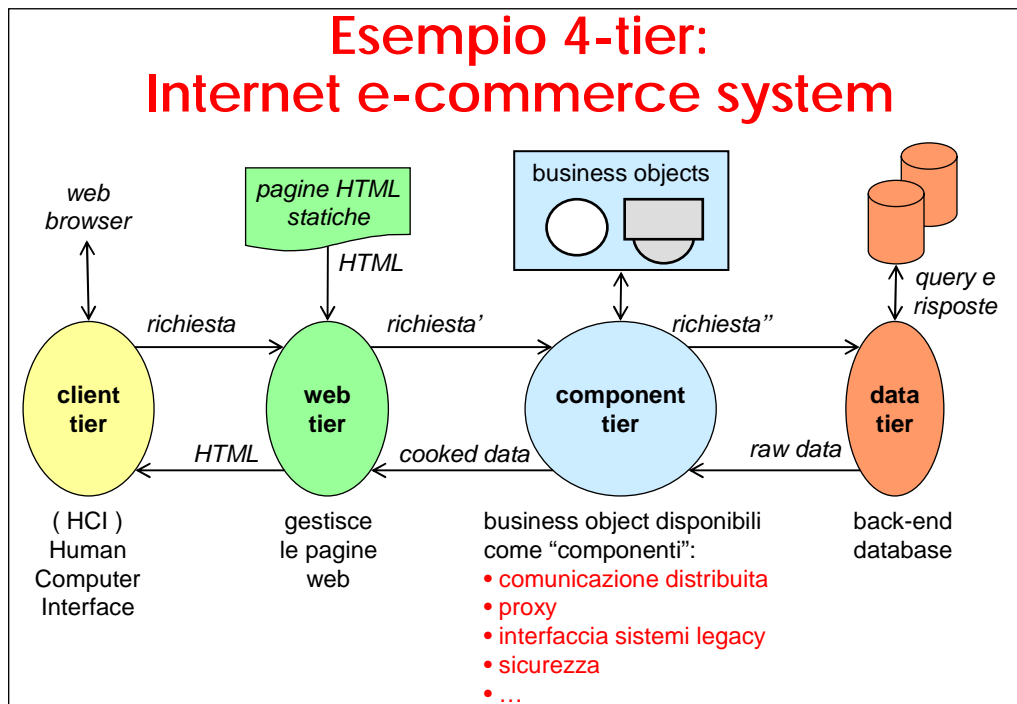
## Architettura C/S 4-tier

- **due componenti / agenti inseriti tra client e server:**
  - web agent (gestisce la presentazione dei dati)
  - component agent (gestisce le logiche applicative)



## Migliorare le prestazioni di rete?

- **architetture 3/4-tier migliorano le prestazioni di calcolo ... ma il front-end resta un collo di bottiglia**
- **come migliorare? l'erogatore del servizio non controlla la parte di rete tra client e front-end**
- **tentativo di miglioramento:**
  - statistica sulla provenienza dei clienti
  - moltiplicare il front-end (uno per ogni rete da cui provengono i miei clienti)
  - come indirizzare i clienti verso il front-end giusto?
    - basandosi su lingua/dominio (es. srv.it, srv.fr)
    - basandosi sul routing (es. DNS modificato Akamai)



## Client tier: browser o applicazione?

### ■ web browser:

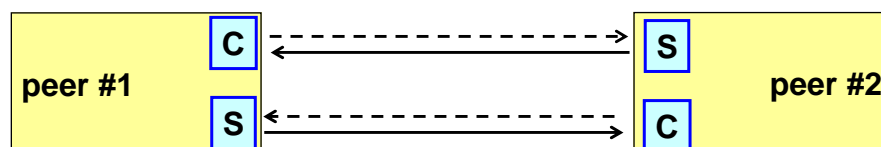
- (V) noto agli utenti e gestito da essi
- (V) comunicazione e dati standard (HTTP, HTML)
- (S) versione incerta di protocollo e dati (minimo comune?)
- (S) prestazioni non elevate (interprete)
- (S) funzionalità limitata (semplice interfaccia grafica)
- (S) estensioni non sempre supportate:
  - applet (Java, Active-X)
  - script client-side (JavaScript, VBscript)
  - plugin (Flash, ...)

## Client tier: browser o applicazione?

- **applicazione client custom / ad-hoc:**
  - (V) funzionalità molto ricca (=richiesta dal server)
  - (V) prestazioni molto elevate
  - (S) addestramento all'uso
  - (S) piattaforme supportate
  - (S) deployment ed aggiornamento
  - (S) assistenza utenti

## Architettura peer-to-peer (P2P)

- **architetture in cui i processi possono fungere simultaneamente da client e da server**
- **vantaggi:**
  - carico di lavoro e di comunicazione distribuito tra tutti i processi
- **svantaggi:**
  - difficoltà di coordinamento / controllo
  - carico di comunicazione realmente distribuito?



## P2P computing

- i client evolvono da meri utenti di servizi a fornitori autonomi di servizi
- per condividere risorse e sfruttare servizi collaborativi
- si sfruttano meglio le capacità di calcolo dei singoli nodi (così si scaricano i server)
- si usano meglio le reti, con comunicazioni dirette tra i nodi (così si evitano congestioni sui link verso i server)

## Architetture P2P

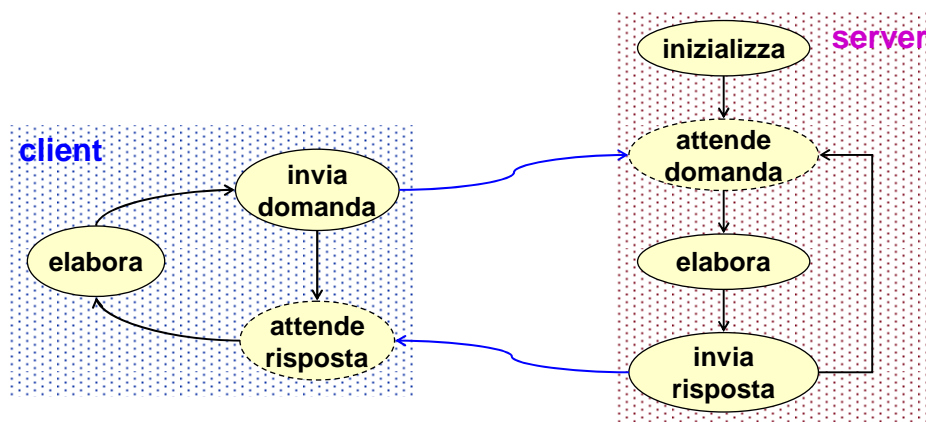
- **collaborative computing**
  - comunità di rete per compiti distribuiti (es. grid computing; aperto o chiuso; anche per dati riservati o elaborazioni con deadline fissa?)
- **edge service**
  - servizi ortogonali come “fattori abilitanti” per la creazione di comunità P2P (es. TOIP, Internet fax)
- **file sharing**
  - per scambiare informazioni sulla rete senza doverle caricare su un server, ma lasciandole là dove si trovano (problema: l'indice)
  - es. Gnutella (gnutella.wego.com), WinMX, Kazaa

## Modelli di server

- l'architettura interna del server influenza molto le prestazioni del sistema complessivo
- bisogna scegliere il modello più adatto al problema applicativo
- non esiste una soluzione buona per tutti gli usi (si rischia che sia troppo complicata)

## Server iterativo

- se il servizio richiesto è di breve durata il server lo fornisce direttamente



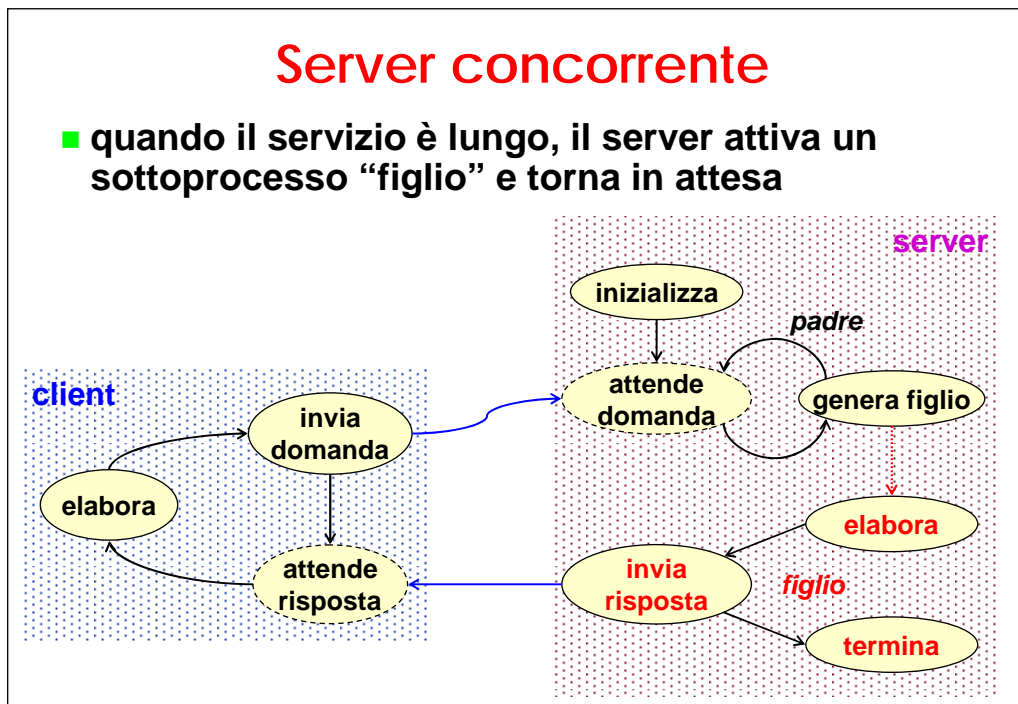
## Esempi di server iterativo

- **servizi standard TCP/IP di breve durata:**
  - daytime (tcp/13 o udp/13) RFC-867
  - qotd (tcp/17 o udp/17) RFC-865
  - time (tcp/37 o udp/37) RFC-868
- **in generale, servizi in cui si vuole fortemente limitare il carico (un solo utente per volta)**
- **vantaggi:**
  - semplicità di programmazione
  - velocità di risposta (quando ci si riesce a collegare!)
- **svantaggi:**
  - limite di carico

## Prestazioni di un server iterativo

- prestazioni non influenzate dal numero di CPU
- sia  $T_E$  il tempo di CPU dell'elaborazione richiesta al server [s] (ipotesi:  $T_E \gg T_R$ )
- prestazioni massime (in condizioni ottimali):
 
$$P = 1 / T_E \text{ servizi / s}$$
- in caso di richieste simultanee da più client, quelli non serviti rientrano in competizione successivamente (a meno che la coda delle domande abbia ampiezza  $> 1$ )
- la latenza [ s ] del servizio dipende dal carico  $W \geq 1$  del nodo che ospita il server:

$$T_E \leq L \leq T_E \times W \quad \text{ovvero} \quad L \sim T_E \times E(W)$$



- ## Esempi di server concorrente
- la maggior parte dei servizi standard TCP/IP:
    - echo (tcp/7 o udp/7) RFC-862
    - discard (tcp/9 o udp/9) RFC-863
    - chargen (tcp/19 o udp/19) RFC-864
    - telnet (tcp/23) RFC-854
    - smtp (tcp/25) RFC-2821
    - ...
  - in generale, i servizi con elaborazione complessa o di durata lunga e/o non prevedibile a priori

## Server concorrente: analisi

### ■ vantaggi:

- carico idealmente illimitato

### ■ svantaggi:

- complessità di programmazione (concorrente)
- lentezza di risposta (creazione di un figlio,  $T_F$ )
- carico max limitato (ogni figlio richiede RAM, cicli di CPU, cicli di accesso a disco, ...)

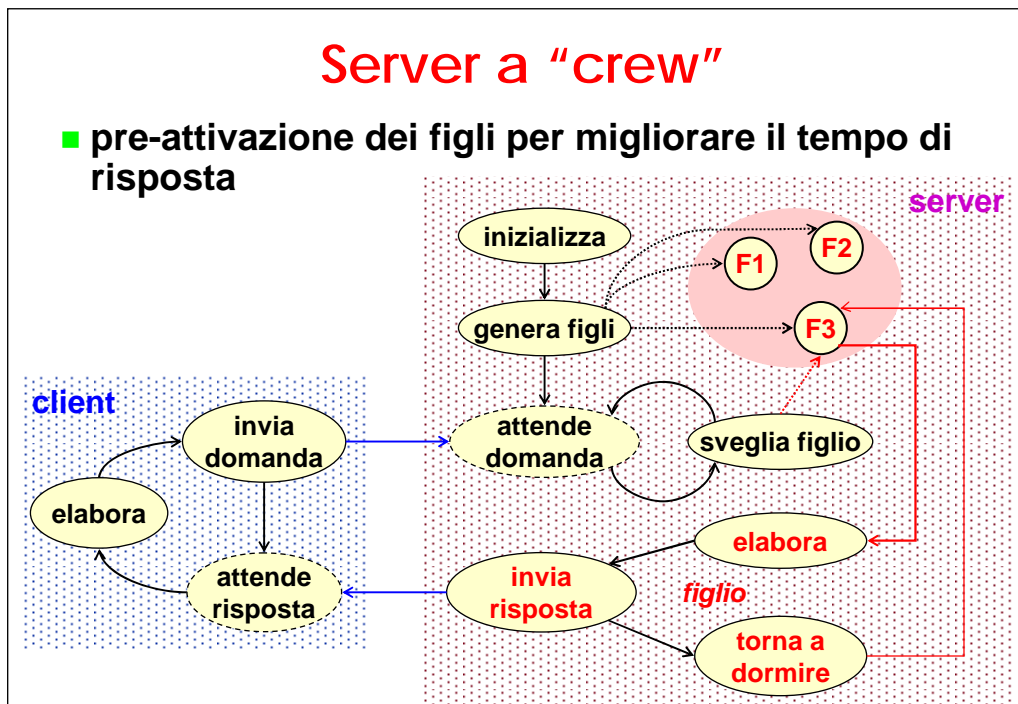
## Prestazioni di un server concorrente

- influenzate dal numero di CPU (sia esso  $C$ )
- sia  $T_F$  il tempo di CPU per creare un figlio [s]
- prestazioni massime (in condizioni ottimali):

$$P = C / (T_F + T_E) \text{ servizi / s}$$

- in caso di richieste simultanee da più client, quelli non serviti rientrano in competizione successivamente (a meno che la coda delle domande abbia ampiezza  $> 1$ )
- la latenza del servizio dipende dal carico  $W$  del nodo che ospita il server:

$$(T_F + T_E) \leq L \leq (T_F + T_E) \times W / C \text{ s}$$



- ### Esempi di server a "crew"
- tutti i servizi concorrenti possono essere realizzati con server a crew
  - servizi di rete ad alte prestazioni:
    - sottoposti ad alto carico (=n. di utenti simultanei)
    - con basso ritardo alla risposta (latenza)
  - esempi tipici:
    - web server per e-commerce
    - DBMS server

## Server a "crew": analisi

### ■ vantaggi:

- carico idealmente illimitato (si possono generare figli addizionali in funzione del carico)
- velocità di risposta (svegliare un figlio è più rapido che crearlo)
- possibilità di limitare il carico massimo (solo figli pre-generati)

### ■ svantaggi:

- complessità di programmazione (concorrente)
- gestione dell'insieme dei figli (children pool)
- sincronizzazione e concorrenza degli accessi alle risorse condivise del server da parte dei vari figli

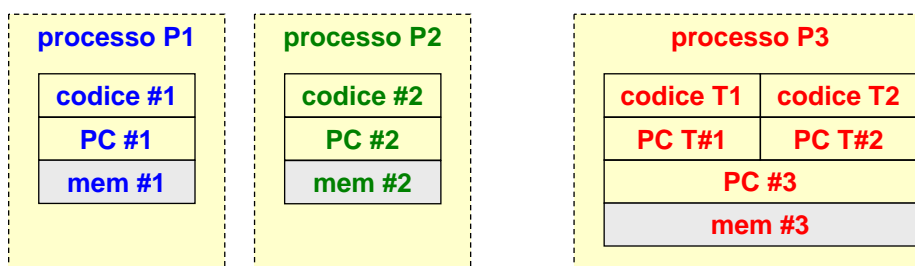
## Prestazioni di un server "a crew"

- analoghe a quelle di un server concorrente, con  $T_F$  sostituito dal tempo necessario ad attivare un figlio  $T_A$  (di solito trascurabile)
- se, una volta esauriti i figli, il server a crew può generarne altri allora le prestazioni sono una combinazione pesata con la probabilità  $G$  di dover generare nuovi figli:

$$P = (1 - G) \times [C / (T_A + T_E)] + G \times [C / (T_F + T_E)]$$

## Programmazione concorrente

- lavoro simultaneo di più moduli di elaborazione sulla stessa CPU
- due modelli principali:
  - processi
  - thread



## Processi vs. thread (I)

- **attivazione di un modulo**
  - [ P ] lenta
  - [ T ] veloce
- **comunicazione tra moduli**
  - [ P ] difficile (richiede IPC, es. pipe, shared memory)
  - [ T ] facile (stesso spazio di indirizzamento)

## Processi vs. thread (II)

### ■ protezione tra moduli

- [ P ] ottima, sia della memoria sia dei cicli di CPU
- [ T ] pessima (e l'accesso a memoria comune richiede sincronizzazione e può causare deadlock)

### ■ debug:

- [ P ] non banale ma possibile
- [ T ] molto difficile (schedulazione non replicabile)