Università degli Studi di Bologna
Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Scuola di Ingegneria

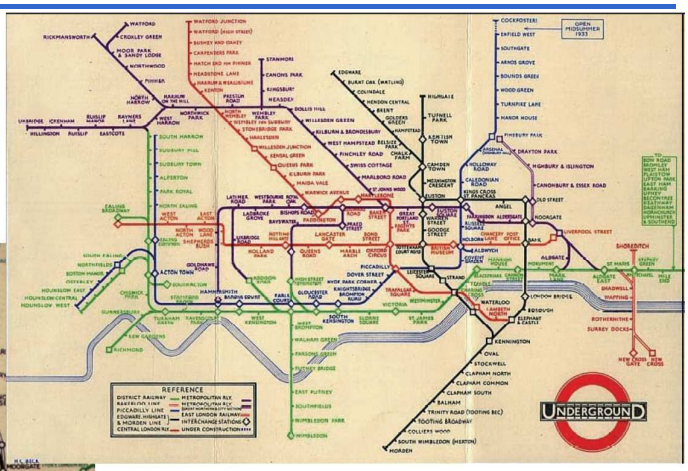
Corso di **Reti di Calcolatori M**

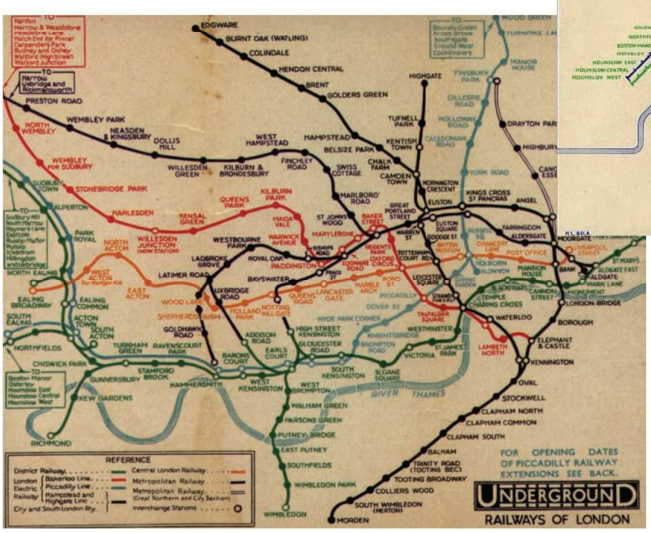
MODELLI alla base della ESECUZIONE in SISTEMI DISTRIBUITI ed APPLICAZIONI

Antonio Corradi

Anno accademico 2014/2015

ASTRAZIONE …





In **sistemi complessi**, di particolare interesse

SISTEMI DISTRIBUITI

Ambito non completamente risolto … o meglio

Area molto utilizzata e **da affrontare continuamente**

Ma con **molte sfide da superare e problemi da risolvere.** Ad esempio:

- **Scalabilità** e **Latenza limitata**
- **Predicibilità** e **Performance**
- **Facilità di supporto a concorrenza e parallelismo**
- **Superamento di failure parziali**
- **Eterogeneità (a diversi livelli)**
- **Integrazione e standard**

…

MODELLI e INTERPRETAZIONE

- **MODELLI approfonditi per l'ESECUZIONE**
  - **Modelli architetturali di supporto e fornitura servizio (o di provisioning e deployment)**
  - **PRAM**
  - **Efficienza e Speed-up**
  - **Qualità del Servizio ossia QoS**
- Esperienze di **interpretazione dei modelli**
  - **Sono facili da maneggiare**
  - **Presentano la realtà** (e la modellano ☺)
  - **Aiutano nello sviluppo di progetti**
  - **I modelli sono sempre significativi e applicabili?**

ALCUNI MODELLI di BASE

Spesso, **pattern, modi, strategie locali sono utili**

modelli e paradigmi ***statici/dinamici***

modelli e strategie ***preventivi /reattivi***

modello di ***esecuzione*** ***nel sistema***

monoutente/multiutente

processore/processori

modello per ***esecuzione attiva***

processi/oggetti **replicazione**

modello delle ***entità per la allocazione***

processi/oggetti **decisioni statiche/dinamiche**

SISTEMI DISTRIBUITI

Nei **sistemi distribuiti** dobbiamo lavorare con un modello

Un sistema distribuito è costituito da risorse

risorse che sono necessarie durante la esecuzione e che ne permettono i risultati

Le risorse sono ad esempio:

- Memoria fisica

- Disco (persistenza)

- Capacità di calcolo (CPU varie, anche molteplici)

- I/O e supporto alla comunicazione

- Anche altro, ma meno obbligatorio (sensori, attuatori, ecc.)

Per supportare programmi e applicazioni e componenti che devono eseguire sulle diverse risorse fisiche

E che diventano risorse a loro volta, usabili per altro e da altri

GESTIONE RISORSE

GESTIONE RISORSE di un **SISTEMA DISTRIBUITO**
definizione di **risorsa**
ogni componente riusabile o meno, sia **hardware,** sia **software, necessario alla applicazione o al sistema**
Classificazione

* risorse **fisiche**	vs.	risorse **logiche**
* risorse **statiche**	vs.	risorse **dinamiche**
* risorse di **basso livello**	vs.	risorse **applicative**

Risorse organizzate e **basate sulla astrazione**
specifica (interfaccia visibile)e **implementazione**
Realizzazioni concentrate e distribuite
tenendo conto della accuratezza delle informazioni

Gestione organizzata in due fasi (***statica e dinamica***)

DESCRIZIONE delle RISORSE

Descrizione delle risorse
- classe della risorsa
- nome della risorsa
- attributi di una risorsa
  - principali e necessari, ulteriori attributi correlati
- qualità del servizio QoS
  - *esecuzione vincolata su una determinata macchina*

con fattori
- costo
- tempo di allocazione
- tempo di ritardo in comunicazione e servizio

Controllo allocazione (statico)
allocazione della risorsa e del nome
Controllo accesso (dinamico)
locale vs. remoto

GESTIONE RISORSE

La gestione in due fasi: statica e dinamica

1) pianificazione della organizzazione e **identificazione**
- allocazione
- disponibilità
- costo

2) controllo delle risorse **(durante l'uso)**
- controllo di accesso e di Qualità
- ottimizzazione
- autenticazione
- controllo di correttezza operazioni ed eccezioni

Esempio: risorse tipiche
- **fisiche:** **processori e sistema di comunicazione**
- **logiche:** **servizi forniti**

GESTIONE RISORSE

Implementazione della Gestione risorse

Costituite di solo dati (passive) vs. gestione con servitore (complesso)
- composte vs. multilivello (***distribuite***)
- mobili vs. immobili
- movimento e replicazione (con consistenza forte vs. debole)

Due modelli principali di condivisione e gestione delle risorse
- **service request**
- **file system distribuito**

Service Request
- **Servizio specifico** con **richiesta esplicita** dell'utente *(C/S)*

File System Distribuito (FSD) o a middleware
- **Servizio unico** con **trasparenza** alla **allocazione delle risorse**

servizi presenti su tutti i nodi, o solo in alcuni nodi (spesso detti ***file server***)
detto spesso *modello ad Agenti*

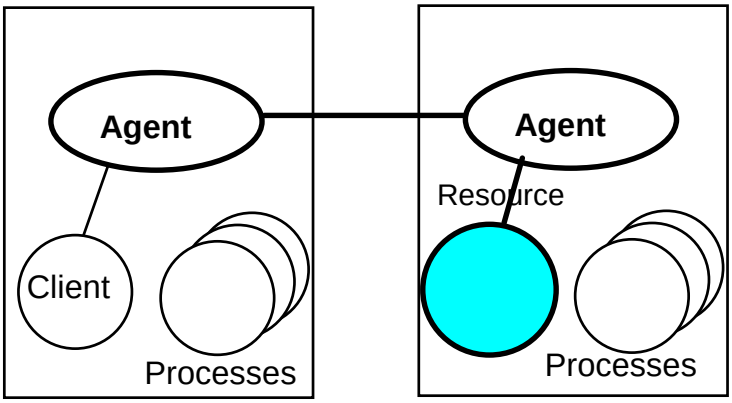
GESTIONE ad AGENTI (FSD)

Unico sistema di agenti per il servizio

agenti per controllare e coordinare il servizio in modo distribuito

Coordinamento tra agenti per l'uso delle risorse

Sono possibili fasi di negoziazione tra gli agenti prima dell'uso della risorsa con possibilità di rifiuto del servizio



PROCESSI e SCHEDULING

Scheduling dei Processi
in due momenti di decisione

Scheduler per la allocazione dei processi al processore

Dispatcher per l'assegnamento del processore ad un processo (meccanismo)

Lo scheduling è la politica che può essere distribuita se condividiamo il carico

e ottenere sia politica locale sia politica globale

Scheduling Locale

Charlotte	round-robin
V-kernel	priorità
Accent	16 livelli di priorità variabile e time-slice

PROCESSI e SCHEDULING

Necessità di *Scheduling Globale*

operazioni remote sui processi

meccanismi

gestione risorse remote

politiche

considerando i PROCESSORI

LOAD SHARING

gestione delle risorse in modo che **nessun processore sia idle** (almeno un processo per processore del set)

LOAD BALANCING

bilanciamento dell'utilizzo delle risorse per ottenere un **carico equilibrato** su tutti i processori partecipanti alla applicazione (**efficienza elevata**)

CREAZIONE REMOTA dei PROCESSI

Operazioni remote sui processi

creazione remota / terminazione remota

esecuzione remota

in SPRITE **fork remota e condivisione variabili (sola lettura)**

Esecuzione remota

possibilità di attivare un processo su un nodo diverso e di interagire con questo (disponibilità ai diversi livelli del sistema)

In **V-kernel**: un comando usa altre workstation libere da carico locale: **a livello utente, trasparenza meno**
V-Kernel
selezione **esplicita** od **implicita** dell'host
<programma><args> @ <nomehost>
<programma><args> @ *

ESECUZIONE REMOTA

Requisiti

- necessità di propagare **informazioni di stato** dei processori
- **non-interferenza** con l'uso locale
- **basso overhead** della esecuzione remota

Eterogeneità come problema

- non esiste uno spazio globale dei nomi per le entità da riferire (che deve essere creato)
- esistono diverse convenzioni per definire i servizi e gli attributi (sintassi dei comandi, codice diverso, formato dei dati, etc.)
- è necessaria una traslazione delle informazioni da uno spazio ad un altro (costo di traslazione)

GESTIONE delle RISORSE

Gestione delle **risorse** del processo in modo **trasparente** ed **indipendente** dalla **allocazione**
considerando

risorse locali semplici (variabili, codice, …)
risorse più complesse (vincolate)
come file aperti e **risorse di comunicazione**

Ci si basa sulla copia per le risorse semplici …
per le risorse complesse (che introducono vincoli e non consentono facili semantiche per copia) …

Necessità di gestori per realizzare la **trasparenza** del servizio

le richieste vanno sempre dirette a gestori *che permettono di distinguere i servizi* ***locali*** *e servizi* ***remoti*** *fornendo una unica interfaccia all'utilizzatore*

ALLOCAZIONE dei PROCESSI

Politiche di allocazione e riallocazione (dinamica)

LOAD SHARING ⇨ Deciso a priori prima della esecuzione (e attuato dopo, alla creazione delle risorse)

Allocazione delle risorse, *senza* ***muovere*** *gli* ***elementi una volta allocati (allocazione statica)***

LOAD BALANCING ⇨ Fatto durante l'esecuzione

Dopo la allocazione e una esecuzione su un nodo, si possono **migrare risorse già allocate** *cercando di migliorare la efficienza globale* ***(allocazione dinamica)***

assumendo la ***conoscenza globale dei processi*** *o* ***meno***
ottimizzando globalmente *costi elevati* *(statico)*
o ***meno*** *costi accettabili dinamici*
lavorando in modo esatto *costi a volta inaccettabili*
o ***approssimato*** **ed** ***euristico***

ALLOCAZIONE dei PROCESSI

In particolare, i costi precisi sono fondamentali per

valutazioni statiche

si possono usare anche **algoritmi complessi** per il calcolo della allocazione (quindi **fuori linea** e **prima** del **deployment**)

algoritmi esatti di allocazione **problema NP**
algoritmi euristici **Genetici, Tabu search**

Spesso queste strategie sono troppo costose per essere applicate durante la esecuzione

valutazioni dinamiche

obiettivo ⇨ **riduzione dell'overhead**

Politiche semplici che devono rispettare la minima intrusione

Strategie locali e a basso costo realizzativo

LOAD SHARING (STATICO)

Considerando una *applicazione costituita di processi*

DETERMINARE

quali processi, quando allocarli, dove allocarli

alla ricerca di *processori liberi* in grado di essere utilizzati nelle implementazioni distribuite
(politica applicata solo alla creazione dei processi)

Organizzazioni possibili basate sulla architettura di interconnessione STATICA vs. DINAMICA

Processori in **ring logico**	statica
processori in **gerarchia logica MICROS**	dinamica
processori liberi (**worm**)	dinamica

LOAD SHARING

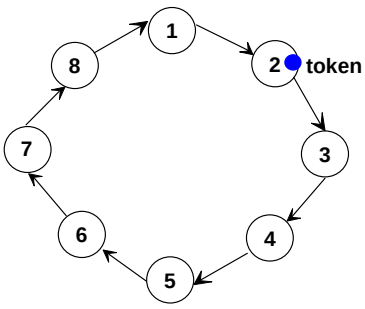
Ring logico

Si considera una struttura logica per la ricerca di spazio per eseguire i nuovi processi **attraverso un anello: sul ring circola un token che autorizzazione chi lo possiede a gestire la allocazione per tutti**, dirimendo problemi e conflitti

Il gestore usa un broadcast iniziale a tutti della richiesta di esecuzione: tutti rispondono in termini di disponibilità

si distribuisce il carico secondo le risposte

Struttura statica e proattiva
ma anche semplice da gestire e da controllare anche in caso di guasti



LOAD SHARING in MICROS

Gerarchia logica MICROS (sistema operativo distribuito)

Obiettivi

* gestione di un **numero di nodi molto elevato**
* numero di **utenti elevato** e di **applicazioni molto varie**
* **indipendenza dalla topologia**
* gestione della **replicazione**

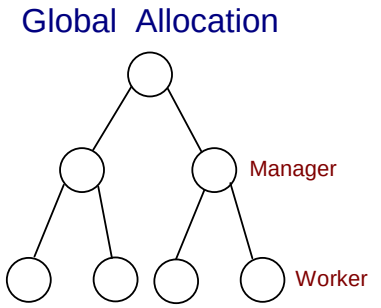
L'architettura è gerarchica ossia a livelli logici: non si prevedono connessioni dirette

Struttura a ruoli in una farm

Worker → funzioni di calcolo (**slave**)

Manager → funzioni di gestione

Numero di livelli dipende dal numero di worker



MICROS LOAD SHARING

Fault tolerance

master → più nodi master

slave → il master deve poter comandare i livelli sottostanti

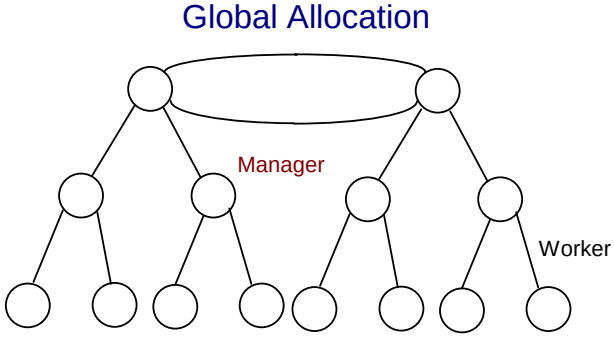
Allocazione statica

Si allocano un numero di processori worker adatti con i relativi manager

Allocazione dinamica

Si possono richiedere altri nodi di elaborazione nella gerarchia

al limite si possono chiedere nuove risorse al livello sovrastante



WORM LOAD SHARING

Worm come approccio dinamico innovativo

- parallelo, tollerante ai guasti, adattativo al rete ed alla topologia
- verso un bilanciamento nell'uso delle risorse

Worm/Verme un insieme di più segmenti, ciascuno un processo, che possono anche **comunicare tra loro** per **load sharing**

Obiettivo del worm

arrivare ad una copia del verme per nodo del sistema
installando la applicazione che sta al disopra (una volta sola)

Il verme si incarica di cercare i nodi liberi attraverso clonazione su nodi liberi, usando messaggi di richiesta e di accettazione (detti probe) mandati dai segmenti che vogliono espandersi

Metodologia decentralizzata

NON si conosce lo stato del verme globalmente

LOAD BALANCING (DINAMICO)

OBIETTIVI della **MIGRAZIONE AUTOMATICA**

- uso delle **risorse più CORRETTO ed EFFICIENTE**

  disomogeneità delle risorse fisiche e decisioni di allocazione delle logiche

- **BILANCIAMENTO del carico computazionale**

  divisione del carico tra i nodi secondo un principio di equilibrio

- **DINAMICITÀ e MOBILITÀ**

  possibilità di fare fronte a una allocazione anche non ottimale, o non più ottimale

anche possibilità di **sopravvivenza a guasti, se copie della risorsa**

Requisiti

Performance	buon uso delle risorse
Efficienza del sistema	overhead limitato
Trasparenza o meno	al livello applicativo: spesso gestione automatica

MONITORING

Il MONITORING come mezzo di controllo e gestione

Identificazione del carico del sistema usando osservazioni sul carico corrente, assumendo continuità della applicazione e gradienti limitati

Raccolta di **informazioni di carico** su
processori, risorse e comunicazione

* ad **eventi**
* **dati statistici e storici**
* osservazioni su **intervallo limitato**

Le informazioni monitorate sono usate per la previsione delle variazioni del carico nel futuro immediato (**ipotesi di continuità**)

Necessità di **limitare le informazioni** da osservare e mantenere ***per limitare intrusione*** **(minima intrusione)**

MIGRAZIONE - Considerazioni

MIGRAZIONE di RISORSE

DATI, di OGGETTI, di AGENTI, …MIGRAZIONE DI PROCESSI

come entità computazionali mobili

PROCESSI spostati da un nodo ad un altro

il processo composto da: lo stato iniziale + i cambiamenti

un sottoinsieme deve essere trasferito da un nodo ad altro

Pre-emption

priorità nell'uso locale

Migrazioni multiple

consentire concorrenza nelle migrazioni per parallelizzare

Evitare dipendenza residue

dispendioso lasciare traccia nel sistema delle migrazioni

Evitare thrashing

movimento di un solo processo che non riesce a proseguire

PROBLEMI in MIGRAZIONE (INTERNI)

In caso di migrazione, il processo deve preparare la fase di mobilità e gestire tutte le risorse che vedeva precedentemente

→ Cambiamento di ambiente della risorsa mobile

- Identificazione dello stato

Il processo deve determinare quali siano le risorse interne da trasportare sulla nuova locazione e cominciare a determinarne lo stato

- Azioni di blocco del processo stesso

Il processo potrebbe avere una parte dello stato non trasportabile da chiudere

Azioni di chiusura di file o code trasportabili (ultime volontà)

Azioni di memorizzazione di risorse che possano dovere essere ritrovate nel nuovo nodo

- Blocco della attività per mobilità

Fine della attività sul vecchio nodo e meccanismi di movimento sul nuovo nodo

PROBLEMI nella MIGRAZIONE (ESTERNI)

In caso di migrazione, durante e dopo la migrazione

… i messaggi in fase di consegna e da spedire a chi migra

→ Cambiamento di nome della risorsa mobile

- Ridirezione dei messaggi *strategia pessimista/proattiva*

Il supporto tiene traccia e bufferizza i messaggi che poi inoltra

Il nodo di partenza tiene traccia della allocazione del processo

i clienti mandano al vecchio nodo

- Riqualificazione dell'allocazione *strategia pessimista/proattiva*

i messaggi per il processo sono mantenuti ma i clienti sono informati della nuova allocazione

Il nodo di partenza tiene traccia della allocazione del processo solo fino al completamento del trasferimento

- Recovery da parte dei clienti *strategia ottimista/reattiva*

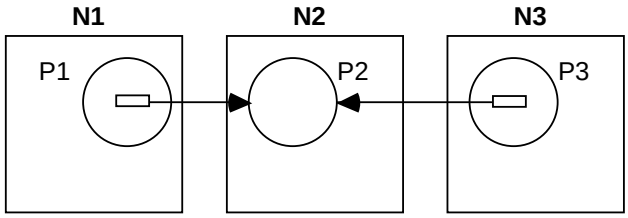
Non si mantiene la nuova allocazione del processo e non si informano i clienti. Il messaggio può fallire: è compito del cliente di ritrovare la nuova allocazione

SISTEMI PIONIERI: DEMOS/MP

DEMOS/MP (1977) / **M**igration **P**rocesses sistema pioniere
Kernel di sistema distribuito
- sistema a **scambio di messaggi**
-comunicazione attraverso entità intermedie (***link***)
  - i link permettono di comunicare con un processo ricevente
  - i link sono **associati** ai processi riceventi
  - i link possono essere **passati** nei messaggi

Caso di **MIGRAZIONE**
un processo P2 ciclico (che non termina) si sposta dal nodo N2 a N3

Situazione Iniziale



protocollo migrazione di DEMOS/MP

fasi di:
- **Blocco di P2**
- **Trasferimento** dello stato di P2 da N2 a N3
  codice e dati (stato iniziale)
  stato (stato corrente)
  messaggi
- **Uso dei vecchi link** da parte dei processi mittenti
  resta in N2 una entità (forwarder) che fa procedere i messaggi

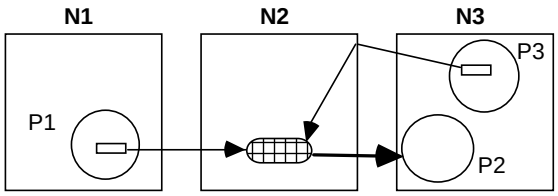
in alternativa: scarto messaggi e problema della ritrasmissione
- **Riqualificazione dei link**
  la informazione della nuova locazione può essere sostituita

A riqualificazione completata di tutti i link, il forwarder scompare

Costo migrazione + costo updating link
Requisito TRASPARENZA della allocazione

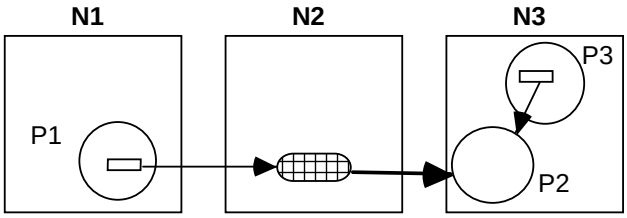
MIGRAZIONE in DEMOS/MP

Situazioni Intermedie

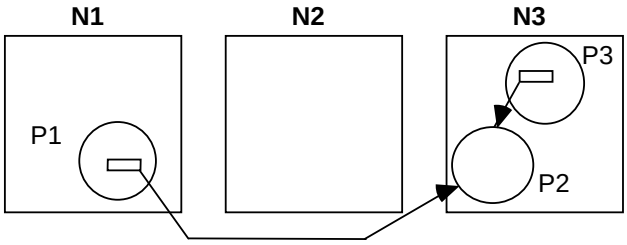


Il forwarder invia alla nuova locazione di P2 …

Riqualificazione del link da P3 a P2



Situazione finale

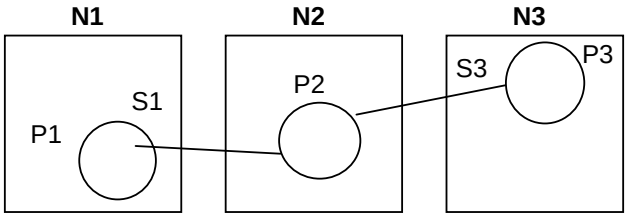


SISTEMI PIONIERI: V-Kernel

V-kernel (1983) Kernel leggero a scambio messaggi
obiettivi: **efficienza e tolleranza guasti**
Sfruttare i nodi idle per eseguire processi
- non tutti i processi sono mobili
- sistema a scambio di messaggi con comunicazione diretta
- primitive di scambio messaggi per favorire sincronicità e C/S

send	receive
<attesa>	<operazione>
receive	send

Situazione Iniziale: un processo **P2** (ciclico) si sposta da un nodo **N2** a **N3**



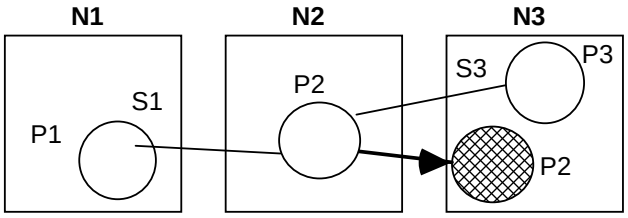
PROTOCOLLO di V-Kernel

- **PreCopia** dello stato di **P2 da N2 a N3 fatta in precedenza**
  Copia iniziale durante la esecuzione e pre-copying in fasi successive delle pagine modificate
- Al blocco di **P2**
  il trasferimento dello stato viene completato: i messaggi che il processo ha ricevuto parte del suo stato e quindi sono copiati
- **Ritrasmissione periodica degli altri messaggi**
  ancora i processi relativi in attesa
  Il processo scarta i messaggi che il sender provvede a reinviare
- **Riqualificazione** delle cache di indirizzo
  in genere via broadcast

TRASPARENZA della ALLOCAZIONE
attraverso l'accesso al kernel locale

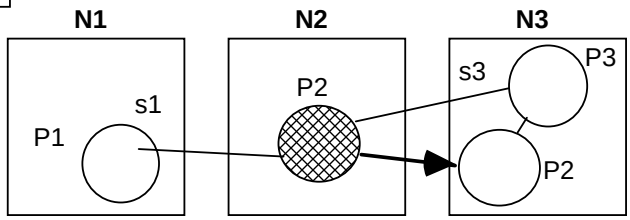
MIGRAZIONE in V-KERNEL

Situazioni Intermedie

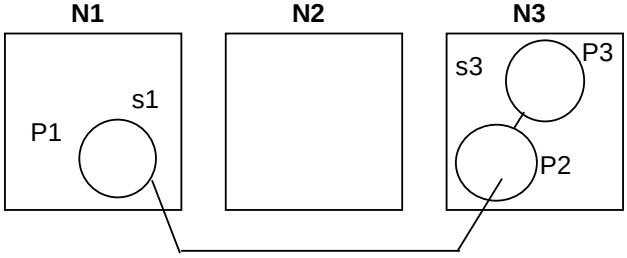


Il processo si pre-copia nella nuova locazione di P2

Riqualificazione dei link da P3 a P2



Situazione finale



MIGRAZIONE

Basata sia su **politiche**, sia su **meccanismi** di supporto

MECCANISMI

dipendenti dal modello computazionale e dallo specifico sistema

POLITICHE

indipendenti dal sistema e general-purpose (non embedded)

Criteri sensati

- ***non tutti i processi migrano***

fissi gli aciclici e i dipendenti dal nodo

- *è opportuno un* ***gestore della migrazione*** *per ogni nodo*

DETERMINARE (per i **processi**)

chi, quando, come, dove migrare

MECCANISMI di MIGRAZIONE

Chi migra?

processi, oggetti passivi (**file**), oggetti attivi, agenti

Costituzione della RISORSA

codice + dati (stato iniziale)

dati (stato corrente)

risorse cui accedere (locali e remote)

Blocco computazione

blocco dello stato dall'esterno: messaggi trasferiti o rifiutati

Trasferimento e Copia

sincronizzazione tra copia vecchia da distruggere e nuova da attivare

Riferimenti obsoleti

riqualificazione o variazione su necessità

COSTO MECCANISMI di MIGRAZIONE

Charlotte - Migrazione di Processi

Tempo$_{\textbf{Migraz}}$ (in msec) = 45 + 12.2 **n** + 9.9 + 1.7 **ln**

n numero di 2 kbyte del processo
ln numero link remoti

(Charlotte)
reliable message send **2kbyte** = **11**ms

tempo migrazione	**processo con**
242 ms	*32 Kbyte senza link*
750 ms	*100Kbyte con 6 link*
6 s	*1 Mbyte*

POLITICHE di MIGRAZIONE

FASI costituenti

VALUTAZIONE carico (**V**)
carico locale vs. globale

TRASFERIMENTO (**T**)
chi trasferire e quando trasferire

LOCAZIONE (**L**)
dove migrare e reinserire il processo

T e **L** sono spesso legate e interdipendenti

Interazione e integrazione con SCHEDULING locale
impatto sullo scheduling del nodo di partenza e del nodo di arrivo
in competizione con risorse locali

POLITICHE di MIGRAZIONE

STATICHE **predefinite** e decise a priori (**basso costo**)

V *carico soglia fissa* (p.e., numero processi)

T movimento del *processo più "nuovo"*

L migrazione da un *nodo sorgente* sempre a un *definito destinatario*

SEMIDINAMICHE predefinite con **limitate dipendenze** dallo **stato corrente** - o anche politiche probabilistiche (costo limitato)

V *carico soglia variabile*

T *identificazione ciclica* tra i processi

L allocazione su *destinatario ciclico*

DINAMICHE strettamente **dipendenti dallo stato corrente**

V confronto con *carichi dei nodi vicini* (carico medio dinamico)

T informazioni sullo *stato dei processi* e decisioni derivate

L ricerca dei *potenziali nodi destinatari con messaggi nel vicinato*

POLITICHE di MIGRAZIONE

POLITICHE SEMPLICI vs. COMPLESSE

V T L per processi **aciclici** vs. **ciclici**

V → a soglia fissa vs. confronto con vicini

T → processo adatto per vicino predeterminato o random

L → uso di **messaggi probe**

random, probabilistiche, cycle, shortest queue

accettazione incondizionata

probing, bidding

accettazione condizionata

probe: verifica di possibilità su vicini **PROBING** (**T e L congiunti**)
identificazione di alcuni candidati a ricevere i processi e valutazione del loro stato

BIDDING (Contract Net)

Protocolli di cooperazione - BIDDING (Contract Net)

protocollo di negoziazione tra nodi coinvolti per cooperare

SENDER-INITIATIVE (**source**)

1) il sender fa un **broadcast** della propria esigenza (**announce**)
2) i possibili receiver offrono la propria **disponibilità** con un (**bid**)
3) il sender **sceglie un receiver** tra i bidder
4) il receiver dà finalmente l'ok **definitivo** (**contract**)
5) **trasferimento** del carico

Il sender non prenota i receiver potenziali che hanno aderito alla richiesta in base allo stato locale corrente

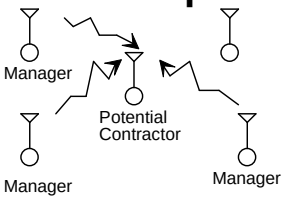
Possibili altre scelte in caso di rifiuto e molti round possibili

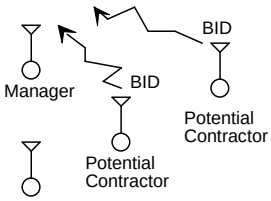
Politiche di negoziazione molto flessibili ma
anche scelte finali non ottimizzate globalmente

PROTOCOLLO di BIDDING

Protocolli a fasi multiple

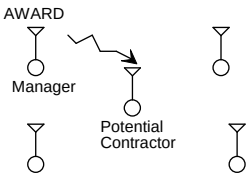
1) Announce richiesta

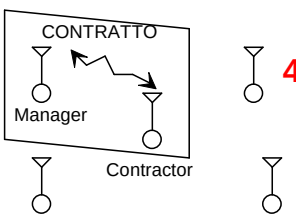




2) Fase di Bidding

3) Scelta del Bidder





4) Contratto finale

		SEND1	SEND2
bid	REC1	0.9	0.7
	REC2	0.7	0.1

DECISIONI di MIGRAZIONE

CENTRALIZZATA entità unica di controllo delle migrazioni

DECENTRALIZZATA coordinamento di entità

raccolta implicita o esplicita di informazioni e decisione distribuita

basata su confronto di informazioni di stato (piggybacking)

favorendo movimenti su base locale degli altri nodi (vicinato)

RESPONSABILITÀ coppia SENDER-RECEIVER

Iniziativa del SENDER: il nodo carico si preoccupa di trovare un opportuno ricevente (RECEIVER)

Iniziativa del RECEIVER: un nodo scarico trova potenziali attività da eseguire identificando il mittente (SENDER)

schemi MISTI

SENDER initiative → più adatta a carichi bassi

RECEIVER initiative → più adatta a carichi medi-elevati

MIGRAZIONE - fattibilità

RISULTATO IMPORTANTE

Anche con politiche semplici si ottengono significativi miglioramenti rispetto al caso senza migrazione

Anche chiudendo tutte le parti di sistema alla migrazione (socket e file), e adottando politiche statiche e semplici

***Politiche più sofisticate* NON ottengono *miglioramenti significativi* tali da controbilanciare la complicazione introdotta**

- **STABILITÀ**
  evitare thrashing
- **EFFICIENZA**
  algoritmo decisione ed attuazione
- **OTTIMALITÀ**
  subottimalità

MIGRAZIONE

Molti sistemi hanno proposto prospettive diverse
Per approfondire a molti livelli la migrazione

*Sistemi con **algoritmi sofisticati** tipo BID*
*Sistemi con **politiche globali***
*Sistemi con obiettivo la **efficienza***
*Sistemi che lavorano in **ambienti eterogenei** (compilati)*
*Sistemi che lavorano in **ambienti interpretati omogenei***
*Sistemi ad **oggetti***

… ancora
Sistemi mobili
***Mobile Computing** ed evoluzioni*

MIGRAZIONE e AGENTI MOBILI

MIGRAZIONE DI **AGENTI MOBILI**

nei modelli di movimento con **agenti mobili** definiamo **attività in cui il movimento è un requisito** ed è **deciso dalla applicazione stessa**
l'obiettivo non necessariamente legato al bilanciamento del carico o a considerazioni di uso di risorse
ma derivato da **specifiche precise di applicazione**

Mobilità motivata da necessità di consultazione di risorse

sistemi mobili e geografici con coordinamento
sistemi globali (basati su Web ed Internet)

criteri

- **gli agenti possono muoversi anche ritornando su nodi già visitati** (per riportare informazioni trovate)
- **non ci sono vincoli di costo** nella decisione di migrazione: la mobilità necessaria in ogni caso
- si devono realizzare **meccanismi efficienti**

MOVIMENTO di CODICE

Verso modelli di mobilità di codice

Oltre al modello cliente servitore, in cui si scambiano dati tra il cliente ed il servitore,e questo fornisce il servizio e risultato

1) Remote Evaluation (REV)

il cliente **manda al servitore** anche (parte del) **codice** da **eseguire** nel servizio corrispondente
(invio di pezzi di codice nuovo)

2) Code On Demand (COD)

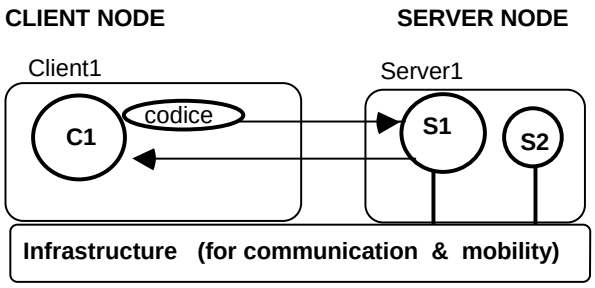
il cliente **richiede dal servitore il codice da eseguire** per un **servizio** eseguito localmente
(Applet Java)

Spostamento permanente di codice e nuova installazione del codice (one-hop)

MOVIMENTO di CODICE

REV il codice si muove sul **servitore** a riqualificarlo e a **fornire comportamenti prima non noti**

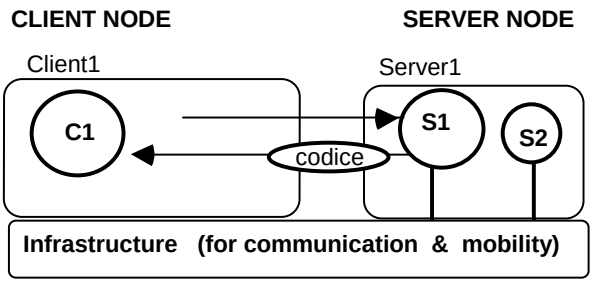
Nel settore degli apparati di telecomunicazioni, si cominciano a considerare migrazioni di codice anche per router e bridge (**reti intelligenti** e **reti attive**)



REMOTE EVALUATION

COD il codice viene portato alla **esecuzione sul cliente**

Si ottiene sempre la **versione più fresca del codice sul cliente** che non deve occuparsi degli aggiornamenti



CODE ON DEMAND

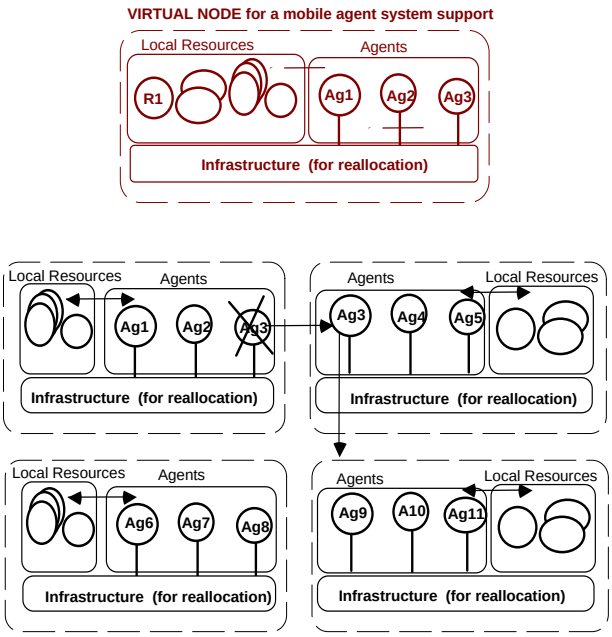
AGENTI MOBILI

SISTEMI ad AGENTI MOBILI
tecnologia in cui, **una o più attività, possono muoversi**, dopo avere cominciato la esecuzione su un nodo determinato, **cambiando la propria allocazione** e **mantenendo lo stato** già raggiunto

Sistemi ad entità multi-hop

L'agente si porta dietro stato e codice che rappresentano la sua storia

Vantaggi in caso di esecuzione locale e sintesi dei risultati che viene trasportata solo su necessità



ALLA BASE DEL CORSO

Nei **sistemi distribuiti** siamo interessati alla **esecuzione** e **operatività**

Ci aspettiamo che ci sia sviluppo prima della esecuzione

Ad esempio, ci sono classi sviluppate separatamente che favoriscono ulteriori sviluppi statici, ma che poi sono da considerarsi un'unico contributo durante la esecuzione

Allora queste non ci interessano

Ci interessa tutto quello che ha impatto durante la esecuzione e che rimane significativo e vitale durante questa, favorendo e abilitando la distribuzione (*e facendoci capire in che modo*)

Ad esempio, ci sono **classi** che poi diventeranno **processi e componenti attivi** e distribuiti per tutta la durata della applicazione: sono i processi che ci interessano e che rappresentano una parte della architettura del sistema run-time

La architettura dinamica ci interessa, e ci interessa capire come e quanto bene funziona

ANCORA DI BASE …

Nei sistemi distribuiti siamo interessati alla **performance** e **qualità**

*Ci aspettiamo che ci siano **risorse implicate** e **casi particolarmente significativi** per una **architettura considerata***

Ad esempio, l'uso di RMI ha un impatto molto forte sul costo e sulla scalabilità del sistema complessivo

Mentre l'uso diretto di socket e strumenti di più basso livello garantisce minore overhead e maggior controllo

Durante la esecuzione ci interessano i colli di bottiglia**, ossia **i punti critici** e le **parti che possono determinare un comportamento del sistema poco adatto o carente

Ad esempio, usare uno strumento come una RMI (o una richiesta remota costosa) per un messaggio scambiato in modo occasionale una tantum (e magari solo una volta) tende a introdurre un potenziale bottleneck da considerare e da controllare in un progetto

La architettura va verificata a priori e a posteriori sul campo e quantificando la esecuzione

MODELLI di ESECUZIONE …

Nei sistemi distribuiti siamo interessati a **operatività**, **performance**, **reale esecuzione distribuita**

Uso di modelli preventivi/reattivi

Comportamenti **preventivi** evitano a priori eventi o situazioni, ma introducono spesso un **costo fisso** sul sistema (spesso calcolabile)

Comportamenti **reattivi** permettono di introdurre minore logica (e **limitare il costo**) in caso gli eventi non si verifichino

Uso di modelli statici/dinamici

Comportamenti **statici non** permettono di **adeguare il sistema** a fronte di **variazioni** (limitate)

Comportamenti **dinamici** permettono di **fare evolvere il sistema** a fronte di variazioni (limitate) ma tendono a causare **costi di gestione più elevati (overhead)**

MODELLI DINAMICI e STATICI

Modelli dinamici / Modelli statici

Il numero degli utenti di una applicazione è predefinito

Gli utenti si possono **aggiungere/togliere**

Il numero dei processi di una applicazione è predefinito

Il numero dei processi può **cambiare durante l'esecuzione**

Il numero dei nodi massimo è predefinito

I processori partecipanti possono anche **aumentare molto**

Il numero dei clienti di un servizio è predefinito

Il numero dei servizi (throughput) non predefinito

I servitori sono noti e predefiniti

I servizi (servitori) devono cambiare in numero e tipo

Servitori intermedi catalogano e attivano servitori

MODELLO di ESECUZIONE

Esecuzione del sistema anche senza applicazioni

Infrastruttura presente sulle macchine

Esecuzione di una o più applicazioni

Monoutente: l'uso di un sistema in modo dedicato è tipico delle fasi prototipale

Multiutente: più utenti consentono di formare un migliore mix di entità eseguibili sul/sui sistemi

modello ***workstation***

si utilizzano preferenzialmente le risorse locali

modello ***processor pool***

si utilizzano le risorse in modo trasparente a secondo del loro utilizzo e disponibilità

Come si comandano le risorse e come sono attribuite?

MODELLI a RISORSE

Alcune risorse (logiche) durante la esecuzione

Processi entità attive capaci di esecuzione di

- **azioni locali** su un ambiente proprio e
- **azioni di comunicazione** con altri processi attraverso *memoria condivisa* e *scambio di messaggi*

Uso di *dati esterni* ai processi stessi (scarso confinamento)

Oggetti entità introdotte per la astrazione, come capacità di

- racchiudere e nascondere **risorse interne** (astrazione dei dati) con **visibilità esterna** delle **operazioni**
- agire su risorse interne alla richiesta di operazioni dall'esterno

Oggetti passivi **astrazioni di dato** su cui eseguono entità esterne

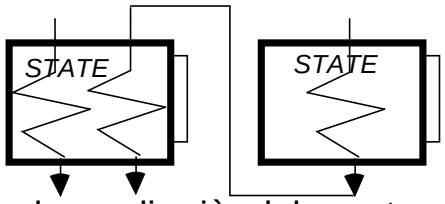
Oggetti attivi **entità attive capaci** di **esecuzione** e **scheduling**

ESECUZIONE e OGGETTI

Oggetti passivi

I modelli ad **oggetti PASSIVI** prevedono che i processi (*esterni agli oggetti stessi*) possano entrare **sugli oggetti** e richiedere la esecuzione dei metodi

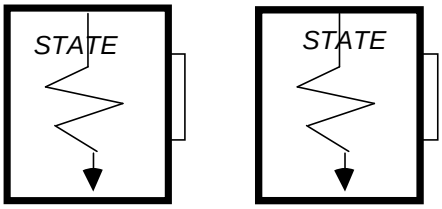
- modello a scarso confinamento
- scarsa protezione
- interferenza tra diversi processi



Oggetti attivi

I modelli ad **oggetti ATTIVI** chiudono di più dal punto di vista della esecuzione: i processi esterni non possono entrare ma solo presentare richieste. Chi esegue sono **solo processi interni** all'oggetto attivo

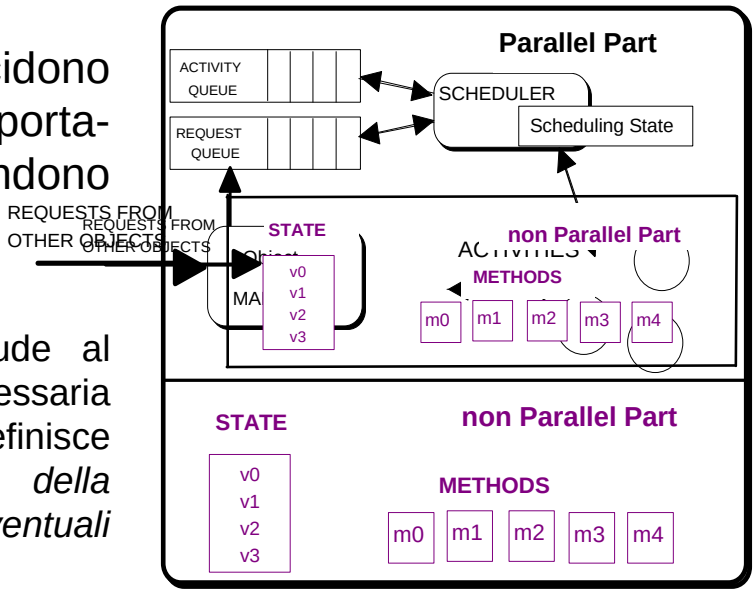
- oggetti protetti
- completa determinazione



OGGETTI ATTIVI

Oggetti attivi

Gli Oggetti Attivi decidono indipendentemente il comportamento interno e lo nascondono confinandolo



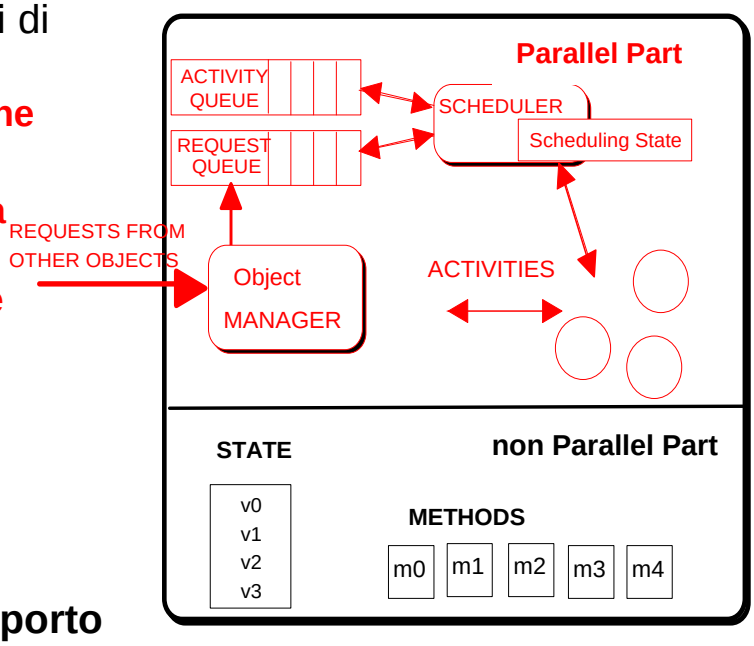
Ogni **oggetto attivo** racchiude al proprio interno la necessaria **capacità di concorrenza** e definisce la *propria gestione locale della concorrenza*, prevenendo *eventuali interferenze*

Quanti processi interni?

OGGETTI ATTIVI

Il supporto aggiunge in modo **automatico** tutte le **funzioni** di autodeterminazione

- La coda delle **richieste esterne**
- La coda delle **attività interne**
- La realizzazione della **politica** di scheduling
- Le **gestione** della **attivazione** dei processi interni
- La gestione della loro **terminazione**
- La consegna dei **risultati**
- La **gestione** degli **errori**



Meccanismi forniti dal supporto e Politiche lasciate all'utente

OGGETTI e CLASSI - digressione

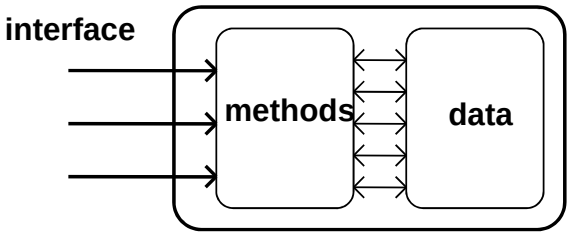
Oggetti passivi come **dati e metodi**

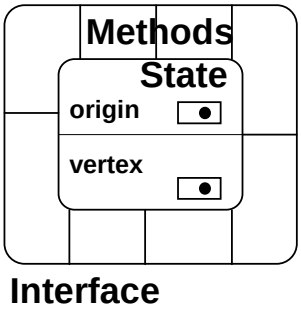
I **metodi** sono resi visibili dalla interfaccia ed invocabili dall'esterno

I **dati** sono tipicamente protetti e non visibili dall'esterno

I dati (attributi) sono

- **dati primitivi** (esempio un intero di nome **origine**)
- **riferimenti ad altri oggetti** (esempio un link tipizzato di nome vertice ad un altro oggetto)





OGGETTI SENZA CLASSI (Attori)

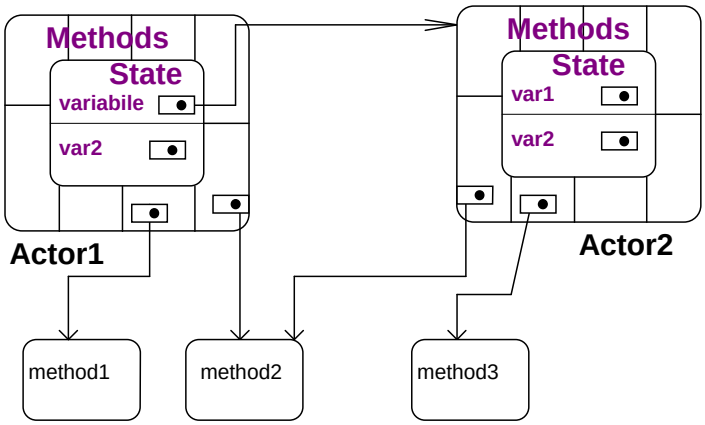
I primi sistemi hanno previsto sia **classi**, come entità descrittive delle istanze, sia la mancanza di **componenti di metalivello** (sistemi classless) con la possibilità che **ogni istanza sia un prototipo a se stante senza nessuna entità descrittiva generale**

In **sistemi ad attori**, detti anche **prototipali**, ogni oggetto è responsabile della propria descrizione e contenuti

Un attore deve riempire i **propri contenuti** sia **variabili** sia **metodi**

Ogni attore è **responsabile** del propri**o intero compor-tamento**

Può quindi cambiare durante la esecuzione i propri metodi e variabili



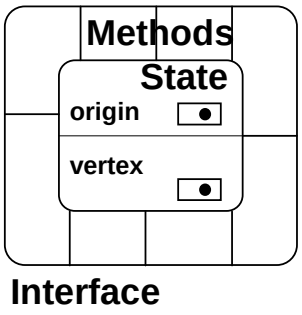
Sistemi meno usati, ma più facilmente provabili formalmente

SISTEMI CON CLASSI (non solo linguaggi)

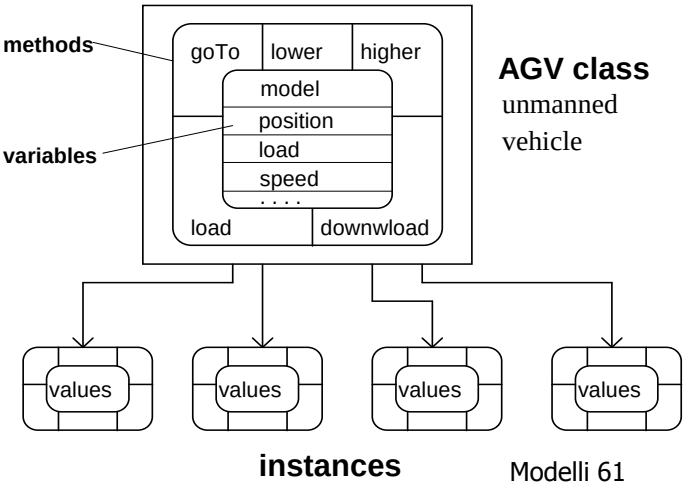
Nei sistemi con **classi**, ci sono le **classi**

- che contengono realmente i metodi in modo unico per tutte le istanze
- che specificano quali dati abbia ogni istanza e di che tipo (*primitivo o meno*)

se i tipi **non sono primitivi** ma **altri oggetti**, la **classe** specifica di che classe devono essere i riferimenti



Ci sono le classi a run-time?



CLASSI negli AMBIENTI RUN-TIME

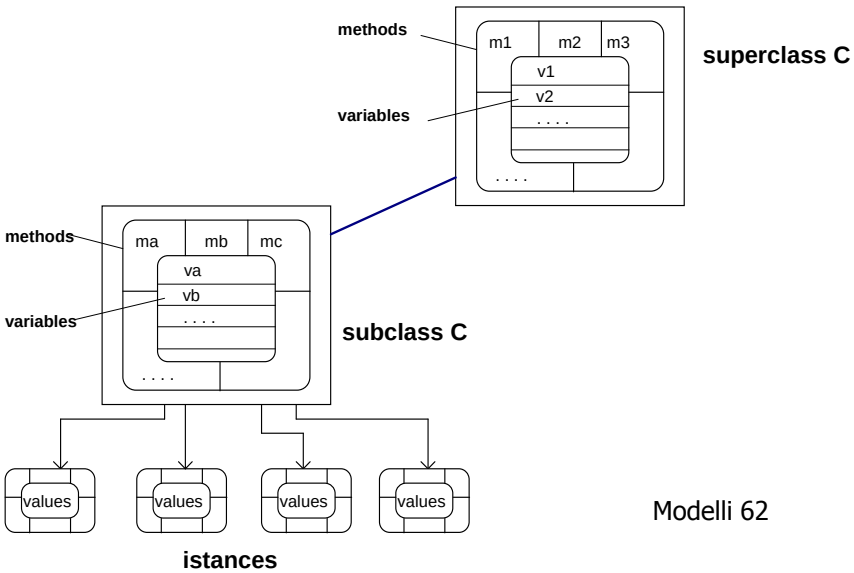
Le classi **sono presenti a run-time** e **caricate in modo dinamico**

Vengono caricate in un heap per ritrovare i metodi e le variabili statiche a run-time

Anche gli oggetti sono caricati in modo dinamico (**heap**) e fanno riferimento alla propria classe per il proprio comportamento

Le classi sono spesso legate dalla **relazione di ereditarietà** e si prevede che una istanza faccia riferimento a **una classe** e **molte superclassi**

(si può chiamare sia m1 di SC sia ma di C)



Ereditarietà SEMPLICE vs. MULTIPLA

Nei linguaggi OO, tra classi si prevede **ereditarietà**

Ereditarietà multipla (genitori multipli)

Ereditarietà semplice (singolo genitore)

tra classi (anche per interfacce)

Vantaggi? Svantaggi?

- capacità espressiva,
- semplicità di espressione
- possibilità di mapping di situazioni specifiche
- estensibilità ad altre proprietà (Aspect Oriented Programming)
- semplicità di supporto
- minore overhead

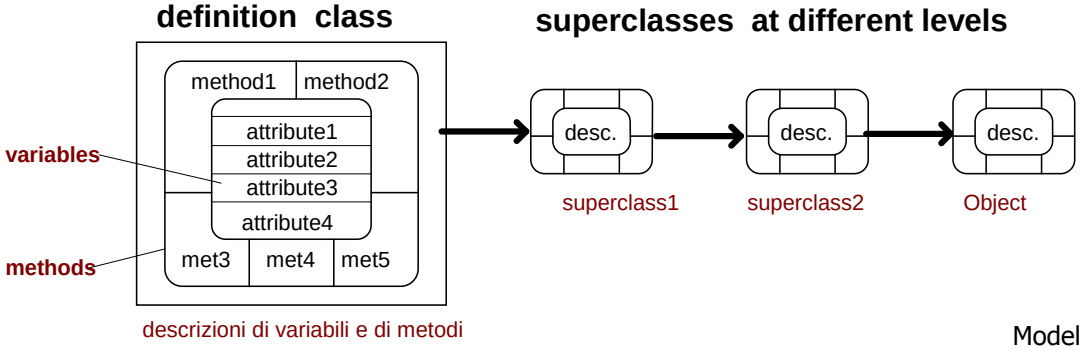
accettazione ed uso da parte degli utenti

EREDITARIETÀ SINGOLO GENITORE

Nei linguaggi OO con ereditarietà semplice tra classi (Java, e molti altri linguaggi OO, …), la ereditarietà produce un albero di classi, la ricerca dinamica di un attributo avviene attraverso una esplorazione dinamica su una lista di classi (dalla corrente)

In questi sistemi, la ricerca del metodo da eseguire è semplificata

- Abbiamo una catena di classi da ricercare e in ordine prestabilito risalendo la catena
- Vale l'**overriding** tenendo conto della **catena di ereditarietà**
- nei sistemi concentrati

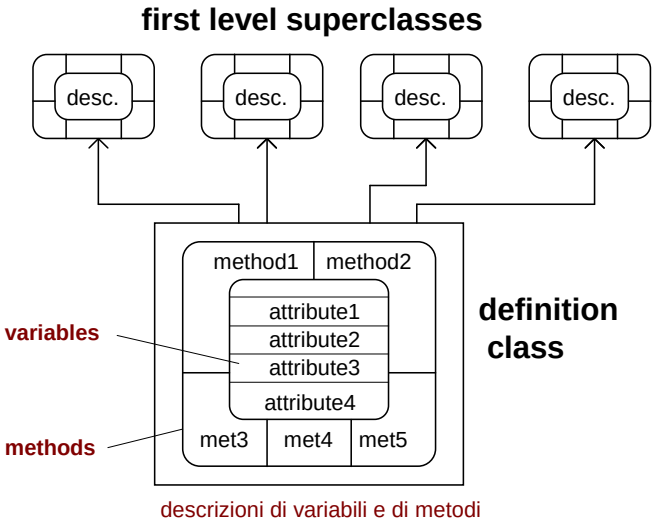


EREDITARIETÀ MULTIPLA

Nei linguaggi OO con ereditarietà multipla tra classi (C++, Vbasic, … e supporti derivati), la ereditarietà produce un grafo di classi e la esplorazione dinamica è su un albero di classi (dalla corrente)

In questi sistemi, la ricerca del metodo da eseguire diventa più complessa: abbiamo molteplici classi da ricercare e in ordine stabilito dall'ordine di ereditarietà

Valgono **Overriding** e **ordine in Ereditarietà** nei sistemi concentrati



CLASSI vs. INTERFACCE

Nelle moderne architetture:

- le **interfacce** sono il **contratto di interazione, unico e non negoziabile**

-le **classi** descrivono le diverse **implementazioni** (e ce ne possono essere molte e differenziate per qualità nel sistema)

Nei sistemi distribuiti si è diffusa da molto tempo la idea di avere **interfacce** come **contratto tra diversi interessati** - che fanno anche sviluppo indipendente - e a tenere queste separate dalla specifica implementazione (magari molteplice)

I middleware sono basati su **interfacce** e meno su **classi** (e altre loro implementazioni differenziata, come i componenti)

Nei linguaggi OO, questa **separazione è arrivata dopo**, ma nei **moderni linguaggi** è stata incorporata rapidamente, specie nei **linguaggi progettati per il distribuito**

CLASSI vs. INTERFACCE

Le **interfacce** sono il contratto tra partecipanti (in Java sono anche entità del programma)

Le **classi** descrivono le implementazioni specifiche (e sono uniche entità descrittive del comportamento delle istanze)

Nei linguaggi OO spesso si prevede **ereditarietà di entrambe**

- Ereditarietà multipla (multiplo genitore) - naturale per interfacce
- Ereditarietà semplice (singolo genitore) - tipica per classi

In Java abbiamo **interfacce in grafo** - e classi che possono implementarne molteplici – e **classi in catena di ereditarietà**, semplificando molto il supporto per le istanze:

- Un'istanza richiede il **caricamento solo di una classe** (di definizione) e di tutte le super**classi. In ogni (istanza di una) classe, i metodi sono in posizione predeterminabile.**
- Ogni metodo si trova dinamicamente in modo semplice **scorrendo la catena delle classi a partire dalla classe di definizione** (facilità nella produrre codice statico e nel supporto dinamico)

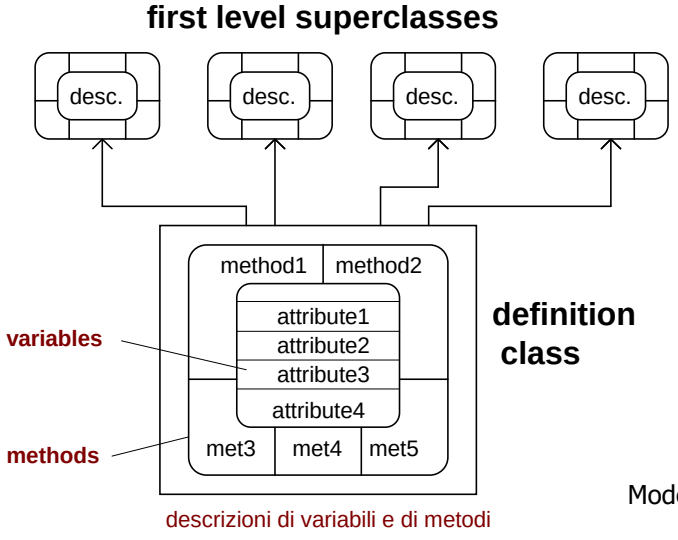
EREDITARIETÀ MULTIPLA

Nei sistemi OO con ereditarietà**, i metodi devono riferire gli attributi correnti** della istanza (descritta dalle classi)

- Ogni istanza quanti attributi e metodi ha?
- Il metodo metodo2 della classe corrente può riferire un attributo di una superclasse?
- Il metodo di una superclasse può riferire un attributo di una sottoclasse?

Spesso uso di notazioni per riferire **attributi specifici** e **metodi specifici**

Lo stesso problema per trovare i metodi da eseguire



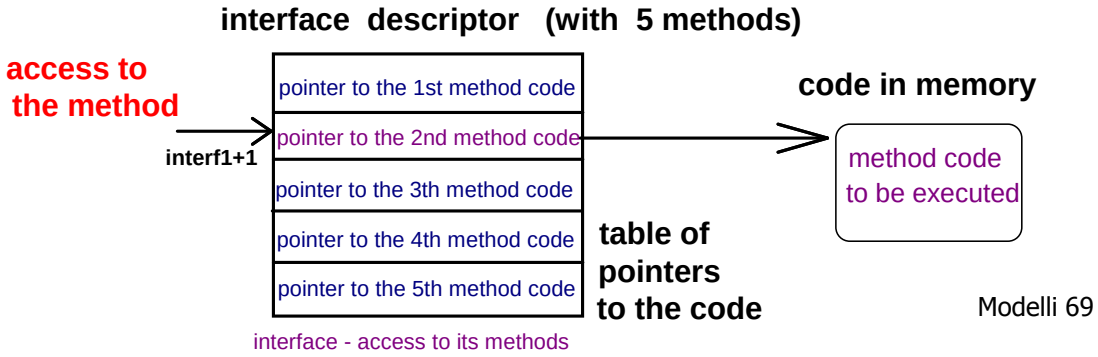
INTERFACCE e METODI

In generale, in sistemi con codice, non solo statico, in **architetture distribuite**, la **interfaccia** è un **raggruppamento di metodi**

A livello di supporto, diventa opportuno avere un **legame dinamico** con il codice dei metodi e ritrovarlo in modo aggregato per **interfaccia** (o **classe**) attraverso **un supporto runtime**

La interfaccia **diventa (in ambiente run-time)** una **tabella** che elenca i **puntatori ai metodi** da eseguire e che viene associata ad ogni oggetto o componente per trovare il codice

L'accesso avviene in modo indiretto attraverso il puntatore in posizione nota e facile da calcolare (statica)



EREDITARIETÀ MULTIPLA e DISTRIBUITO

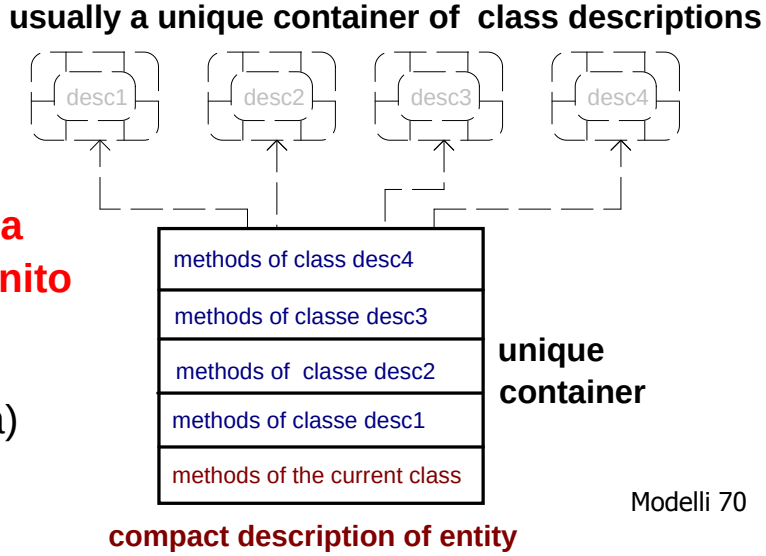
Nei supporti con ereditarietà multipla e in architetture distribuite, molto spesso la **classe** (o il **componente** o l'**interfaccia**) diventa **un contenitore unico** che raggruppa tutta la descrizione della entità, per ragioni di efficienza e di maggiore località

Diventa difficile determinare in modo statico la posizione di un metodo specifico

I metodi possono essere riferiti solo tenendo conto del **mappaggio per la classe specifica**

e non in modo **predefinito**

(un metodo ritrovato sempre attraverso la sua classe / interfaccia)



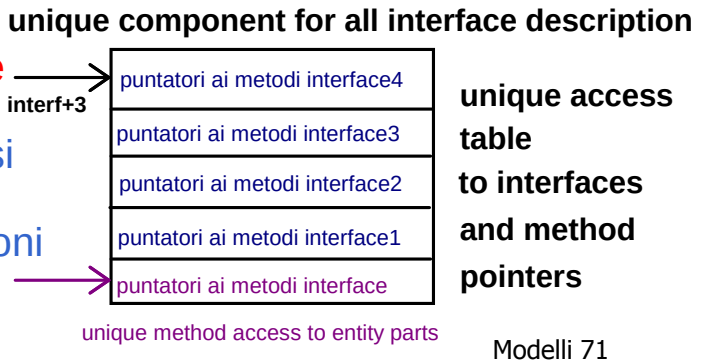
CLASSI MULTIPLE e USO dei METODI

Ancora con ereditarietà multipla e in architetture distribuite,

- In caso di **compilatori**, gli attributi devono essere risolti per ogni classe prima della esecuzione, con un mapping statico dei nomi (non sempre si riesce)
- In caso di **interpreti (o di ambienti run-time)**, è più complesso risolvere ogni entità e lo si deve fare durante la esecuzione
- **Necessità di trovare un modo di accedere al codice dei metodi in modo generale e da fare a runtime (per mantenere il codice)**

Come farlo?

Uso di una tabella degli indici dei metodi divisa per interfacce che permette di dare un accesso veloce ai metodi stessi durantel'esecuzione senza avere calcolato prima le posizioni delle classi e dei metodi stessi



CLASSI MULTIPLE e USO dei METODI

Con componenti con ereditarietà multipla e in architetture distribuite,

- I **metodi sono ritrovati dinamicamente in modo indiretto**, in ogni caso, attraverso l'accesso ad una tabella dei metodi (e interfacce)
- La **tabella registra tutte le classi per quel componente** e per ciascuna inserisce a run-time **il puntatore al metodo stesso**
- L'accesso indiretto è spesso usato in sistemi dinamici

Il ritrovare il metodo richiede due passi di ritrovamento:
- nel componente la **posizione** della **classe di interesse** (**dinamico**)
- di ritrovare lo **spiazzamento** del metodo ricercato (**statico**)

Si può trovare ogni metodo in base alla **posizione corrente** della **interfaccia** nella classe corrente di riferimento

I metodi di ogni classe sono ritrovati partendo dalla classe stessa in cui sono riferiti, isolandola e considerandola in modo separato

Dobbiamo avere un modo di isolare la classe e di poterla riferire
Una API potrebbe essere

```
ClassofInterest GetClassRepresentation(Classe)
```

INTERFACCE MULTIPLE e COMPONENTI

Tutto quello che abbiamo detto per ereditarietà multipla di classi in architetture distribuite, si può applicare al caso di

componenti che implementano delle interfacce di metodi legacy predefiniti

come i componenti DCOM distribuiti

I metodi di ogni interfaccia sono ritrovati partendo dalla interfaccia in cui sono riferiti, isolandola e considerandola in modo separato

Abbiamo una API che isola in un componente corrente, una interfaccia di interesse

```
HRESULT QueryInterface(const GUID Identifier;
      out void ** InterfaceIsolatedPtr);
```

Ogni componente deve supportare la QueryInterface…

che è parte di una interfaccia di ogni componente

Anzi, che è parte della interfaccia di base di ogni componente

SEMANTICA per RIFERIMENTO

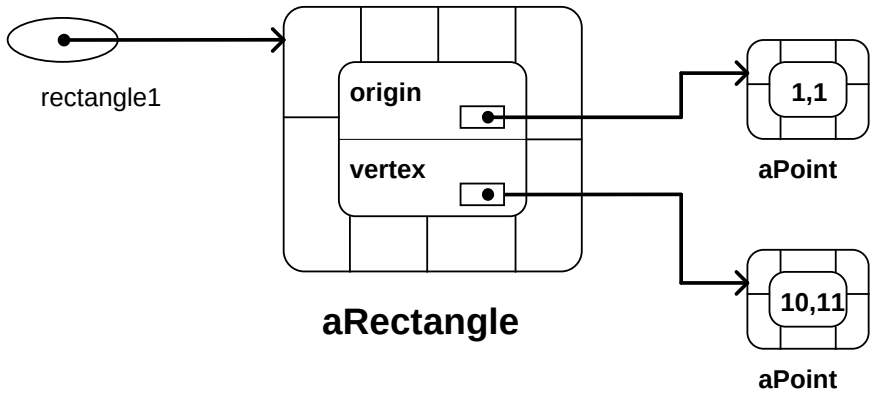
Gli oggetti non contengono altri oggetti ma **possono puntarvi**

In sistemi ad oggetti, gli attributi non primitivi hanno ***semantica per riferimento cioè contengono solo riferimenti***

Quindi attraverso una **variabile** (con un tipo) posso riferire una altra istanza

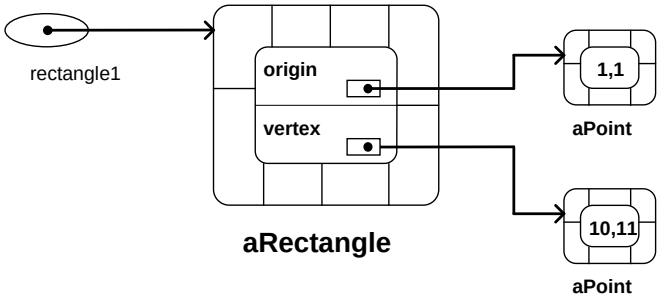
Cambiando valore della variabile posso riferire successivamente una altra istanza

Gli **oggetti passivi** durante la esecuzione determinano un grafo di riferimenti tra di loro



RIFERIMENTI locali e REMOTI (?)

In un sistema ad oggetti locale, il grafo si sviluppa nell'ambito della stessa **macchina virtuale locale con variabili per riferimento**



Attraverso una variabile posso riferire una altra istanza **solo locale**

In genere i sistemi più diffusi (Java) assumono che un programma o applicazione faccia solo riferimenti **interni** alla macchina virtuale e **locali**

Come si possono ritrovare entità esterne?

Si usano protocolli esterni di comunicazione introdotti per i sistemi distribuiti con agganci locali

Ad esempio: **TCP/IP ed il suo sistema di nomi** (e **socket**)

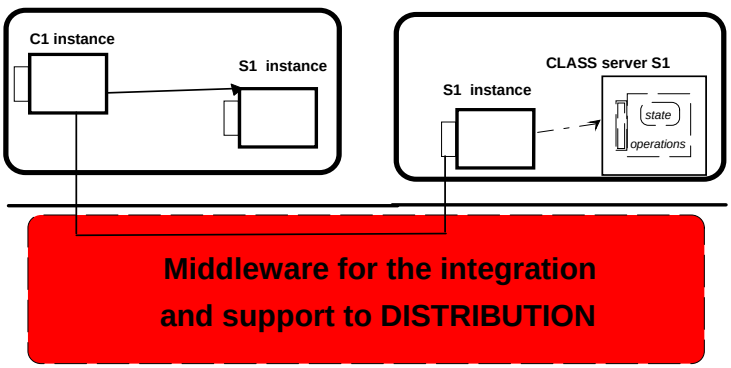
RIFERIMENTI REMOTI

In un sistema ad oggetti, usiamo il supporto di un ambiente esterno per potere coordinare macchine (virtuali) diverse

Un oggetto C1 sul nodo 1 può riferire usando il supporto di Java le istanze locali di S1

Se si vuole riferire una istanza remota abbiamo bisogno di un supporto intermediario che estenda la visibilità

In alcuni casi i **riferimenti locali e remoti** sono resi uniformi attraverso **intermediari locali** (**proxy**) che si incaricano di mascherare il supporto

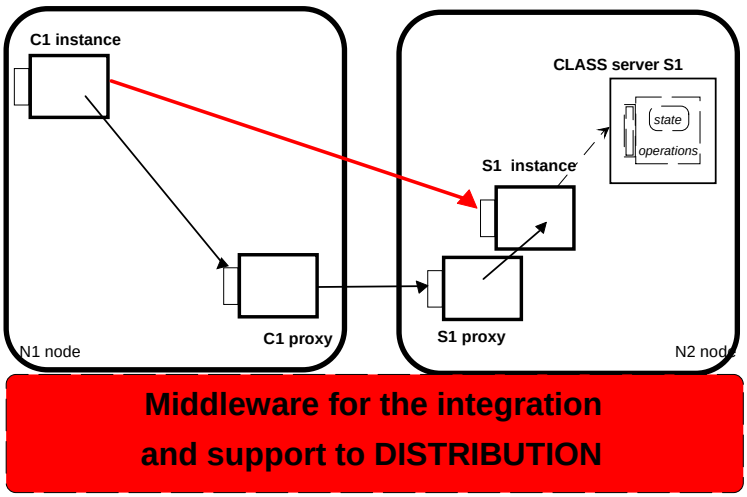


RIFERIMENTI REMOTI via RMI

Tra due sistemi Java possiamo usare Remote Method Invocation (RMI) che costruiscono due proxy,
- uno dalla parte del cliente (stub)
- uno dalla parte del servitore (skeleton)

I **proxy** sono spesso generati automaticamente e facilitano il progetto della parte utente che ragiona indipendentemente dalla distribuzione

In modo del tutto simile in altri **ambienti uniformi** con **ampio supporto** (**CORBA**, **DCOM**, ecc.)

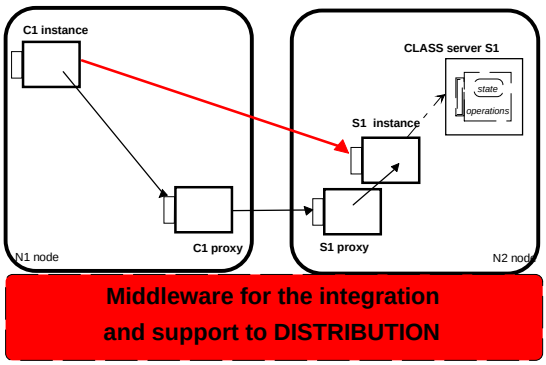


RIFERIMENTI REMOTI via PROXY

Due **macchine virtuali Java** usano **PROXY** per ottenere visibilità remote dei riferimenti per usare RMI

Molte necessità di supporto e problemi:
- Come si ottiene la conoscenza del servitore? (sistema di nomi)
- Dove sono le classi accessorie?
- Come ottenerle (durante la esecuzione)?
- E se ci sono inconsistenze?
- E se il server non fosse attivo?
- E se non mantenesse lo stato?

Sui riferimenti remoti:
- due riferimenti per lo stesso oggetto?
- due riferimenti per lo stesso servizio?
- …

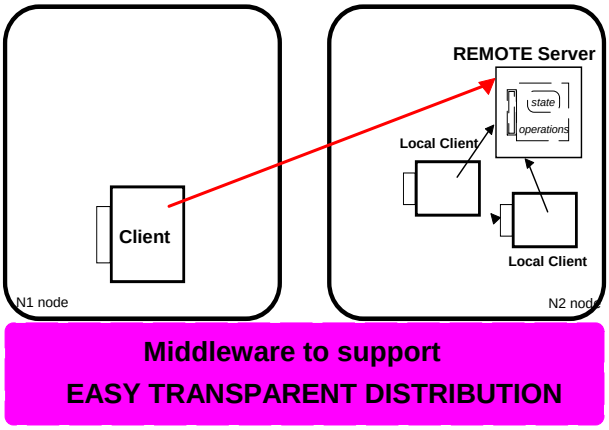


RIFERIMENTI REMOTI e MIDDLEWARE

Un punto centrale in **tutti** i **middleware** che aiutano l'utente a non occuparsi di dettagli è **come si consente un riferimento remoto, che permetta di accedere a entità non locali in modo trasparente**

Bisogna sicuramente per ciascuno valutare il meccanismo di supporto

- Quanto costa l'accesso?
- Quanto costa l'organizzazione di supporto del middleware?
- Come ottenere riferimenti remoti?
- Sono possibili inconsistenze?
- Quali compiti ha il supporto?
- …



INTERMEDIARI e PROXY

PROXY

In una comunicazione si possono interporre **intermediari** sia dalla parte del richiedente sia dalla parte del fornitore del servizio

PROXY

del cliente o del servitore

proxy

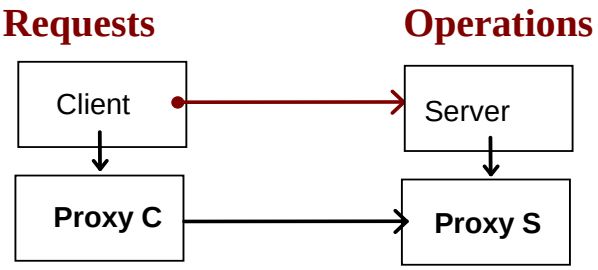
C/S stub e skeleton

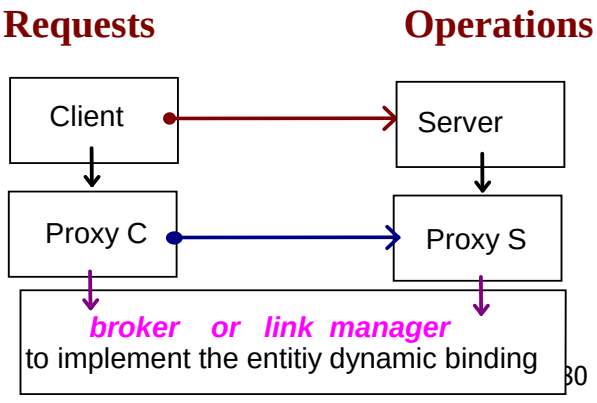
interceptor

per aggiungere funzionalità

broker

simile al container





MODELLI A CONTENIMENTO

CONTENIMENTO

Spesso molte funzionalità possono essere non controllate direttamente ma lasciate come responsabilità ad una **entità delegata supervisore** (**contenitore**) che se ne occupa

- spesso introducendo politiche di default
- evitando che si verifichino errori
- controllando eventuali eventi

I **contenitori** (entità con molti nomi, dette anche **CONTAINER**, **ENGINE**, **MIDDLEWARE**, …) possono occuparsi di azioni automatiche da cui viene sgravato l'utilizzatore che deve specificare solo la parte contenuta tipicamente

di alto livello, *non ripetitiva*,

fortemente dipendente dalla logica applicativa

MODELLI A CONTENIMENTO

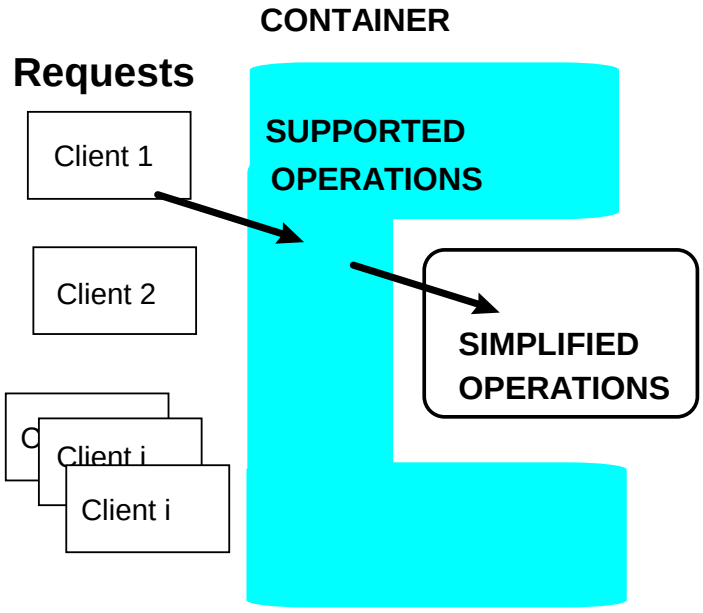
CONTAINER

Un **servizio utente** potrebbe essere integrato in un ambiente (**middleware**) che si occupa in modo autonomo di molti aspetti diversi

Vedi

CORBA tutti aspetti C/S
Engine per framework a GUI
Container per servlet
Supporto per componenti

Container possono ospitare **componenti più trasportabili e mobili**



DELEGA al CONTAINER (Middleware)

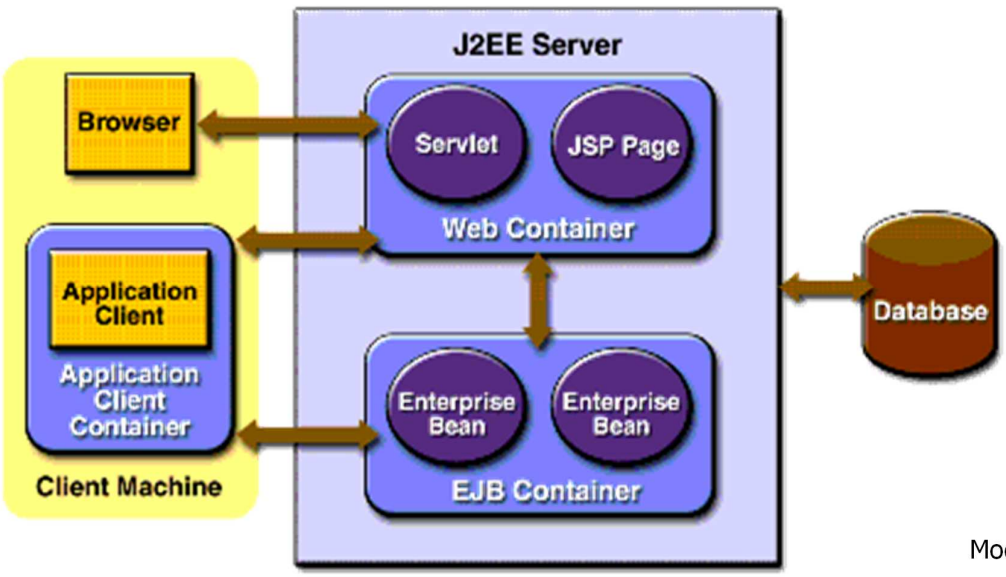
Il **container** può fornire "*automaticamente*" molte delle funzioni per supportare il **servizio** dell'**utente**

- **Supporto al ciclo di vita**
  - **Attivazione del servitore / deattivazione /**
  - **Mantenimento dello stato**
  - **Persistenza e recupero delle informazioni (interfaccia DB)**
- **Supporto al sistema dei nomi**
  - **Discovery del servitore / servizio /**
  - **Federazione con altri container**
- **Supporto alla qualità del servizio**
  - **Tolleranza ai guasti, selezione tra possibili deployment**
  - **Controllo della QoS richiesta e ottenuta**

…

J2EE – Java 2 Enterprise Edition

Un container potrebbe essere anche capace di facilitare sia l'esecuzione di componenti diversi, come
servlet, **JSP**, **bean** di varia architettura e tipo



OGGETTI vs. COMPONENTI

Un punto centrale nel distribuito è anche la difficoltà del supporto agli **oggetti** di un linguaggio e come **si possa facilmente** ottenere un **modello di uso più auto-contenuto e semplice** da capire e da usare

Modello ad oggetti poco confinato e molto dipendente dall'ambiente di esecuzione (a grana fine)

Bisogna sicuramente **limitare l'impatto dell'oggetto** e la **sua forte interazione** con **l'ambiente contenente**

Modello a componenti (a grana più grossa)

Per definire entità che siano più facilmente auto-contenute e trasportabili da un ambiente ad un altro

Definizione di componente: "**astrazione statica di una entità confinata con porte di collegamento verso l'esterno**"

In questi sistemi spesso interazione via container (vedi dopo)

COMPONENTI

Un componente

- **Statico** in quanto indipendente dalla applicazione e con vita propria
- **Astratto** senza visibilità della specifica struttura interna mostrando solo porte di ingresso e uscita
- **Comunicante in modo disciplinato con porte** come uniche entità riconosciute per la interazione, da e per l'esterno (**IN** e **OUT**)

Effetto di **migliore riusabilità**, con facile trasportabilità da un ambiente ad un altro (non ci sono interazioni nascoste, ma sono tutte visibili), **possibilità di sostituzione** di una implementazione con un'altra (aggancio dinamico) senza intervenire sull'ambiente,

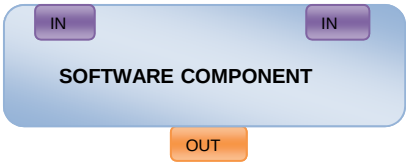
Verso la realizzazione di Servizi e SOA (**S**ervice **O**riented **A**rchitecture o **SOA**) ⇨ porte come metodi offerti e invocati, ma anche come proprietà visibili ed accessibili

ANCORA COMPONENTI

Un componente ancora

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

Michael Feathers

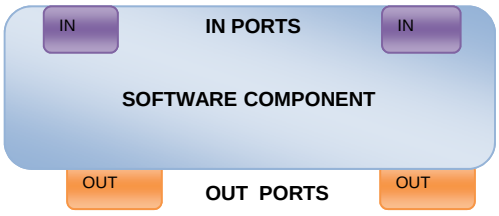


Un **componente** è tipicamente una entità a **grana più grossa** di un oggetto, molto più **autocontenuta** e capace di **operare** in **ambienti diversi**…

spesso viene a lavorare in un **container**, ossia un **server di supporto** capace di interagire con il componente e di fornirgli **molte azioni necessarie** con una buona divisione dei compiti

J2EE, EJB come contenitori capaci di ospitare componenti progettati per loro

PROPRIETÀ di un COMPONENTE



Un componente ha una **interfaccia molto disciplinata e** deve dichiarare il **contratto di interazione** attraverso **porte** che regolano le **richieste accettate in ingresso (porte in)** e le richieste che **può fare verso l'esterno (porte in)**

Questo regola in modo preciso e non variabile la interazione con l'esterno tutta esplicita (e non nascosta)

Un componente è **autocontenuto** ma deve gestire solo **alcune funzionalità** e **delegare alcune funzionalità** ad un **contenitore** che è capace di fornirgliele e lo **racchiude**

Interazione regolata e disciplinata, e con anche capacità di esecuzione autonoma o meno

SISTEMI a COMPONENTI

Un sistema a componenti si basa esclusivamente su componenti che hanno queste proprietà insieme con i **container racchiudenti**

- **Incapsulamento** della implementazione
- **Condivisione delle risorse** che devono essere assegnabili dinamicamente
- **Composizione** formando nuovi componenti a partire dai componenti esistenti
- **Ciclo di vita** come diverse possibilità di esistenza del componente
- **Attività** come espressione di output attraverso la interazione di componenti
- **Controllo** come presenza di una funzione di monitoraggio e controllo dei componenti
- **Mobilità** come possibilità di introdurre e rimuovere componenti in modo dinamico

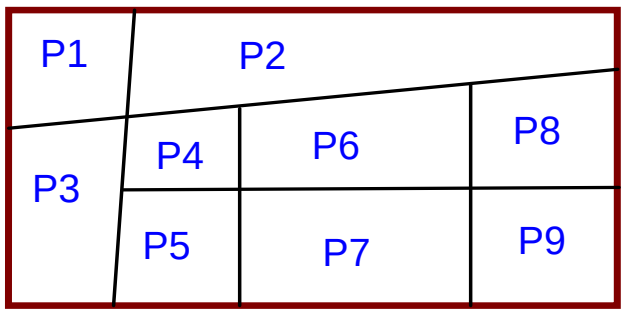
DEPLOYMENT di APPLICAZIONE

Una applicazione viene sviluppata determinando **oggetti** e **classi** necessarie ai requisiti e, se non si lavora su una unica macchina, si decide un **deployment su più macchine** che comporta di

- **partizionare la applicazione** in **aggregazioni costituenti** e
- **basarsi** su un supporto per i **riferimenti remoti**

La applicazione si divide in risorse che rappresentano in partizioni (P1 – P9) da mappare sul deployment

Application



Possible partitioning of the resources

RISORSE di una APPLICAZIONE

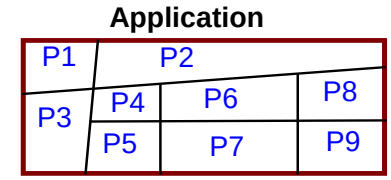
Una applicazione precede risorse molto diverse, logiche e concrete:
processori, rete, e anche processi, oggetti, componenti, ….

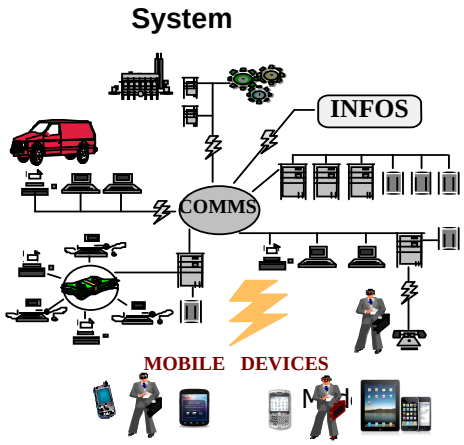
Le risorse applicative che consideriamo sono molto varie:

- **processi**
- **componenti**
- **oggetti e classi**

Le risorse di sistema da considerare sono molte e diverse:

- **processori**
- **reti**
- **interconnessioni di cluster**
- **Cloud**





PARTIZIONI della APPLICAZIONE

Una applicazione deve essere portata in un deployment su un certo numero di processori e si deve decidere come raggruppare i suoi componenti in partizioni per i processori stessi

La applicazione prevede risorse

sia statiche (rappresentate nel lucido precedente) facilmente raggruppabili secondo necessità, in modo da avviare il tutto con i componenti allocati
sia dinamiche (non rappresentate nel lucido) che saranno create durante la esecuzione o potrebbero anche non essere create in alcuni run

Si pensi ai processi o alle risorse che dipendono dalla esecuzione e che possono essere create solo in alcune esecuzioni, dipendentemente dallo stato e dal progresso della applicazione stessa

MODELLO di ALLOCAZIONE

Allocazione …. o Deployment

le entità di una applicazione possono essere

o *statiche* o *dinamiche* (**o *miste***)

Allocazione statica: data una specifica configurazione (o deployment), le risorse sono decise **prima della allocazione**

Allocazione dinamica: la allocazione delle risorse è decisa **durante la esecuzione ⇨ sistemi dinamici**

Risorse statiche (sempre decise in modo statico)
Risorse dinamiche decise anche in modo statico … e

In sistemi dinamici, si creano risorse dinamiche non previste e si può pensare di riallocare risorse esistenti (migrazione):
le risorse possono muoversi sulla configurazione durante la esecuzione

Risorse dinamiche decise anche in modo dinamico

SUPPORTO DEPLOYMENT

- ***MANUALE***

→ **l'utente** determina **ogni singolo oggetto** e lo **trasferisce sui nodi appropriati con sequenza appropriata di comandi**

- ***APPROCCIO con FILE SCRIPT***

→ **si devono eseguire** alcuni **file di script** (qualche linguaggio shell, bash, perl, ecc.) **racchiudono la sequenza dei comandi per arrivare alla configurazione che presenta dipendenze tra oggetti**

- ***APPROCCIO basato su MODELLO o MODEL-DRIVEN***

→ **supporto automatico alla configurazione attraverso linguaggi dichiarativi o modelli di funzionamento della configurazione** da ottenere (ad esempio SmartFrog e Radia)

MODELLI di ALLOCAZIONE

- ***APPROCCIO ESPLICITO*** **(comandato dall'utente)**

→ l'utente prevede il **mappaggio per ogni risorsa** potenzialmente da creare **prima della esecuzione**

- ***APPROCCIO IMPLICITO*** **(automatico)**

→ il sistema si occupa del **mappaggio delle risorse della applicazione**

(anche al deployment e durante la esecuzione)

- ***APPROCCIO MISTO***

→ **il sistema adotta una *politica di default* applicata sia inizialmente per le risorse statiche sia dinamicamente** per la allocazione delle **nuove risorse e la migrazione** di quelle già **esistenti**

→ **eventuali indicazioni dell'utente** sono tenute in conto per **migliorare le prestazioni**

MODELLI di ALLOCAZIONE

- ***APPROCCIO ESPLICITO***

Costo elevato: l'utente che **prima della esecuzione** deve prevedere un **mappaggio per ogni risorsa, anche che non sarà creata**

- ***APPROCCIO IMPLICITO***

Costo limitato: il sistema si occupa del **mappaggio solo delle parti necessarie → le statiche e le dinamiche su bisogno**

- ***APPROCCIO MISTO***

Costo variabile, bilanciabile e adattabile:

il sistema adotta una **politica per ogni risorsa, o statica o dinamica**

decisioni statiche possono essere ottimizzate prima del run-time

decisioni dinamiche possono essere a costo diverso, a secondo del carico del sistema

il sistema può anche usare politiche che tengono conto di indicazioni utente, se è il caso:

un oggetto deve essere coresidente con un altro, se il nodo non troppo carico oppure sul nodo N4 …

SISTEMI CONTEXT - AWARE

Il **contesto** deriva da ambiti mobili in cui gli **utilizzatori possono muoversi mentre stanno usufruendo di servizi**

contesto di computing: connettività di rete, di banda, costi di comunicazione, risorse vicine, …

contesto utente: profilo utente, storia delle scelte e preferenze utente, situazione sociale corrente, ….

contesto fisico: posizione, condizione di traffico, velocità, livello di rumore, temperatura, luce correnti, …

contesto in tempo: comportamento differenziato in ore del giorno, lavoro e relax, …

Tutte le proprietà che possono essere considerate come indicatori per fornire un **servizio migliore** fanno parte del contesto e permettono di adattare la fruizione al contesto corrente e alle esigenze utente

I sistemi possono usare le informazioni di contesto per fornire un servizio **migliore agli utenti che lo richiedono** o anche che **non ne sono consapevoli** se si riesce a capire quali possono essere le **preferenze che possono guidare l'adattamento**

CONTEXT – AWARE COMPUTING

Il concetto di contesto permette di personalizzare il servizio in modo da consentire un adattamento che può diventare anche molto spinto

non solo in base alle **preferenze utente** ma anche **all'ambiente sociale** dell'utente stesso

- **dove sei**
- **con chi sei**
- **quali risorse sono presenti e vicine**

Per esempio, se siete al lavoro e si avvicina un superiore si possono mostrare e focalizzare le risorse di produttività sul computer

Se invece siete solo tra colleghi, si possono anche mostrare strumenti di presenza e di entertainment e ambienti di game

I sistemi possono usare il concetto di Contesto per ottenere e prendere decisioni adattate alla situazione e anche ottenere riconfigurazioni automatiche in caso sia necessario

CONTEXT - AWARENESS

Molte definizioni di contesto

Set of environment states and settings that either determines application behavior or where an application event occurs and is relevant for the user

Riconfigurazione utente (awareness passiva)

Il sistema presenta le variazioni all'utente che ha il compito di decidere se accettare il nuovo comportamento e incorporarlo esplicitamente nel sistema

Riconfigurazione automatica (awareness attiva)

Il sistema adatta il comportamento senza aiuto dell'utente

Gradi diversi of **context-awareness utente** per i diversi tipi di utenza: differenziazione del comportamento e del coinvolgimento utente

SISTEMI LOCATION - AWARE

Alcuni sistemi sono capaci di **adattarsi in base alla locazione delle risorse coinvolte e dell'utilizzatore, intesa come localizzazione in senso geografico**

Questi sistemi, detti **location-aware** or **location-based devono fornire servizi adattati in ogni condizione di operatività**

- outdoor e indoor
- con la precisione richiesta
- con i costi ammessi e previsti

Sono basati su molte tecnologie diverse, più o meno costose: GPS, dispositivi ad hoc, basate su wireless network a domini (celle Wi-Fi, Bluetooth, …), basate su indicatori correlati alla posizione corrente in mobilità (come sensori RFID, ecc.) e anche ai sensori montati sui telefoni cellulari (usati come wireless sensor network)

Tecniche allo stato dell'arte tendono ad **usare la fusione e combinazione di sensori diversi** per identificare il posizionamento

LOCATION- e CONTEXT-AWARENESS

La **locazione** può essere trattata e gestita a due livelli

Indicatori di basso livello (fisici), per identificare, sentire e ottenere informazioni:

tempo, utenti e risorse vicini, banda, orientazione, …

Indicatori di alto livello (simbolici), o logici cioè correlati a informazioni di contesto di alto livello:

attività logiche, azioni specificate in tempo, abitudini utente e agenda relativa, informazioni visuali, …

I due livelli devono essere integrati e messi in stretta relazione

Locazione come indicatore fondamentale per il supporto al **concetto di contesto** e **produrre forme di conoscenza addizionali riguardo al comportamento, abitudini, profilo, e bisogni utente**

Spesso si usano tutte le forme possibili per arrivare a questa integrazione, dalle tecniche AI, inferenze, induzioni, e anche sistemi di regole per assistere nella elaborazione delle informazioni

ALLOCAZIONE OFF-THE-SHELF

I **Data center conto terzi** spesso forniscono **allocazioni standard o pronte**

Risorse: le risorse sono allocate in modo esclusivo e per il tempo stabilito, anche se non usate

Il **modello Cloud permette** una prospettiva diversa:

Ci sono **risorse disponibili** e si paga **per-use**

In modo **differenziato**, possiamo avere

- **utenti esperti** che decidono in modo informato che risorse e in che modo vengano fornite (aggiunte e tolte)
- **utenti meno smart** che hanno a disposizione pacchetti standard con configurazioni **standard e pronte**

Risorse fornite su bisogno in modo elastico e flessibile, seguendo il bisogno ad ogni momento e con la possibilità di verifica delle risorse ad ogni momento

CASO del CLOUD

Progetto di una applicazione che ottiene dei **servizi on-demand** ottenuti via **Web su richiesta dell'utente** che non si deve **preoccupare** (troppo) della loro **gestione**

Le risorse logiche e fisiche sono sul Cloud nei diversi data center

L'utente deve sicuramente usare **Risorse-aaS** (**Risorse as a Service**) e si aspetta un comportamento molto dinamico dalla parte del servizio

⇨Se necessario il data center deve predisporre in modo **più o meno automatico nuove risorse** sia logiche sia fisiche

→Questo rende la architettura molto **elastica adattabile e flessibile**

I problemi di sono lasciati al gestore del data center

INTERESSE per il DEPLOYMENT

Scelta di un deployment o di un altro

→ **Può avere un grande impatto durante la specifica esecuzione e deve essere tenuto in conto**

Pensate a delle risorse che debbano comunicare,

- dobbiamo considerare strumenti di comunicazione internodo se le risorse potranno essere allocate su nodi diversi

- dobbiamo scegliere gli strumenti di comunicazione più adatti per la allocazione che stiamo determinando (pensate ad architetture diverse ed eterogenee di supporto)

- dobbiamo anche ottimizzare gli strumenti di comunicazione quando le risorse sono presenti sulla stessa macchina, differenziando comunicazioni inter-nodo e intra-nodo (come spesso fanno i middleware esistenti)

- dobbiamo verificare che il deployment sia adatto agli strumenti di comunicazione scelti e non produca problemi (identificando ed eliminando colli di bottiglia e casi critici)

CASO di RMI (SCALABILITÀ)

Vincolo di deployment in RMI

→ Il **registry** deve essere sullo stesso nodo del server e i clienti devono comunicare con lui per ritrovare il riferimento remoto

→ Questo rende la architettura vincolata a una conoscenza dei **nodi di servizio (**e i servizi **NON scalabili)**

Possiamo fare un progetto con **una diversa architettura**,

I riferimenti remoti possono anche essere derivati da un passaggio di parametri (magari con un unico mediatore)

- possiamo prevedere una architettura in cui ai clienti i riferimenti ai server sono passati attraverso una funzione di attivazione iniziale e senza conoscenze pregresse

- eliminando il collo di bottiglia del nodo del server e del registry, che possono introdurre dei forti limiti alla **scalabilità** e rappresentare un vincolo pesante

Così possiamo introdurre nodi senza avere problemi di vincoli, in caso di crescita del sistema (**estensibilità**) e di necessità di introdurre nodi

CASO di RMI (ANCORA)

Vincolo di RMI per Garbage Collector

→ la JVM del server deve controllare che i clienti abbiano ancora i riferimenti remoti per realizzare un GC distribuito

→ Questo rende la architettura molto poco scalabile **(NON scalabile** in caso di molti riferimenti**)** e anche con problemi **(poco corretta)**

Come posso sapere se il riferimento remoto va in giro e viene distribuito?

Vincolo di RMI su inter / intra nodo

→ se un server ed un client sono coresidenti sullo stesso processore, le JVM non consentono ottimizzazioni e si deve sempre usare il riferimento attraverso stub e skeleton, senza alcuna possibilità di scorciatoie

→ Questo rende la architettura molto poco adattabile **(NON flessibile)** e con problemi di efficienza **(poco scalabile e non adatta al distribuito)**

Modello C/S

Cliente/Servitore e richiesta operazione
Intrinsecamente distribuito su nodi diversi

regole di comunicazione di **alto livello**:

modello asimmetrico con il cliente che conosce il servitore e interagisce in modo sincrono (risultato) e bloccante (attesa del risultato) (a default)

Modello con forte accoppiamento:

compresenza di chi interagisce

Ovviamente ci interessano solo modelli che siano intrinsecamente distribuibili e distribuiti e che portino a deployment realmente distribuiti

Ci sono molti punti deboli del C/S e difficoltà di uso tipicamente superate da variazioni per esigenze specifiche

MODELLI - OLTRE IL C/S

Molte variazioni del Cliente/Servitore

Modelli **pull (sincrono non bloccante)**
(faccio io il recupero del risultato con o senza attesa)

Modelli **push (sincrono non bloccante)**
(il server mi allunga il risultato e io lo recupero poi localmente)

Modelli a **delega** per la attesa della risposta **(sincrono non bloccante)**
(faccio *aspettare un altro al posto mio* e recupero il risultato da questo)

Modelli di **notifica** per la risposta
(il delegato mi notifica che una risposta è arrivata)

Modelli ad **eventi (anche asincrono)**
(un evento viene reso noto agli interessati consumatori e generato da produttori)

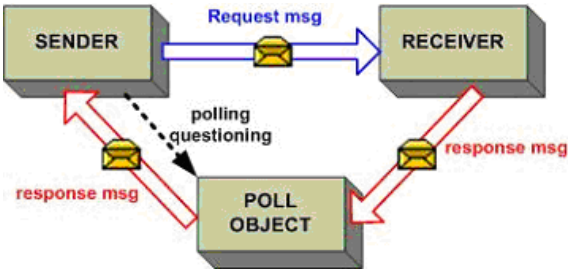
Modelli di **provisioning**
(oltre al servitore e al cliente, una serie di intermediari interessati)

DELEGA - RECUPERO RISULTATO…

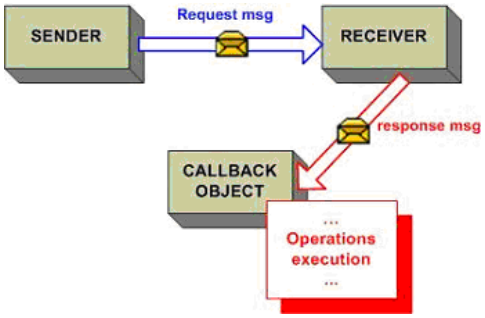
In un modello sincrono non bloccante, introduzione di entità per la interazione

Rispetto ad un puro meccanismo cliente servitore, **si usano oggetti di Poll e Call-Back come intermediari**

Oggetto Poll



Oggetto Call-Back



Operazioni corte ed attese limitate

Operazioni lunghe e indipendenti dal chiamante

Si deve definire bene cosa sono e come sono strutturati

SCAMBIO MESSAGGI

Modello a scambio MESSAGGI

Molto flessibile e primitivo

Scambio di informazioni: proprietà

- **sincrono/asincrono** (risultato o meno)
- **simmetrico/asimmetrico** (conoscenza pari o meno)
- **indiretto/diretto** (entità intermedia o meno)

Implementazione

- **bloccante/non bloccante** (blocco del mittente in attesa)
- **bufferizzato/ non** (batch di messaggi)
- **reliable/unreliable** (senza o con perdita di messaggi)

Modalità a **ricevente multiplo o di gruppo**

- **multicast (MX) e broadcast (BX)**

A volte messaggi come solo informazioni elementari (segnali) di sincronizzazione (e non comunicazione, con informazioni)

MODI dello SCAMBIO MESSAGGI

Modello a scambio MESSAGGI

Molto vario nel distribuito

Rendez-vous

Scambio di messaggi sincrono, bloccante, simmetrico, accoppiato, non bufferizzato, e **uno ad uno** (più di un C/S)

Con entità intermedia (canale, …)

Scambio di messaggi asincrono, non bloccante, asimmetrico, **disaccoppiato** (molto meno stretto di un C/S)

Con entità intermedia e azione di gruppo (canale, …)

Scambio di messaggi asincrono, non bloccante, asimmetrico, disaccoppiato e **molti a molti** (eventi)

C/S vs SCAMBIO MESSAGGI

Cliente/Servitore

Modello con forte accoppiamento:

compresenza di chi interagisce

Meccanismo adatto per comunicazioni semplici ma poco flessibili

di **alto livello** (ossia adatta per usi applicativi)

con difficoltà in situazioni diverse, vedi multicast e broadcast

Scambio messaggi Sender/Receiver

Modello con limitato accoppiamento:

non compresenza grazie al buffering

Comunicazione primitiva, flessibile, espressiva, magari difficile da maneggiare

di **basso livello** (ossia adatta per usi di sistema):

forme **molto varie**, anche con più facile supporto a forme di **multicast e broadcast**

DIS / ACCOPPIAMENTO

Gli strumenti di comunicazione possono imporre o meno dei ***vincoli sulle entità*** che devono interagire

Questi vincoli possono anche introdurre forti limiti su chi interagisce e introdurre necessità di conoscenze che a volte non sono necessarie

Accoppiamento in vari modi

- **spazio**

Le entità si devono conoscere (reciprocamente) e essere co-locate

- **tempo**

Le entità interagenti devono essere compresenti (presenti insieme nello stesso instante)

- **sincronizzazione**

Le entità interagenti devono aspettarsi a vicenda e sono soggette a blocchi reciproci

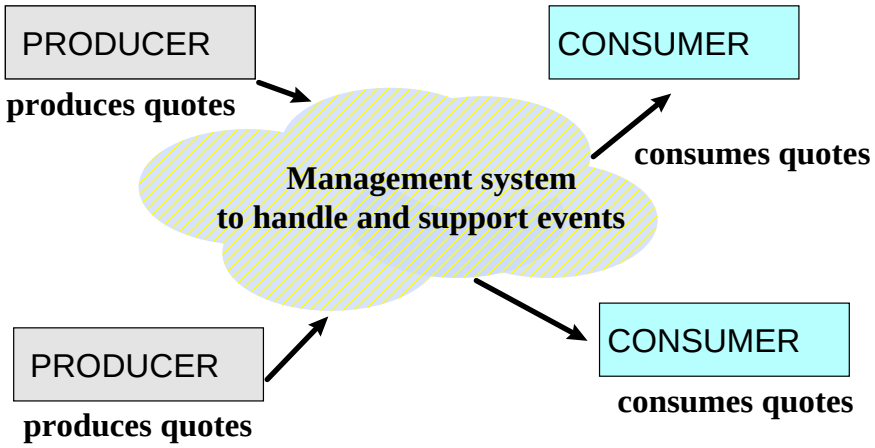
Il disaccoppiamento diventa un fattore per l'abilitazione di maggiore flessibilità e per arrivare a modelli che possono sfruttare la potenziale distribuzione

EVENTI e PUBLISH-SUBSCRIBE

Disaccoppiamento tra entità interagenti

Gli **eventi** permettono ai **produttori** di generare eventi (**publish**) e di disinteressarsi della consegna

I **consumatori** si sono registrati come interesse ad un evento (**subscribe**) e sono avvisati della occorrenza



Produttori e **consumatori** non devono essere **presenti insieme**

Da Reti L-A
FRAMEWORK: EVENTI LOCALI

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

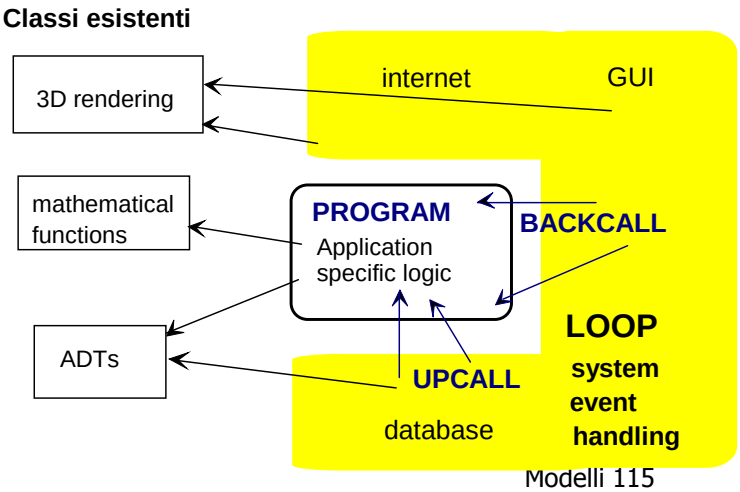
Il Framework **tende a rovesciare il controllo** (*per eventi di sistema*)

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 all'utente sono dette **backcall** o **upcall** assimilabili a **eventi asincroni** generati dal supporto che le applicazioni devono gestire tramite una funzione di gestione specificata dall'utente



SISTEMI ad EVENTI (DISTRIBUITI)

I sistemi ad eventi devono essere pensati e progettati **senza vincoli di località di nessun tipo**

Il modello ha la sua forza nella non località delle entità interagenti

implementazioni solo locali (*ad esempio che prevedono il vincolo della condivisione del codice tra produttore e consumatore*) sono downsizing arbitrari e non sensati del modello

Realizzare un sistema ad eventi non tenendo conto del potenziale disaccoppiamento, …

vuole dire usare male **il modello, una delle cose peggiori che possiamo fare usando una tecnologia**

Se si vincolano gli eventi alla co-residenza e alla compresenza delle entità interagenti, stiamo determinando dei deployment che contrastano con il modello di base che stiamo usando

SISTEMI ad EVENTI: INDICATORI

I sistemi ad eventi sono pensati per **grandi sistemi** e alcuni **indicatori sono fondamentali**

- **Costo nella distribuzione degli eventi (da limitare)**
- **Performance (da ottimizzare)**
- **Scalabilità (alta)**
- **Tempi di Latenza (da limitare)**
- **Pervasività del servizio (alta)**
- **Sviluppo ed esecuzione indipendente (elevata)**
- **Tolleranza ai guasti (massima)**

Si devono quindi pensare a sistemi ad eventi **realizzabili** in cui **gli indicatori**, per tutte le possibili realizzazioni distribuite, presentino **valori accettabili**, magari **'costanti'**… beh **almeno prevedibili**

EVOLUZIONE dei SISTEMI ad EVENTI

Eventi primitivi

Gli **eventi** sono spesso considerati **segnali on/off senza contenuto informativo**

eventi di interrupt o segnali associati ad una funzione di gestione

Eventi con contenuto

Alcuni **eventi** hanno **un contenuto** e si possono pensare anche a **filtri di interesse** e a ragionare sul **contenuto**

RSS con registrazione su temi specifici o multicast differenziato su gruppi di destinatari diversi con registrazioni differenziate

Eventi con qualità - Quality of Service

Alcuni **eventi** hanno requisiti **da specificare in modo differenziato per i diversi utilizzatori** e si possono pensare anche supporti differenziati (persistenza, tempo di mantenimento, reliability, …)

Eventi persistenti*: gli utenti non perdono eventi anche se off-line e ricevono tutti gli eventi non ricevuti*

Priorità degli eventi*, Spazio e numero di eventi in coda, …*

SISTEMI PUBLISH-SUBSCRIBE

Sistemi PUB-SUB identificati come **sistemi evoluti basati su modelli ad eventi** e **scambi di messaggi** per sfruttarne la flessibilità ed il disaccoppiamento di interazione

aumentando **scalabilità e distribuibilità**

Il modello PUB-SUB ha anche ulteriori flessibilità …

Filtraggio dei messaggi sulla base di

topic-based: sulla base del soggetto di interesse (si pensi a un interesse specificato tra diversi canali possibili, RSS specifico)

content-based: sul soggetto del contenuto del messaggio (si pensi a una parola chiave o una relazione di interesse)

type-based: sul tipo di messaggio (si pensi distinguere tipi diversi e potere selezionare a priori)

Qualità di Servizio (QoS) sui messaggi

Persistenza, Priorità, Garanzia di mantenimento e durata, …

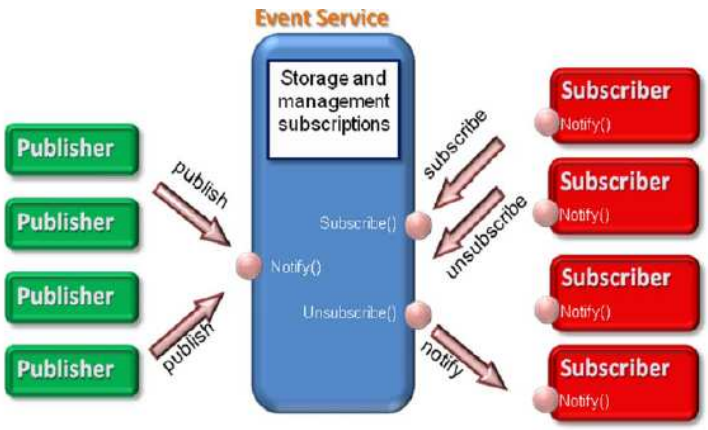
SISTEMI PUBLISH-SUBSCRIBE

I sistemi PUB-SUB reali prevedono operazioni di sottoscrizione per i consumatori

I **produttori** o **publisher** forniscono eventi (potrebbero chiedere quali siano i sottoscrittori correnti)

I **consumatori** o **subscriber** che si sono sottoscritti devono riceverle attraverso una notifica

Una **infrastruttura** deve garantire la operatività



MODELLI DISACCOPPIATI - TUPLE

MODELLO a TUPLE per la INTERAZIONE

modello generale per **comunicazione** e **sincronizzazione**

progettato come *astrazione della memoria condivisa + comunicazione*

Uno **spazio di tuple** è un insieme **strutturato di relazioni**, intese come contenitori di *attributi e valori* per PUB-SUB

Su uno spazio di tuple si possono depositare / estrarre **informazioni di alto livello** senza **causare interferenze**

Una possibile relazione: **messaggio** (**dachi**, **achi**, **corpo**)

Lo spazio serve da contenitore per **valori di tuple** in accordo al pattern specificato (ai *tipi degli attributi,* qui stringhe ASCII)

Tuple per lo spazio *messaggio*: {**Antonio**, **Giovanni**, **msg1**}

{**Giovanni**, **Antonio**, **msg1**} {**Antonio**, **Giovanni**, **msg2**} …

Non ci sono limiti alle tuple che si possono depositare e che possono rimanere nello spazio delle tuple *senza limiti di tempo* o di *memoria*

TUPLE - Linda (Gelernter)

Operazioni di **In** e **Out sullo spazio**

Su **spazi di tuple** sono sempre possibili operazioni di **scrittura** e **lettura concorrenti corrette** con accesso in base al contenuto degli attributi

In estrazione di una tupla dallo spazio e **Out** inserimento di una tupla

- La **Out emette la tupla** sullo spazio a disposizione di richieste e lì rimane fino ad un consumo corrispondente (una sola In)
- La **In estrae una tupla** dallo spazio adatta alle esigenze, se esiste
- Se non esiste, aspetta fino a che non se ne riceve una
- Il **match avviene in base al pattern** sui valori degli attributi

In caso di match con più tuple, una sola viene estratta **non deterministicamente**

Out: messaggio (P, Q, testo1)

In: messaggio (?dachi, Q, ?corpo)

La **In** aspetta una tupla in cui il secondo attributo presenti la stringa Q e produce valori per **dachi**(=P) e **corpo**(=testo1)

DISACCOPPIAMENTO TUPLE

Spazi di tuple

Portano ad una comunicazione **disaccoppiata e poco sincrona**

In tempo

Un processo potrebbe lasciare tuple in un sistema distribuito e solo **molto dopo** i relativi consumatori potrebbero arrivare a prelevare le tuple stesse

In conoscenza reciproca (spazio e sincronizzazione)

i consumatori non devono conoscere in alcun modo i processi produttori e non possono interferire (*una operazione estrae una tuple, eventuali altre aspettano*)

In qualità - QoS

Gli spazi delle tuple sono **persistenti** e tendono a mantenere le **tuple depositate senza limiti di memoria** (come requisiti) senza introdurre interferenza tra processi

Spazi di tuple sono messi a disposizione per favorire **comunicazioni ben fatte** di **alto livello**

Javaspaces, …

TRASPARENZA di una PROPRIETÀ

TRASPARENZA (vs. VISIBILITÀ)

Accesso	omogeneità accesso alle risorse locali e remote
Allocazione	uso indipendente dalla località o meno delle risorse
Nome	indipendenza del nome dal nodo di residenza
Esecuzione	non distinguibilità dell'uso di risorse locali e remote
Performance	non degrado nell'uso dei servizi locali o remoti
Fault	capacità di fornire servizi anche in caso di guasti
Replicazione	capacità di usare risorse molteplici per un servizio unificato con migliore QoS

La **trasparenza** è sempre un requisito da ottenere?

ad **ogni costo**, per **ogni livello di sistema**, per **ogni applicazione** e **strumento**

(??) **Location-awareness** per la generazione di servizi che dipendono dalla **visibilità** della **allocazione** corrente

TINA-Consortium – oltre il C/S

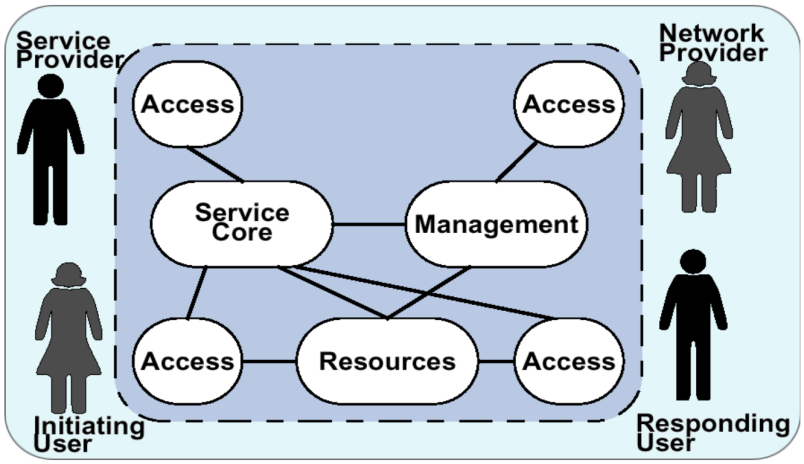
Telecommunications Information Networking Architecture

TINA-C - Consortium definisce modelli di servizio e disponibilità

Si considerano **utenti** di un servizio (ad es. comunicazione con altri utenti)

Questo è reso possibile da una serie di altri agenti in gioco: molteplicità di servizi e da una collaborazione di provider e facilitatori degli stessi.

Si considerano tutti i **provider di rete e di servizio**



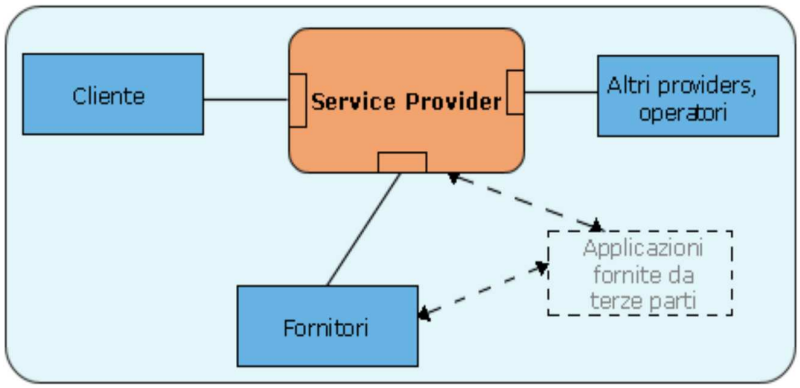
TINA-C – PROVISIONING e QoS

Accordo e negoziazione tra parti per il servizio

coinvolgendo **risorse di comunicazione** e tenendo conto della necessità di fare la **gestione delle risorse** durante il servizio

Le entità in gioco decidono un livello di QoS da mantenere come prerequisito per il servizio

Se il servizio viene completamente erogato con la QoS negoziata viene considerato significativo, avvenuto con successo, e da pagare



Modello astratto (concentrato) – RAM

Una macchina ad accesso random (**R**andom **A**ccess **M**achine) è costituita da:

- **un** programma inalterabile composto di istruzioni in sequenza
- **una** memoria composta di una sequenza di parole, ciascuna capace di contenere un intero
- **un** solo accumulatore capace di operare
- **un** nastro di input (read-only)
- **un** nastro di output (write-only)

Modello RAM

Durante un passo la RAM esegue una istruzione del programma in sequenza, a parte i salti

Istruzioni ad esempio: *read, write, load, store, add, sub, mul, div, jump (acc), ..., halt*

Modi di indirizzamento: *immediato, diretto, indiretto*

La RAM è una **macchina special-purpose** per la risoluzione di un problema specifico

limiti:

- *non* c'è *limite* alla *memoria* di programma
- le *istruzioni richiedono tutte lo stesso tempo*

Modello PRAM

Una ***macchina parallela ad accesso random*** (**P**arallel **R**andom **A**ccess **M**achine) è costituita da una collezione di RAM

Una **PRAM di dimensione P** è composta da:

- **P** programmi inalterabili ciascuno fatto di istruzioni in sequenza
- **una sola** memoria composta di una sequenza di parole capaci di contenere un intero
- **P** accumulatori capace di operare
- **un** nastro di input ed **uno** di output

Il modello è **Uniform memory access** (**UMA**), solo 1 livello di memoria

Durante un passo, la PRAM esegue **P** istruzioni, **una per ogni programma** (modello MIMD o meglio MIMD sincrono)

Modello PRAM

DISTINZIONE semantica

in base a come si eseguono le operazioni su una stessa cella di memoria
Operazioni di lettura/scrittura: operazioni ***simultanee*** (**C**oncorrenti) e ***sequenzializzate*** (**E**sclusive)

Concurrent **R**ead **E**xclusive **W**rite (**CREW** PRAM)
Exclusive **R**ead **E**xclusive **W**rite (**EREW** PRAM)
Concurrent **R**ead **C**oncurrent **W**rite (**CRCW** PRAM)
Exclusive **R**ead **C**oncurrent **W**rite (**ERCW** PRAM)

	read	write
CREW	concurrent	exclusive
EREW	exclusive	exclusive
CRCW	concurrent	concurrent
ERCW	exclusive	concurrent

Interferenza Operazioni Concorrenti

Operazioni sequenziali: ritardo sui programmi

Principio di serializzazione ⇨

gli effetti come se **una operazione** fosse fatta per **prima nel primo passo**, poi un'altra in successione nel successivo, poi un'altra ancora, etc., tutte in un passo di istruzione

Operazioni concorrenti:
vanno insieme senza interferenza … con quali effetti?

Limiti del modello:

- le istruzioni richiedono lo stesso tempo
- il tempo di operazione non dipende da P
  accesso a **memoria comune** in un tempo che non cresce con P
- il tempo di operazione di input/output trascurato
  accesso ad un unico nastro non cresce con P

I/O bottleneck ⇨ Uso di concorrenza per gestire il sottosistema I/O

Modello MP-RAM

Una ***RAM con Message Passing*** (**M**essage **P**assing-**R**andom **A**ccess **M**achine) è una collezione di RAM, ciascuna con una **memoria privata** e connessione con canali **punto a punto**

Il modello è **Non Uniform memory access** (**NUMA**), più livelli di accesso (memoria e rete)

Una **MP-RAM** di **dimensione P** è composta da:

- **P** programmi inalterabili ciascuno fatto di istruzioni in sequenza
- **P** memorie composte di una sequenza di parole
- **P** accumulatori capaci di operare sulla propria memoria
- **un** nastro di input ed **uno** di output
- **un** grafo di interconnessione punto-a-punto (ad albero, a stella, ecc.)

Ogni nodo ha un certo *numero* di vicini ed un *grado di interconnessione*

Modello MP-RAM

Maggiore il **grado di interconnessione**, **maggiore** il **costo della architettura**, **minore necessità** di **routing** intermedio

Possibilità di **comunicazione bidirezionale**

Le istruzioni sono accresciute con: *send* e *receive neighbor*

Durante un passo, la MP-RAM esegue P istruzioni, una per ogni programma

DISTINZIONE semantica

cosa succede se una **receive** arriva prima della **send** sul canale punto-a-punto corrispondente?

semantica sincrona

chi prima arriva, aspetta l'altro (se punto a punto)

rendez-vous

Confronto Modelli PRAM e MP-RAM

Espressività	PRAM	>>	MP-RAM
Globalità	**PRAM**	>>	MP-RAM
Località	**MP-RAM**	>>	P-RAM
Realizzabilità	MP-RAM	>>	P-RAM

i modelli PRAM e MP-RAM si possono emulare reciprocamente

PRAM realizza il multicast/broadcast come un insieme di spazi coordinato per i diversi canali (canali come dato ed un flag)

send e receive come istruzioni che agiscono sul flag e poi sul valore da ricevere/inviare

MP-RAM realizza la memoria comune usando passi intermedi, routing e temporizzazione

Modello PRAM e MP-RAM

Problema

Il modello PRAM è troppo *potente ed astratto* **(globale)**

Il modello MP-RAM è più vicino alla implementazione **(locale)**

Gli algoritmi sono più veloci sul modello che nei sistemi reali modellati

Modelli **globali** o **non ristretti**

non impongono nessun vincolo espressivo, favorendo la espressività del modello, ma introducendo difficoltà nella implementazione

Modelli **locali** o **ristretti**

prevedono vincoli espressivi (si comunica solo coi vicini) ma si facilita la implementazione

Modelli COMPUTAZIONALI

COMPLESSITÀ degli algoritmi

dipendenza dalla dimensione del problema **N**

complessità in tempo CT(n) (abbreviato in **T(N)**)

complessità in spazio CS(n)

Pensiamo a soluzioni **parallele** con più **processori** (**parallelismo di grado P**) che sono considerati per tutta la specifica esecuzione e che possono ospitare computazione (**calcolo** ossia parti dell'algoritmo)

COMPLESSITÀ

T(1,N) soluzione **sequenziale** $T_1(N)$

T(P,N) soluzione **parallela con P** processori $T_P(N)$

INDICATORI SINTETICI

SPEED-UP *Miglioramento dal sequenziale al parallelo*

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

EFFICIENZA *Uso delle risorse*

E(P,N) = Speed-up / Numero Processori

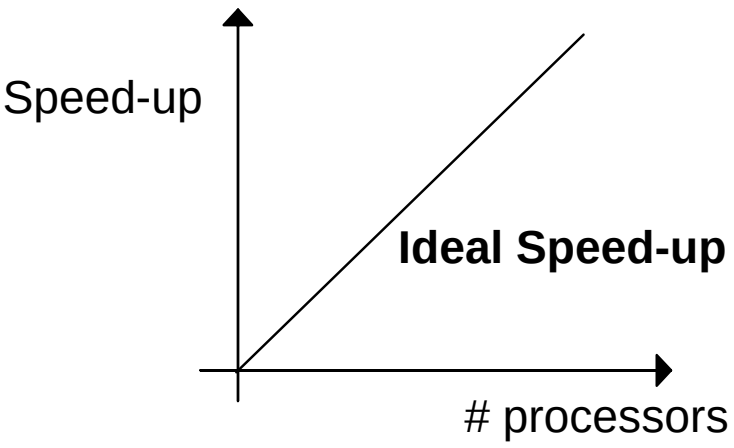
E(P,N) = $S_P(N)$ / P $E_P(N) = T_1(N) / P \; T_P(N)$

*$S_P(N)$ al **massimo P*** *ed* *$E_P(N)$ al **massimo 1***

Si parla di speed-up come miglioramento introducendo una qualche variazione nelle risorse impegnate**: in questo caso, parallelismo reale**

INDICATORI IDEALI

Stiamo ragionando su **valori medi**
SPEED-UP ideale ed EFFICIENZA



Ci interessa l'intero range di risultati, tenendo presente che possono esistere casi specifici di speed-up anomalo *non significativi e solo casi particolari* ⇨ dipendenti dall'algoritmo

FATTORE DI CARICO

legge di Grosh

Il migliore deployment per un programma è

una esecuzione sequenzializzata su un unico processore

Correlazione tra N e P: possiamo considerare

N indipendente da P, o dipendente da P

FATTORE di CARICO (loading factor) L = N / P

dependent size	(N funzione di P)
independent size	***(più interessante al crescere di N)***
identity size	(N == P)

OBIETTIVO

Trovare la migliore approssimazione per ogni algoritmo che ci interessa esaminare

SPEED-UP

Massimo **speed-up** ottenibile nel passaggio dal sequenziale al parallelo… quindi **massimo vantaggio nel parallelismo**

legge di Amdhal

lo speed-up è limitato dalla parte sequenziale

Un programma si divide in due parti:
una **parte *parallela*** ed una parte ***sequenziale***

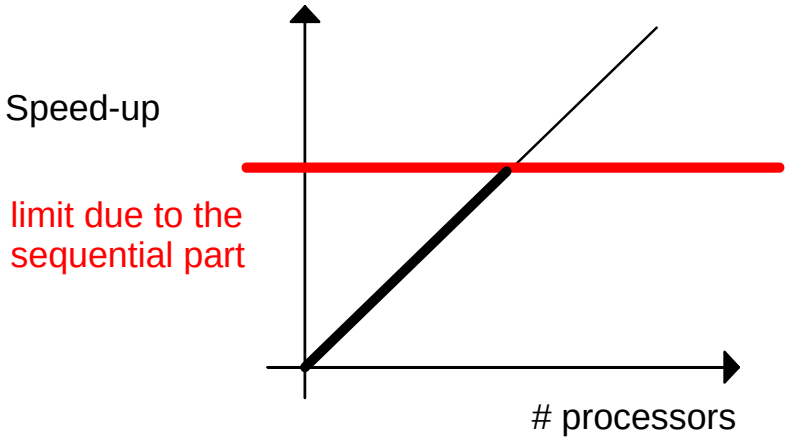
Se un programma è costituito di 100 operazioni e solo
80 possono andare in parallelo e
20 richiedono sequenzialità

Anche con 80 processori →
lo speed-up non può arrivare oltre 5

in situazione ideale, con *migliore allocazione possibile per i due indicatori,* ***speed-up fisso, efficienza varia***

INDICATORI PIÙ VICINI AL REALE

Considerando **SPEED-UP** ed **EFFICIENZA**



Non possiamo superare il limite di speedup dovuto ad Amdhal

Lo speedup viene ad essere limitato, così come la efficienza

Prima zona lineare poi speed-up costante ed efficienza cala

SPEED-UP (OTTIMO?)

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

il valore di ***P*** tale per cui si ottiene la minore complessità del problema (cioè T minimo)

in genere, questo avviene con **N/P** molto **elevato**, cioè se ogni processore è **molto carico** e con **carico significativo da svolgere (limite la parte sequenziale)**

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

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

Limite di **Amdhal** (legge di Amdhal) considera il rapporto tra le due parti dell'algoritmo in termini di **computazione** e identifica il collo di bottiglia nella parte sequenziale

Un CASO di studio (N==P)

Problema di dimensione N con uso di P processori
per il caso di problema di *somma di N interi*
Complessità del sequenziale O(N)
Complessità del modello parallelo *identity size* (N == P)

Un numero di processori pari a P e connessioni ad albero: ogni nodo foglia passa il valore somma dei due valori base; ogni nodo passa i valori dal basso come somma verso alto; la radice fornisce il risultato all'utente

$N = 2^{H+1}$ ~= $P = \mathbf{2^{H+1}-1}$ (N valori ~= P processori in albero)

$H = O(\log_2 P) = O(\mathbf{\log_2 N})$ cioè $H = \mathbf{\log_2 N} =\sim \mathbf{\log_2 P}$

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

I valori fluiscono dalle foglie in su ed ogni nodo li somma ad ogni passo (solo tempo di comunicazione)

Il CASO di studio (N==P)

Efficienza tende a zero

$L = N / P = 1$

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

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

Maggiore il numero di processori (lo speedup aumenta) ma minore l'efficienza

I processori lavorano per una frazione sempre minore dell'intero tempo di soluzione ($\mathbf{E_P(N)}$ cala al crescere di P)

Il CASO di studio (indipendent size)

Problema di dimensione N con uso di P processori

Se possiamo dividere il problema parallelizzandolo con un certo fattore di carico

Una fase locale di lavoro (somme, Comp) ed una fase di scambio di informazioni (Comm) per combinare i risultati

L = N/P

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

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

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

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

N>>P **Lo speed-up tende a P e la efficienza tende a 1**

ANCORA CASO di STUDIO

il caso di somma di N numeri con P processori con carico locale e comunicazione

Consideriamo unitario e uguale il costo delle somme e della comunicazione

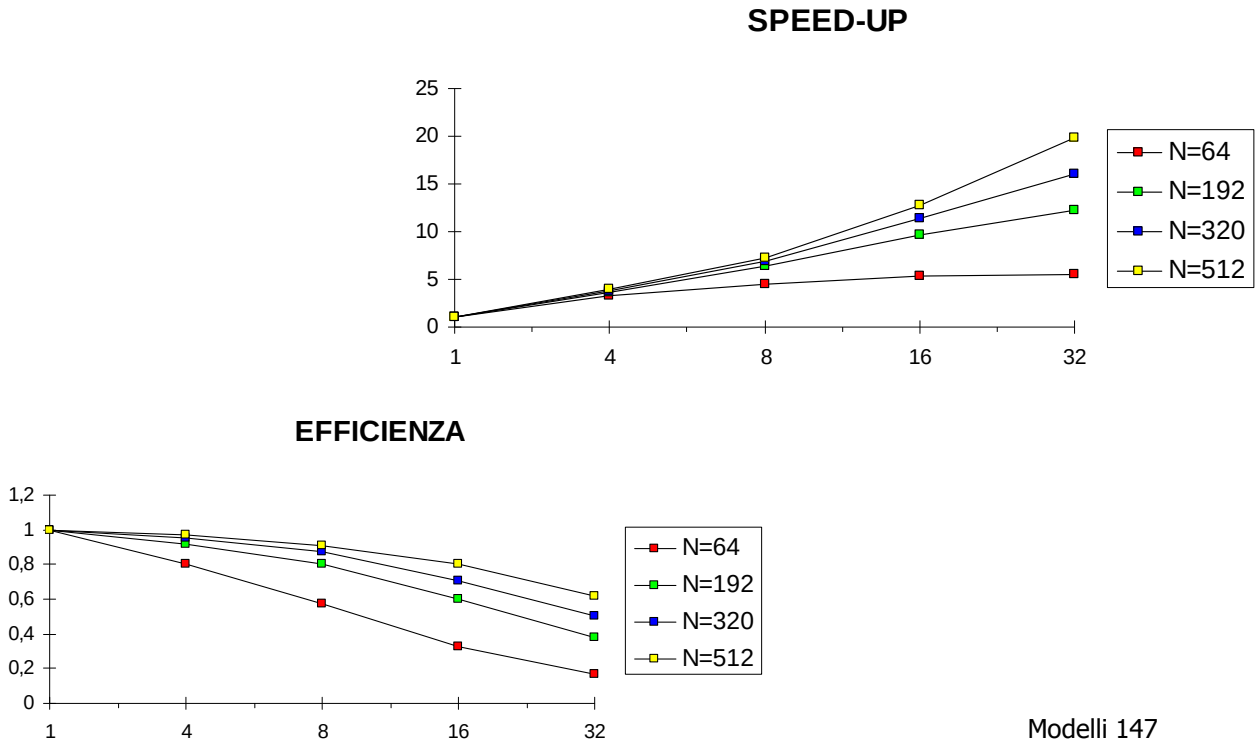
$\mathbf{T_P(N)} =\sim N/P + 2 \log_2 P$ nodi totali $P = 2^{H+1}-1$

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

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

I due indicatori dipendono sia dal numero di processori sia dalla dimensione del problema

Graficamente



Modelli 147

INDICATORI SPEED-UP ed EFFICIENZA

PROBLEMI

- schematizziamo a meno di fattori costanti
- a volte il caso peggiore può essere più significativo
- trascuriamo completamente molti elementi

Trascuriamo
movimento di dati I/O e
mappaggio (deployment specifico)

Nel mondo reale →
sono necessarie comunicazioni (non solo a regime, ma anche prima e dopo del regime)

Trasferimento iniziale dei dati
Stampa e gestione valori intermedi
Raccolta dei risultati

Modelli 148

Ancora sul CASO di studio

Complessità del modello parallelo *heavily loaded limit*
Se L cresce $\mathbf{T_{P\,HL}(P,N)} = O(L + \log_2 P) \Rightarrow O_{HL}(L)$

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

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

Cioè, intuitivamente se carichiamo ogni nodo molto ⇨
Il fattore di carico L è molto elevato

Si può raggiungere anche
uno speed-up ideale ed un'efficienza ideale
poiché abbiamo caricato bene tutti i processori e nessuno è usato poco e male

MAPPAGGIO

Assumiamo di avere mappato il problema nel modo migliore (**configurazione** e **deployment**)
Spesso non si possono fare allocazioni così facili
problemi dinamici nella comunicazione dopo la allocazione

Si può considerare la funzione di **Overhead Totale, o T_0**
cioè tenendo in conto
*le **risorse** ed il **tempo** speso in **comunicazione***

$\mathbf{T_1(N)}$ tempo sequenziale di esecuzione
$\mathbf{T_p(N)}$ tempo parallelo di esecuzione
$\mathbf{T_0(N)} = T_0(T_1, P) = P * T_P(N) - T_1(N) = |P * T_P(N) - T_1(N)|$
Se si lavorasse con efficienza massima, overhead nullo
$\mathbf{T_0(N) = 0} \Rightarrow P * T_P(N) = T_1(N)$

TEMPO di OVERHEAD

$T_0(N) >= 0$ ⇨ $T_1(N) <= P * T_P(N)$ ossia

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

T_0 rappresenta il lavoro perso

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

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

$E_P(N)$ = S / P = **$T_1(N) / (T_0(N) + T_1(N))$**

$E_P(N)$ = $1 / (T_0(N)/T_1(N) + 1)$ = **$1 / (1 + T_0(N)/T_1(N))$**

Si dovrebbero fare misure precise in uno dei due sensi e verificare le reali dipendenze: $T_0(N)$ da N e da P?

ANCORA per il CASO di STUDIO

Nel caso della somma di N numeri con P processori

Consideriamo unitario il costo somma e comunicazione

$T_P(N)$ =~ $N/P + 2 \log_2 P$ nodi totali $P = 2^{H+1}-1$

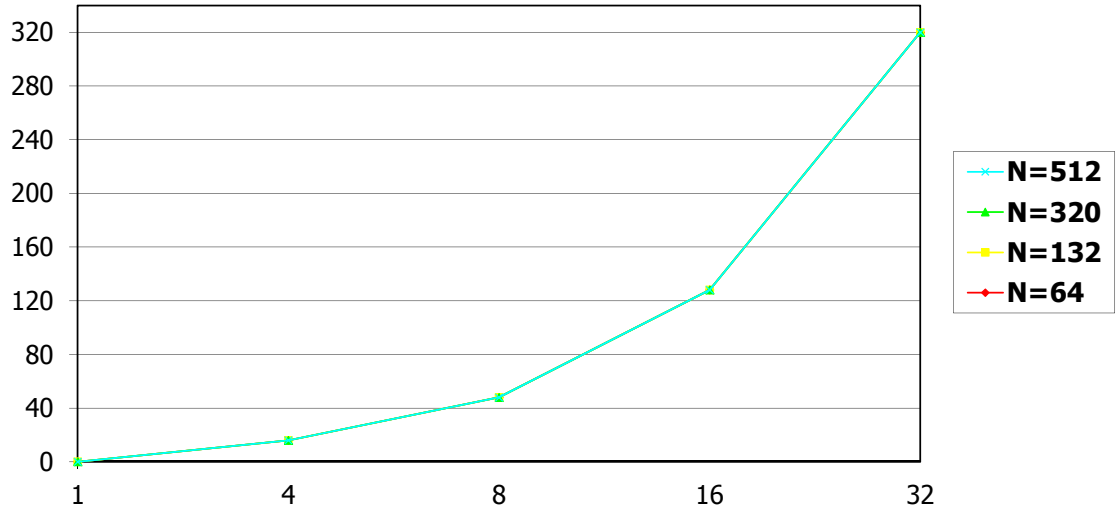
$T_0(N,P)$ = $P\ T_P(N) - T_1(N)$ =~ $P\ (N/P + 2 \log_2 P) – N$

$T_0(N,P)$ =~ **$2 P \log_2 P$**

L'overhead dipende in modo rilevante **dal numero di processori impegnati**

L'aumento è motivato dalla necessità di coordinarne il lavoro, in fase iniziale, e per la raccolta dei risultati in fase di completamento

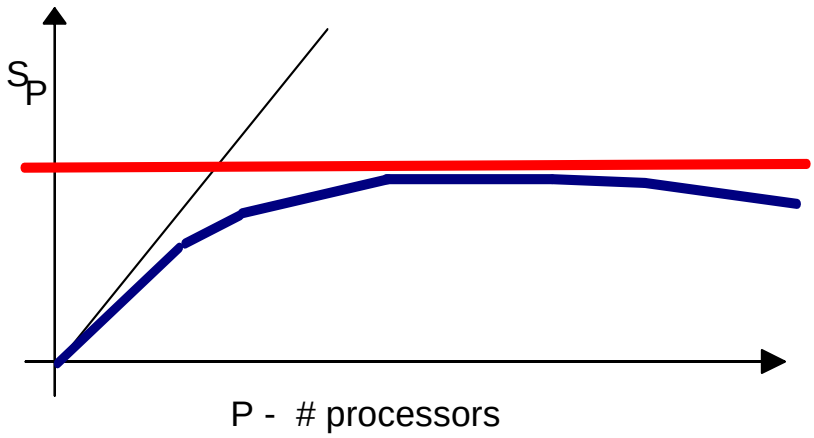
Graficamente T_o



Grafici sovrapposti

INDICATORI REALI

Considerando lo **SPEED-UP in modo meno ottimizzato**



Non possiamo mantenere lo speed up ideale

Lo speedup viene ad essere limitato dall'overhead

Prima zona lineare, poi speedup costante, poi calo a causa dell'overhead

ISOEFFICIENZA

$E_P(N,P)$ = **1/** ($T_0(N)/T_1(N)$ + **1)** **$T_1(N)$** è il lavoro utile

Obiettivo ⇨ **mantenere l'efficienza costante**

$T_0(N)/T_1(N)$ = ***(1 – E) / E*** $T_0(N) = (1 - E) / E \; T_1(N)$

$T0(N,P) = ((1 – E)/ E) \; T_1(N,P) =$ K $T1(N)$

$T_0(N,P) =$ K $T_1(N)$ considerando il **fattore K costante**

La costante (?) K caratterizza il sistema

Nel caso precedente (1 nodo /1 valore) K non costante
Nell'albero, K dipende da P ed N e vale (**$2 P \log_2 P / N$**)

FATTORE ISOEFFICIENZA

Funzione isoefficienza
Se teniamo costante N e variamo P, **K determina se un sistema parallelo possa mantenere un'efficienza costante**
→ cioè **uno speed-up ideale**

Se K è piccola ⇨ ***possibile alta scalabilità***
Se K è elevata ⇨ ***poca scalabilità***
K può non essere costante ⇨ sistemi non scalabili

Nell'albero, K vale $2 P \log_2 P / N$

sistema scarsamente scalabile
I sistemi reali sono tutti scarsamenti scalabili (sic ☹)

CASO per RIFLESSIONE

Data una applicazione costituita da Q processi
con infiniti processori a disposizione
come gestire la allocazione dei processori?

argomentate su quanti processori usare

Come sono i processi?
Quale è la interazione?
Quanto è necessario caricare ogni singola macchina?
La applicazione basata su oggetti?
E le classi sono replicate?

La legge di Grosh dice che è bene usare un unico processore se la cosa è possibile e praticabile **MAI!**

STIMOLI per RIFLESSIONE

Se volessimo considerare l'esperienza dei sistemi concentrati di calcolo

heavily loaded limit come buona situazione
buona efficienza con processori molto carichi

Quale è la vostra esperienza di utenti di PC e workstation?
Cosa dice la legge di Grosh?
Che senso ha parlare di processi leggeri e pesanti?
Cosa ci guida per la efficienza?
Come calcolare il fattore di carico in termini di processi per processore? Numero di processi per processori

Ma **quanti processi** sono ragionevoli?