

MANAGEMENT DEI PROCESSI

PROCESSI *entità di esecuzione nel sistema*

Processo sequenziale

*come esecuzione sequenziale di azioni sui dati
tipicamente mappato su un unico processore*

Le **risorse** usate dal processo sono fornite
*o alla creazione del processo
o su richiesta dello stesso*
meglio **se le azioni sono locali o vicine**

Sono necessari costrutti o primitive
per *definizione dei processi
creazione/distruzione dei processi
acquisizione/rilascio delle risorse*

Meccanismi per i processi
descrittori dei processi
stati dei processi

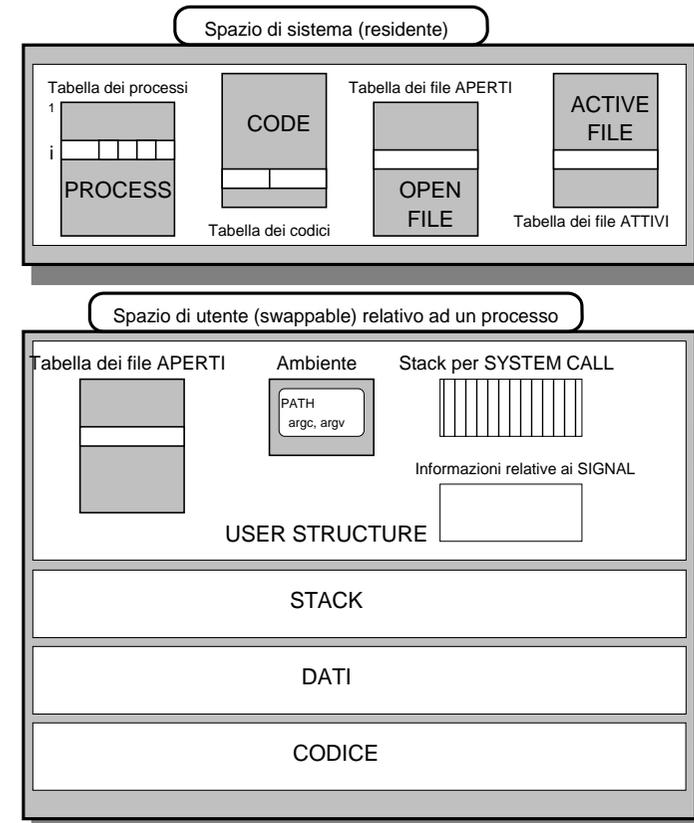
Nel distribuito

Problema fondamentale

Allocazione dei processi ai processori
(statica e dinamica)

modello dei processi

processi pesanti/ processi leggeri
Unix



processi **fissi e mobili**

fissi residenti dove sono stati creati
mobili possono essere riallocati

processi **statici e dinamici**

statici creati all'inizio della applicazione
dinamici creati quando sono necessari

Peso dei processi

Sono processi anche i **thread** o **lightweight processes**

Processi pesanti:

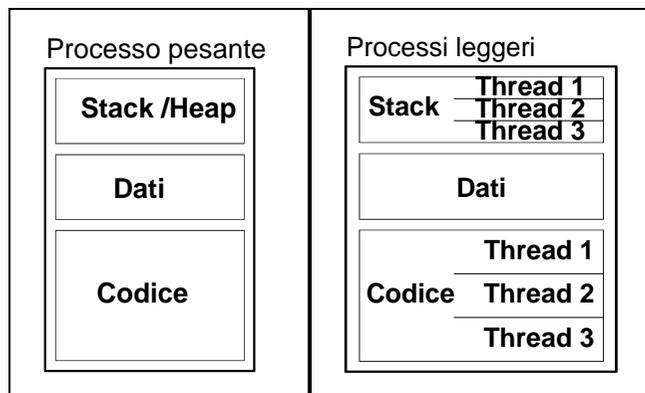
costo elevato dello scheduling

Processi leggeri e medi:

riduzione del costo

più facile condivisione e comunicazione

Condivisione delle risorse tra processi utente

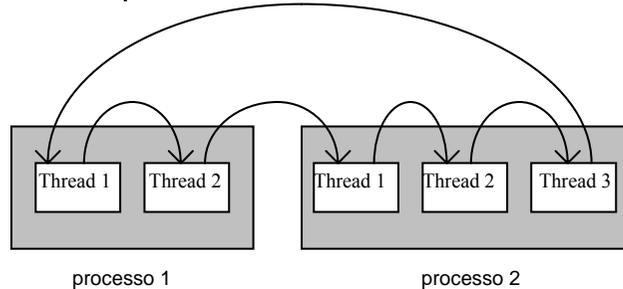


UNIX, Charlotte solo processi pesanti

V-Kernel processi pesanti e medi

Mach processi leggeri e medi

Lo **scheduler** quindi deve tenere conto dei **thread**



In caso di sistemi esistenti

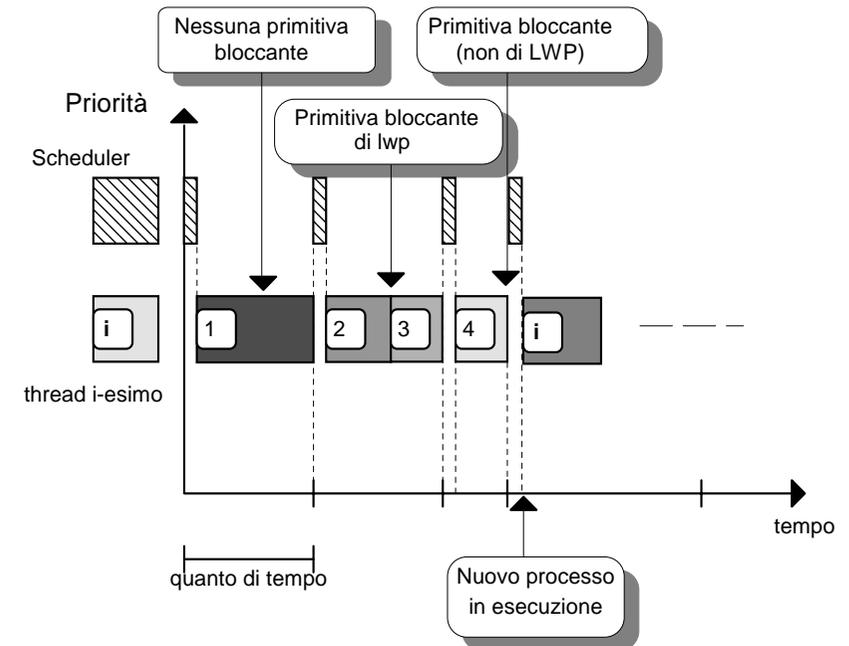
La realizzazione dei **processi leggeri** senza modifica del kernel introduce

Problema delle **primitive sospensive**

Se un thread esegue una primitiva sospensiva, viene sospeso l'intero processo (anche tutti gli altri thread)

==>

Si introducono soluzioni ad-hoc



Ossia i thread non possono fare primitive sospensive (tipo read) ma solo primitive non bloccanti e mettersi da parte esplicitamente

Soluzioni

- 1) Uso di **primitive ad-hoc**, che svolgono la azione di sospensione del thread e ne riattivano un altro, senza interessare il kernel
- 2) Uso di primitive non sospensive solamente **select e primitive asincrone**
In un **wrapper** che avvolge la chiamata alla primitiva e garantisce la non sospensione

lwp LightWeight Processes (per SUN BSD)

libreria al disopra del kernel
con thread in priorità

Gestione dei thread LWP

Inizializzazione

```
int pod_setmaxpri(maxpri)
int maxpri;
```

Il main è il thread con la massima priorità disponibile e prosegue in concorrenza a thread creati

```
lwp_create(tid, fun, prio, flag, stack,
           nargs, arg1, .., argn)

thread_t *tid;
void (*fun)(); /* codice */
int prio; /* priorità thread */
int flag;
stkalign *stack;
int nargs;
int arg1, ..., argn;
```

scheduling dei thread

Un **thread** esegue come dispatcher degli altri e realizza una **politica di scheduling**

round-robin:

```
while(1)
{ lwp_sleep(TIME_SLICE);
  lwp_resched(PRIORITA);
}
```

anche altri schemi

- un **thread** generato per ogni richiesta dall'esterno ed un **dispatcher implicito** tra **thread**
- scheduler dei thread come una **pipeline** di processi sempre presenti e unquonato dedicato a ciascuno

I package devono fornire anche strumenti di sincronizzazione tra **thread**

ad esempio

semafori di mutua esclusione
variabili condizione
sospensioni temporizzate

DCE Distributed Computing Environment

definito a livello di **Open System Foundation OSF**

package standard per fornire i thread
disponibile in System V

Funzioni di:

- creazione/distruzione di thread
 - creazione di template per i thread
 - gestione dei costrutti di mutua esclusione
 - gestione delle variabili condizione
 - creazione e gestione di variabili condivise dai thread
 - gestione delle autorizzazioni
-
- altri servizi: autenticazione, gestione risorse

Non sono necessarie funzioni di **kernel** ad-hoc per le **primitive**

Il sistema DCE rappresenta un accordo tra realtà di mercato proprietario diverso e standard accademici accettati

RPC integrate con i processi leggeri: ogni operazione viene eseguita in un **thread** generato

Soluzioni in kernel vs. applicative

User-level
kernel level

☺ efficienza rispetto a processi pesanti

gestioni nel kernel: kernel-level thread (Mach, V)

☺ integrazione degli strumenti nel kernel

☹ meno efficienti di realizzazioni a livello applicativo
politiche general-purpose

gestioni applicative: User-level thread

☺ possibilità di variare le politiche secondo necessità

☹ meno efficienti di realizzazioni ad hoc
interferenza con le politiche di kernel

gestioni miste: FastThread

☺ possibilità di integrare le politiche secondo necessità e di ottenere i vantaggi di entrambe

Supporto a thread a due livelli

Si comincia ad affermare l'idea di avere gestioni a due livelli dei thread

thread a **livello di applicazione**

thread a **livello di kernel**

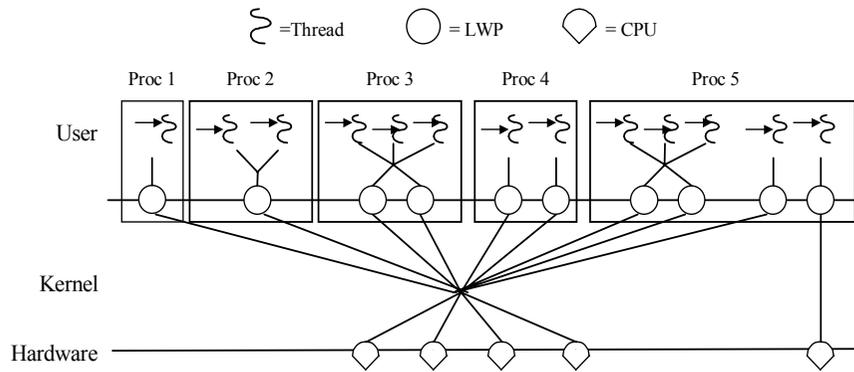
L'utente deve specificare anche come le cose possono essere messe insieme

Solaris

Gli utenti definiscono i *processi applicativi*

Sono tenuti a definire per ciascun processo pesante

- i **thread** logici che li compongono
- gli **lwp** processori virtuali che richiedono ed i legami con i processori disponibili

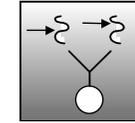


Lo schema consente di legare i thread applicativi in modi molto differenziati ai processori in esecuzione

Scheduling delle attività

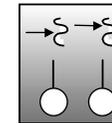
SunOS 5.x prevede molti possibili modelli di esecuzione

1) molti a uno



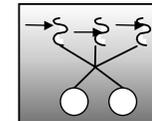
Più Thread utente schedulati su un unico LWP
primitive bloccanti -> blocco di tutto il processo
scheduling a livello utente
(HP-UX e DCE)

2) uno a uno



Un Thread utente per ogni LWP
primitive bloccanti -> blocco del thread
scheduling a livello kernel
IBM AIX, Microsoft Windows NT, IBM OS/2, LINUX

3) molti a molti

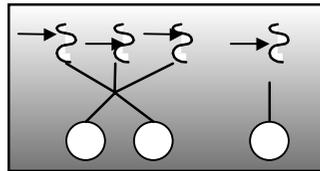


diversi Thread utente per ogni LWP
primitive bloccanti -> nessun blocco del thread
DEC Unix, Silicon Graphics IRIX

Scheduling con modello a due livelli

SUNOS 5.x

Solaris 2.x



scheduling a due livelli

con eventuale binding di un **thread** ad un **LWP**

Esigenze di **real-time**

in caso di **multiprocessore**

Particolarmente studiato per ottenere il numero di IWP necessari alla migliore esecuzione

In caso che non ce ne siano disponibili, possono essere creati automaticamente su necessità

se ci sono thread da eseguire

`thr_setconcurrency()`

Primitive di thread

API molto differenziate e poco portabili

non sono compatibili POSIX

che propone primitiva simili, ma diverse

(anche per diverse versioni)

I thread hanno anche memoria comune tra loro o privata

visibile a gruppi

oltre ai mutex, lock, condizioni, etc.

API thread

```
int thr_create (void *stack_base,
               size_t stack_size,
               void *(*start_routine) (void *),
               void *arg, long flags,
               thread_t *new_thread);

void thr_exit (void *status);
int thr_join (thread_t wait_for,
              thread_t *departed,
              void **status);

void thr_yield (void);
int thr_suspend (thread_t target_thread);
int thr_continue (thread_t target_thread);

int thr_keycreate (thread_key_t *keyp,
                  void (*destructor) (void *value));
int thr_setspecific (thread_key_t key,
                    void *value);
int thr_getspecific (thread_key_t key,
                    void **valuep);

int thr_getconcurrency (void);
int thr_setconcurrency (int new_level);

int thr_getprio (thread_t target_th, int *pri);
int thr_setprio (thread_t target_th, int pri);
int mutex_init (mutex_t *mp, int type,
                void *arg);
int mutex_lock/mutex_unlock/mutex_trylock/
mutex_destroy (mutex_t *mp);

int cond_init, cond_wait, cond_signal,
cond_broadcast , cond_timedwait
int sema_init, sema_wait, sema_trywait
etc.
```

Gruppi di Processi

Insieme di Processi che sono visti come una unica entità astratta per alcune azioni

Ad esempio: modello multi-server a multicast

Semantica di Gruppo

- un gruppo deve essere dinamico
- un processo può appartenere a più gruppi
- un gruppo deve essere riconosciuto in termini di comunicazione

Realizzazione attraverso il broadcast

Gruppi chiusi

solo i componenti possono inviare al gruppo

Gruppi aperti

la astrazione di gruppo è visibile ad ogni altra entità

Gruppi chiusi

per la realizzazione del parallelismo

Gruppi aperti

per la realizzazione di cliente/servitori multipli

Gruppi di Processi

Struttura ed organizzazione dei gruppi

Processi pari

tutti i processi sono pari

Processi gerarchici

presenza di un coordinatore e di coordinati

Appartenenza ad un gruppo

Un processo gestore

Esiste un processo che consente di entrare a fare parte di un gruppo e di lasciare il gruppo

Realizzazione Distribuita

tutti i processi mantengono la lista di appartenenza
Identificazione dei guasti distribuita

ISIS

con le primitive di broadcast rappresenta lo stato dell'arte a livello di comunicazione di gruppo

FBCast	(FIFO Bx)
CBCast	(Causal Bx)
ABCast	(Atomic Bx)
GBCast	(Group Bx)

Scheduling dei Processi

Scheduler

allocazione dei processi al processore

Dispatcher

assegnamento del processore ad un processo

Lo scheduling è la parte che può essere distribuita

politica locale

politica globale

Scheduling Locale

Charlotte round-robin

V-kernel priorità

Accent 16 livelli di priorità variabile e time-slice

Necessità di Scheduling Globale

operazioni remote sui processi

meccanismi

gestione risorse remote

politiche

LOAD SHARING

utilizzo delle risorse in modo che nessun processore sia idle

LOAD BALANCING

bilanciare l'uso delle risorse per ottenere un carico equilibrato

Politiche di allocazione

STATICHE e DINAMICHE

LOAD SHARING

Fatto a priori e durante l'esecuzione
senza muovere gli elementi una volta allocati

LOAD BALANCING

Fatto durante l'esecuzione
migrando elementi anche dopo che hanno eseguito

*assumendo la **conoscenza 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***

In particolare, i costi sono fondamentali per

valutazioni statiche

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

- algoritmi esatti **problema NP**
- algoritmi euristici

Genetici, Tabu search

Spesso queste strategie non si possono applicare durante la esecuzione

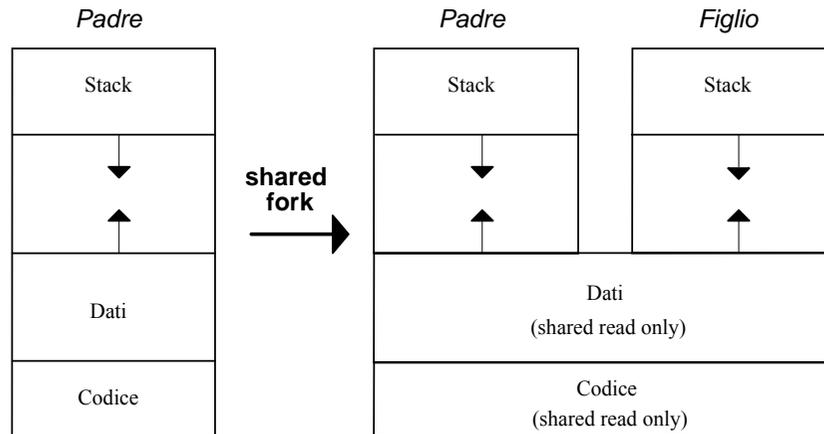
valutazioni dinamiche

obiettivo ==> **riduzione dell'overhead**

Operazioni remote sui processi

creazione remota / terminazione remota
esecuzione remota

in SPRITE **fork remota**
e condivisione variabili



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

file aperti
risorse di comunicazione

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

*le richieste vano sempre diretta a gestori che distinguono i servizi **locali** e servizi **remoti** fornendo una unica interfaccia all'utilizzatore*

Esecuzione remota

possibilità di attivare un processo su un nodo diverso e di interagire disponibile 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> @ *`

Eterogeneità

- non esiste uno spazio globale dei nomi per le entità da riferire
- esistono diverse convenzioni per definire i servizi e gli attributi (sintassi dei comandi, etc.)
- è necessaria una traslazione delle informazioni da uno spazio ad un altro

Requisiti

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

IMPLEMENTAZIONE

Distinguiamo **servitori** di calcolo e **clienti**
GESTORI con un servizio di **registrazione**

- i nodi servitori si registrano come fornitori di servizi
- i nodi clienti accedono alle informazioni

Due fasi:

Inizializzazione

La richiesta per la creazione di un nuovo processo segue i normali meccanismi

Si interessa un nodo servitore remoto, anziché locale

Richiesta in **broadcast a più servitori**

Un servitore richiede al kernel la creazione e
inizializzazione

Esecuzione

Non c'è differenza rispetto al caso locale:

- spazio inizializzato allo stesso modo
- ogni riferimento è indipendente dalla località
- kernel e gestori sono omogenei in locale o remoto

Terminazione

- uso di primitive di abort
- distruzione dei costrutti per il processo (porte)

GESTIONE RISORSE

*definizione di **risorsa***

ogni componente riusabile o meno, sia hardware, sia software necessario alla applicazione o al sistema

Classificazione

- **risorse** basate sulla **astrazione**
specifica (interfaccia visibile) e
implementazione
- **risorse fisiche** vs. **risorse logiche**
tutte distribuite
- **risorse di basso livello** vs. **risorse applicative**

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

- **pianificazione** della organizzazione e identificazione
allocazione
disponibilità
costo
- **controllo** delle risorse (loro uso)
controllo di accesso
ottimizzazione
autenticazione
controllo di correttezza operazioni ed eccezioni

Realizzazioni

concentrate e distribuite
accuratezza delle informazioni

Esempi: risorse tipiche

processori e sistema di comunicazione

Criteria di performance

Delay ritardo nel produrre il servizio

service time

durata media di un servizio

waiting time

tempo di attesa per un servizio

response time

tempo di risposta per il completamento di un servizio

Tutti gli indicatori di tempo sono anche indicatori del carico del processore

throughput

servizi nell'unità di tempo (di osservazione)
a diversi livelli

Si possono introdurre anche altri criteri legati alla gestione economica della risorsa

costo

tenere conto dei costi aggiuntivi di sistema conseguenti ad una azione di sistema

minima intrusione =>

limitare l'overhead del supporto

Descrizione delle risorse

attributi di una risorsa

classe della risorsa

nome della risorsa

altri attributi correlati

esecuzione vincolata su una determinata macchina

con **fattori**

qualità del servizio

tempo di allocazione

costo

tempo di ritardo in comunicazione

gestione risorse

uso di servitore (complesso) vs. solo dati

composte vs. multilivello

mobili vs. immobili

in caso di movimento e replicazione

con consistenza forte vs. con consistenza debole

Controllo allocazione

allocazione della risorsa e del nome

Controllo accesso

locale vs. remoto

Condivisione delle risorse

Due modelli principali di condivisione delle risorse

service request

file system distribuito

Service Request

Servizio specifico con **richiesta esplicita** dell'utente

modello Cliente/Servitore (C/S)

File System Distribuito (FSD)

Servizio unico con **trasparenza** alla **allocazione delle risorse**

servizi completamente presenti, o solo in alcuni nodi (spesso detti **file server**)

modello ad Agenti

Modelli di risoluzione del problema

Servizi (ed informazioni)

semplici vs. **multipli** (oggetti)

statici vs.

dinamici

centralizzati vs.

distribuiti

Modello a server

Un unico servitore (centralizzato o distribuito)

- per tutte le risorse
- che gestisce una sola classe di risorse

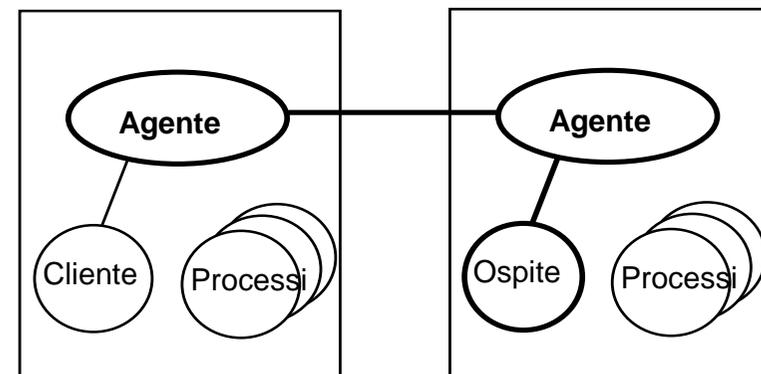
Uso di tabelle per le informazioni delle risorse

Affidabilità ==> Replicazione

Unico sistema di agenti per il servizio

agente 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**

LOAD SHARING

DETERMINARE

quali processi, quando, dove muovere

Implementazioni distribuite

alla ricerca di processori liberi in grado di essere utilizzati

Organizzazioni possibili

interconnessione

STATICA vs. DINAMICA

processori in ring logico

statica

processori in gerarchia logica MICROS

dinamica

processori liberi (worm)

dinamica

Ring logico

Ancora una struttura logica per la ricerca con un token in un anello

Si usa un broadcast iniziale a tutti della richiesta di esecuzione

si distribuisce il carico secondo le risposte

struttura statica

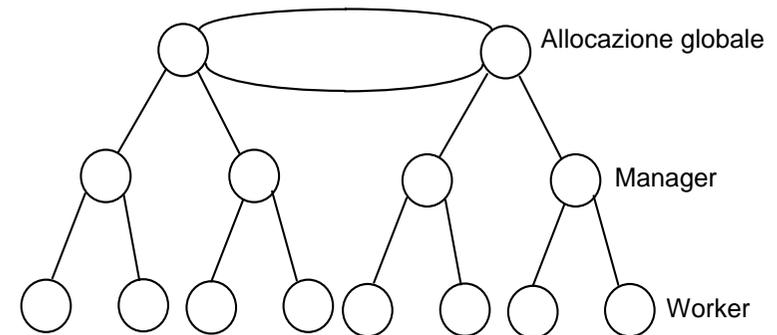
Gerarchia logica MICROS

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 ma logici: non ci sono connessioni dirette



Worker ==> funzioni di calcolo ed I/O (**slave**)

Manager ==> funzioni di gestione

Numero di livelli dipende dal numero di worker

Guasti

master ==> più nodi master

slave ==> il master deve poter comandare i livelli sottostanti

Allocazione statica

Si allocano un numero di processori worker adatti ed 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

Approccio nuovo

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

Worm/verme fatto di uno o più segmenti, cioè processi che possono anche comunicare tra loro

Obiettivo del worm

arrivare ad una copia del verme per nodo

Applicazione sta al disopra

Un verme si incarica di cercare i nodi liberi attraverso

clonazione su nodi liberi

uso *di messaggi di richiesta di accettazione* (detti **probe**) mandati dai segmenti che vogliono espandersi

Metodologia decentralizzata

NON si conosce lo stato del verme globalmente

LOAD BALANCING

MIGRAZIONE => OBIETTIVI

- uso delle risorse **più CORRETTO ed EFFICIENTE**
disomogeneità delle risorse fisiche
file partizionati e presenti su nodi specifici
- **BILANCIAMENTO del carico computazionale**
divisione del carico tra i nodi secondo necessità
- **TOLLERANZA ai guasti**
una risorsa può essere trasferita in altro nodo prima del crash totale
- **DINAMICITÀ e MOBILITÀ**
possibilità di fare fronte a una allocazione anche non ottimale, o non più ottimale

Requisiti

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

Eterogeneità

diverse architetture e modelli computazionali

caratteristiche dell'ambiente

- capacità di osservazione (monitor)
- affidabilità
- dinamicità

MONITORING

Identificazione del carico del sistema usando osservazioni sul carico corrente

assumendo *continuità della applicazione e
gradienti limitati*

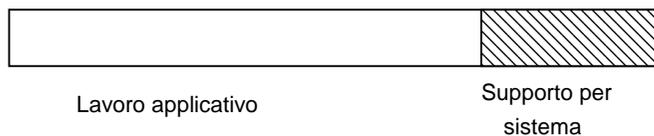
Raccolta di informazioni di carico su processori, risorse e comunicazione

- ad eventi
- osservazioni su intervallo limitato
- dati statistici

*Le informazioni monitorate sono usate per la **previsione**
delle variazioni del carico nel futuro*

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

Percentuale di occupazione delle risorse



monitoring può non servire nel load sharing

PRINCIPIO di NON INTRUSIONE

i livelli di supporto di sistema non devono apportare una
variazione sostanziale sulla performance dell'applicazione
Si deve quindi assolutamente limitare il loro overhead

Altri requisiti per la migrazione

- **Pre-emption**
priorità nell'uso locale
- **Evitare dipendenza residue**
lasciare traccia nel sistema delle migrazioni può
diventare costoso
- **Evitare thrashing**
movimento di un solo processo che non riesce a
proseguire
- **Migrazioni multiple**
consentire concorrenza nella migrazione per
parallelizzare: gli strumenti più complessi

MIGRAZIONE DI PROCESSI

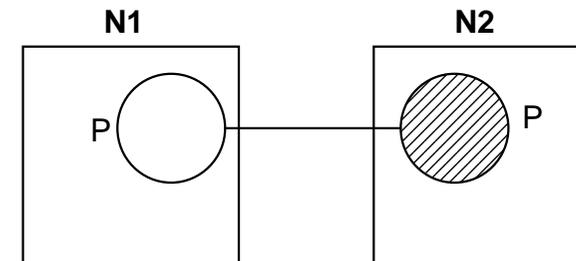
entità computazionali **mobili**

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



MIGRAZIONE DI DATI
MIGRAZIONE DI OGGETTI
MIGRAZIONE DI AGENTI

In caso di migrazione

e durante la migrazione

*PROBLEMA dei MESSAGGI in fase di consegna
messaggi da spedire a chi sta migrando*

Cambiamento di nome della risorsa mobile

strategie pessimiste/proattive

- **Ridirezione dei messaggi**

bufferizzazione dei messaggi arrivati per il processo da parte del kernel, che riceve poi i messaggi
Il nodo di partenza tiene traccia della allocazione del processo; i clienti mandano al vecchio nodo

- **Riqualficazione dell'allocazione**

i messaggi arrivati per il processo sono mantenuti ma i clienti sono informati della nuova allocazione
Si mandano messaggi a tutti i potenziali clienti
Il nodo di partenza tiene traccia della allocazione del processo solo fino al completamento del trasferimento

strategie ottimiste/reattive

- **Recovery da parte dei clienti**

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

DEMOS/MP (1977)

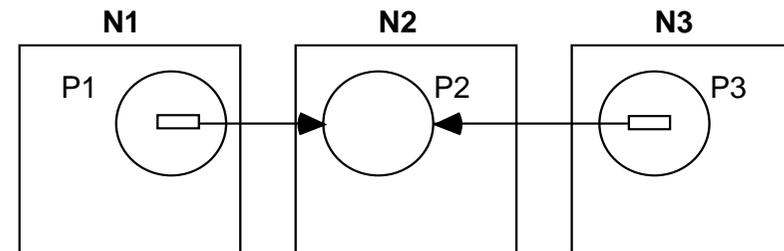
Kernel di sistema distribuito

- sistema a **scambio di messaggi**
- comunicazione attraverso entità intermedie (**link**)
- i link sono **associati** ai processi riceventi
- i link possono essere **passati** nei messaggi

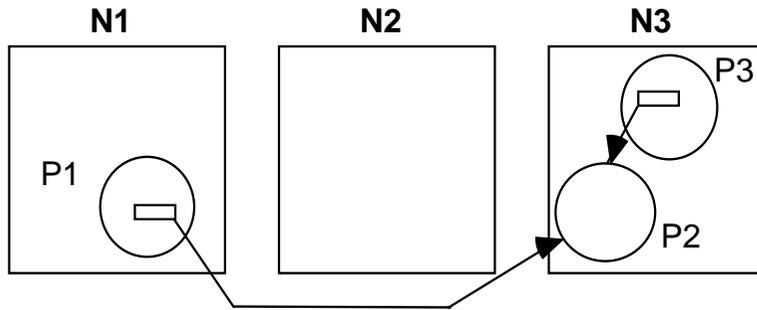
Caso di **MIGRAZIONE**

un processo P2 (ciclico) si sposta da un nodo N2 a N3

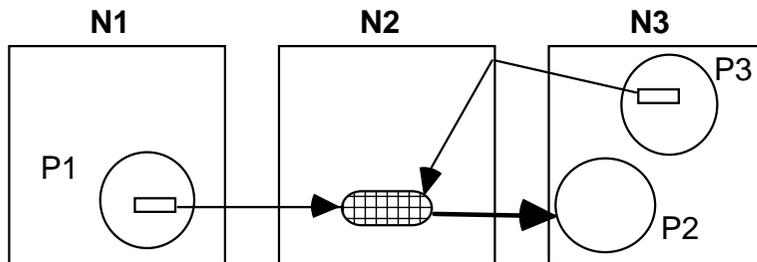
Situazione Iniziale



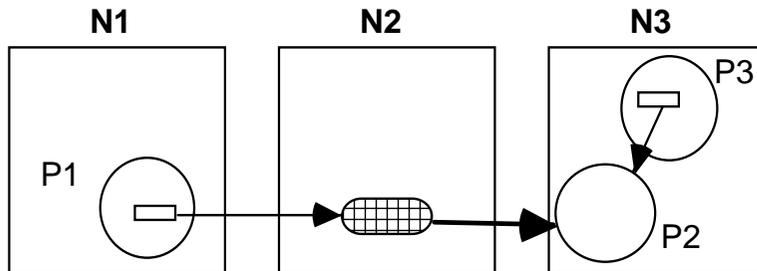
Situazione finale



Situazioni Intermedie



Il forwarder invia alla nuova locazione di P2



Riqualficazione del link da P3 a P2

PROTOCOLLO di MIGRAZIONE

Necessità di:

- **Blocco** di P2
 - **Trasferimento** dello stato di P2 da N2 a N3
 - codice e dati (stato iniziale)
 - stato (stato corrente)
 - messaggi
 - vecchi **link**
 - resta in N2 una entità che fa procedere i messaggi
- forwarding address**

in alternativa: scarto messaggi
problema della *ritrasmissione*

- i link possono essere **riqualificati (riqualificazione)**
la informazione della locazione corrente può essere sostituita alla obsoleta

a *riqualificazione* completata di tutti i link,
il forwarding address scompare

Costo **migrazione** + costo **updating** link

Requisito
TRASPARENZA della allocazione

V kernel (1983)

Kernel di sistema a scambio messaggi

obiettivi: **efficienza e tolleranza errori**

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

<attesa>

receive

receive

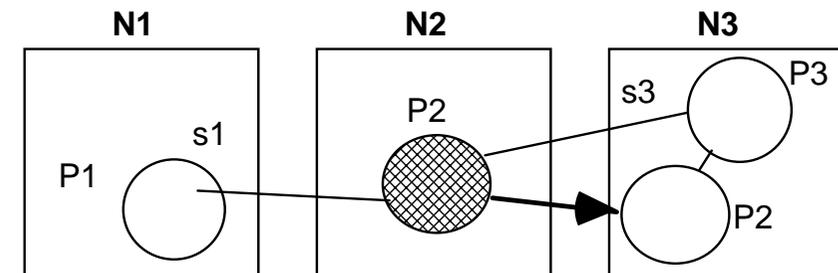
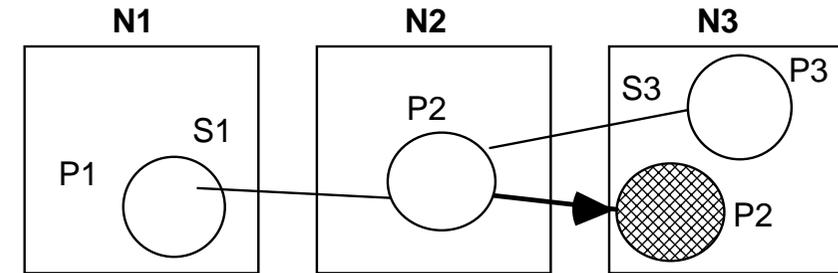
<operazione>

reply

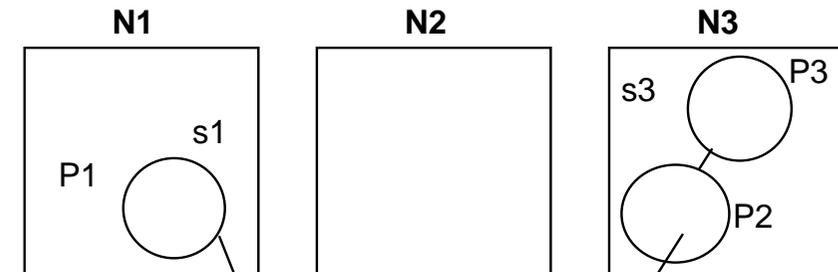
Caso di **MIGRAZIONE**

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

V kernel



I processi reinviano i messaggi (P1 e P3)



Completo spostamento di P2

PROTOCOLLO

Possibilità di:

- **Copia** dello stato di P2 da N2 a N3
fatta in precedenza
precopying 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
- si riqualificano le **cache di indirizzo**
in genere via *broadcast*

TRASPARENZA della ALLOCAZIONE

attraverso l'accesso al **kernel locale**

MIGRAZIONE di FILE

associata alla replicazione

alcuni problemi **comuni**, altri **risolti**

altri **introdotti** dalla **replicazione**

Altri sistemi (p.e. MOS), cominciano a
determinare e **sperimentare** delle
politiche di migrazione, oltre a definire i **meccanismi**

MECCANISMI

dipendenti dal modello computazionale
dallo specifico sistema

POLITICHE

indipendenti dal sistema
e non **embedded** nel kernel
ma **general-purpose**

criteri

- non tutti i processi migrano
fissi gli aciclici e i dipendenti dal nodo
- ci può essere un gestore della migrazione per
nodo

DETERMINARE

quando, chi, come, dove migrare

MECCANISMI di MIGRAZIONE

CHI migra?

- processi
- oggetti passivi (file)
- oggetti

Costituzione della RISORSA

- codice + dati (stato iniziale)
- stato corrente
- risorse cui accedere
- risorse locali e remote

BLOCCO computazione

- blocco** dello stato
- dall'esterno
- messaggi
- trasferiti
- rifiutati

TRASFERIMENTO e COPIA

- Sincronizzazione tra copia vecchia da distruggere e nuova da attivare
- Riferimenti obsoleti
- riqualificazione**
- variazione su necessità**

MIGRAZIONE - POLITICHE

FASI costituenti

VALUTAZIONE carico (V)

- carico locale
- globale

TRASFERIMENTO (T)

- chi trasferire
- quando trasferire

LOCAZIONE (L)

- dove migrare e
- reinserire il processo

T e L sono spesso legate e interdipendenti

SCHEDULING

- impatto sullo scheduling
- del nodo di partenza
- del nodo di arrivo

POLITICHE di MIGRAZIONE

STATICHE

predefinite e **decise a priori**

- V** carico **soglia** fisso (p.e. numero processi)
- T** movimento del processo più "nuovo"
- L** migrazione da un certo nodo sorgente sempre a un predefinito destinatario

SEMIDINAMICHE

predefinite con **limitate dipendenze** dallo stato corrente (o anche politiche probabilistiche)

- V** carico soglia variabile
- T** identificazione ciclica tra i processi
- L** allocazione su destinatario ciclico

DINAMICHE

non predefinite ma **strettamente dipendenti** dallo stato corrente

- V** confronto con carichi dei nodi vicini (**carico medio**)
- T** informazioni sullo stato dei processi e decisioni conseguenti
- L** ricerca dei potenziali nodi destinatari

Politiche di CARICO, TRASFERIMENTO e LOCAZIONE

Processi aciclici vs. ciclici

POLITICHE SEMPLICI vs. COMPLESSE

CARICO ==> a soglia fissa
confronto con vicini

TRASFERIMENTO ==>
processo adatto per vicino predeterminato
o random
probe: verifica di alcuni vicini

LOCAZIONE ==> uso di **probe**
random
probabilistiche *accettazione*
cycle *incondizionata*
shortest queue

probing *accettazione*
bidding *condizionata*

PROBING (T e L congiunti)

identificazione di alcuni candidati a ricevere i processi
e valutazione del loro stato

DECISIONE di MIGRAZIONE

CENTRALIZZATA

unica entità **controlla** i movimenti ==>
collo di bottiglia

DECENTRALIZZATA

raccolta implicita o **esplicita** di informazioni
decisione basata su confronto di informazioni
piggybacking di informazioni di carico
confronto su base locale con stato degli altri nodi (vicinato) e non azioni globali di scambio

RESPONSABILITÀ della MIGRAZIONE

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**

Protocolli di cooperazione

BIDDING (Contract Net)

protocollo di negoziazione tra possibili nodi coinvolti per cooperare

SENDER-INITIATIVE (source)

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

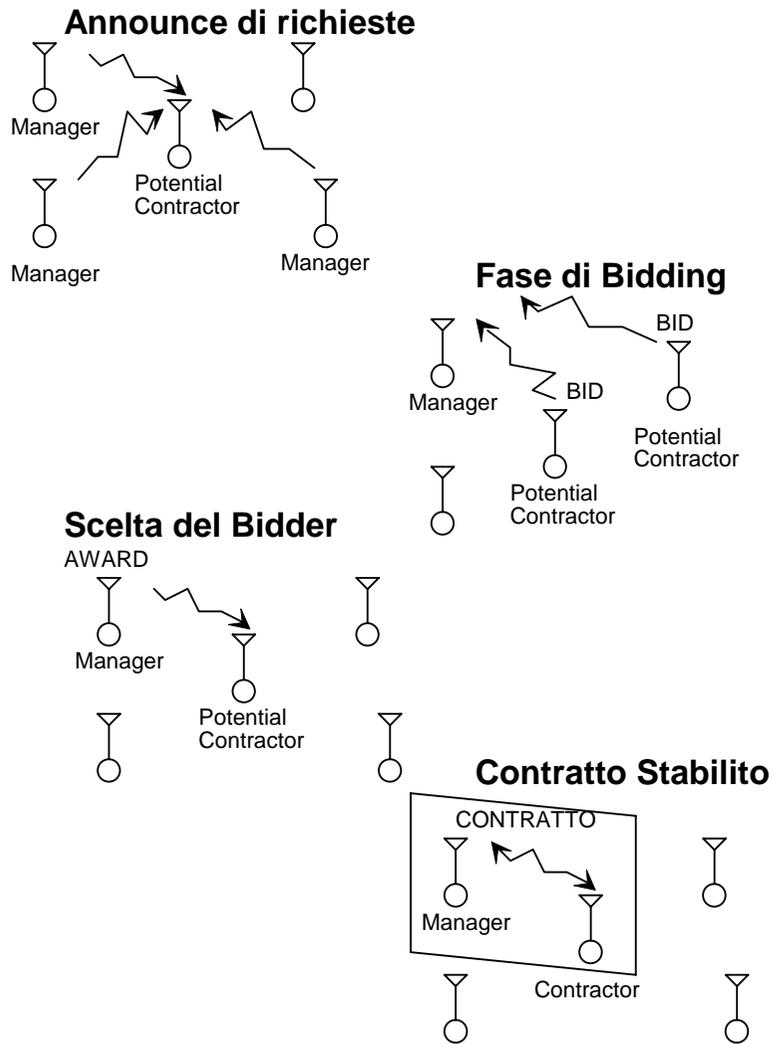
Il sender sa che i receiver potenziali non sono **prenotati** ma hanno solo aderito alla richiesta in base allo stato locale

Il sender **sceglie** tra i potenziali, pronto ad altre scelte in caso di rifiuto e **molti round** possibili

Molti problemi di implementazione

- come determinare **interesse** e **criteri di scelta**
- determinare **vicinato**
- **deadline** del bid
- **limitare** la trasmissione di informazione
- **non ottimalità** decisioni

BIDDING ==>SELEZIONE MUTUA molto FLESSIBILE ma COSTOSA



RICOSTRUIRE STATO GLOBALE via
COOPERAZIONE DEI NODI INTERESSATI

		SEND1	SEND2
bid	REC1	0.9	0.7
	REC2	0.7	0.1

COSTO del meccanismo di migrazione

Charlotte - Migrazione di Processi

TempoMigraz (msec) =
45 +

12.2 n (*n* = numero di 2 kbyte del processo)
+ { *fattore link remoti* }

9.9 +
1.7 ln (*ln* = numero link remoti)

(Charlotte)

reliable message send 2kbyte = 11ms

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

RISULTATO IMPORTANTE

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

Politiche più sofisticate

e.g. muovere il processo più a lungo in attesa
non ottengono miglioramenti significativi
da controbilanciare la complicazione introdotta

- STABILITÀ

evitare thrashing

- EFFICIENZA

algoritmo decisione ed attuazione

- OTTIMALITÀ

subottimalità

In caso di sistemi **REAL-TIME**

valutazioni molto più **critiche** e
politiche anche più **sofisticate** di necessità

DRAFTING Protocol [Ni]

protocollo di bidding
efficiente e corretto

Classi di carico

alto, normale, basso

Iniziativa **al receiver**

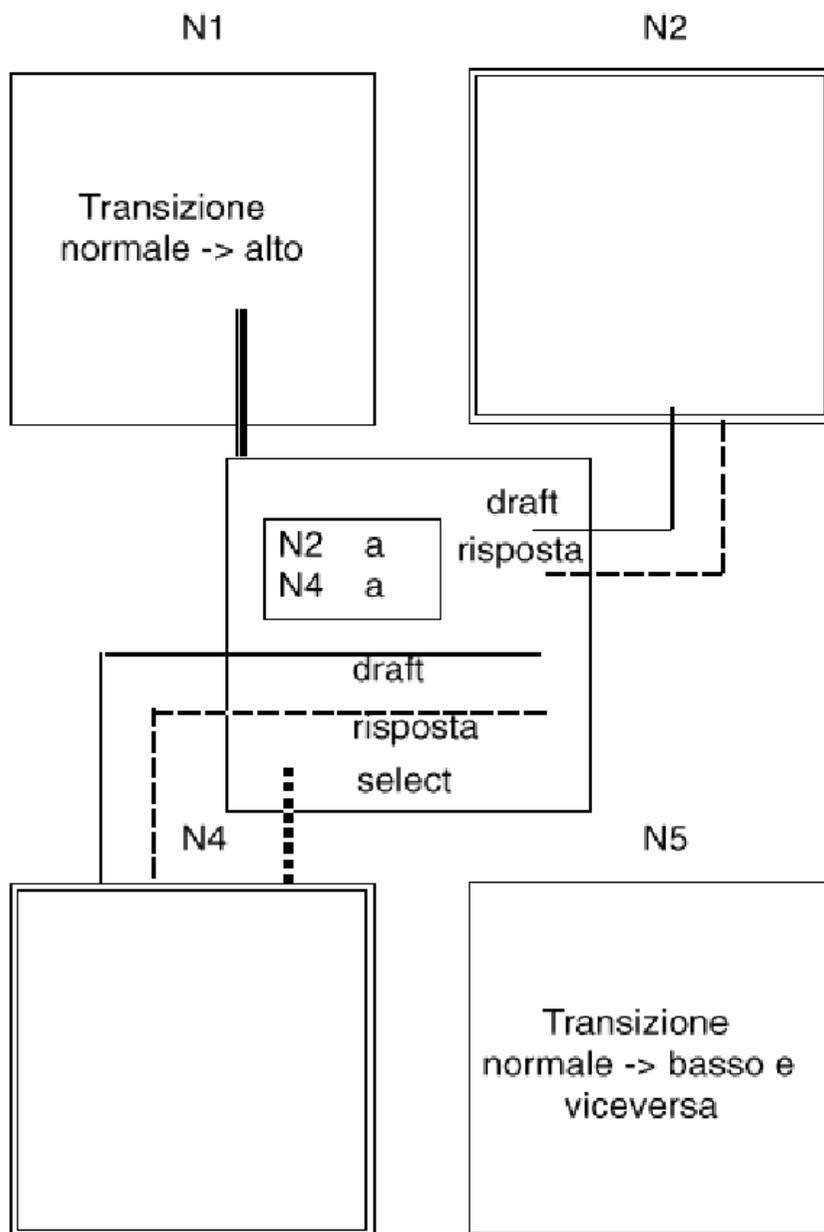
il receiver è a carico **basso** e
stimola i nodi che suppone a carico **alto**

OBIETTIVO:

minimizzare lo scambio di informazioni tra i nodi pur mantenendo una visione locale **consistente** con lo stato corrente

Tabella di carico dei nodi mantenuta in ogni nodo

- 1) ricevente invia la propria disponibilità (**draft**) ai nodi a carico elevato
- 2) questi rispondono al draft, fornendo elementi di valutazione del processo da muovere
- 3) il receiver sceglie da chi ricevere (**select**)
- 4) il sender individuato migra il processo



Obiettivo:

**limitare scambio informazioni
e ottenere decisioni corrette**

Propagazione dei cambiamenti di stato con
broadcast o *piggybacking* (o entrambi)

limitando l'invio dei messaggi di cambiamento di stato
per le transizioni

normale -> basso solo se prima nodo alto **broadcast**

normale -> alto o broadcast o piggyback

Fase di **IDENTIFICAZIONE** della
SOGLIA di stato

Il protocollo iniziato dal receiver consente a questo di
stimare le necessità dei nodi a carico elevato

- I nodi a carico elevato **non** devono eseguire l'algoritmo
- algoritmo fair ==> tutti i receiver rispondono alla stessa sollecitazione (**fairness**)
- soglie garantiscono **stabilità**

MOS

sistema operativo con facility di migrazione
processi **preempted**

- il **carico locale** per ogni processo noto a priori
(approccio a compilazione)
- algoritmo **probabilistico** per limitare lo scambio di informazioni di carico tra nodi
- trasferimento solo dopo una **soglia** di esecuzione locale

STABILITÀ, EFFICIENZA

Charlotte

sistema con processi **preemptable** e **diverse** politiche

- decentralizzate basate sul **bidding** diretto o inverso
- iniziativa del **sender/receiver**

meccanismi di supporto forniti dal kernel

PRIMITIVE di:

- raccolta statistiche locali (intervalli 50-80ms per 100 intervalli)
- **MigrateOut, MigrateIn**
- **CancelMigration**

Il sistema riqualifica automaticamente i link

STABILITÀ, FLESSIBILITÀ, MOLTEPLICITÀ

MOLTEPLICITÀ

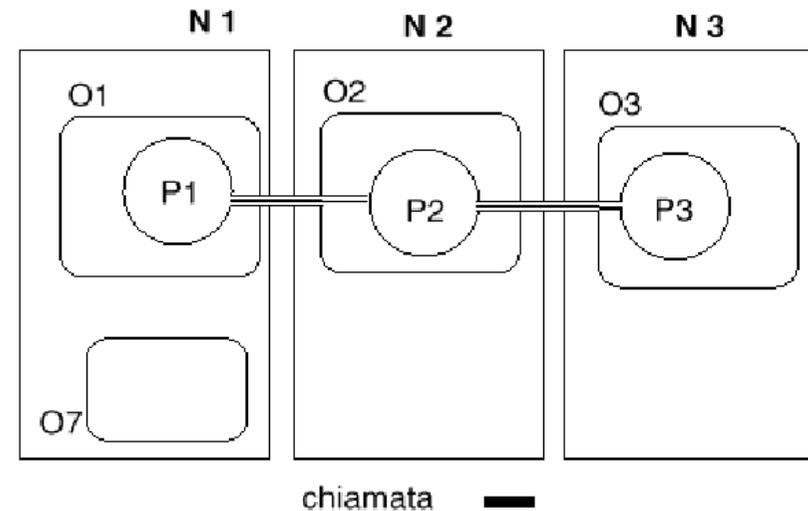
un sistema ad oggetti EMERALD

Oggetti => **ambiente confinato** di esecuzione

Computazione => **richieste sincrone** tra oggetti

Un oggetto può ospitare più di una attività per volta
(**concorrenza inter- oggetto**)

Una richiesta si '**muove**' sugli oggetti richiesti andando a generare processi localmente agli oggetti interessati alle attività



passaggio tra ambienti diversi

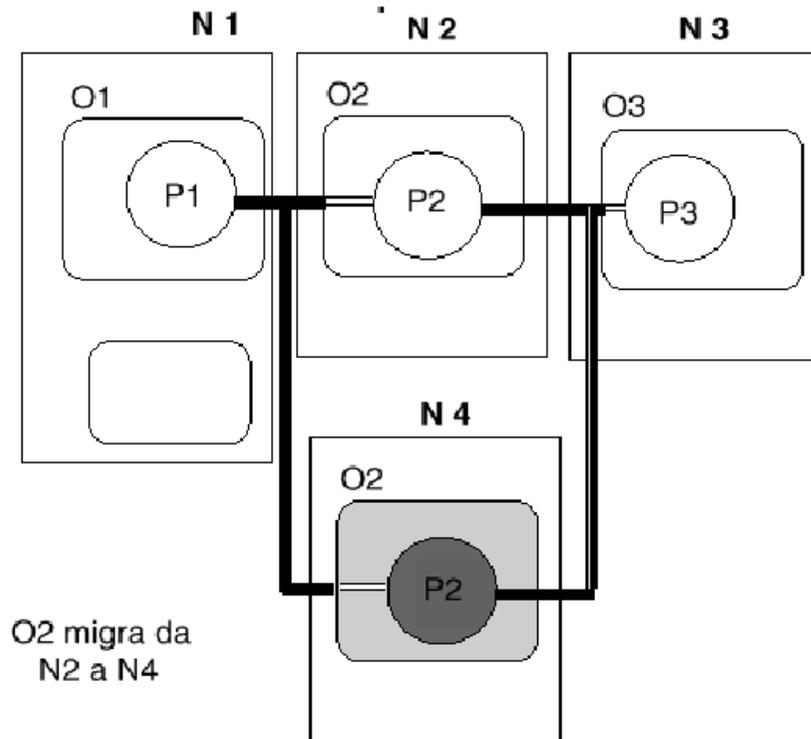
TRASPARENZA e PORTABILITÀ

La MIGRAZIONE avviene per interi oggetti

Migrazione esplicita

Se l'oggetto O2 migra

- **scatena un meccanismo di migrazione**
lo stato si compone di riferimenti ad altri oggetti
- produce **effetti** sulle chiamate **ancora pendenti** e sui **potenziali nuovi invocati**



Migrazione implicita

si ottiene con il passaggio dei parametri

Oggetto. Operazione (parametri)

I parametri di una operazione sono a loro volta **oggetti**

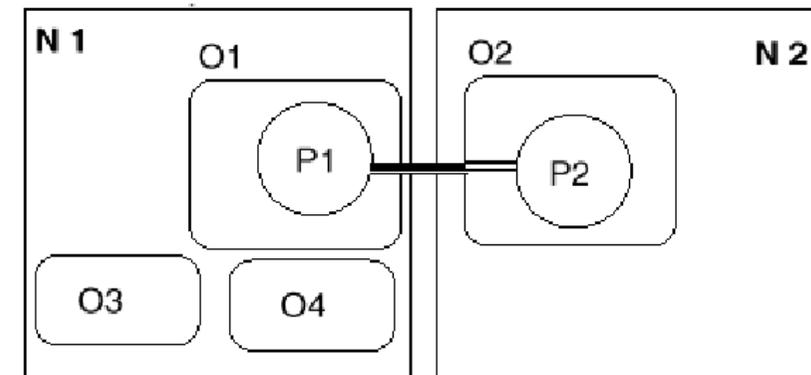
La richiesta di **operazione remota** può

- generare chiamate a sua volta
- causare lo spostamento

temporaneo (call by visit)

definitivo (call by move)

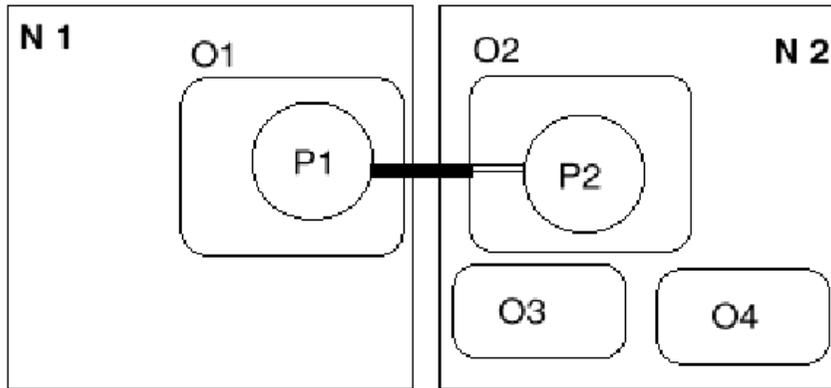
di alcuni degli oggetti passati come parametri



EMERALD lega la **politica di migrazione** alla logica di **strutturazione degli oggetti** specificata dall'utente

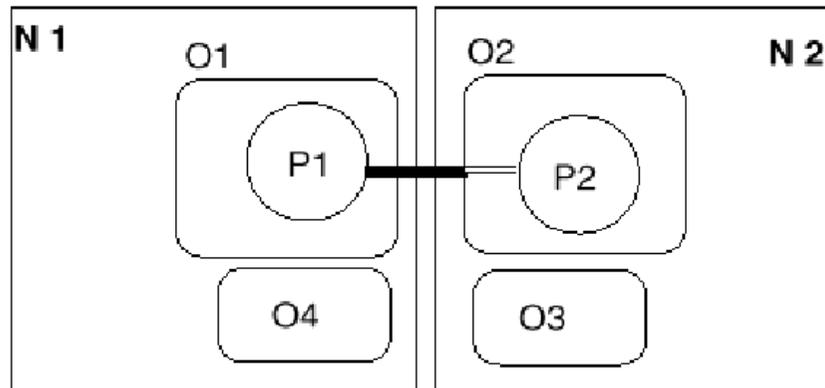
le relazioni ed i riferimenti tra gli oggetti guidano la migrazione e la riallocazione

Durante la esecuzione remota



Migrazione di O3 ed O4

Al termine della esecuzione remota



O3 Call by move

O4 Call by visit

MIGRAZIONE DI AGENTI

nei modelli di **movimento con agenti**

definiamo **attività in cui il movimento** è un requisito
l'obiettivo non è legato al bilanciamento del carico o a
considerazioni di uso di risorse

derivato da specifiche precise di applicazione

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

criteri

- gli agenti devono muoversi anche ritornando su *nodi già visitati* (per riportare informazioni trovate)
- non ci sono *vincoli di costo* nella decisione di migrazione, fatta in ogni caso
- necessità di realizzare *meccanismi efficienti*

Esempi di applicazioni

- **network management** non cliente servitore (vedi snmp o CMIP OSI): uso di agenti che propagano ai diversi nodi le esigenze e raccolgono informazioni
- **supporto** di ricerche per **informazioni su web**: sistemi di caching di dati, ricerca più intelligente di informazioni, gestori per verificare esistenza di siti prima che un utente vi acceda (commercio elettronico)
- sistemi di lavoro **cooperativo** (Computer Supported Cooperative Work): uso di agenti per il coordinamento necessario per ottenere viste ed eventi comuni