Università degli Studi di Roma “Tor Vergata”
Facoltà di Ingegneria

Processi nei Sistemi Distribuiti

Corso di Sistemi Distribuiti

Valeria Cardellini
Anno accademico 2008/09

Processi nei SD

- Un sistema software distribuito è composto da un insieme di processi in esecuzione su più nodi del sistema
- Un algoritmo distribuito può essere definito come un insieme $\{P_1, P_2, \ldots, P_n\}$ dove P_i è un processo
- I processi possono comunicare, sincronizzarsi e cooperare con le stesse modalità sia se sono in esecuzione su nodi remoti, sia sullo stesso nodo

Processori, processi e thread

- Costruire *processori virtuali* a livello software, al di sopra dei processori fisici
- Processore: fornisce un insieme di istruzioni con la capacità di eseguire automaticamente una serie di istruzioni
- Thread: un processore software minimale nel cui contesto possono essere eseguite una serie di istruzioni
  - Il salvataggio del contesto del thread implica l'interruzione dell'esecuzione corrente ed il salvataggio di tutti i dati necessari a continuare l'esecuzione successivamente
- Processo: un processore software nel cui contesto possono essere eseguiti uno o più thread
  - Eseguire un thread significa eseguire una serie di istruzioni nel contesto di quel thread

Cambio di contesto

- Contesto del processore
  - Insieme minimale di valori memorizzati nei registri del processore, usati per l'esecuzione di una serie di istruzioni (ad es. registri interni, stack pointer, program counter, …)
- Contesto del thread
  - Insieme minimale di valori memorizzati nei registri e in memoria, usati per l'esecuzione di una serie di istruzioni (ad es. contesto del processore, stato)
- Contesto del processo
  - Insieme minimale di valori memorizzati nei registri ed in memoria, usati per l'esecuzione di un thread (ad es. contesto del thread, registro MMU)
- I thread condividono lo spazio di indirizzamento
  - Il cambio di contesto può essere fatto in modo completamente indipendente dal SO
- Il cambio di contesto dei processi è generalmente più costoso, in quanto coinvolge il SO

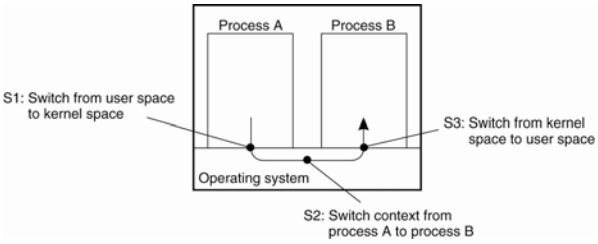
Processi e thread

- I processi offrono trasparenza alla concorrenza, ma ad costo relativamente alto in termini di prestazioni
- I thread offrono concorrenza con un minor grado di trasparenza
  - Applicazioni con prestazioni migliori (in molti casi) ma con codifica e debug più difficile
  - Con i thread in generale ottimizzazione nell'esecuzione dei programmi, tramite un miglioramento delle fasi di allocazione ed esecuzione (condivisione dello stato e dello spazio degli indirizzi)

Vantaggi del multithreading

- Non occorre bloccare l'esecuzione dell'intero processo ad ogni chiamata di sistema bloccante
- Possibilità di sfruttare il parallelismo disponibile nei multiprocessori
- Comunicazione più economica rispetto a IPC

Cambi di contesto legati all'uso di un meccanismo di IPC



- Più adatti per applicazioni di grandi dimensioni
- Modalità alternative per fornire il supporto di thread
  - Implementazione a livello utente
  - Implementazione a livello kernel
  - LWP (lightweight process)
  - Attivazioni dello scheduler

Implementazione di thread

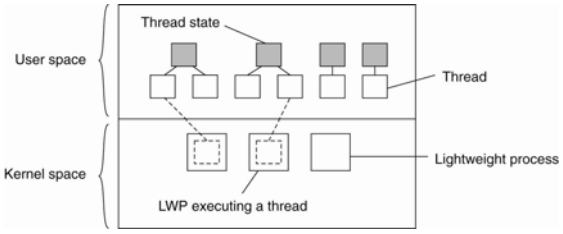
- A livello dello spazio utente (anche schema N:1)
  - La libreria viene eseguita completamente in modalità utente
  - Tutte le operazioni sui thread sono completamente gestite all'interno di un singolo processo
    - Non è richiesto l'intervento del SO per creare, terminare, schedulare per l'esecuzione, sospendere o risvegliare un thread
    - L'implementazione può essere molto efficiente (tempo di cambio di contesto molto basso)
  - **Tutti** i servizi forniti dal kernel sono per conto del processo in cui il thread risiede
    - Se il kernel decide di bloccare un thread, viene bloccato l'intero processo a cui appartiene il thread
  - *In pratica*: scegliamo di usare i thread quando ci sono molti eventi esterni (I/O) da gestire, ma i thread si bloccano su una base per-evento: se il kernel non distingue i thread, come può supportare la segnalazione di eventi?
  - Ulteriore svantaggio: mancanza di parallelizzazione su multiprocessore

Implementazione di thread (2)

- A livello del kernel (anche schema 1:1)
  - Il kernel contiene l'implementazione del pacchetto di thread
  - Tutte le operazioni sui thread sono chiamate di sistema
  - Le operazioni che bloccano un thread non sono più un problema: il kernel schedula un altro thread disponibile all'interno dello stesso processo
  - La gestione di eventi esterni è semplice: il kernel (che cattura tutti gli eventi) schedula il thread associato con l'evento
  - Il problema principale è la perdita di efficienza dovuta al fatto che ogni operazione del thread richiede una trap al kernel
- Soluzione possibile: cercare di combinare i concetti di thread a livello utente ed a livello kernel (anche schema N:M, N >= M)

Implementazione di thread (3)

- Processi leggeri (o thread Solaris)
  - Idea di base: introdurre un approccio a due fasi → **processi leggeri** (LWP) che possono eseguire thread a livello utente



  - Quando un thread di livello utente fa una chiamata di sistema, il LWP che sta eseguendo il thread si blocca (il thread è legato all'LWP)
  - Il kernel può schedulare un altro LWP che ha un thread eseguibile
  - Quando un thread chiama un'operazione di livello utente bloccante, si esegue un cambio di contesto ad un thread eseguibile nello stesso LWP
  - Quando non ci sono thread da schedulare, un LWP può rimanere idle o essere rimosso dal kernel

Thread e SD: lato client

- Client multithread: l'obiettivo principale è nascondere la latenza delle comunicazioni distribuite
- Esempio: un browser è generalmente multithread
  - Ogni risorsa inclusa in una pagina è gestita da un thread, che si occupa di una singola richiesta e risposta HTTP
- Esempio: multiple chiamate richiesta/risposta ad altre macchine (RPC)
  - Il client effettua molteplici chiamate contemporaneamente, ciascuna gestita da un diverso thread
  - Attende finché non ottiene tutti i risultati
  - Speedup lineare se le chiamate sono a server diversi

Thread e SD: lato server

- Server multithread: gli obiettivi principali sono maggiore efficienza prestazionale e maggiore modularità architetturale (specializzazione, semplicità)
- Prestazioni
  - Per gestire una richiesta in ingresso è molto più economico avviare un thread piuttosto che un processo
  - Un server single-thread impedisce la scalabilità del server verso un sistema con architettura multiprocessore
  - Come nei client multithread: nascondere la latenza di rete reagendo alla richiesta successiva mentre si risponde alla precedente
- Struttura migliore:
  - Molti tipi di server hanno una richiesta di I/O elevata: l'uso di chiamate di sistema bloccanti (ben note) semplifica la struttura complessiva
  - I programmi multithread tendono ad essere di dimensioni minori e più semplici da capire per il controllo di flusso semplificato

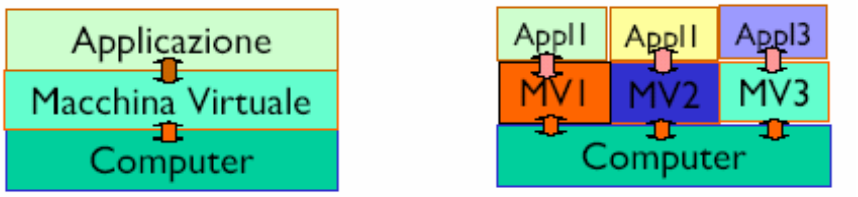
Virtualizzazione

- Virtualizzazione: un livello alto di astrazione che nasconde i dettagli dell'implementazione sottostante
- Virtualizzazione: astrazione di risorse computazionali
  - Si presenta all'utilizzatore una visione diversa da quella reale
  - Virtualizzazione della piattaforma
  - Virtualizzazione delle risorse di sistema

- Le tecnologie di virtualizzazione comprendono una varietà di meccanismi e tecniche usate per risolvere problemi di:
  - Sicurezza, prestazioni ed affidabilità
  - Come? Disaccoppiando l'architettura ed il comportamento delle risorse hardware e software percepiti dall'utente dalla loro realizzazione fisica

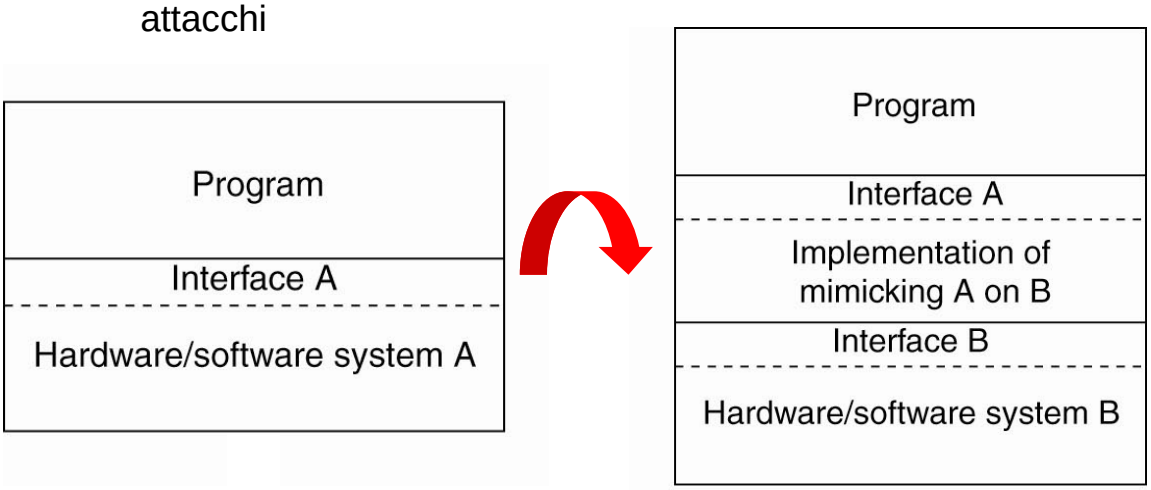
Macchina virtuale

- Il concetto di macchina virtuale (VM, Virtual Machine) è un'idea "vecchia", essendo stato definito negli anni '60 in un contesto centralizzato
  - Ideato per consentire al software esistente di essere eseguito su mainframe molto costosi
  - Ad es. il mainframe IBM System/360-67
- Una VM permette di rappresentare le risorse hardware diversamente dai loro limiti fisici
- Una singola macchina fisica può essere rappresentata e usata come differenti ambienti di elaborazione



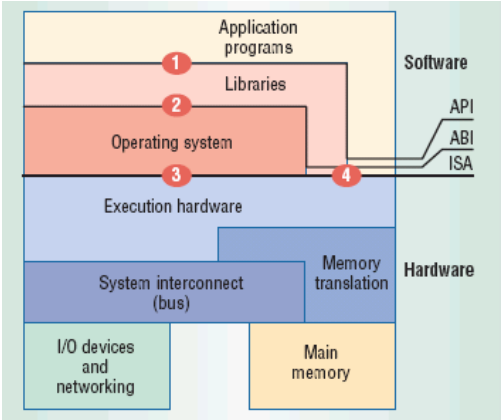
Virtualizzazione (2)

- La virtualizzazione è ritornata in auge alla fine degli '90 e sta assumendo maggiore importanza
  - L'hardware cambia più velocemente del software (middleware e applicazioni)
  - Facilita la portabilità e la migrazione di codice
  - Isolamento di componenti malfunzionanti o soggetti ad attacchi



Architetture delle macchine virtuali

- La virtualizzazione può aver luogo a livelli diversi
- Dipende fortemente dalle interfacce offerte dai vari componenti del sistema
  - Interfaccia tra hardware e software (istruzioni macchina invocabili da ogni programma o user ISA) [interfaccia 4]
  - Interfaccia tra hardware e software (istruzioni macchina invocabili solo da programmi privilegiati o system ISA) [interfaccia 3]
  - Chiamate di sistema [interfaccia 2]
    - ABI (Application Binary Interface): interfaccia 2 + interfaccia 4
  - Chiamate di libreria (API) [interfaccia 1]
- Obiettivo della virtualizzazione:
  - Imitare il comportamento di queste interfacce

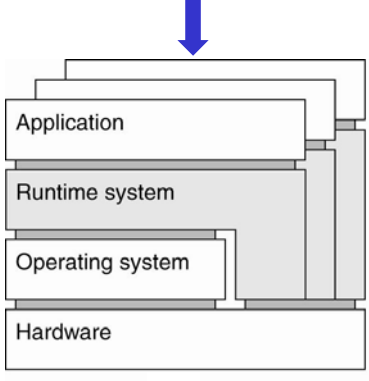


Riferimento: J.E. Smith, R. Nair, "The architecture of virtual machines", *IEEE Computer*, May 2005.

Tipi di virtualizzazione

- Due tipi di virtualizzazione:
  - **Macchina virtuale di processo**
  - **Monitor della macchina virtuale** (VMM, Virtual Machine Monitor) o anche hypervisor
- Macchina virtuale di processo
  - Il programma è compilato in un codice intermedio (portabile), che viene successivamente eseguito nel sistema runtime
  - Virtualizzazione solo per un singolo processo
    - La VM di processo è una piattaforma virtuale che esegue un singolo processo
    - Fornisce un ambiente ABI o API virtuale per le applicazioni utente
  - Esempi: Java VM

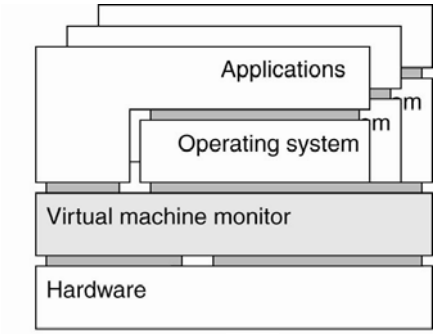
Istanze multiple di combinazioni <applicazione, sistema runtime>



Monitor della macchina virtuale

- Uno strato software separato che scherma completamente l'hardware sottostante ed imita l'insieme di istruzioni
- Sul VMM possono essere eseguiti indipendentemente e simultaneamente sistemi operativi diversi
- Esempi: VMware, Microsoft Virtual PC, VirtualBox, Xen

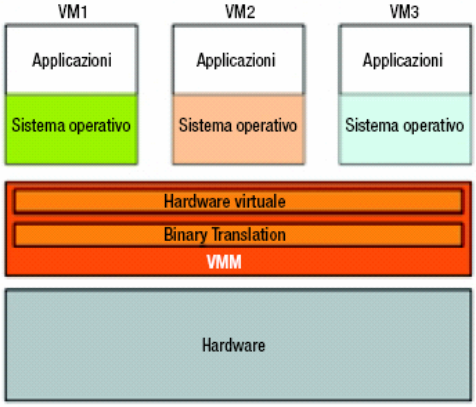
Istanze multiple di combinazioni <applicazioni, sistema operativo>



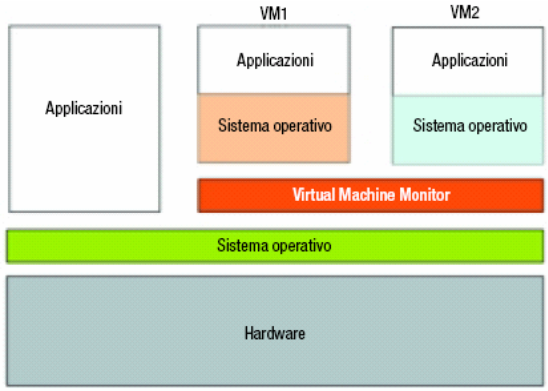
Riferimento: M. Boari, M. Balboni, "Tecniche di virtualizzazione: teoria e pratica", *Mondo Digitale*, Marzo 2007.

Monitor della macchina virtuale (2)

- In quale livello dell'architettura di sistema collocare il VMM?
  - Può essere implementato direttamente sull'hardware (VM classica o di sistema) oppure su un sistema operativo esistente (VM ospitata)
    - Esempi di VM di sistema: VMware ESX, XEN
    - Esempi di VM di ospitata: VMware Server, Virtual PC



VM di sistema



VM ospitata

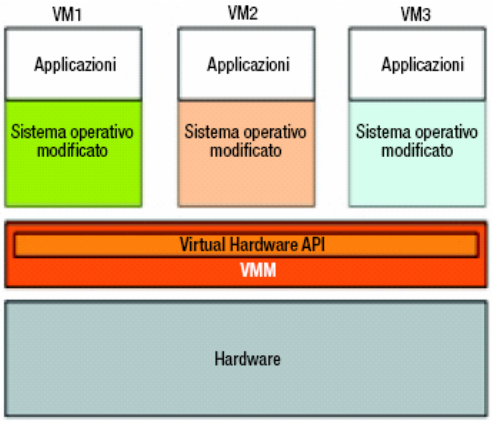
Monitor della macchina virtuale (3)

- Quale modalità di dialogo per l'accesso alle risorse fisiche tra la macchina virtuale ed il VMM?
  - Virtualizzazione completa
  - Paravirtualizzazione
- Virtualizzazione completa
  - Il VMM espone ad ogni macchina virtuale un'interfaccia hardware simulata *funzionalmente identica* a quella della sottostante macchina fisica
- Paravirtualizzazione
  - Il VMM espone ad ogni macchina virtuale un'interfaccia hardware simulata *funzionalmente simile* (ma non identica) a quella della sottostante macchina fisica
  - Simile all'approccio classico per VMM
  - Non viene emulato l'hardware, ma viene creato uno strato minimale di software per assicurare la gestione delle singole istanze di macchine virtuali e il loro isolamento

Paravirtualizzazione

- Xen è l'esempio più noto di paravirtualizzazione

  http://www.cl.cam.ac.uk/research/srg/netos/xen/
  - Il VMM offre al SO guest un'interfaccia virtuale (hypercall API) alla quale il SO guest deve riferirsi per aver accesso alle risorse
  - Occorre rendere compatibile con Xen il kernel ed i driver del SO ospite; le applicazioni rimangono invece invariate
    - Soluzione preclusa a molti sistemi operativi commerciali, a meno di non avere un processore che supporti la virtualizzazione nativa (Intel VT e AMD-V)
  - Overhead molto basso: in grado di fornire prestazioni molto simili a quelle dell'esecuzione non virtualizzata



PlanetLab

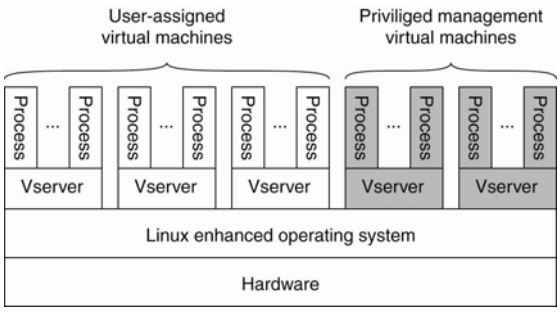
- PlanetLab: un sistema distribuito collaborativo caratterizzato da virtualizzazione distribuita

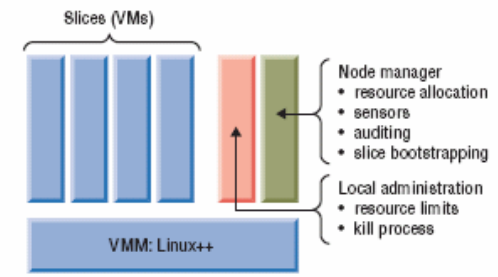
  http://www.planet-lab.org/

  – Ampio insieme di macchine sparse su Internet
    - Quasi 1000 nodi in circa 470 siti
  – Usato come testbed per sperimentare sistemi ed applicazioni distribuite su scala planetaria in un ambiente reale
  – Alcuni esempi di sistemi ed applicazioni testati su PlanetLab
    - Reti overlay
    - Misure di rete (Scriptroute, I3, …)
    - Multicast di livello applicativo (Scribe, …)
    - Distributed Hash Table (Chord, Tapestry, Pastry, Bamboo, …)
    - Content Distributed Network (CoDeeN, …)
    - Virtualizzazione ed isolamento (Denali, …)
    - Processamento distribuito di query (PIER, IrisLog, Sophia, ..)
    - Allocazione di risorse

Nodi in PlanetLab

- Organizzazione di base di un nodo PlanetLab
  – Ogni nodo ospita una o più macchine virtuali
  – VMM, parte di un SO Linux esteso
    - Virtualizzazione a livello di chiamate di sistema
  – Vserver: ambiente separato in cui può essere eseguito un gruppo di processi
    - Completa indipendenza, concorrenza ed isolamento tra processi in vserver diversi





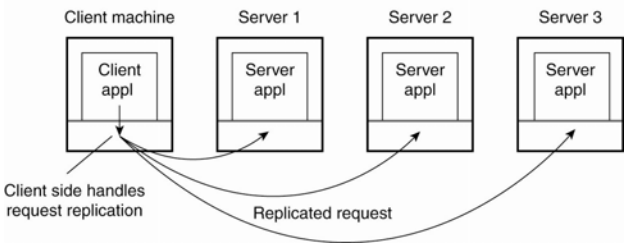
Riferimento: L. Peterson, T. Roscoe, “The design principles of PlanetLab”, *Operating Systems Review, 40(1):11-16, Jan. 2006*.

Virtualizzazione distribuita in PlanetLab

- Un'applicazione in PlanetLab viene eseguita in una slice della piattaforma
  - Un insieme di vserver (in esecuzione su nodi diversi), su ciascuno dei quali l'applicazione riceve una frazione delle risorse del nodo sotto forma di VM
  - Assimilabile ad un cluster di server virtuali
  - Virtualizzazione distribuita: insieme distribuito di VM che sono trattate dal sistema come un'entità singola
  - Programmi appartenenti a slice diverse, ma in esecuzione sullo stesso nodo, non interferiscono gli uni con gli altri
- Molteplici esperimenti possono essere eseguiti simultaneamente su PlanetLab

Problematiche lato client

- Il client comprende componenti per ottenere la trasparenza della distribuzione
  - Trasparenza di accesso: generalmente gestita attraverso la generazione di un client stub
  - Trasparenza all'ubicazione, alla migrazione e al riposizionamento: generalmente il lato client tiene traccia della locazione effettiva
  - Trasparenza alla replica: molteplici invocazioni gestite dal client stub



  - Trasparenza ai guasti: spesso gestita solo lato client (cercando di mascherare malfunzionamenti del server o della comunicazione)

Problematiche lato server

- Già analizzate diverse questioni inerenti la progettazione di un server, quali
  - Server iterativi e concorrenti
  - Distribuzione orizzontale e verticale
- Come può il client conoscere la porta su cui contattare il server?
  - Porta preassegnata e nota
  - Porta assegnata dinamicamente (interazione con daemon)
    - Daemon in ascolto su porta preassegnata delle richieste per alcuni servizi
    - Richieste per lo stesso servizio girate su porta assegnata dinamicamente di cui viene informato il server corrispondente
  - Attivazione dinamica di server (interazione con superserver)
    - Per richieste sporadiche non conviene mantenere server attivi
    - Il superserver ascolta le porte corrispondenti e per ogni richiesta in arrivo risveglia (o crea dinamicamente) il server corrispondente
    - Ad es. inetd di UNIX e GNU/Linux

Problematiche lato server (2)

- Interrompibilità del server: è possibile interrompere un server una volta che ha accettato una richiesta di servizio?
- Soluzione 1: interruzione della connessione
  - Comporta l'interruzione del servizio
- Soluzione 2: dati "out-of-band"
  - Usare una porta di controllo separata
    - Il server ha un thread (o processo) in attesa di messaggi urgenti
    - Quando arriva un messaggio urgente, la richiesta di servizio associata viene sospesa
  - Oppure usare i meccanismi di comunicazione out-of-band forniti dal livello di trasporto
    - Ad es. TCP permette di trasmettere dati urgenti sulla stessa connessione

Problematiche lato server (3)

- Server con o senza stato
- Server senza stato (stateless)
  - Non mantiene informazioni accurate sullo stato del client e non deve informarlo di eventuali cambi di stato lato server
    - Ad es. non invalida la cache di un client
    - Ad es. non registra che un file è stato aperto
  - Conseguenze:
    - Client e server sono completamente indipendenti
    - Ridotte le inconsistenze di stato dovuto a crash del client o del server
    - Possibile perdita di prestazioni, ad es. il server non può anticipare il comportamento del client
- Server con stato (stateful)
  - Mantiene informazioni persistenti sullo stato del client
    - Ad es. registra che un file è stato aperto, in modo da poterne fare il prefetching
    - Ad es. conosce il contenuto della cache di un client e consente ad un client di mantenere una copia locale di dati condivisi

Migrazione del codice

- Nei SD la comunicazione può non essere limitata al passaggio dei dati, ma riguardare anche il passaggio di programmi, anche durante la loro esecuzione

- Motivazione della migrazione
  - Bilanciare o condividere il carico di lavoro
  - Risparmiare risorse di rete e ridurre il tempo di risposata processando i dati vicino a dove risiedono
  - Sfruttare il parallelismo ma senza le difficoltà della programmazione parallela
  - Configurare dinamicamente il SD

Modelli per la migrazione del codice

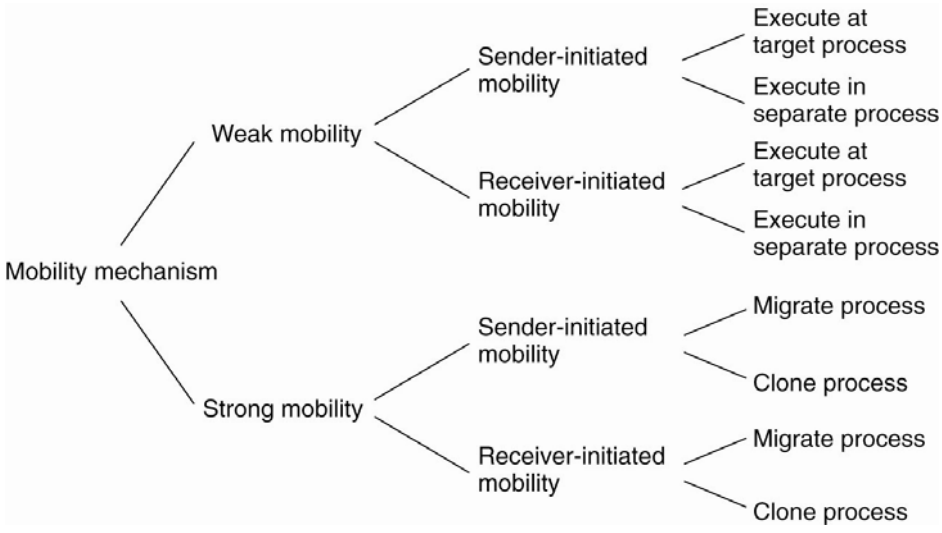
- Processo composto da tre segmenti
  - Segmento del codice
    - Le istruzioni del programma in esecuzione
  - Segmento delle risorse
    - I riferimenti alle risorse esterne di cui il processo ha bisogno
  - Segmento dell'esecuzione
    - Lo stato del processo (stack, PC, dati privati)

- Alcune alternative per la migrazione
  - Mobilità leggera o mobilità forte
    - Leggera: trasferito solo il segmento del codice
    - Forte: trasferito anche il segmento dell'esecuzione
  - Migrazione iniziata dal mittente o avviata dal destinatario
  - Nuovo processo per eseguire il codice migrato o clonazione

Approfondimento nel corso di Sistemi Informatici Mobili
  - Articolo di Fuggetta et al., 1998

Modelli per la migrazione del codice (2)

- Alternative per la migrazione del codice



Migrazione e risorse locali

- Collegamento processo/risorsa
  - Collegamento per identificatore
    - Ad es. il processo è collegato ad un socket
  - Collegamento per valore
    - Occorre solo il valore di una risorsa, ad es. una libreria standard
  - Collegamento per tipo
    - Occorre solo una risorsa di tipo specifico
- Collegamento risorsa/macchina
  - Risorsa unattached
    - Facile da spostare, ad es. file
  - Risorsa fastened
    - Più costosa da spostare, ad. una base di dati
  - Risorsa fissa

Migrazione nei sistemi eterogenei

- In ambienti eterogenei, la macchina target può non essere in gradi di eseguire il codice
- La definizione del contesto di processo, thread e processore è fortemente dipendente dall'hardware, dal SO e dal sistema runtime
- Soluzione migliore: usare una macchina virtuale
  - Macchina virtuale di processo
  - Monitor di macchina virtuale