PROs/CONs

- Virtual Machine:
  +Compatibilità
  +Interoperabilità
  +Portabilità
  +Migrazione
  +Flessibilità
  +Consolidamento dei server
  +Isolamento di componenti guasti o soggetti ad attacco→più sicurezza e affidabilità
  +Isolamento delle prestazioni
  +Bilanciamento del carico
- VMM di tipo 2:
  +Non bisogna modificare il SO guest
  +Sfrutta il SO host per accedere a periferiche ed usare servizi di basso livello (e.g. scheduling risorse)
  -Più superficie di attacco rispetto a tipo 1, espone anche OS dell'host
  -Degrado delle prestazioni rispetto a tipo 1
- Virtualizzazione completa:
  +Non serve modificare SO guest
  +Isolamento completo tra istanze di VM
  -Implementazione VMM più complessa
  -Per implementazione efficiente c'è bisogno della collaborazione del processore
- Paravirtualizzazione:
  +Soluzione relativamente più semplice e pratica
  +Meno overhead della fast binary translation
  +Non ha bisogno di funzionalità estese della CPU come nella virtualizzazione assistita dall'HW
  -Richiede disponibilità codice sorgente OS
  -Costo per mantenere SO paravirtualizzato (non può essere eseguito direttamente su HW)
- Shadow page table:
  - Il sistema operativo guest si aspetta uno spazio di memoria contiguo che inizia dall'indirizzo 0, ma la memoria della macchina sottostante potrebbe non essere contigua: il VMM deve preservare questa illusione.
  - L'implementazione della SPT è complessa.
  - Le SPT richiedono parecchio spazio di memoria e causano un overhead sul tempo di esecuzione poiché, per essere mantenute, richiedono che le operazioni del sistema operativo guest vengano intercettate.
  - SPTs need to be kept consistent with guest PTs
  +Migliorano le performance perché in questo caso l'overhead si ha solo sulla scrittura.
- Xen:
  +Modello di hypervisor leggero (poche righe di codice)
  +Migliorato continuamente
  +Gestione flessibile, si possono configurare parametri per modificare le performance
  +Basso overhead rispetto a bare metal senza virtualizzazione
  +Supporta la migrazione live delle VM
  -Performance nelle operazioni di IO restano un problema
- Resizing dinamico:
  + Meno costoso dello shutdown e riavvio
  -Non è supportato da tutti i prodotti di virtualizzazione né da tutti i SO guest

- Migrazione:
  + Consolidamento ifrastruttura
  +Flessibilità nel failover
  +Bilanciamento del carico
  -Overhead di migrazione
  -Non supportata da tutti i VMM
  -Migrazione in ambito WAN non banale
- Pre-copy:
  -Non va bene per app memory-intensive, tempo di downtime alto
- Post-copy:
  +Riduce downtime e tempo totale di migrazione
  -VM sorgente deve restare attiva
  -Degrado performance legato ai page faults
- Ibrido:
  +La fase di pre-copy riduce il numero di page faults
- Virtualizzazione OS:
  +I containers occupano meno spazio delle VM, quindi su uno stesso host posso averne di più
  +Meno degrado prestazioni
  +Meno tempo per startup e shutdown
  +Footprint più piccola
  +Abilità di condividere pagine di memoria fra più containers
  +Più portabilità ed interoperabilità
  +Meno dipendenza da ambiente di esecuzione
  -Meno isolamento di una VM
  -Meno flessibilità
  -Meno sicurezza
  -Supporto solo per app supportate da OS sottostante
  -Non posso avere OS diversi su containers diversi, solo versioni diverse del SO dell'host
- Unikernel:
  +Leggero
  +Sicuro
  +Veloce (no context switch e avvio veloce)
  -Una singola app alla volta in esecuzione
  -Un singolo linguaggio di programmazione a runtime
  -Sforzo significativo per portare app su unikernel
  -Pochi strumenti di debug
- Volumi Docker:
  + Completamente gestiti da Docker
  + Facile farne backup e migrarli
  +Gestiti con CLI o Docker API
  + Funzionano sia su containers Linux che Windows
  + Condivisibili tra più containers
  + Contenuto può essere cifrato
  + Contenuto può essere prepopolato
  + Meglio rispetto a scrivere dati nel layer scrivibile dei containers

---

Microservizi e serverless computing:

- Microservizi su containers:
  + Scale in/out cambiando il numero dei containers per un'istanza
  + Scale up/down cambiando le risorse per un'istanza
  + Isolamento istanze di microservizi
  + Applicare limiti alle risorse per un microservizio
  + Build e avvio rapidi
  -Necessitano di orchestrazione di containers
- Microservizi:
  +Aumentano agilità del sw
  +Unità indipendenti di sviluppo, deployment, operazioni e versioning
  +Tutte le interazioni tra microservizi avvengono via API, che incapsula dettagli implementativi
  +Sfruttano virtualizzazione basata su containers
  +Maggiore scalabilità e isolamento Guasti
  +Maggior riusabilità
  +Maggiore sicurezza dei dati
  +Sviluppo e consegna più veloci
  +Maggior autonomia
  -Hanno aumentato il traffic di rete
  -Le chiamate tra servizi su rete costano di più in termini di latenza di rete
  -Maggior complessità operazional (e.g. deploy) e nel testing e debugging
- Orchestrazione:
  +I microservizi non devono conoscersi fra loro
  -SPOF
  -Collo di bottiglia
  -Forte accoppiamento
  -Aumenta latenza di rete
- Coreografia:
  +No elemento di centralizzazione
  +Meno accoppiamento, meno complessità operazionale e più flessibilità
  -Più complesso sviluppare servizi di garanzia di delivery
  -Più complesso tracking dei servizi
  -I microservizi devono trovarsi per comunicare
- Circuit breaker:
  +Evita effetto a cascata failures
  -Non dà garanzia di ricevere una risposta
- Database per service:
  +Aiuta a disaccoppiare microservizi
  +Ogni microservizio può usare il tipo di DB migliore per lui
  -Più complessità nell'implementare le transazioni
  -Complessità maggiore nella gestione di molteplici DBs: si può ovviare a questo problema in due modi o non usando un DB per ogni servizio oppure usando tabelle private per servizio, schemi per servizio o DB-server per servizio.
- CQRS:
  -Aumenta complessità
- Log aggregation:
  -Elemento di centralizzazione → SPOF e collo di bottiglia
  -Grande mole di dati da gestire

-La sincronizzazione dei clock dei vari microservizi può essere complessa

- Distributed requests tracing:
  -Aggregare e tracciare dati distribuiti richiede un'architettura complessa a sostegno
- Serverless:
  -Limitazioni relative a supporto linguaggio di programmazione
  -Limiti risorse
  -No standard
  -Cold start
  -Vendor lock-in

Sincronizzazione:

- Algoritmo di Cristian:
  -SPOF
  -Accuratezza buona solo con Tround piccolo
  -Time server hackerato dà valore sbagliato
- ME basato su autorizzazione:
  - Approccio centralizzato:
    +Garantisce ME
    +Fairness
    +Semplice da implementare
    -Coordinatore è SPOF e collo di bottiglia
    -Se processo fallisce mentre è in CS si perde il messaggio di release
  - Lamport distribuito:
    +Garantisce ME
    +Fairness
    +Liveness (no deadlock)
    +No elementi di centralizzazione
    -No garanzia ordering
    -Se un processo fallisce nessun altro può entrare in CS
    -Ogni processo può essere collo di bottiglia
  - Ricart Agrawala:
    +No elementi di centralizzazione
    +Meno messaggi di Lamport
    +Safety, liveness, fairness
    -No garanzia ordering
    -Se un processo fallisce nessun altro può entrare in CS
    -Ogni processo può essere collo di bottiglia
- ME basata su token:
  - Approccio decentralizzato:
    + Se l'anello è unidirezionale, viene garantita anche fairness
    + Rispetto a token centralizzato, la gestione del token è condivisa
    -Consumo di banda di rete per trasmettere il token anche quando nessuno vuole entrare in CS
    -In caso di perdita del token occorre rigenerarlo
    -Guasti temporanei possono portare alla creazione di token multipli
  - Approccio centralizzato:

+Garantisce ME, fairness, liveness e ordering
-SPOF e collo di bottiglia
-Crash coordinatore porta a dover eleggerne un altro

- ME basato su quorum:
  +Safety
  +Più efficiente di Ricart Agrawala su larga scala (3*sqrt(N) messaggi)
  -Non soddisfa liveness
- Algoritmi di elezione:
  Bully: O(N) best case, O(N^2) worst case
  Anello Lynch e Fredrickson: O(2*N) ma messaggi più grandi del bully

---

Consistenza e replicazione:

- Consistenza finale:
  +Semplice e poco costosa
  +Usata nel DNS
  +Letture/scritture veloci
  -No illusione singola copia
  -Possibile inconsistenza per scritture conflittuali →Algoritmo di riconciliazione
- Remote write:
  +Elevata tolleranza ai guasti
  -Write lenta
  -Poca scalabilità
  Aggiornamento bloccante (consistenza linearizzabile):
  +Maggiore tolleranza ai guasti
  -Il client deve aspettare l'update di tutte le repliche
  Aggiornamento non bloccante (consistenza sequenziale):
  +Client deve aspettare scrittura solo su replica primaria
  +Più adatto a scalare, più repliche distribuite geograficamente
  -Meno tolleranza ai guasti
- Local Write:
  +Client attende solo aggiornamento locale
  -Meno tolleranza ai guasti
