Sistemi distribuiti

Sistemi distribuiti

Un **sistema distribuito** è un **insieme di entità separate fisicamente** (su macchine differenti); ognuna di esse ha una certa potenza computazionale (indipendente dalle altre); sono in grado di comunicare e di coordinarsi tra di loro per arrivare ad un obiettivo comune e agli occhi degli utenti si presentano come un unico sistema (es. zoom ha più computer connessi tramite un server che coordina; gmail)

➔ perché sviluppiamo sistemi distribuiti? sviluppare server con processori molto grandi è svantaggioso, conviene sfruttare tanti microprocessori che utilizzano reti sempre più affidabili

I sistemi distribuiti sono usati in tantissimi campi tra cui: finanza e commercio (Amazon), informazioni (Wikipedia e social networks), industrie creative e di intrattenimento (Youtube e online gaming), sanità (app Immuni), apprendimento (Google Classroom), trasporti e logistica (Google Maps e Waze), scienza (The Grid), gestione dell'ambiente (monitorare terremoti, tsunami)

Esempi di sistemi distribuiti:

- Local Area Network and Intranet
- Database Management System
- Automatic Teller Machine Network
- Internet/World-Wide Web
- Pervasive Systems and Ubiquitous Computing
- Service Oriented Architecture
- Virtual networks
- Peer-to-peer (P2P)
- Cloud Computing
- Big Data Computing

esempio di sistema distribuito: Massively Multiplayer Online Games (MMOGs)

→ offrono un'esperienza immersiva dove un gran numero di utenti interagiscono tramite internet in un mondo virtuale comune a tutti i giocatori

→ includono aree di gioco complesse, sistemi per la socializzazione e l'interazione tra utenti e sistemi finanziari interni

→ richiedono risposte veloci, propagazioni degli eventi in real time, visione del mondo intero consistente

Vantaggi e svantaggi

Vantaggi di sistemi di sistemi distribuiti rispetto ai sistemi centralizzati:

1. Economico: un insieme di microprocessori offre una qualità/prezzo migliore rispetto a un singolo mainframe (limiti fisici della scheda madre, costi fili di rame, raffreddamento stanza del server...)
2. Velocità: un sistema distribuito può avere un potere computazionale totale maggiore rispetto a un mainframe
3. alcune applicazioni sono congenitamente distribuite (es. computer delle casse del supermercato)
4. Affidabilità: se un'entità crasha, il sistema per intero può sopravvivere
5. Crescita incrementale: posso aggiungere potenza computazionale aggiungendo moduli
6. La principale sorgente di dati è data dall'esistenza di un gran numero di PC e IoTs e il bisogno per gli utenti di collaborare e condividere informazioni

Vantaggi di sistemi distribuiti rispetto ai PC:

1. Condivisione dati: permette a molti utenti di accedere a un database comune
2. Condivisione delle risorse
3. Comunicazione: migliora la comunicazione umano-umano ed è possibile comunicare anche con i bot
4. Flessibilità: poter spostare il carico di lavoro tra macchine (es. calcoli paralleli)

Svantaggi di sistemi distribuiti:

1. Software: difficoltà di sviluppare software per un sistema distribuito, deve essere sviluppato da più persone
2. Network: possono esserci reti congestionate o perdite di pacchetti
3. Sicurezza: l'accesso ai dati rimane semplice anche per quelli privati

L'obiettivo primario è quello di condividere dati e risorse, ma per questo ci possono essere due problemi:

1. Sincronizzazione
2. Coordinazione → bisogna considerare:
    a. problemi di concorrenza sia temporale che spaziale
    b. non esiste un clock globale
    c. possibili fallimenti
    d. latenze non stimabili

queste limitazioni restringono l'insieme di problemi di coordinazione che si possono risolvere in un'impostazione distribuita

Problemi di design

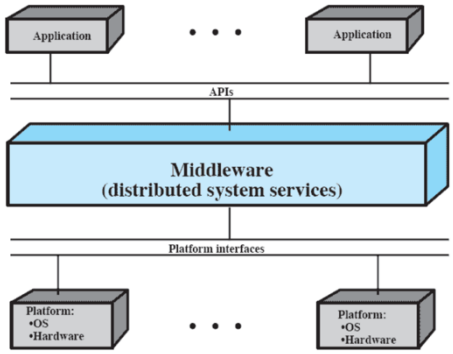
Nel caso di sistemi distribuiti bisogna controllare alcuni problemi di design:

1. Eterogeneità → nelle reti di dispositivi ci sono tantissimi differenti calcolatori con differenti processori e/o sistemi operativi, linguaggi diversi; anche le reti possono essere diverse; risolvo in parte rendendo standard la comunicazione
2. Apertura → necessità che i sistemi siano usabili (bisogna fornire interfacce standard)
3. Sicurezza → garantire confidenzialità (privacy), integrità (protezione contro i cambiamenti non autorizzati) e disponibilità (un dato condiviso tra 2+ entità deve essere disponibile ad ognuna di esse)
4. Scalabilità → i sistemi crescono col tempo/diventano obsoleti; bisogna avere un "business plan" per controllare il costo delle risorse fisiche, perdite di performance, prevenire che le risorse software terminino ed evitare bottlenecks (es. singolo mail server, centralizzare DNS, algoritmi centralizzati)
5. Affidabilità → l'affidabilità su un sistema distribuito deve essere maggiore rispetto a quella di un sistema centralizzato; deve esserci una tolleranza per i fallimenti (oltre che la rilevazione) e una politica di gestione di essi
6. Concorrenza → sia spaziale che temporale
7. Flessibilità → deve essere semplice poter far cambiamenti
8. Performance → ci possono essere delle performance loss dovute a ritardi nella comunicazione (per la parallelizzazione) o a sistemi di tolleranza di fallimenti
    - fine-grain parallelism: tanti piccoli task parallelizzabili; svantaggio → creo una grande trasmissione di dati (latenza di comunicazione)
    - coarse-grain parallelism: programma poco parallelizzato
9. Trasparenza → l'utente deve avere l'impressione di interagire con un'unica interfaccia
    - access transparency: l'utente deve poter accedere ai dati sempre nella stessa maniera
    - location transparency: l'utente non deve sapere dove si trovano fisicamente le risorse
    - concurrency transparency: l'utente non deve accorgersi dei problemi di concorrenza con altri utenti
    - replication transparency: il sistema operativo può fare copie aggiuntive di file e risorse senza che l'utente lo sappia
    - failure transparency: se ci sono dei fallimenti vengono nascosti cosicché l'utente e l'applicazione possano completare i loro task anche se ci sono stati errori

- migration transparency: le risorse possono spostarsi senza che debbano cambiare nome
- performance transparency: se cambia il carico di lavoro, il sistema si deve adattare senza perdere in performance
- scaling transparency: il sistema e le applicazioni si possono ampliare senza cambiamenti nella struttura del sistema o degli algoritmi delle applicazioni
- parallelism transparency: uso automatico del parallelismo senza doverlo programmare esplicitamente

➔ non sempre l'utente vuole una trasparenza completa, dipende dalla situazione

Middleware



Come realizzo un sistema distribuito? Tra hardware e software metto un ulteriore strato: il middleware

Middleware → cuore del sistema distribuito; comunica con le singole entità e deve avere delle interfacce verso tutti i sistemi operativi

I problemi nello sviluppo del middleware sono simili a quelli di un sistema distribuito:

- eterogeneità → sistema operativo, velocità di clock, rappresentazione dei dati, memoria, architettura hardware
- asincronia locale → carico di un nodo, hardware differente, interrupt
- mancanza di informazioni globali → non è possibile sapere in anticipo come è strutturato tutto il sistema (può essere aggiornato/modificato)
- asincronia di rete → la latenza di propagazione non è fissa
- fallimenti nei nodi della rete
- mancanza di un ordine globale di eventi → non si possono programmare gli eventi che accadranno da computer differenti
- consistenza vs disponibilità vs partizione di rete → deve essere presa una decisione su come distribuire le informazioni

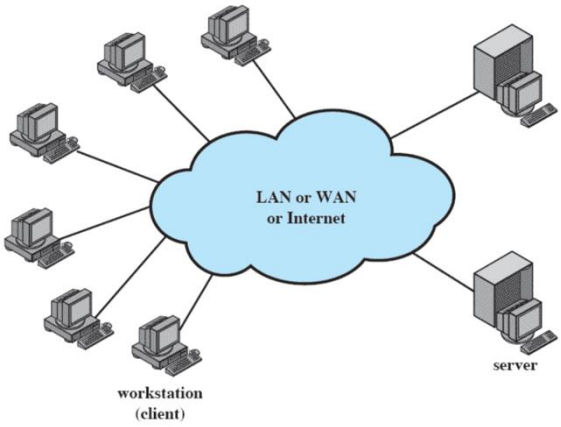
Per avere reali benefici in un sistema distribuito c'è bisogno di un insieme di tool che permettono di accedere in modo uniforme a tutte le risorse del sistema, le quali possono essere localizzate ovunque nel sistema distribuito

Questo insieme di tool permette ai programmatori di fare delle applicazioni che siano uniformi (es. nell'accesso ai dati) → saranno definite al livello di middleware delle interfacce standard da fornire all'applicazione e che permettono di interagire coi

sistemi operativi (N.B. è il middleware che di deve adattare al sistema operativo e non viceversa) e standardizzare le comunicazioni software il più possibile

➔ il middleware fornirà delle API per le applicazioni e le interfacce di piattaforma

Client-server computing



Le macchine client sono generalmente dei pc single-user, mentre i server sono macchine che forniscono uno o più servizi ai client (più performanti)

Solitamente i client sono connessi ai server tramite LAN, WAN o Internet

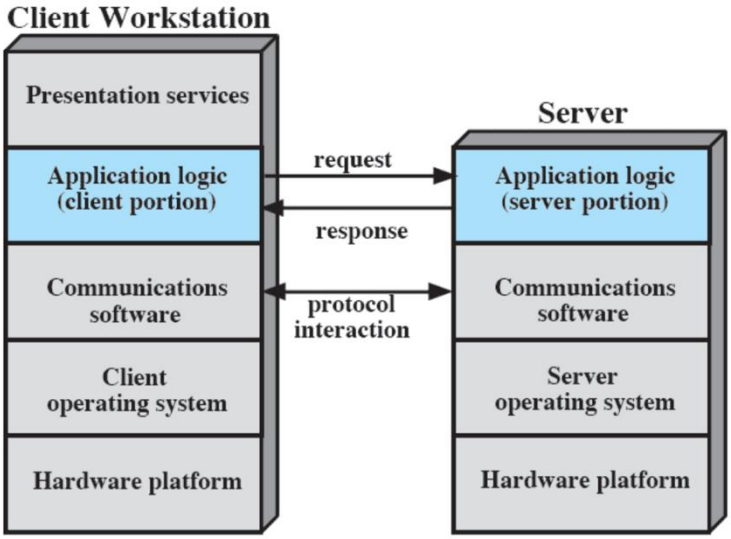
Una configurazione client/server differisce da altri tipi di sistemi distribuiti per:

- alta volontà di fornire sistemi user-friendly → facilito l'utilizzo per l'utente tramite interfacce (anche grafiche) per accedere facilmente ai servizi del server
- bisogno di centralizzare i database
- componente di networking

→ demando al server tutto ciò che è calcolo (accesso ai dati, elaborazione dati)

→ sul client viene spostata la logica e la presentazione dell'applicazione

La caratteristica chiave di un'architettura client server è la possibilità di dividere i task dell'applicazione tra client e server; per operare questa divisione bisogna tener conto delle differenze di sistema operativo e hardware tra client e server, che sono irrilevanti se i due condividono gli stessi protocolli di comunicazione e supportano le stesse applicazioni



Applicazioni distribuite

4 componenti generali:

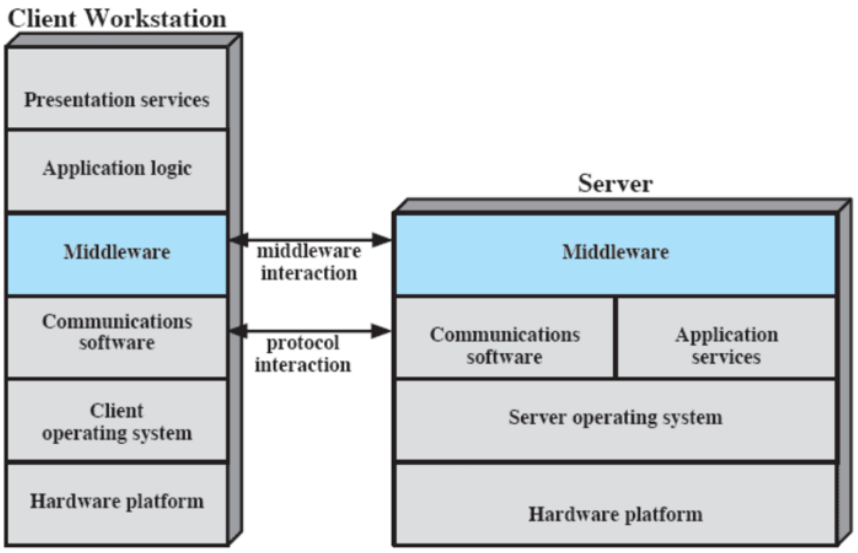
- logica di presentazione = interfaccia utente
- logica dei processi di I/O = validazione dei dati
- logica dei processi del business = verifica delle regole del business e fare calcoli
- logica di immagazzinamento dati = vincoli di primary key, integrità referenziale e recupero dati

Anche se quasi tutte le applicazioni hanno queste quattro componenti generali, non è necessario che facciano parte dello stesso programma, risiedano nello stesso computer, siano scritte nello stesso linguaggio o dallo stesso gruppo di programmatori

Quando si sviluppano componenti delle applicazioni distribuite bisogna farsi delle domande e operare alcune scelte:

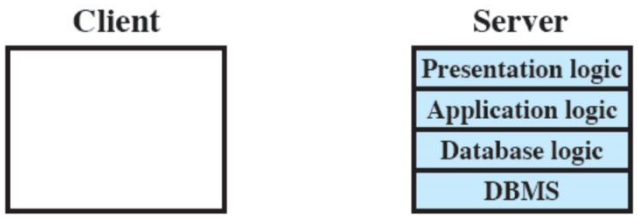
1. in che linguaggio deve essere scritta la componente?
2. dove deve risiedere nell'hardware la componente?
3. quanto spesso verrà cambiata la componente?
4. chi è responsabile del mantenimento del componente?
5. Quanto potrà durare l'applicazione nel mercato?

Il middleware si dovrebbe posizionare tra la parte di comunicazione e la parte di logica a livello del client; a livello di server conviene invece che i servizi dell'applicazione siano a contatto con il sistema operativo del server dato che si accederà spesso ai dati salvati sul server

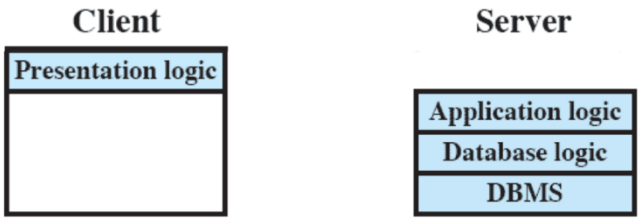


4 classi generali per applicazioni client/server:

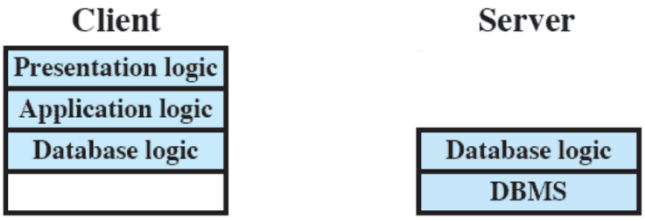
1. Host-based processing → lato client c'è solo l'interfaccia (mostro solo dati)



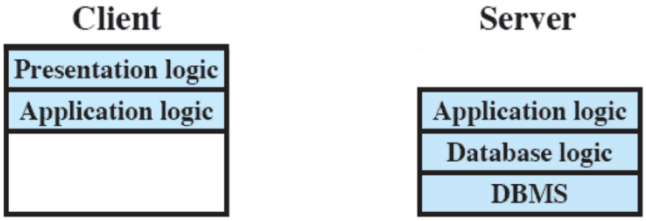
2. Server-based processing → ho solo la presentazione nel client (app tramite browser)



3. Client-based processing → la logica dell'applicazione è tutta lato client (dropbox)



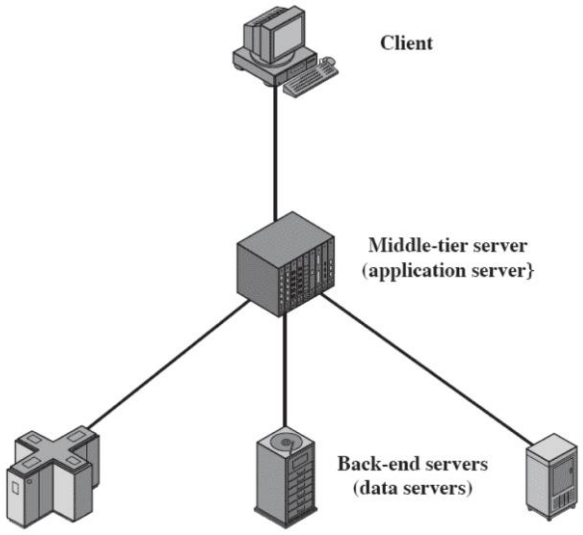
4. Cooperative processing → una parte della logica dell'applicazione è fatta lato client (cloud Google); fat vs. thin client (client leggero ma con meno logica o più pesante ma con più logica?)



Three-tier Client/Server Architecture

Architettura a tre livelli con:

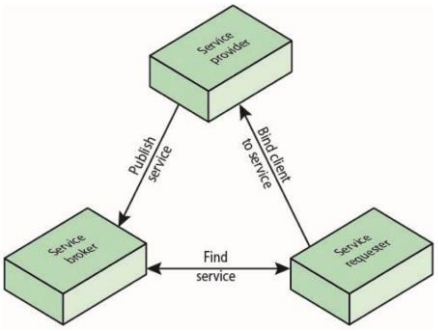
1. client thin
2. server intermedio (middle-tier server) che ha lo scopo di gateway tra i vari servizi, convertitore di protocolli e unificatore di risultati che vengono da più protocolli
3. 1+ server back-end



Service-Oriented Architecture (SOA)

Architettura che lavora in base ai servizi offerti; le funzioni vengono organizzate in modo modulare → ci saranno funzioni in comune tra vari moduli che possono essere riutilizzate senza bisogno di copie; necessita di interfacce standardizzate

→ un service provider (eroga un servizio) manda il proprio servizio a un service broker; chi ha bisogno di quel servizio (service requester) chiederà al service broker dove si trova quel servizio e l'interfaccia in modo da poter utilizzare il servizio e una volta che l'avrà ottenuto potrà fare la richiesta al service provider



→ simile all'architettura three-tier ma il service broker non fa da intermediario come il middle-tier server

Message Passing

Un middleware ha bisogno per funzionare l'utilizzo delle interfacce ossia i messaggi; tutte le comunicazioni tra differenti parti del sistema verranno quindi implementate tramite messaggi

Un processo chiamerà il modulo di message passing che genererà il messaggio e lo invierà al modulo di message passing del processo ricevente

Remote Procedure Call

Un processo fa una chiamata a una funzione che risiede ed è sviluppata, implementata ed eseguita in un altro sistema come se fosse una funzione sul proprio sistema

➔ permette a programmi su differenti macchine di interagire usando le funzioni

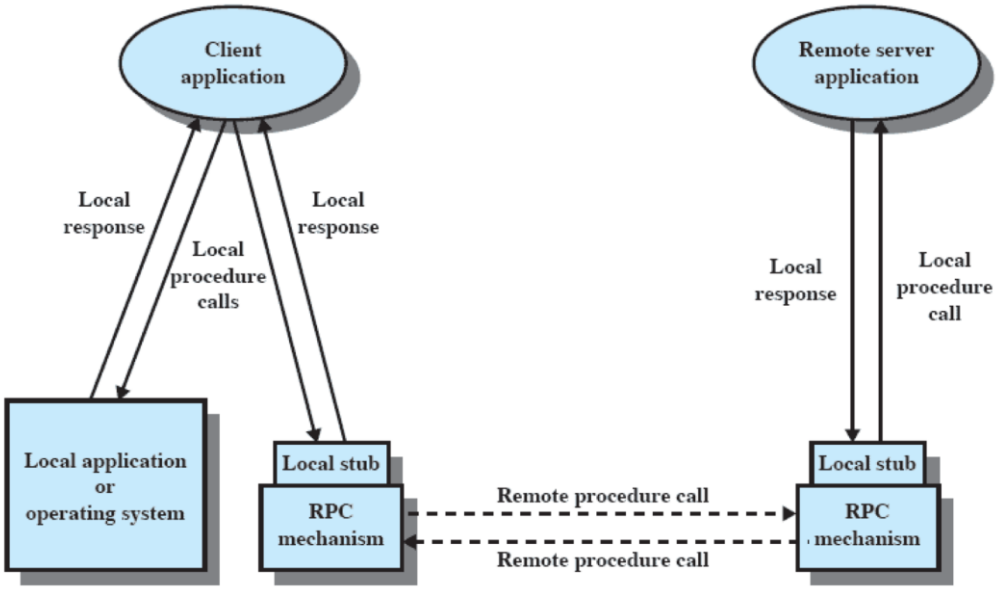
→ sono ormai degli standard; il vantaggio di standardizzarle è che permette a tutti di poterle utilizzare; avendo definito uno standard nel momento in cui viene sviluppata l'applicazione permette facilmente di generare il codice che riguarda la parte di comunicazione

l'applicazione utilizza gli stub → una parte della funzione è implementata lato client (contiene tutta la parte per generare la comunicazione) e l'altra parte è implementata lato server

Dal punto di vista dell'utente le RPCs sono utilizzate come delle API

Quando viene fatta una chiamata a funzione locale, essa viene eseguita localmente tramite il sistema operativo che restituisce un risultato; nel momento in cui viene eseguita una funzione esterna, la funzione viene chiamata come se fosse locale e tramite il meccanismo di RPC viene inviata la richiesta al server che la trasmetterà al modulo che gestisce le chiamate locali; la risposta poi verrà rinviata allo stub del server che comunicherà con lo stub del client per restituire il risultato all'applicazione

- le tempistiche possono essere maggiori rispetto a una chiamata locale dato che introduco un overhead di comunicazione che però è necessario perché il client potrebbe non avere le informazioni e/o la potenza di calcolo per gestire localmente la chiamata



La chiamata a funzione esterna non è esattamente come la chiamata a funzione locale; la differenza sta nel passaggio dei parametri che possono essere passati o per valore o per riferimento

- il passaggio per valore è semplice perché basta copiare il valore e poi trasmetterlo; il valore verrà quindi ricevuto ed elaborato
  N.B. La rappresentazione/formato di un parametro e del messaggio può essere differente se i linguaggi di programmazione tra client e server sono differenti (es. little endian vs. big endian)
- il passaggio per riferimento è più complicato perché:
  - se il riferimento è nella memoria del client, il server non avrà accesso alla memoria del client → bisogna creare una copia della memoria
  - se il riferimento è nella memoria del server, implica che il client deve conoscerne la posizione in memoria (overhead di lavoro)

  questo tipo di passaggio viene utilizzato se devo gestire dati che sono replicati su più sistemi

All'interno di una remote procedure call bisogna considerare anche il **binding** tra client e server; può essere di due tipi:

- persistente → dopo essersi connessi con il server tutte le prossime RPCs avverranno sulla stessa connessione; bisogna poi ricordarsi di chiudere la connessione altrimenti il server dopo un certo tempo di inattività la chiude in automatico
- non persistente → l'architettura chiama la bind con il server, esegue le comunicazioni necessarie e quando avrà ottenuto risposta dalle RPCs la socket viene chiusa; c'è il vantaggio di non tenere connessioni aperte ma ogni volta che il client deve fare una richiesta al server c'è bisogno di riconnettersi

Le RPCs possono essere:

- RPCs sincrone → si comportano come una chiamata a funzione ⇨ finché la funzione non è terminata (return) il chiamante non procede; effettuare una RPC sincrona significa bloccare il chiamante finché non ha ricevuto la risposta
  vantaggio: se per andare avanti nel calcolo il chiamante ha bisogno del risultato si blocca e non prosegue fino a che quel risultato non viene ritornato quindi non verranno fatti dei calcoli sbagliati su una variabile temporanea
  svantaggio: non posso sfruttare il parallelismo ⇨ performance peggiori
- RPCs asincrone → il chiamante non viene bloccato quindi posso fare più RPCs dallo stesso thread
  vantaggio: se ho tanti dati questi vengono parallelizzati senza bloccarsi
  svantaggio: dopo aver parallelizzato devo attendere il risultato ⇨ o busy waiting (while !ricevuto) o elaboro un risultato con un dato che non ho ricevuto (possibile errore); la soluzione è fare una chiamata sincrona dopo 1+ chiamate asincrone → il chiamante è bloccato finché tutte le precedenti chiamate non sono state elaborate

Una remote procedure call può fallire per vari motivi:

- il messaggio che riguarda la chiamata o il messaggio che contiene la risposta è stato perso (es. a causa dei problemi della rete)
- dopo aver fatto la chiamata, il chiamante o il chiamato crasha → la remote procedure call non arriva a termine

- cosa fare? ci sono differenti soluzioni per gestire questi fault
  1. **at least once** → il server deve gestire almeno una RPC; se il chiamante non riceve risposta la invoca nuovamente (ovviamente la chiamata dovrà essere asincrona); il server non memorizza le richieste fatte dal client
     - semplice da implementare → basta mettere un timer dopo aver fatto la richiesta, se non ricevo la risposta invio nuovamente la richiesta finché

non ricevo un ack; se 1+ ack vengono persi, il server potrebbe eseguire la chiamata più volte
    - funziona solo per operazioni idempotenti → stessa richiesta, stesso risultato (es. richiesta dell'id di un utente; controes. richiesta dell'ultimo post di un utente perché ne posso avere più risposte ⇨ quale gestisco?)
2. **at most once** → il server servirà al massimo una richiesta e tutte le successive verranno ignorate ⇨ implica una certa gestione da parte del server (deve tenere traccia delle richieste pervenute; se una richiesta non è soddisfatta per un errore del server, successivamente questa richiesta verrà ignorata)
    - vantaggio: non sovraccarico il server di richieste inutili
    - svantaggio: richiede il rilevamento di pacchetti duplicati
    - funziona anche per funzioni non idempotenti
3. **exactly once** → è il comportamento ideale in quanto emula una chiamata locale ma necessita di un'implementazione molto complessa perché il server deve capire che ha già ricevuto una determinata richiesta e quindi deve solo rinviare i risultati ⇨ viene introdotto un disco dove sono immagazzinati tutti i risultati delle RPC (logging)

Cluster

I cluster sono sistemi distribuiti formati da più computer che lavorano insieme come un'unica risorsa computazionale che crea l'illusione di essere un'unica macchina; sono un'alternativa al symmetric multi-processing (ossia macchine che fanno elaborazione parallela)

I computer che formano un cluster sono chiamati **nodi** e sono formati da scheda madre, processore, dissipatori e memoria RAM; sono quindi macchine ottimizzate per essere in inserite in armadi (rack) ma che possono eseguire programmi indipendentemente dal cluster; hanno solo piccoli monitor di gestione locale o il loro accesso viene fatto da remoto; se necessario vengono messi anche più rack uno accanto all'altro a seconda della potenza necessaria

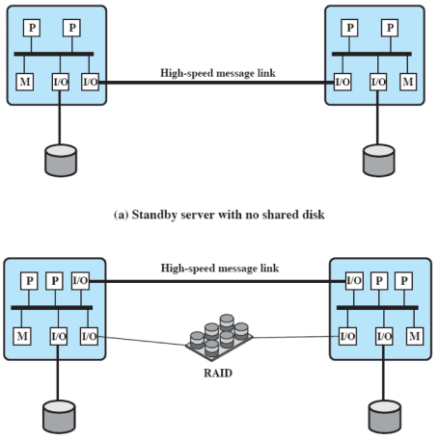
Vantaggi di un cluster:

- assoluta scalabilità → è possibile creare grandi cluster che superano la potenza computazionale di grandi supercomputer
- scalabilità incrementale → è possibile aggiungere nuovi sistemi al cluster in piccoli incrementi
- grande disponibilità → se un nodo ha un fault è possibile spostare su un altro nodo il calcolo; quando si costruisce un cluster si tende a uniformare il sistema operativo, l'architettura etc. quindi la gestione dei fault risulta più semplice

- rapporto performance/prezzo estremamente vantaggioso rispetto al supercomputer in una singola macchina

Nella gestione dei dischi il cluster può essere configurato in due modi:

1. ogni singolo computer ha il suo hard disk e sono tra di loro connessi tutti quanti tramite un canale ad alta velocità
2. utilizzo un piano dell'armadio dedicato ai dischi condivisi (raid) e ognuno dei computer può avere il suo disco locale



La configurazione di un cluster può essere fatta in molti modi differenti:

(dato che il cluster è un server che fornisce servizi posso identificare dei nodi come primari e dei nodi come secondari)

- Passive Standby → i nodi secondari sono una riserva per i nodi primari in caso di fault (se il primario si rompe); è facile da implementare perché ci sarà semplicemente un controller a livello di cluster, ma ha un costo notevole perché i nodi secondari sono inutilizzati ma sempre disponibili per sostituire i primari
- Active Secondary → i nodi secondari sono utilizzati per dei task, ma aumenta la complessità del sistema perché nel momento in cui un nodo primario ha un fault, il secondario potrebbe essere già troppo carico per gestire tutti i servizi quindi o tutto il lavoro del nodo secondario viene spostato in un altro secondario oppure distribuisco il lavoro del nodo primario su più nodi secondari ⇨ potrebbe causare rallentamenti e per operazioni in real time potrebbe non essere accettabile

dal punto di vista dei dischi possiamo avere più soluzioni:

- ogni nodo ha il suo disco; se un dato di un nodo serve anche ad un altro nodo, il dato va copiato; nel momento in cui tutti e due i nodi lavorano sullo stesso dato modificandolo, bisogna anche sincronizzare questi valori → il dato risulta immediatamente disponibile a entrambi i nodi ma ci sarà un overhead della rete e del nodo dovuto al fatto che bisogna trasferire ogni volta il dato modificato sulla rete e riscriverlo sull'altro disco
- in alternativa posso utilizzare dei dischi comuni per tutti i nodi; in questo caso ci sono due differenti soluzioni:
  1 i dischi comuni sono frazionati e ogni nodo ha una parte dedicata → se un dato deve passare da un nodo all'altro, deve essere copiato senza necessità di trasmissione

2. tutta la memoria è condivisa ma deve essere gestita la mutua esclusione distribuita tra i vari nodi

Ci sono due modi per gestire i failure:

1. fallover → se un nodo fallisce, la richiesta viene trasferita su un altro nodo
2. fallback → se un nodo fallisce, viene riavviato e vengono ripristinate tutte le richieste che erano in esecuzione su quel nodo; è più facilmente gestibile rispetto all'altro metodo ma può capitare che la richiesta fallisca per un danno al nodo per cui anche se esso viene riavviato, la richiesta fallirà di nuovo

➔ se ci fosse un errore di programmazione che fa fallire le richieste, qualsiasi gestione sarebbe fallimentare

Nel caso di cluster orientati alla disponibilità, si cerca anche di bilanciare il carico di lavoro → ci sarà un'entità che distribuisce il carico di lavoro tra tutti i nodi in modo da non sovraccaricare un nodo mentre altri sono scarichi; questo viene gestito dal middleware, che deve anche riconoscere quali servizi possono essere svolti da quali nodi

Ci sono 3 modi per ottenere la parallelizzazione nei cluster:

- compilatori parallelizzanti → un compilatore acquisisce il codice e a tempo di compilazione determina quali parti dell'applicazione possono essere eseguite in parallelo; le performance dipendono dalla natura del problema e da quanto bene il compilatore è progettato
- applicazioni parallele → il programmatore progetta un'applicazione che lavori in parallelo; il programmatore ha una grande responsabilità ma è il modo migliore per sfruttare i cluster per alcune applicazioni
- calcolo parametrico → approccio utilizzato se un'applicazione è un algoritmo o un programma che deve essere eseguito tante volte con dati differenti → i dati verranno suddivisi in modo che l'applicazione venga replicata su più nodi e su ogni nodo vengono distribuiti i dati; per rendere questo approccio efficace i lavori devono essere organizzati, eseguiti e gestiti in un certo ordine