DISTRIBUTED SYSTEMS

Insieme di processori connessi in rete.

Vantaggi

- Scalabilità
- Modularità ed eterogeneità
- Condivisione dati e risorse
- Struttura geografica
- Basso costo

Caratteristiche

- Assenza di clock condiviso
- Assenza di memoria condivisa
- Assenza di rilevatore di errori

Teorema CAP

Qualsiasi sistema connesso in rete con dati condivisi deve avere almeno 2 di queste 3 proprietà:

- (C) **Consistency**: tutti i dati nei vari nodi del sistema non presentano incongruenze. Se un dato è stato aggiornato in un nodo, tutti gli altri dovranno essere aggiornati immediatamente.
- (A) **Availability**: a qualunque richiesta deve corrispondere una risposta (costante disponibilità di dati), anche se parte del sistema non è funzionante.
- (P) **Partition tolerance**: il sistema deve continuare a funzionare anche se parte di esso non è funzionante.

Latenza

Ritardo nella comunicazione. Nel teorema CAP viene apparentemente ignorata, ma influenza molto la **partition tolerance**.

Problemi e limiti della computazione distribuita

- Programmazione OOP
- Programmazione procedurale

Modello di computazione distribuita

Consiste in un **sistema asincrono** composto da un insieme di processi, i quali:

- Sono composti a loro volta da stati ed eventi che alterano gli stati stessi.
- Inviano messaggi attraverso canali unidirezionali, con buffer infiniti e privi di errori.
- Ritardi arbitrari (ma finiti)
- Nessuna assunzione sull'ordinamento dei messaggi.

Modelli utilizzati - Interleaving

Si assume ordinamento totale tra gli eventi (tempo fisico).

Modelli utilizzati - Happened before

Si assume ordinamento parziale tra gli eventi (tempo logico).

- **Causalità diretta**: se ($e \lessdot f$) oppure ($e \rightsquigarrow f$), allora $e \rightarrow f$:
  - ($e \lessdot f$) vuol dire che e avviene prima di f, nello stesso processo.
    - **Esempio**:
  - ($e \rightsquigarrow f$) vuol dire che e avviene prima di f, in processi diversi, e comunicano tramite messaggi.
    - **Esempio**: e è l'invio di un messaggio nel processo P1, f è la ricezione del messaggio nel processo P2.
- **Transitività**: se esiste g tale che ($e \rightarrow g$) e ($g \rightarrow f$), allora $e \rightarrow f$.

Modelli utilizzati - Happened before (Meccanismi implementativi)

- **Logical clocks**: una funzione C di un processo Pi assegna un numero $Ci(a)$ ad ogni evento a, secondo la **clock condition**:
  - Se ($a \rightarrow b$) allora ($Ci(a) \rightarrow Cj(b)$), con a evento del processo Pi e b evento del processo Pj. E' inoltre sempre vero che se ($a \rightarrow b$) allora ($Ci(a) < Cj(b)$), quindi se un evento è happened before un altro, allora il valore del clock logico del primo evento è minore del secondo. Ma **non è vero il contrario**
- **Vector clock**: risolvono la limitazione dell'affermazione precedente. Un vettore VC, di dimensione uguale al numero di processi considerati, viene assegnato ad ogni evento.
  - ($a \rightarrow b$) $\iff$ ($VC(a) < VC(b)$)

Modelli utilizzati - Potential causality

L'idea di base è che non tutti gli eventi di un sistema concorrente siano direttamente correlati tra loro. Alcuni di essi possono essere ad esempio causalmente in relazione oppure indipendenti/concorrenti.

- **Concorrenza tra eventi**: ($e \parallel f$) = not ($e \rightarrow f$) and not ($f \rightarrow e$).

Problemi dei sistemi distribuiti - Mutua esclusione

Si intende la mutua esclusione tra processi distruibuiti, ma bisogna comunque rispettare i principi di safety, liveness e fairness.

- **Soluzione con algoritmo centralizzato**: si utilizza un coordinatore, il quale gestisce le richieste secondo la relazione happened before e impiega i vector clock. Il coordinatore ritarda ogni richiesta finchè le precedenti non vengono ricevute.
  - **Elezione del leader/coordinatore (algoritmo di Chang-Roberts)**: viene imposta una topologia logica ad anello. Ogni processo ha un PID univoco. Alla fine il leader sarà quello col PID più alto. L'elezione parte da un processo, il quale (se non ha ancora ricevuto un PID più alto del proprio) invia il proprio PID ad uno dei processi vicini. Se riceve un PID più alto del proprio, lo inoltra. Se riceve il suo stesso PID, allora si dichiara leader, informando tutti gli altri tramite un messaggio.

- **Soluzione con algoritmo decentralizzato (Agrawala)**: per accedere ad una risorsa, un processo invia un messaggio con timestamp a tutti gli altri processi.
    - Se un processo *P1* **non** è interessato a quella risorsa, invia un OK come risposta.
    - Se un processo *P1* è interessato ma ha un timestamp **superiore** a quello ricevuto da *P2*, invia un OK come risposta e si mette in coda per accedere successivamente alla risorsa.
    - Se un processo *P1* è interessato ma ha un timestamp **inferiore** a quello ricevuto da *P2*, allora ha la precedenza su quest'ultimo, il quale viene aggiunto alla coda dei processi in attesa.
        - *P2* accederà alla risorsa solo quando riceverà un OK da tutti gli altri che hanno la precedenza.

Problemi dei sistemi distribuiti - Ordinamento dei messaggi

Essendoci massimo non-determinismo nell'ordine dei messaggi inviati, l'unica soluzione è effettuare un ordinamento causale. I vector clock vengono estesi ad una matrice *m* di interi, la quale permette ad un processo *Pi* di tenere del numero di messaggi inviati da un processo *Pj* ad un altro *Pk* (*m[j,k]*).

- **Algoritmo**
    - Quando *Pi* invia un messaggio a *Pj*, incrementa *m[i,j]* e invia la matrice insieme al messaggio.
    - Un messaggio può essere ricevuto (è **eligible**) da un processo *Pj* quando il numero di messaggi inviati da *Pi* a *Pj* è minore o uguale al valore presente nella matrice di *Pi* (logicamente, i messaggi ricevuti da *Pj* non devono essere più dei messaggi inviati da *Pi* a *Pj*)
    - Se un messaggio non può essere ricevuto, viene aggiunto ad un buffer finchè non diventerà finalmente eligible.
- **Alternativa all'ordinamento causale: Ordinamento totale**: più robusto del precedente. Utilizza i seguenti meccanismi, utili a rendere sincrono un sistema distribuito:
    - **Pulses**: equivalente di un contatore distribuito. Garantisce che un messaggio inviato ad un impulso *i*, verrà ricevuto in quello stesso impulso *i*.
    - **Synchronizers**: meccanismo che indica quando un processo può generare un pulse. Ogni processo invia un solo messaggio per ogni pulse. Se un processo riceve un messaggio dal pulse successivo a quello corrente, attende prima il messaggio sincronizzato (quindi dal pulse corrente).

Problemi dei sistemi distribuiti - Determinazione dello stato globale

Calcolarlo in tempo reale è impossibile. La soluzione migliore è quella di raccogliere e ordinare causalmente gli stati passati, applicando il modello happened before. Lo stato globale risultante ottenuto in questo modo è detto **global snapshot**.

- **Soluzione (Chandy-Lamport)**:
    - Ogni processo ha uno stato e dei canali. A loro volta i canali hanno uno stato (messaggi in transito) e sono unidirezionali, con coda FIFO.
    - Ogni processo è inizialmente di colore **bianco**.
    - Ogni processo salva il proprio **stato locale** e diventa di colore **rosso**.
    - Una volta rosso, ogni processo invia un **messaggio marker** su tutti i canali e registra quelli che riceve.
    - Se un processo **riceve un marker per la prima volta**, salva il proprio stato locale e diventa di colore rosso, per poi continuare eseguire il punto precedente.

    - Se un processo **riceve un marker_su un canale sul quale l'aveva già ricevuto**, smette di registrarli.
    - Il tutto termina quando tutti i processi **hanno ricevuto un marker su tutti i canali**.

Problemi dei sistemi distribuiti - Consenso

Trovare un accordo tra i processi distribuiti riguardo il valore di una proprietà, un'azione da eseguire o altro. Un ulteriore problema può essere la gestione di failure che possono verificarsi durante il calcolo del consenso stesso. Si eseguono infatti più round per eliminare processi che prendono decisioni sbagliate a seguito di failure.

- **Algoritmi per reti asincrone**: inesistenti. Anche solo un fallimento di un processo rende impossibile risolvere il problema del consenso (**FLP result**).
- **Algoritmi per reti sincrone**:
    - **Basic**: ogni processo invia i propri valori a tutti gli altri e riceve a sua volta tutti i valori da loro. Si prende una decisione in base a questi valori, dopo aver ripetuto la procedura più volte.
        - Crash => algoritmo funziona ugualmente.
        - Byzantine problem (processo fornisce arbitrariamente dati errati) => algoritmo non funziona.
    - **Byzantine General Agreement**: risolve il byzantine problem.
        - Vengono eseguiti $f+1$ round (con f numero di processi faulty).
        - Si nomina a rotazione un coordinatore (**king**), che in almeno 1 round non deve essere faulty.
        - Ogni processo scambia i propri valori con tutti gli altri e si effettua una stima.
        - Quando un processo riceve il valore dal king, decide se mantenere il proprio valore o utilizzare quello del king.
        - Se il vettore V (contenente gli N valori di tutti i processi) contiene $N/2 + f$ copie di un valore, viene scelto quest'ultimo, altrimenti viene scelto quello del king.
    - **Paxos**: protocollo utilizzato quando si ha a che fare con una rete di nodi non affidabili.
    - **Raft**: equivalente a Paxos in termini di fault-tolerance e performance, ma più semplice da capire e implementare. Consiste nel distribuire una macchina a stati su un cluster di sistemi computazionali.

MOM (Message Oriented Middlewares)

Consentono lo scambio di messaggi anche tra applicazioni non in esecuzione nello stesso momento, grazie all'utilizzo di code e brokers, i quali permetto la ricezione di tali messaggi in un secondo momento.

Service Oriented Computing

Approccio attualmente più utilizzato per sviluppare sistemi distribuiti.
Servizio: componente software indipendente, accessibile via rete, il quale fornisce una funzionalità.
Possibili implementazioni sono:

- **SOA**
- **REST-full**
- **Microservizi**

SOA (Service Oriented Architectures)

La *business logic* in questo caso è scomposta in parti (servizi) più piccole, autonome, indipendenti e distribuite.

- Possibile utilizzare linguaggi e tecnologie diverse per ogni servizio.
- Ogni servizio è localizzabile tramite un endpoint **(URI = Universal Resource Identifier)**
- Per comunicare tra loro, i servizi devono avere un *contract* (**API**) chiaro e aderire ad uno standard.
- La comunicazione avviene tramite messaggi, scambiabili tramite **richieste HTTP** o tramite **SOAP (Simple Object Access Protocol)**
- Ogni servizio può interagire tramite messaggi con un **Service Consumer** (componente software)
- **Proprietà dei servizi**:
    - Riutilizzabili
    - Condividono contratto formale
    - Progettati per interagire senza necessità di dipendenze tra loro
    - Logica astratta esposta
    - Componibili
    - Autonomi
    - Stateless
    - Comprensibili

SOA - Vantaggi

Questo tipo di architettura evita il problema dell'**integration spaghetti** (problemi di dipendenza causati da integrazione di funzionalità senza un preciso ordine). Inoltre migliorano:

- Adattabilità
- Manutenibilità
- Riusabilità (non nel vero senso del termine, bensì non è necessario scartare tutto per implementare nuove funzionalità)

Si evita anche il problema dell'**object soup**: vengono definiti i confini tra pezzi di logica, i quali rappresentano diverse aree di business.

Web Service

E' un esempio di SOA basato su:

- *XML*
- *SOAP*
- *WSDL*

Garantisce interoperabilità tra middleware eterogenei. Provider e customer utilizzano messaggi per scambiarsi richieste/risposte sotto forma di documenti. Interoperabilità tra messaggi e Remote Procedure Calls.

Web Service - SOAP

SOAP è un linguaggio XML che **definisce formato/struttura di un messaggio**. Un documento SOAP è composto da:

- Header (info generali)
- Body (payload)
- Endpoint mittente
- Endpoint destinatario

Web Service - WSDL

WSDL è un linguaggio XML che definisce sintatticamente l'interfaccia:

- tipologia di porta (specifica operazioni astratte)
- collegamenti tra l'insieme di operazioni con il protocollo di trasporto concreto e i formati di serializzazione (HTTP)

SOA - API

Interfacce che permetto a diverse applicazioni software di comunicare tra loro, facilitando l'accesso/scambio di dati o funzionalità in modo standardizzato. Consentono ad un'applicazione di utilizzare i servizi offerti da un'altra, senza conoscere i dettagli implementativi.

REST

REpresentational State Transfer: tecnologia/architettura platform-indipendent, sviluppata per consentire una comunicazione di componenti non dipendenti tra loro, tramite interfacce e utilizzando protocolli web standard. E' una tecnologia molto semplice e scalabile.

Web Service - RESTful

Si basa sui principi REST per implementare le RPC attraverso il web. Ha un **contratto** contenente:

- **URI** della risorsa (da dove a dove i dati vengono trasferiti)
- **Metodi** (meccanismi di protocollo) per trasferire i dati.
  - HTTP, ad esempio, ha i metodi *GET*, *POST*, *PUT*, *DELETE*...
- **Tipo dei dati** da trasferire (XML, JSON)

Tipologie di servizi

- **Entity services**: stabiliscono confini funzionali associati ad una o più *entità* di business.
- **Task services**: stabiliscono confini funzionali associati ad uno o più *task* di business.
- **Utility services**: raggruppano *competenze* correlate di servizi che potrebbero avere ampi confini funzionali.

Stili architetturali:

- **Stratificato**: ogni livello ha una propria responsabilità e dipende dai livelli sottostanti (esempio a 3 livelli: *Presentation* + *Business logic* + *Persistence*)
- **Esagonale**: la *business logic* è posizionata centralmente. L'applicazione presenta degli adattatori per far comunicare le applicazioni esterne con la *business logic*, ma quest'ultima non dipende dagli adattatori (vale invece il contrario)
- **Monolitico**: l'implementazione è un singolo componente, ma la logica può avere ad esempio un'architettura esagonale. La *business logic* presenta una o più porte inbound o outbound, per permettere l'interazione con l'esterno. Le porte inbound sono di fatto delle *API* esposte.
- **Microservizi**: consiste nello sviluppare una singola applicazione assemblando molteplici componenti (servizi)
    - Ogni componente ha una propria *business logic*.
    - Una modifica nell'implementazione di un componente non impatta sugli altri, anche se in comunicazione.
    - I componenti sono deployabili singolarmente, senza fare il deploy dell'intera applicazione.

API dei servizi

Definiscono operazioni invocabili dagli altri. Le operazioni possono essere:

- *Comandi*: per aggiornare i dati.
- *Query*: per richiedere dati.
- *Eventi*: pubblicati dal servizio quando i dati cambiano.

Tecnologie ICP per le API

Le Inter-Communication Communications possono sfruttare le seguenti tecnologie:

- *REST*: per il trasporto di RPC o con websocket per interazioni event-oriented.
- *MOM* asincrone.
- *gRPC*: framework RPC ad alte prestazioni.
- *GRAPHQL*: linguaggio a query per API.

Interazione tra servizi

Temporalmente parlando:

- **Sincrona**: client aspetta la risposta del servizio, anche bloccandosi.
- **Asincrona**: client non aspetta la risposta del servizio (che potrebbe non esserci, ma in ogni caso non viene inviata immediatamente)

Numericamente parlando:

- **One-to-one**: ogni richiesta di un client viene processata da un solo servizio.
    - *Request/response* (sincrona)
    - *Request/response* (asincrona)
    - *One-way notification*: il client invia una richiesta ad un servizio, ma non è prevista alcuna risposta.

- **One-to-many**: ogni richiesta di un client viene processata da più servizi.
    - *Publish/subscribe*: un client pubblica un messaggio di notifica che verrà accolto dai servizi interessati.
    - *Pubish/async responses*: un client pubblica una richiesta e attende le risposte dei servizi interessati, ma entro un certo lasso di tempo.

Cloud Computing

Attualmente è la principale scelta di design per le applicazioni distribuite su larga scala. Garantisce:

- *Scalabilità*
- *Eterogeneità*
- *Disponibilità*
- *Costi minori* per le aziende.
- *Rischi minori* per le aziende.

Cloud Computing - Modelli

- **SaaS (Software as a service)**
    - Provider mette a disposizione un SW e permette l'accesso tramite una web app.
- **PaaS (Platform as a service)**
    - Provider mette a disposizione una piattaforma per inizializzare, eseguire e gestire app e piattaforme di computazione, senza doversi occupare dell'infrastruttura.
    - La sua evoluzione è **FaaS (Function as a service)**: serverless computing, quindi nessun server o OS da configurare e gestire. Alta scalabilità, disponibilità e fault tolerance. Utile quando bisogna gestire problemi semplici, stateless e prevedibili.
- **IaaS (Infrastructure as a service)**
    - Provider mette a disposizione HW (server, rete, memorie), ma gli utilizzatori devono installare e gestire OS e SW.