

# Scalable and Distributed Systems

## Lezione 1: Un po di definizioni

**Distributed:** Capacità del sistema di distribuire task su più macchine locate in diverse sedi fisiche.

**Distributed System:** Un **sistema distribuito** è un **insieme di macchine autonome** che appare ai suoi utenti come un **unico sistema coerente**. I nodi possono **agire** in modo **indipendente** l'uno dall'altro, ma devono perseguire **obiettivi comuni** coordinandosi scambiandosi **messaggi**.

**Nodi:** Macchine che partecipano al sistema distribuito, ognuna con propri processi e risorse elaborative.

**Scalability:** Capacità del sistema di gestire un carico di lavoro crescente **aumentando le risorse**, in modo che le prestazioni non si degradino.

**Scalabilità verticale:** Aggiunta di potenza (CPU, RAM) a un singolo nodo.

**Scalabilità orizzontale:** Aggiunta di nuovi nodi al sistema.

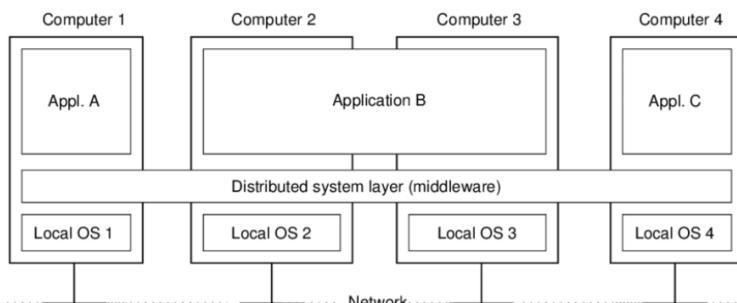
**Fault Tolerance:** Capacità del sistema di continuare a funzionare correttamente nonostante il guasto di uno o più nodi.

**Consistency:** Garanzia che ogni nodo disponga di una vista coerente dei dati.

**Strong Scalability:** Capacità di un **sistema** di ridurre proporzionalmente i **tempi di elaborazione** al crescere delle **risorse** aggiunte, a parità di carico.

**Weak Scalability:** Capacità di un sistema di mantenere **costanti le prestazioni** quando alla crescita del carico di lavoro corrisponde un aumento proporzionale delle risorse, a parità di tempo di elaborazione.

**Middleware:** Un **middleware** è uno strato sw **intermedio** che si colloca **tra le applicazioni e l'infrastruttura di rete/SO**, con lo scopo di **nascondere la complessità della distribuzione** e fornire **servizi comuni** per far cooperare più componenti distribuiti come se fossero locali.



**Principio di Accessibilità delle Risorse:** Garantire che **ogni risorsa remota** sia **accessibile** alle applicazioni esattamente come una **risorsa locale**, nascondendo la complessità della sua collocazione.

**Principio della “Distribution Transparency”:** La **trasparenza della distribuzione** è l'obiettivo di **nascondere** agli utenti il fatto che dati e risorse siano **fisicamente distribuiti** su più computer, anche a grande distanza.

**Principio Be Open:** Un sistema distribuito **aperto** è un sistema che **offre componenti** facilmente **utilizzabili** in **altri sistemi**, e potenzialmente a sua volta è spesso costituito da componenti provenienti da fonti esterne.

**Principio Be Scalable:** Un sistema distribuito “scalable” è progettato per adattarsi a variazioni significative di carico di lavoro, garantendo che sia possibile aggiungere nuove istanze o potenziare quelli esistenti (CPU, RAM) senza impatti negativi sulle prestazioni.

## How to scale

**Scalabilità geografica:** Un sistema è geograficamente scalabile quando utenti e risorse possono essere **lontani** fra loro, ma i **ritardi** di comunicazione risultano quasi **impercettibili**.

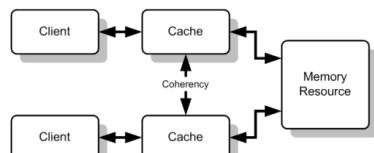
**Scalabilità amministrativa:** Un sistema è amministrativamente scalabile se rimane **facilmente gestibile** anche quando **copre molte organizzazioni** amministrative indipendenti.

**Nascondere le latenze di comunicazione:** Tecnica di **scalabilità geografica** che consiste nell'organizzare un'applicazione in modo da non **restare inattiva** in **attesa** delle risposte da un **servizio remoto**, ma **continuare ad eseguire** altri compiti utili grazie all'uso di **comunicazione asincrona**.

**Partizionamento e distribuzione:** Tecnica di scalabilità che consiste nel **suddividere** un componente in **parti più piccole** e **distribuire** queste parti **su più nodi** del sistema.

**Replicazione:** Tecnica di scalabilità che consiste nel **creare copie di componenti o dati in più nodi** di un sistema distribuito. Se un server va giù, un altro con la copia del componente può continuare a fornire il servizio. Le richieste degli utenti possono essere distribuite tra più copie.

**Caching:** Tipo particolare di **replicazione**, in cui **il client** conserva **localmente** una **copia** (cache) in modo da poterla riutilizzare senza doverla richiedere ogni volta al server, riducendo i tempi di accesso alla risorsa.



**Problema consistenza replicazione:** Se esistono più copie di una risorsa e ne viene modificata una sola, le altre diventano **non aggiornate**.

**Consistenza forte nel caching:** Ogni volta che una copia di una risorsa viene aggiornata, **tutte le altre copie devono essere immediatamente aggiornate**. Questo implica due problemi. Appena un client modifica una copia, tutte le altre devono ricevere l'aggiornamento **senza ritardi**; Se due modifiche avvengono contemporaneamente in posti diversi, bisogna garantire che **tutte le copie** le applichino nello **stesso ordine**, ovunque nel sistema.

## Leggi

**Speedup:** Incremento di tempo per una data esecuzione utilizzando N processori rispetto che 1.

$$\text{Speedup} = \frac{\text{Tempo con 1 processore}}{\text{Tempo con N processori}}$$

**Parte seriale:** Ogni programma ha una **parte seriale** che **non può essere divisa** che deve essere eseguita sequenzialmente, senza poter essere divisi tra più processori.

**Legge di Amdahl: S(N)** è lo speedup massimo utilizzando N processori, **P** è la frazione del programma **parallelizzabile**, **1-P** è la parte **seriale** (non parallelizzabile), **N** è il numero di **processori**.

$$S(N) = \frac{1}{(1 - P) + \frac{P}{N}}$$

Si ha il limite inferiore:  $\lim_{N \rightarrow \infty} S(N) = \frac{1}{1-P}$ .

Anche se aumenti il numero di processori, la parte sequenziale limita il guadagno complessivo. Con P=1 il programma è totalmente parallelizzabile, e lo speedup cresce linearmente con N.

**Esempio semplice:** Se P=0.9 e quindi il 10% rimane sequenziale, il massimo speedup,

$$\frac{1}{1-0.9} = 10$$

anche con infiniti processori, sarà:

**Legge di Gustafson:** Si ha sempre  $S(N) = \text{speedup}$  con N processori, P = frazione parallelizzabile del programma,  $1-P$  = parte sequenziale.

$$S(N) = N - (1-P)(N-1)$$

**Es.** Supponiamo che  $1-P=0.05$ , N= 100 processori.

Con **Gustafson**:

$$S(100) = 100 - 0.05 \cdot 99 = 100 - 4.95 = 95.05x \text{ di speedup.}$$

Con **Amdahl**:

$$\lim_{N \rightarrow \infty} S(N) = \frac{1}{0.05} = 20$$

Se il problema cresce, la parte parallela cresce molto più della parte seriale. I carichi di lavoro possano **crescere** quando sono disponibili più processori, aumentando così l'efficienza complessiva.

**Legge di Little:** Sia **L** = numero medio di elementi nel sistema (es. clienti in coda), **λ** = tasso medio di arrivo (es. clienti al minuto), **W** = tempo medio di permanenza di un elemento nel sistema. Si ha **L=λ·W**

Il **numero medio di entità** presenti in un sistema è uguale al **tasso di arrivo** moltiplicato per il **tempo medio che ogni entità trascorre nel sistema**.

**Es:** In un ristorante arrivano in media **10 clienti ogni ora** ( $\lambda=10$ ). Ogni cliente resta nel locale in media **30 minuti = 0,5 ore** ( $W=0.5$ ). Allora, il numero medio di clienti presenti sarà:  $L=10 \cdot 0.5=5$  **clienti nel ristorante in media**.

## Types of Distributed Systems

**Cluster computing:** Sistema distribuito in cui **più computer simili, connessi** tramite una **rete locale** ad alta velocità e con lo **stesso SO, lavorano insieme** per eseguire compiti di calcolo intensivo in modo coordinato.



**Grid computing:** Sistema distribuito in cui più sistemi informatici, spesso **eterogenei** e appartenenti a **domini amministrativi diversi** sparsi geograficamente, sono **federati** per **condividere risorse** di elaborazione al fine di **eseguire compiti** complessi.



**Cloud:** Modello di **erogazione** di risorse informatiche in cui **l'intera infrastruttura** è fornita come **servizio**, consentendo di configurare dinamicamente ciò che serve a partire da servizi disponibili in rete.

## Lezione 4° - Protocols for time management and Logical Clocks

**Assenza di un orologio globale:** Ogni nodo possiede un proprio orologio interno.

Questi orologi non sono perfettamente sincronizzati. Di conseguenza, non esiste un “tempo assoluto” valido per tutti i nodi contemporaneamente. Inoltre i messaggi inviati tra nodi subiscono ritardi dovuti alla rete.

I nodi funzionano in maniera autonoma, senza un’autorità centrale che imponga l’ordine temporale degli eventi. Devono coordinarsi e stabilire un ordine logico degli eventi.

L’ordine causa-effetto degli eventi è molto importante. Se un’azione A provoca l’azione B, il sistema deve registrare A prima di B, anche se avvengono su macchine diverse. Se le operazioni arrivano fuori ordine, i dati possono diventare incoerenti. (Leggere un saldo bancario aggiornato prima che venga registrato un versamento).

E’ importante quindi mantenere un **riferimento temporale coerente** tra nodi fisicamente separati, al fine di garantire il corretto ordinamento degli eventi.

**Ritardo di propagazione:** Tempo che impiega un segnale ottico a viaggiare lungo il cavo fisico.

**Ritardo di accodamento:** Tempo che i pacchetti devono aspettare in una coda perché il router è temporaneamente congestionato.

**Ritardo di elaborazione:** Tempo necessario al dispositivo (router, switch, server) per leggere e interpretare il pacchetto.

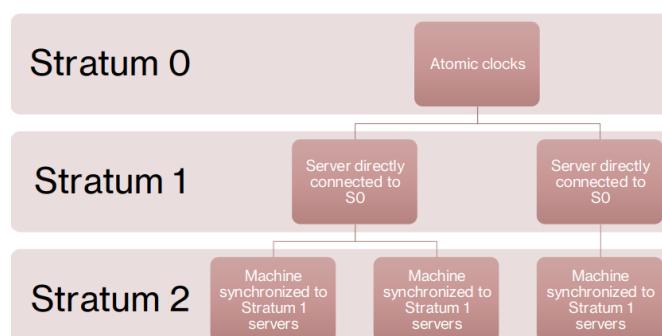
### NTP

**Network Time Protocol:** Protocollo per **sincronizzare gli orologi** dei computer in un sistema distribuito con un **riferimento temporale affidabile**. NTP lavora secondo un’architettura **gerarchica a strati**.

→ **Stratum 0:** Contiene le **fonti di tempo primarie** (orologi atomici).

→ **Stratum 1:** Contiene i **server NTP** collegati **direttamente** a stratum 0.

→ **Stratum 2 e successivi:** Contiene i computer **al livello inferiore** sincronizzati con i server NTP.

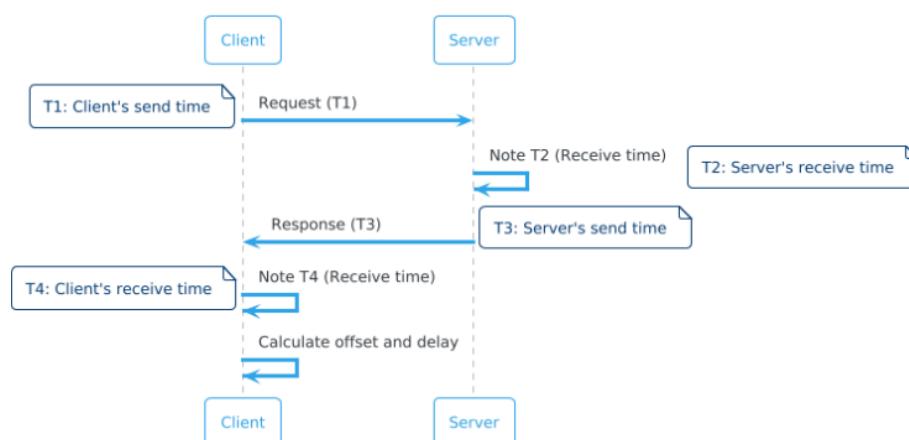


### Funzionamento

**Client’s send time T1:** Il **client** invia una **request** al server e annota l’istante in cui parte il messaggio (**T1**).

**Server’s receive time T2:** Il **server** riceve la request e annota l’orario in cui l’ha ricevuta (**T2**).

**Server’s send time:** Dopo aver elaborato la request, il server invia la **response** e annota l’istante in cui la invia (**T3**).



**Round Trip Delay:** Tempo effettivo di **viaggio** (rete) **andata e ritorno**. E' dato da  $d=(T4-T1)-(T3-T2)$ , ovvero il tempo totale del viaggio meno il tempo impiegato dal server per elaborare la risposta.

**Clock Offset:** Differenza media tra l'orologio del client e quello del server.

$$o=(T2-T1)+(T3-T4)/2 \quad ((\text{Treq server} - \text{Treq client}) + (\text{Tresp server} - \text{Tresp client})) / 2$$

Es.

Tempo client: 1627745605.250 s

Tempo server: 1627745607.500 s

L'orologio del client è **indietro di circa 2.250 secondi** rispetto al server.

T1 (**client send time**) = 1627745605.250 → quando il client invia la richiesta.

T2 (**server receive time**) = 1627745607.450 → quando il server riceve la richiesta.

T3 (**server send time**) = 1627745607.451 → quando il server risponde.

T4 (**client receive time**) = 1627745605.451 → quando il client riceve la risposta.

$$\theta = (T2-T1)+(T3-T4)/2 = (1627745607.450 - 1627745605.250) + (1627745607.451 - 1627745605.451) / 2 = (2.200 + 2) / 2 = \mathbf{2.100}$$

$$\delta = (T4-T1)-(T3-T2) = (1627745605.451 - 1627745605.250) - (1627745607.451 - 1627745607.450) = 0.201 - 0.001 = \mathbf{0.200} \rightarrow \text{tempo totale di andata e ritorno dei pacchetti sulla rete.}$$

Nuovo tempo client = 1627745605.250 + **2.100** = 1627745607.350 +- 0.200

Tempo server: 1627745607.500 s

La differenza rimasta è di **0.150 secondi**, che corrisponde al piccolo errore dovuto al ritardo di rete.

Una volta deciso l'offset corretto, **NTP non sposta bruscamente l'orologio del client**, ma applica piccole correzioni incrementali, in modo **graduale e stabile**, così da non disturbare i processi in esecuzione.

## Precision Time Protocol

**PTP:** Protocollo di sincronizzazione temporale progettato per ottenere **un'accuratezza** nell'ordine dei **microsecondi**, utilizzato in ambiti dove la **precisione temporale** è **critica**.

**Architettura base:** Il PTP usa un'architettura **gerarchica master-slave**.

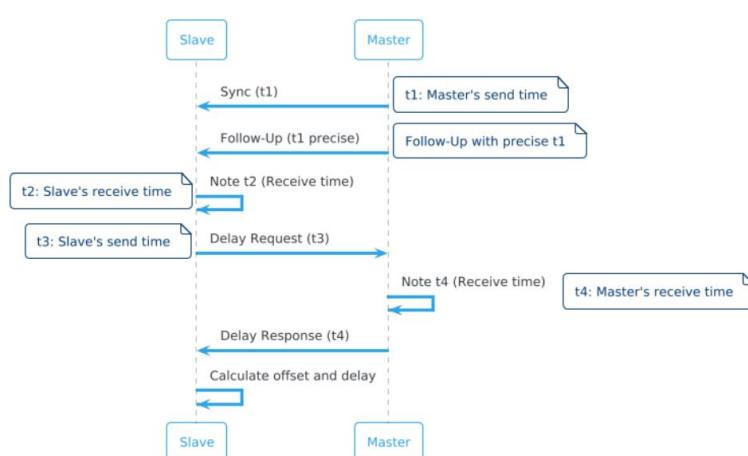
**Master Clock:** Orologio di riferimento sincronizzato con fonti molto precise (GPS).

**Slave Clock:** Orologi dei dispositivi che si sincronizzano con il master.

**Grandmaster Clock:** Se ci sono più master nella rete, uno solo viene scelto come **Grandmaster**, cioè l'orologio con l'autorità più alta in termini di accuratezza.

**Boundary Clocks:** Sono dispositivi "intermedi" che si sincronizzano con un master, ma a loro volta fanno da master per altri dispositivi, per **propagare la sincronizzazione**.

**Transparent Clocks:** Non sono né master né slave, ma si limitano a **inoltrare i messaggi PTP**, correggendo i timestamp tenendo conto dei ritardi introdotti dai **router**.



## Scambio di messaggi:

**Funzionamento:** Viene inviato dal **master** allo slave un **Sync Message** contenente l'**orario corrente** del master. Il master salva **t1**, l'istante in cui ha inviato il sync.

Il master invia poi allo slave un **Follow-Up Message**, contenente l'orario esatto **t1** in cui era **partito il Sync**. Lo slave annota **t2**, l'istante in cui riceve il **sync**.

Lo **slave** invia al master un **Delay Request Message**, per misurare il **ritardo di rete**. Lo slave annota **t3**, l'istante in cui invia il *Delay Request*.

Il master annota **t4**, l'istante in cui ha ricevuto il *Delay Request*. Il master invia **t4** allo slave sotto forma di **Delay response**. Lo slave tramite **t1**, **t2**, **t3**, **t4** si sincronizza al master.

**Clock Offset (scostamento dell'orologio):** Differenza media tra l'orologio del master e dello slave.  $o=(T2-T1)-(T4-T3)/2$

**Round Trip Delay:** Ritardo causato dalla rete, calcolato come tempo totale del viaggio meno il tempo impiegato dal server. Indica il ritardo causato dalla rete.

$$d=(T4-T1)+(T3-T2)/2$$

## Logical Clock

**Orologi logici:** I **logical clocks** sono un meccanismo per **ordinare gli eventi** in base alle loro **relazioni causali**, in modo indipendente dagli orologi fisici delle macchine.

**Lamport Timestamps:** Logical clocks che assegnano ad **ogni evento un numero** che riflette il suo **ordine logico** rispetto agli altri eventi.

Ogni processo mantiene un **contatore** intero che si **incrementa** a ogni operazione effettuata. Quando un processo invia un **messaggio**, **include** in esso il **valore attuale** del proprio contatore. Il processo che **riceve** il messaggio **confronta** il valore del contatore **ricevuto** con il **proprio contatore** locale. Aggiorna il proprio contatore a: **max(contatore locale, contatore ricevuto)+1**.

**Es:** Siano P1 e P2 due processi in un sistema distribuito.

P1 scrive su un file. Contatore P1: 0 → 1

P1 invia un messaggio a P2. Contatore P1: 1 → 2. Il messaggio contiene **timestamp 2**.

P2 riceve il messaggio. Contatore P2 prima della ricezione: 0. Contatore P2:

$$\max(0,2)+1=3.$$

P2 legge un file. Contatore P2: 3 → 4. **P2 risponde a P1**. Contatore P2: 4 → 5. Invia messaggio con **timestamp 5**.

**P1 riceve il messaggio.** Contatore P1 prima: 2. Contatore P1:  $\max(2,5)+1=6$ .

**Ordine logico degli eventi:** P1 scrive (1) → P1 invia (2) → P2 riceve (3) → P2 legge (4) → P2 invia (5) → P1 riceve (6).

**Mancanza di ordinamento totale:** Due **eventi concorrenti** (cioè che non hanno alcuna relazione causale) possono avere lo **stesso timestamp**, o uno può risultare “prima” dell’altro anche se in realtà non hanno alcun legame.

**Vector Clock:** Ogni **processo** mantiene un **vettore di contatori**. Se nel sistema ci sono **N** processi, ogni vettore ha **N** elementi. Ogni elemento del vettore relativo ad un processo rappresenta il **numero di eventi** conosciuti dal processo per quel **processo**.

**Evento locale:** Quando un processo compie un evento, **incrementa il proprio contatore** all’interno del vettore.

**Invio di un messaggio:** Il processo incrementa il proprio contatore e allega **l’intero vettore** al messaggio.

**Ricezione di un messaggio:** Il processo ricevente aggiorna il proprio vettore facendo la **massimizzazione elemento per elemento** tra il proprio vettore e quello ricevuto. Poi incrementa il contatore corrispondente a sé stesso per registrare l’evento di ricezione.

**ES:** Numero di processi N=2.

Orologi iniziali: VC1=[0,0] VC2=[0,0]

Supponiamo che P1P<sub>1</sub> faccia un evento interno. Aggiorna la sua voce:

$$VC1[1]=VC1[1]+1=1$$

Resultato: VC1=[1,0]

P1 invia un messaggio a P2, **includendo** il suo orologio VC1=[1,0] nel messaggio.

Quando P2P<sub>2</sub> riceve il messaggio, aggiorna elemento per elemento con il **massimo**:

$$VC2[2]=\max(VC2[2], VC1[2])=\max(0,0)=0$$

Quindi prima dell’incremento: VC2=[1,0]

Poi incrementa la propria voce: VC2[2]=VC2[2]+1=1

Resultato finale: VC1=[1,0], VC2=[1,1]

Questo meccanismo garantisce che gli orologi vettoriali riflettano accuratamente le relazioni causalì tra gli eventi nel sistema.

**Happens-before (accade-prima):** Siano  $a$  un evento del processo  $P_i$  e  $b$  un evento del processo  $P_j$ , e  $VC(a)$ ,  $VC(b)$  i rispettivi vector clock. Un evento  $a$  è accade-prima di un evento  $b$ , indicato come **se e solo se**:  $VC(a) < VC(b)$ , ovvero il vettore nel processo  $P_i$  al momento di  $a$  è inferiore in tutti i suoi valori al vettore di  $P_j$  al momento di  $b$ .

**Concorrenza:** Due eventi  $a$  e  $b$  sono concorrenti se nessuno dei due accade-prima dell'altro, ovvero quando i loro orologi vettoriali **non sono confrontabili** in tutti gli elementi.

**Esempio:** Supponiamo di avere due processi P1 e P2.

Evento **a** in P1:  $VC(a) = [1, 0]$

Evento **b** in P2 dopo aver ricevuto un messaggio da P1:  $VC(b) = [1, 1]$

Confronto:  $VC(a)[1] = 1 \leq 1 = VC(b)[1]$  ✓

$VC(a)[2] = 0 \leq 1 = VC(b)[2]$  ✓

E almeno un valore è strettamente minore ( $0 < 1$ )

Significa che l'evento **a** in P1 ha preceduto l'evento **b** in P2.

**Confronto con concorrenza**

Se invece avessi:

$VC(x) = [1, 0]; VC(y) = [0, 1]$

Allora:

Non è vero che  $VC(x) \leq VC(y)$  (perché  $1 \neq 0$ ). Non è vero che  $VC(y) \leq VC(x)$  (perché  $1 \neq 0$ ).

Quindi **x e y sono concorrenti**: nessuno accade prima dell'altro.

## Lezione 5 - Mutual exclusion in Distributed Systems

**Esempio:** Due processi aggiornano contemporaneamente lo stesso saldo di un conto. Senza sincronizzazione: il risultato finale può essere sbagliato (perdita o duplicazione di denaro). Con sincronizzazione: gli aggiornamenti vengono gestiti in modo ordinato: consistenza garantita.

**Mutual Exclusion: Proprietà** che garantisce che **solo un processo alla volta** possa entrare in una **sezione critica**.

**Sezione critica:** È il blocco di codice che utilizza una risorsa condivisa.

Gli scopi della mutua esclusione sono **prevenire corruzione dei dati** e **assicurare che le operazioni sui dati condivisi siano eseguite in modo atomico**.

### Goals of Mutual Exclusion

**Safety:** Garantisce che **solo un processo alla volta** possa trovarsi nella **critical section**.

**Liveness:** Garantisce che ogni processo che vuole entrare nella critical section **prima o poi potrà farlo. (No starvation)**.

**Fairness:** Le richieste di ingresso nella critical section vengono servite **nell'ordine in cui sono state fatte** (first-come, first-served).

### Lamport's Bakery Algorithm:

**Assegnazione biglietto:** Un processo che vuole entrare nella critical section sceglie un **numero di ticket = max(ticket di tutti) + 1**.

**Attesa turno:** Il processo aspetta che **tutti i processi con numero più piccolo** abbiano finito. Se due processi hanno lo stesso numero, vince quello con **ID di processo più piccolo**.

**Critical Section:** Il processo con il numero di ticket più basso entra nella **critical section**. Tutti gli altri attendono.

**Rilascio biglietto:** Quando il processo esce dalla critical section, **resetta il suo ticket** (lo mette a 0), indicando che non sta più partecipando alla "fila".

**Esempio:** Supponiamo tre processi P1, P2, P3: P1 prende ticket 1. P2 prende ticket 2. P3

prende ticket 3. Ordine di ingresso: P1 → P2 → P3.

Se P2 e P3 prendono il numero **contemporaneamente**: Entrambi leggono max=1 →

ticket=2. Per decidere, si guarda l'**ID del processo** → quindi **P2 (ID minore)** entra prima

di P3.

## Strutture dati usate:

**choosing[N]:** Array di booleani che indica se un processo sta scegliendo un numero di ticket.

**number[N]:** Array di interi che contiene i numeri di ticket assegnati ai processi.

*Require: N is the number of processes*

*Ensure: Mutual exclusion among processes*

```
choosing[N] → {false, false, ..., false} // Initialize choosing flags for each process  
number[N] → {0, 0, ..., 0}           // Initialize ticket numbers for each process
```

...

Process (i):

```
lock(i);
```

*critical session code*

```
unlock(i);
```

```
void lock(int i) {  
    choosing[i] = true;  
    number[i] = 1 + max(number[0], ..., number[N-1]);  
    choosing[i] = false;  
  
    for (int j = 0; j < N; j++) {  
        while (choosing[j]);  
        // Attende che l'altro processo j abbia finito di scegliere il numero.  
  
        while (number[j] != 0 &&  
               (number[j] < number[i] ||  
                (number[j] == number[i] && j < i)));  
        // Attende se:  
        // - j ha un numero più piccolo (quindi ha la priorità), oppure  
        // - j ha lo stesso numero ma ID più piccolo (tie-breaking).  
    }  
}
```

**Spiegazione:** Il processo annuncia che sta scegliendo il ticket, assegna il proprio ticket:  $1 + \max$ , e annuncia che ha finito di scegliere il ticket. Scansiona tutti gli altri processi j. Aspetta per ogni processo che non stia scegliendo il numero, per non leggere number[j] a metà. Rimane in attesa se il numero scelto da j è inferiore, o se è uguale ma il processo j ha un identificativo inferiore. Nella unlock il processo rilascia il ticket settandolo a 0.

```
void unlock(int i) {  
    number[i] = 0; // Rilascia il ticket → non è più in attesa  
}
```

**Basato su software:** L'Algoritmo viene implementato interamente tramite **codice e costrutti logici**, senza fare affidamento su meccanismi hw stile spinlock.

**Svantaggi:** Alto overhead di comunicazione a causa del **controllo continuo dei valori dei ticket**.

## Lamport Distributed Mutual Exclusion Algorithm

**Logical Clocks:** Ogni processo mantiene un **Lamport clock**, che viene incrementato ogni volta che il processo invia una **REQUEST**, oppure la **riceve**.

**Richiesta di accesso:** Quando un processo vuole entrare nella **critical section**, invia un messaggio **REQUEST(timestamp, id)** a tutti gli altri processi. Le richieste vengono ordinate in base al timestamp, che è il **Lamport clock**, quindi la più vecchia ha la priorità.

**Attesa delle risposte:** Ogni altro processo, quando riceve la richiesta, decide se rispondere subito oppure aspettare, se ha una richiesta con priorità più alta in corso.

**Ingresso nella sezione critica:** Quando il processo ha raccolto **tutti i REPLY**, significa che nessun altro ha la priorità, quindi può entrare nella **critical section**.

**Rilascio della sezione critica:** Dopo aver completato il lavoro, il processo invia un messaggio **RELEASE** a tutti, che notifica agli altri che la risorsa è libera, così possono aggiornare le loro code.

- ◆ **Esempio**

Supponiamo 3 processi (P1, P2, P3):

1. P1 manda **REQUEST(5, P1)** → inserito in tutte le code.
2. P2 manda **REQUEST(6, P2)**.
3. P3 manda **REQUEST(7, P3)**.
  - Ordine nelle code: P1 → P2 → P3.
  - P1 entra per primo, quando esce manda **RELEASE**.
  - Le code vengono aggiornate → P2 è il prossimo.

**Es:** **P1** invia un messaggio **REQUEST** con il suo timestamp a **P2 e P3**. **P2 e P3** rispondono a **P1** con un **REPLY**. Una volta che **P1** ha ricevuto tutti i messaggi **REPLY**, entra nella **sezione critica**. Dopo aver terminato, **P1** invia un messaggio **RELEASE** a **P2 e P3**.

**Complessità in messaggi:** In totale, servono **3(N – 1)** messaggi per un singolo ingresso in sezione critica, ovvero **(N – 1) REQUEST messages + (N – 1) REPLY messages + (N – 1) RELEASE messages**. Svantaggio.

## Lezione: Advanced algorithms for Mutual exclusion

### Token Based Approach

**Algoritmo di Suzuki-Kasami:** Abbiamo un **token** che circola tra i processi, il processo che **possiede il token** ha accesso esclusivo alla **sezione critica**. Quando un processo vuole entrare nella sezione critica, invia **messaggi di richiesta** a tutti gli altri processi.

### Strutture dati processi

**Array Request Number RNi[N]:** Array mantenuto da ogni processo *i*, tale che l'elemento  $RNi[j]$  contiene l'**ultimo numero di richiesta ricevuto dal processo j**. Serve per sapere se la richiesta di un processo *j* è nuova o già servita.

### Strutture dati token

**Array Last Request Number LN[N]:**  $LN[j]$  indica l'**ultimo numero di richiesta di j** per cui il token è già stato concesso. In questo modo, il sistema sa quali richieste sono **già state soddisfatte** e non deve ridare il token due volte per la stessa richiesta.

**Coda Q:** Coda che contiene gli **ID dei processi in attesa del token**, usata per gestire l'ordine di assegnazione del token.

### Algoritmo

**Richiesta di entrare nella Critical Section:** Quando un **processo i** vuole entrare nella critical section, **controlla se ha il token**. Se ce l'ha, **entra direttamente**, altrimenti **deve richiedere il token**.

In tal caso, **incrementa il contatore delle sue richieste**  $RNi[i]=RNi[i]+1$ , e **invia il messaggio di richiesta a tutti** gli altri processi,  $REQUEST(i,RNi[i])$ .

**Rilascio della Critical Section:** Quando il processo *i* ha **terminato** la sezione critica, imposta l'ultimo numero di richiesta concesso dal token per *i* come quello salvato internamente, impostando  $LN[i]=RNi[i]$ , indicando che la sua ultima richiesta è stata servita.

Successivamente, **controlla** le richieste pendenti degli altri. Per ogni processo *k* che **non è già nella coda Q**: Se  $RNi[k]=LN[k]+1$ , allora il processo *k* ha una richiesta pendente **non ancora servita**. In questo caso, *k* viene aggiunto alla coda Q.

**Gestione del token:** Se la coda Q **non è vuota**: Estrae un processo *j* da Q. **Invia il token a j**. Se la coda Q **è vuota**, **i mantiene il token** fino a quando un altro processo non farà richiesta.

**Es.** Il **token** è in mano a **P1**. Tutti i contatori iniziano da 0:  $RN1=RN2=RN3=[0,0,0]$ .

$LN=[0,0,0]$ .  $Q=[]$  (vuota).

**P2 vuole entrare nella CS:** P2 non ha il token, quindi: Incrementa il proprio contatore:

$RN2[2]=RN2[2]+1=1$ . Invia a tutti un messaggio: **REQUEST(2,1)**. Gli altri processi aggiornano i loro RN: P1 e P3 memorizzano che **P2 ha fatto una nuova richiesta**.

**P1 termina la sua CS:** P1 aggiorna il token:  $LN[1] = RN_1[1] = 1$  (se aveva richiesto).

Controlla le richieste: Vede che  $RN[2]=LN[2]+1=1$ , quindi P2 ha una richiesta pendente.

Aggiunge P2 alla coda:  $Q=[2]$ . Invia il **token a P2**.

**P2 entra nella CS:** P2 riceve il token, ed entra subito nella CS.

**P3 vuole entrare nella CS mentre P2 è dentro:** P3 non ha il token, quindi: Incrementa  $RN3[3]=1$ . **REQUEST(3,1)** a tutti. Gli altri processi aggiornano le loro copie  **$RN[3]$** .

**P2 esce dalla CS:** P2 aggiorna il token:  $LN[2]=RN2[2]=1$ . Controlla le richieste:

Vede che  $RN[3]=LN[3]+1=1$ , quindi P3 è in attesa. Aggiorna la coda:  $Q=[3]$ . Invia il token a P3.

**Prestazioni dei messaggi:** Se un processo **ha già il token**, può entrare direttamente nella **critical section** senza dover mandare messaggi. Se un processo **non ha il token**, deve inviare: **N-1 messaggi di richiesta** (a tutti gli altri processi). **1 messaggio di reply** con cui gli viene inviato il token. Quindi, in totale **N messaggi** per ottenere la CS. Sono pochi quando si hanno pochi processi.

**Perdita del token:** Se il **token si perde**, nessuno **potrà più** entrare nella sezione critica.

**Timeout:** Se un processo **aspetta troppo** senza ricevere il token, suppone che sia stato perso e inizia una **procedura di rigenerazione**.

## No-Token approach

**Ricart-Agrawala Algorithm:** Quando un processo i vuole entrare nella **sezione critica**, invia un **messaggio di Request** a tutti gli altri processi. **Ogni Request** è etichettata con un **timestamp logico**. Gli altri processi se non vogliono entrare nella sezione critica o non hanno una richiesta con timestamp minore, inviano una **Reply**. Il processo i resta in attesa, e quando ha ricevuto le **Reply** da tutti i processi, può entrare nella critical section.

**Es:** Immaginiamo 3 processi: P1,P2,P3.

**P1 vuole entrare nella CS:** Invia **REQUEST(P1)** a P2 e P3.

**P2 e P3 ricevono la richiesta:** Se non vogliono entrare nella CS o hanno una richiesta con timestamp maggiore, rispondono subito con **REPLY**. Se invece vogliono entrare anch'essi e la loro richiesta ha **priorità più alta** (timestamp minore), possono **differire la risposta** (la invieranno dopo).

**P1 riceve tutte le risposte:** Ora può entrare nella **critical section**.

**P1 esce dalla CS:** Invia **REPLY** a tutti i processi le cui richieste aveva messo in attesa (quelle "deferred").

**Message Complexity:** Per ogni richiesta alla CS servono: N-1 messaggi di **request** (agli altri processi), e N-1 messaggi di **reply**, per un totale di **2(N - 1) messaggi**.

**Network failures:** Se un messaggio si perde o un nodo della rete è lento, il processo richiedente resta bloccato in attesa.

## Quorum Approach

**Maekawa's Algorithm:** Ogni processo comunica solo con un **sottoinsieme di processi** chiamato **quorum**.

Un **quorum** è un **sottoinsieme di processi** a cui un processo deve chiedere il permesso per entrare nella zona critica. **Qualsiasi coppia di quorum ha almeno un membro in comune**. Quel processo in comune può dare il permesso a uno solo dei due quorum, in caso due processi di entrambi i quorum richiedono l'accesso contemporaneamente.

**Es:** Sistema con 5 processi: P1, P2, P3, P4, P5.

$Quorum(P1) = \{P1, P2, P3\}$

$Quorum(P2) = \{P2, P4, P5\}$

$Quorum(P3) = \{P1, P3, P4\}$

Se P2 e P3 vogliono accedere insieme alla CS, e hanno un processo in comune (P4), P4 darà il permesso solo a uno dei due.

**Rischi di deadlock:** Se tutti i processi vogliono entrare nella critical section in contemporanea, può avvenire una negazione circolare da parte dei processi in comune dei quorum, con conseguente deadlock.

Es.  
Quorum(P1) = {P1, P2}  
Quorum(P2) = {P2, P3}  
Quorum(P3) = {P3, P1}

Tutti i processi vogliono entrare nella CS quasi in contemporanea: **P1** chiede il permesso a {P1, P2}. **P2** chiede il permesso a {P2, P3}. **P3** chiede il permesso a {P3, P1}.

**P1** vota per sé stesso. **P2** vota per sé stesso. **P3** vota per sé stesso.

P1 manda richiesta a **P2**, ma P2 ha già votato per sé stesso → ✗ rifiuta.

P2 manda richiesta a **P3**, ma P3 ha già votato per sé stesso → ✗ rifiuta.

P3 manda richiesta a **P1**, ma P1 ha già votato per sé stesso → ✗ rifiuta.

P1 ha 1 voto (da sé stesso), ma non quello di P2 → bloccato.

P2 ha 1 voto (da sé stesso), ma non quello di P3 → bloccato.

P3 ha 1 voto (da sé stesso), ma non quello di P1 → bloccato.

**Timeout:** Se un processo aspetta troppo a lungo, rilascia i permessi e riprova.

**Priorità al timestamp:** L'accesso va alla richiesta con timestamp più vecchio.

**Cambio quorum:** Cambiare dinamicamente i quorum per risolvere conflitti.

**Vantaggi:** Il numero di messaggi per richiesta è  $O(\sqrt{N})$  invece di  $O(N)$ , ideale per sistemi con **molti processi** e **alta contesa**.

## Lesson - Deadlock detection and resolution in distributed systems

**Deadlock:** Un *deadlock* si verifica quando ciascun processo aspetta qualcosa che un altro processo deve rilasciare, e questo crea un **ciclo di attesa circolare** in cui **nessuno può andare avanti**.

**Es:** **P1** ha una risorsa **R1** ed è in attesa di ottenere **R2**. **P2** ha una risorsa **R2** ed è in attesa di ottenere **R1**. Nessuno dei due può rilasciare la risorsa perché entrambi stanno aspettando l'altro, rimanendo entrambi bloccati per sempre.

**Condizioni per il Deadlock:** Perché si verifichi un **deadlock**, devono essere vere **tutte e quattro** le seguenti condizioni **contemporaneamente**.

**Mutual Exclusion:** Solo **un processo alla volta** può usare una certa risorsa, che non può essere condivisa.

**Hold and Wait:** Un processo **può trattenere** alcune risorse già ottenute **mentre attende** di ottenerne altre.

**No Preemption:** Le risorse **non possono essere sottratte forzatamente** a un processo, ma devono essere **rilasciate volontariamente**.

**Circular Wait:** Esiste una **catena chiusa di processi**, in cui ogni processo **detiene** una risorsa, e **attende** un'altra risorsa **posseduta dal processo successivo** nella catena.

*Es:* P1 → ha R1, aspetta R2. P2 → ha R2, aspetta R3. P3 → ha R3, aspetta R1. → Deadlock.

**Prevenzione del Deadlock:** Si cerca di **evitare che un deadlock possa accadere**, impedendo che **una delle quattro condizioni di Coffman** sia vera. Se almeno una di queste non si verifica, il deadlock è impossibile.

**Acquisire tutte le risorse prima di iniziare:** Ogni processo deve richiedere tutto ciò di cui ha bisogno prima di cominciare. Se non può ottenerle tutte subito, aspetta.

**Deadlock Avoidance:** Il sistema **non impedisce** ai processi di chiedere risorse, ma **controlla dinamicamente** ogni richiesta per verificare se concederla porterebbe il sistema in una **situazione pericolosa (unsafe)**. Solo se lo stato rimane **“sicuro” (safe)**, la risorsa viene concessa. (Algoritmo del banchiere, no sistemi distribuiti).

### A deadlock avvenuto rilevato

**Preemption:** Se un processo tiene una risorsa ma ne richiede un'altra, il sistema può **togliergli temporaneamente** quella che ha. Poi la riavrà più tardi.

**Kill:** **Terminare uno o più processi coinvolti nel deadlock** (spesso si sceglie quello meno “costoso” da riavviare in termine di danni).

**Rollback and Recovery:** Ogni processo **salva periodicamente il proprio stato** (detto **checkpoint**). Se avviene un deadlock, il sistema **torna all'ultimo checkpoint** privo di deadlock e **riprende l'esecuzione da lì**.

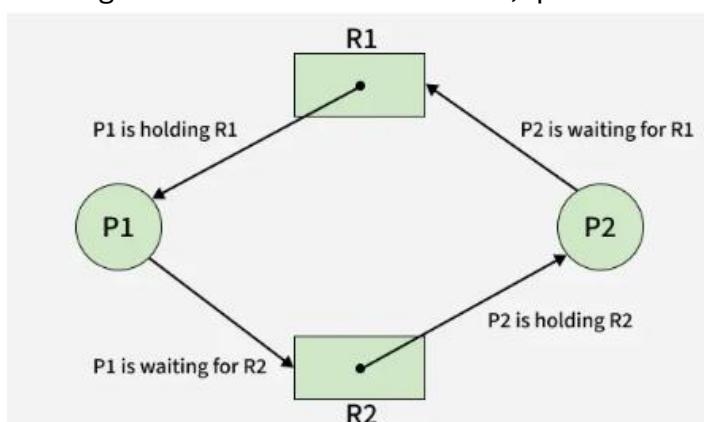
### Deadlock Detection in Resource Allocation Graphs

**Resource Allocation Graph:** Grafo utilizzato per rilevare se c'è un deadlock.

**Nodi: Processi**, indicati come **P1, P2, P3**, e **Risorse**, indicate come **R1, R2, R3**.

**Archi: Da processo a risorsa** ( $P \rightarrow R$ ), in cui il processo P **richiede** quella risorsa R, o da **risorsa a processo** ( $R \rightarrow P$ ), in cui la risorsa R è **assegnata** a quel processo P.

Se nel grafo si forma un **ciclo chiuso**, questo indica un **potenziale deadlock**.

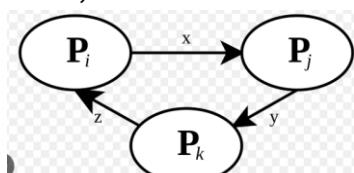


**Es:** La risorsa R1 è assegnata a P1 ( $R1 \rightarrow P1$ ). Il processo P1 richiede la risorsa R2 ( $P1 \rightarrow R2$ ). La risorsa R2 è assegnata a P2 ( $R2 \rightarrow P2$ ). Il processo P2 richiede la risorsa R1 ( $P2 \rightarrow R1$ ). Nessuno dei due può andare avanti, perché P1 aspetta R2 (che ha P2), mentre P2 aspetta R1 (che ha P1). Si crea un **ciclo di attesa circolare**, cioè un **deadlock**

**Wait-for Graph: Versione semplificata** del RAG, che elimina i nodi delle risorse e mantiene solo le relazioni di attesa tra processi.

I **nodi** rappresentano solo i processi (P1, P2, P3, ...). Gli **archi** rappresentano le **relazioni di attesa** tra processi. Una freccia  $P1 \rightarrow P2$  significa “Il processo P1 sta aspettando una risorsa che è posseduta da P2.”

Si parte da un **RAG** e si rimuovono i nodi delle risorse. Si sostituiscono quindi i percorsi  $P \rightarrow R \rightarrow Q$  con un singolo arco  $P \rightarrow Q$ , eliminando R. Se nel Wait-for Graph **esiste un ciclo**, allora c’è un **deadlock**.



**Es:** P1 → P2 → P3 → P1: P1 sta aspettando P2, P2 sta aspettando P3, P3 sta aspettando P1. **Deadlock**.

**Distributed Deadlock Detection Approach:** Tutti i nodi **collaborano** tra loro per rilevare i deadlock. Ogni nodo tiene traccia **localmente** delle risorse e dei processi che conosce, **scambia informazioni** con gli altri nodi, e partecipa al **rilevamento collettivo** dei cicli nel sistema globale. Questo continuo scambio di informazioni permette di **ricostruire insieme**, se esiste un **ciclo di attesa globale**. Viene utilizzato un **wait-for graph** per mantenere il modello del sistema **globale**.

*Es:* Nodo A rileva che P1 aspetta P2. Nodo B rileva che P2 aspetta P3. Nodo C rileva che P3 aspetta P1. Nessun nodo da solo vede il ciclo, ma globalmente, scambiandosi i

## Models of Deadlock

**Single Resource Model:** Ogni **processo** può avere al massimo una sola richiesta **attiva** alla volta. Ogni **risorsa** può essere **assegnata a un solo processo** alla volta.

Nel **Wait-for Graph**, ogni processo è rappresentato da un **nodo**. Un arco  $P1 \rightarrow P2$  significa che **P1 sta aspettando una risorsa detenuta da P2**. Ogni processo può aspettare **solo un altro processo**, quindi il **grado uscente massimo** di ogni nodo è 1.

*Es:* P1 → P2 → P3 → P1

**The AND Model:** Un processo può chiedere più risorse contemporaneamente, ma può procedere solo se le ottiene tutte insieme.

Nel WFG un processo può avere **più archi uscenti** perché può aspettare più risorse contemporaneamente. Anche se **non c'è un ciclo** nel WFG, **può comunque esserci un deadlock**, perché il processo **aspetta più risorse contemporaneamente**, e queste risorse **sono sparse tra più processi**, che a loro volta potrebbero essere bloccati per altre ragioni.

**Es:** P1 → P2, P1 → P3, P2 → P4, P3 → P4. P1 ha bisogno **sia di una risorsa da P2 che da P3**. P2 e P3 stanno entrambi aspettando P4. P4 è bloccato o non rilascerà mai le risorse. No ciclo ma deadlock.

**OR Model:** Un processo può richiedere più risorse, ma gli basta una sola delle risorse richieste per procedere. Nel WFG, un processo può avere **più archi uscenti**, ma non serve che **tutti** siano risolti, **basta che se ne liberi uno**. Quindi un ciclo **non implica necessariamente** un deadlock, perché un processo potrebbe “sbloccare” il ciclo ricevendo **una sola** delle risorse che aspetta.

#### The AND-OR Model

Un processo può chiedere un insieme di risorse in modo combinato, dove alcune richieste devono essere soddisfatte **tutte insieme (AND)** e altre possono essere soddisfatte **in alternativa (OR)**.

**Es:** Il processo **P1** richiede: **x AND (y OR z)**. P1 **deve avere la risorsa x**, e **deve ottenere almeno una** tra le risorse y e z.

### Classification of distributed deadlock detection algorithms

**Path-Pushing Algorithm:** Ogni nodo **costruisce** il proprio **WFG locale**. Quando un nodo **scopre** che un suo **processo sta aspettando** un processo su un **nodo remoto**, **invia un messaggio** a quel nodo per aggiornarlo di essere in attesa.

Quando il nodo remoto riceve il messaggio, **aggiorna** il proprio **WFG locale**, aggiungendo l'attesa dal nodo esterno. Viene **aggiornato** così una parte del **WFG globale**, e col tempo in modo distribuito viene ricostruito il WFG globale. Quando un nodo scopre che un **percorso forma un ciclo**, allora viene dichiarato un **deadlock**.

**Es:** Nodo 1: P1 → P2, Nodo 2: P2 → P3, Nodo 1 invia a Nodo 2 il percorso **P1 → P2**, per informarlo che P2 è atteso da P1. Ora Nodo 2 sa: P1 → P2 → P3. P3 ora sta aspettando P1. Nodo 2 invia a Nodo 3 il percorso **P1 → P2 → P3**. Nodo 3 lo aggiorna aggiungendo la sua dipendenza: P1 → P2 → P3 → P1. Nodo 3 nota che il percorso **torna a P1**, cioè **un ciclo completo** è stato formato. **Deadlock**.

**Edge-Chasing Algorithm:** I processi si mandano messaggi **probe** per verificare se si trovano in un ciclo di attesa. Quando un processo **P1 è bloccato** in attesa di una risorsa da **P2**, invia un **probe message** a P2. (Probe: initiator = P1, sender = P1, receiver = P2). Se **P2 sta aspettando anche lui** un altro processo **P3**, allora **P2 propaga** il probe a P3. (probe: initiator = P1, sender = P2, receiver = P3).

Se un processo **non è bloccato**, **scarta il probe** appena lo riceve, perché non fa parte di alcun ciclo di attesa. Solo i processi **bloccati** continuano a propagare i probe.

Se a un certo punto il probe **ritorna al processo iniziale (P1)**, significa che si è formato un **ciclo di attesa**: P1 → P2 → P3 → P1. Quindi, il sistema **dichiara un deadlock**.

**Diffusion Computation Algorithm:** Quando un processo **P<sub>i</sub>** diventa **bloccato** in attesa di risorse da altri processi, diventa **l'iniziatore** della rilevazione del deadlock e **invia query messages** a tutti i processi **da cui dipende**.

Ogni processo che riceve una **query**, se è **attivo** **scarta la query**, mentre se è **bloccato**, allora **memorizza** da chi ha ricevuto la query, e **propaga** la query a tutti i processi da cui **lui stesso** sta aspettando risorse.

Quando un processo riceve una query, ci sono due casi.

Se è la **prima query** ricevuta per questa rilevazione, il processo non risponde subito, ma prima **manda** a sua volta **query** ai processi da cui dipende nel grafo, aspetta **tutte le risposte** da quei processi, e solo quando ha ricevuto **tutte le reply** dai suoi “figli”, invia **una reply** al suo “genitore”.

Se un processo riceve **un'altra query** per la stessa *rilevazione*, significa che qualcun altro gli ha mandato lo stesso messaggio. In questo caso, **risponde subito** con una reply, senza ri-propagare la query, in modo da evitare **loop infiniti** e messaggi duplicati.

**Condizione di deadlock:** L'iniziatore capisce che c'è un **deadlock** quando ha ricevuto **una risposta** per **ogni query** che aveva inviato, ma **non si è sbloccato** nel frattempo.

Questo significa che tutti i processi contattati sono rimasti bloccati, nessuno ha potuto liberare risorse, e quindi **esiste un ciclo di attesa. Deadlock**.

**Es:** P1 → P2 → P3 → P1

P1 è bloccato, quindi avvia il rilevamento. P1 invia **query(P1 → P2)**.

P2 riceve la query, è anche lui bloccato. → P2 memorizza che la query arriva da P1. → P2 invia **query(P2 → P3)**.

P3 riceve la query, è anche lui bloccato. → P3 invia **query(P3 → P1)** (perché sta aspettando P1).

P1 riceve una query da P3 (riconosce se stesso nell'iniziatore). → Questo indica che si è formato un ciclo. → P1 capisce che tutti i processi coinvolti sono bloccati. **Deadlock rilevato.**

Ora i processi inviano **reply** indietro: P3 manda reply a P2, P2 manda reply a P1.

Quando P1 ha ricevuto tutte le reply → **conferma il deadlock**.

**Global State Detection-Based Algorithms:** Lo **stato globale** è composto dallo **stato locale** di ciascun processo (quali risorse ha, se è attivo o bloccato), e lo **stato dei messaggi in transito** tra i nodi. Un **snapshot** è una “**fotografia logica**” del sistema, che raccoglie gli stati locali e i messaggi in transito **in modo coerente**.

Lo snapshot **non fotografa un momento reale preciso**, perché i nodi sono asincroni e non condividono un orologio globale. Tuttavia, se nel sistema una **proprietà stabile** (come il deadlock) era vera prima dello snapshot, allora sarà **ancora vera nello snapshot**.

Si **raccoglie uno snapshot** coerente dello stato globale. Si **ricostruisce il Wait-for Graph globale**, unendo le informazioni di tutti i nodi. Si **analizza** il grafo risultante. Se c'è un **ciclo di attesa**, viene rilevato un **deadlock rilevato**.

## Lezione – Self stabilization

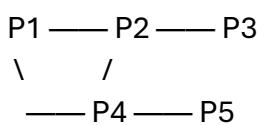
E' possibile che un **sistema distribuito** entri in uno **stato "illegittimo**" di errore. Un messaggio può essere **perso** durante la comunicazione, un nodo può **fallire temporaneamente**, o le variabili locali dei nodi possono **divergere** e non essere più coerenti.

**Self stabilization:** Indipendentemente dallo stato iniziale, il sistema è garantito a **convergere** verso uno **stato legittimo** entro un **tempo limitato, da solo**, senza alcun intervento esterno.

**System model:** Un sistema distribuito è formato da un insieme di  $n$  macchine, dette **processori**  $P_1, P_2, \dots, P_n$ . Ognuno di questi processori può **eseguire calcoli**, e **scambiare messaggi** con altri processori.

I processori non sono tutti collegati tra loro, ma solo con alcuni **vicini**. Due processori  $P_i$  e  $P_j$  sono **vicini** se **possono comunicare direttamente, scambiandosi messaggi** tra loro.

**Rappresentazione come grafo:** Ogni **nodo**  $v_i \in V$  rappresenta un **processore**  $P_i$ . Ogni **arco** tra due nodi rappresenta un **canale di comunicazione** tra due processori vicini.



**Canale di comunicazione FIFO:** Ogni canale di comunicazione  $Q_{ij}$  tra due processori viene rappresentato dalla coda FIFO dei messaggi in transito che il processore  $P_i$  ha inviato al suo vicino  $P_j$ , ma che  $P_j$  non ha ancora ricevuto.

**Stato di un processore:** Ogni processore  $P_i$  ha un proprio **stato interno**, che include variabili locali (variabili, flag, token, ecc.), informazioni ricevute dai vicini, decisioni sul prossimo messaggio da inviare.

**Stato globale del sistema:** A un dato istante di tempo, lo **stato completo del sistema distribuito** è determinato dallo **stato interno di ogni processore**  $P_i$  e dal **contenuto di ogni coda**  $Q_{ij}$  di messaggi in transito.  $c = (s_1, s_2, \dots, s_n, q_{1,2}, q_{1,3}, \dots, q_{ij}, \dots, q_{n,n-1})$ ,

**Es:**  $P1 \text{ --- } P2 \text{ --- } P3$ ,  $c = (s_1, s_2, s_3, q_{1,2}, q_{2,1}, q_{2,3}, q_{3,2})$

$s_1, s_2, s_3$  sono gli **stati interni** dei tre processori;  $q_{1,2}$  = messaggi da  $P_1$  a  $P_2$  non ancora ricevuti;  $q_{2,3}$  = messaggi da  $P_2$  a  $P_3$  in transito, ecc.

**Comportamento dinamico del sistema:** È composto dal set di stati e dalle transizioni tra stati.

**Set di stati:** Tutte le configurazioni  $c$  possibili che il sistema può teoricamente assumere. Alcune sono **leggitive**, altre **illeggitive**.

**Transizione tra stati:** Indica come il sistema passa da una configurazione all'altra nel tempo.  $c \xrightarrow{\text{azione}} c'$  significa partendo dalla configurazione  $c$ , dopo una certa azione, il sistema passa alla configurazione  $c'$ .

**Weak Fairness:** Se un'azione è **sempre pronta** a essere eseguita da un certo momento in poi, allora **prima o poi dovrà essere eseguita**.

**Strong Fairness:** Se un'azione diventa **abilitata infinite volte**, allora verrà eseguita infinite volte.

**Diametro della rete  $\delta$ :** Presi due nodi qualsiasi della rete, preso il percorso più breve che li collega, il **diametro  $\delta$**  è la lunghezza del percorso più lungo tra tutti quelli minimi possibili. Con  $P_1 - P_2 - P_3 - P_4 - P_5$ , il diametro  $\delta = 4$ , cioè servono 4 "salti" per andare da  $P_1$  a  $P_5$ .

$\Delta$ : Indica il **numero massimo di vicini** che un nodo può avere.

**Final failure:** Ultima volta che un nodo smette di funzionare.

**Tempo di stabilizzazione:** Tempo che il sistema impiega per ritornare in uno stato legittimo dal **final failure**.

**Definizione formale:** Consideriamo un **sistema distribuito**  $S$ , e indichiamo con  $P$  una proprietà che descrive le **condizioni corrette** in cui il sistema dovrebbe trovarsi.

**Stati legittimi:** Tutti gli stati che soddisfano il **predicato  $P \rightarrow$** , in cui il sistema funziona correttamente.

**Stati illegittimi:** Tutti gli stati che non soddisfano  $P \rightarrow$ , in cui il sistema è in errore.

Il sistema  $S$  deve rispettare **due proprietà fondamentali**:

**Closure:** " $P$  è chiuso rispetto all'esecuzione di  $S$ ", ovvero se il sistema si trova in uno stato legittimo ( $P$ ), allora **tutti gli stati successivi** (ad un'azione) **rimarranno legittimi**.

$P(s) \text{ e } s \xrightarrow{S} s' \Rightarrow P(s')$

**Convergence:** "Da qualsiasi stato globale, anche illegittimo,  $S$  raggiunge uno stato che soddisfa  $P$  in un numero finito di passi."

**Definizione Stabilization (Arora & Gouda):** Versione più flessibile, che non richiede di partire da qualsiasi stato arbitrario, ma solo da **alcuni stati di partenza accettabili**.

**Q:** rappresenta uno **stato iniziale valido**.

**P:** rappresenta lo **stato corretto**.

Il sistema  $S$  **stabilizza**  $Q$  in  $P$ ,  $Q \rightarrow P$  se valgono le proprietà closure e convergence, ma con  $Q$  come condizione iniziale.

**Closure:** Una volta raggiunto  $P$ , il sistema **rimane in**  $P$  per sempre.

$$P(s) \text{ e } s \xrightarrow{S} s' \Rightarrow P(s')$$

**Convergence:** Non da qualsiasi stato, bensì da **qualsiasi stato che soddisfa**  $Q$ , il sistema **raggiungerà**  $P$  in un numero finito di passi.

**Es:  $P$ :** tutte le lampadine hanno lo stesso stato, cioè: (tutte accese) oppure (tutte spente). Questo è lo **stato desiderato** (sistema coerente).

**Q:** tutte le lampadine sono collegate alla rete elettrica e possono comunicare tra loro.

Questo è lo **stato iniziale accettabile**: il sistema può funzionare, ma non è detto che le lampadine siano ancora sincronizzate.

Ogni lampadina osserva le altre: se trova che qualcuna è in uno stato diverso, cambia il proprio stato per **adeguarsi alla maggioranza**. Se tutte sono uguali, **mantiene lo stato**.

**Convergence (da Q a P):** Se partiamo da uno stato che **soddisfa**  $Q$  (cioè tutte le lampadine sono connesse e possono comunicare), allora **entro un numero finito di passi** il sistema raggiungerà  $P$ : tutte le lampadine finiranno per avere lo stesso stato (tutte accese o tutte spente).

$$\Rightarrow Q(s) \Rightarrow \exists s': P(s') \text{ entro un numero finito di passi}$$

Es: Stato iniziale: A=on, B=off, C=off (ma tutte comunicano, quindi  $Q$  è vero). Dopo qualche passo, si allineano  $\rightarrow A=off, B=off, C=off \Rightarrow P$  **soddisfatto**.

**Closure** Una volta che tutte le lampadine sono sincronizzate (cioè  $P$  è vero), nessuna regola del sistema può rompere questa condizione. Infatti, se tutte sono uguali, **nessuna cambia più stato da sola**.  $P(s) \text{ e } s \xrightarrow{S} s' \Rightarrow P(s')$

**Quindi il sistema “stabilizza”  $Q$  in  $P$ :**  $Q \rightarrow P$ . Se le lampadine **sono tutte connesse** ( $Q$ ), allora il sistema **garantisce** che dopo un po' saranno **tutte sincronizzate** ( $P$ ), e una volta sincronizzate **resteranno così** (closure).

**Transient Failure:** Errore **temporaneo**, che cambia **lo stato attuale** del sistema. Se il sistema è **self-stabilizzante** tornerà **da lì da solo** a uno stato **legittimo**.

**Reachable Set:** Insieme di tutti gli **stati che il programma può effettivamente raggiungere** partendo dagli **stati iniziali legittimi**.

Se il sistema si trova in uno stato appartenente al reachable set e fa una transizione, anche lo stato successivo **appartiene al reachable set**.  $s \in R \wedge s \xrightarrow{S} s' \Rightarrow s' \in R$

**Randomizzazione:** Ogni processo, con una certa probabilità, prende decisioni diverse, in modo da rompere la **simmetria** naturalmente.

**Randomized self-stabilization:** Sistema self-stabilizzante in cui **il numero medio di round** necessari per raggiungere uno stato corretto è inferiore a una **costante**  $k$ .

**Progettazione degli algoritmi self stabilizzanti**

**Numero di stati:** Ogni nodo è una **macchina a stati finiti**. Più stati ha, più memoria serve, quindi dobbiamo ridurre **al minimo il numero di stati** mantenendo comunque la capacità di auto-stabilizzarsi.

**Central Daemon:** Scheduler che decide **quale processo** farà la prossima mossa quando più processi sono pronti per eseguire una mossa.

**Distributed daemon:** Ogni macchina **decide autonomamente** se fare una mossa, in base al proprio stato e ai vicini.

**Algoritmo distribuito uniforme:** Tutti i nodi eseguono **lo stesso codice**. Nei sistemi auto-stabilizzanti serve almeno **una macchina “eccezionale”**, che **segue regole**

**diverse per rompere la simmetria**, altrimenti il sistema potrebbe restare “bloccato” in uno stato perfettamente simmetrico e non evolvere mai.

## Sistema di Dijkstra

**Struttura del sistema:** Il sistema è formato da **n macchine a stati finiti**, disposte in **forma di anello (ring)**, ognuna collegata alla successiva, e l’ultima collegata alla prima. Ogni macchina può vedere solo lo stato dei suoi vicini, non quello dell’intero sistema. Una macchina ha il **privilegio** quando può **cambiare** il proprio **stato**, secondo una regola predefinita. Es: “Se il mio stato è diverso da quello del mio vicino, allora cambio il mio stato per allinearmi.”

È possibile che **più macchine** abbiano il privilegio contemporaneamente. In tal caso il **central daemon**, che osserva tutti i nodi, sceglie **arbitrariamente** quale macchina privilegiata eseguirà la prossima mossa.

Dijkstra definisce un **legitimate state** come una **configurazione del sistema** che soddisfa le seguenti proprietà.

**Liveness:** Deve esserci **almeno un privilegio** attivo nel sistema (almeno una macchina deve poter fare una mossa).

**Closure:** Se il sistema si trova in uno stato legittimo e una macchina fa una mossa, il nuovo stato deve essere **ancora legittimo**.

**No starvation:** Ogni macchina deve, prima o poi, ottenere il privilegio un numero infinito di volte.

**Reachability:** Da qualunque stato legittimo, si può arrivare a qualsiasi altro stato legittimo con una sequenza di mosse.

**Stato legittimo:** Stato in cui **esattamente una macchina ha il privilegio**.

**First solution Dijkstra Algorithm:** Abbiamo un **anello di n macchine**, numerate da 0 a  $n-1$ . Ciascuna macchina ha uno **stato interno** scelto da un insieme finito di valori  $K = \{0, 1, 2, \dots, k-1\}$ . L’obiettivo del sistema è **stabilizzarsi** in una configurazione particolare, partendo anche da uno stato casuale.

**Comportamento del sistema:** Tutte le macchine **tranne la macchina 0** eseguono lo stesso algoritmo. Ogni macchina confronta **il proprio stato (S)** con **lo stato del vicino a sinistra (L)**.

**Se  $S = L$ :** La macchina **non fa nulla**, a meno che **non sia la macchina 0**.

**Se  $S \neq L$ :** La macchina esegue  $S := L$ , copiando lo stato del vicino sx.

**Macchina 0:** Quando il suo stato è **uguale** a quello del vicino a sinistra, **incrementa il proprio stato:**  $S := (S + 1) \bmod K$ , in modo da rompere la simmetria.

Pseudocodice

Per la macchina 0 (eccezionale):

If  $S == L$  then

$S = (S + 1) \bmod K$

Per le altre macchine:

If  $S \neq L$  then

$S = L$

Un **central demon** sceglie ad ogni passo, **una macchina che è privilegiata** e le consente di eseguire la sua regola.

Tutte le macchine tendono a **rendere uguale il proprio stato a quello del vicino sinistro**, quindi col tempo tutti si “allineano”. **La macchina 0** ogni tanto cambia stato, mantenendo in movimento il token.

**Es Prof:** Supponiamo che **la macchina 6** sia la prima a fare una mossa. Il suo stato è **diverso** da quello della **macchina 5**, quindi la macchina 6 è **privilegiata**. Dopo aver eseguito la mossa, **la macchina 6 copia lo stato della macchina 5**, rendendolo uguale. A questo punto, la macchina 6 **perde il suo privilegio**. Supponiamo che **la macchina 7** abbia uno stato diverso da quello della macchina 6; allora il **demon** le assegna il privilegio. La macchina 7, **copia lo stato della macchina 6**. Di conseguenza, **le macchine 5, 6 e 7 hanno ora lo stesso stato**.

Alla fine, tutte le macchine si comportano allo stesso modo, quindi **progressivamente tutti gli stati diventano uguali**. Quando succede questo: **solo la macchina 0** sarà privilegiata, perché la sua condizione  $L = S$  è vera.

La macchina 0 esegue la sua regola:  $S := (S + 1) \bmod K$ , incrementando lo stato di 1. Ora la macchina 1 diventa privilegiata, e ricomincia il ciclo. In questo modo, il **token** circola attorno all'anello, passando da una macchina alla successiva. Il sistema rimane stabile: c'è sempre una sola macchina privilegiata alla volta, il comportamento è ciclico e controllato.

### Soluzione 3 Stati

Nell'anello di  $n$  macchine, ognuna ha uno stato  $S \in \{0, 1, 2\}$ . Abbiamo **due** macchine eccezionali, la **macchina 0** e la **macchina n-1**. Tutte le altre  $1 \leq i \leq n - 2$  seguono la stessa regola comune.

#### Bottom machine (macchina 0)

If  $R == S+1 \bmod 3$

then  $S = S - 1 \bmod 3$

La macchina 0 guarda solo il proprio stato  $S$  e quello del **vicino destro**  $R$ . Se lo stato di  $R$  è **un'unità avanti** rispetto al suo, allora **decrementa il proprio stato** di 1 ( $\bmod 3$ ).

1. If  $s=0$  and  $r=1$ , then the state of  $s$  is changed to 2.
2. If  $s=1$  and  $r=2$ , then the state of  $s$  is changed to 0.
3. If  $s=2$  and  $r=0$ , then the state of  $s$  is changed to 1.

#### Top machine (macchina n-1)

If  $L == R$  and  $((S \neq L+1 \bmod 3)$

then  $S = L + 1 \bmod 3$

La macchina  $n-1$  guarda **entrambi** i vicini, sx  $L$  e dx  $R$ . Se i due vicini sono **nello stesso stato**, e il proprio stato  $S$  **non è** quello successivo a  $L$ , allora **aggiorna  $S$  a  $L + 1 \bmod 3$** .

$L = R$	$(L+1) \bmod 3$	Possibili $S$	Nuovo $S$
0	1	[0, 2]	1
1	2	[0, 1]	2
2	0	[1, 2]	0

#### Le altre macchine ( $1 \leq i \leq n-2$ )

If  $L == (S+1) \bmod 3$

then  $S := L$

If  $R == (S+1) \bmod 3$

then  $S := R$

Se il vicino sx  $L$  è un passo avanti al proprio stato  $S$ , copiano  $L$ . Altrimenti controllano il vicino dx  $R$  e fanno lo stesso.

1. If  $s=0$  and  $L=1$ , then  $s=1$ .
2. If  $s=1$  and  $L=2$ , then  $s=2$ .
3. If  $s=2$  and  $L=0$ , then  $s=1$ .

Le macchine intermedie si adeguano gradualmente ai loro vicini. Quando tutti gli stati tendono a uniformarsi, entra in gioco la **macchina 0**, che modifica il proprio stato creando un disallineamento, che si propaga lungo l'anello. La **macchina n-1** reagisce in modo complementare, chiudendo il ciclo e assicurando che il token continui a circolare.

The **bottom machine**, machine 0:

```
If (S+1) mod 3 = R
then
  S := (S-1) mod 3
```

The **top machine**, machine  $n-1$ :

```
If L = R and (L+1) mod 3 ≠ S
then
  S := (L+1) mod 3
```

The **other machines**:

```
If (S+1) mod 3 = L
then
  S := L
```

```
If (S+1) mod 3 = R
then
  S := R
```

State of machine 0	State of machine 1	State of machine 2	State of machine 3	Privileged machines	Machine to make move
0	1	0	2	0, 2, 3	0
2	1	0	2	1, 2	1
2	2	0	2	1	1
2	0	0	2	0	0
1	0	0	2	1	1
1	1	0	2	2	2
1	1	1	2	2	2
1	1	2	2	1	1
1	2	2	2	0	0
0	2	2	2	1	1
0	0	2	2	2	2
0	0	0	2	3	3
0	0	0	1	2	2

**Costs of Self-Stabilization:** Non c'è alcun **limite superiore** al numero di **transizioni** necessarie perché il sistema passi da uno **stato non sicuro** a uno **stato sicuro**. Questo significa che, in teoria, il sistema può impiegare anche **molto tempo** prima di stabilizzarsi.

**Convergence Span:** Numero **massimo** di transizioni che il sistema può eseguire, partendo da uno stato **potenzialmente errato**, prima di raggiungere **uno stato sicuro**.  
**Response Span:** Numero **massimo** di transizioni necessarie per passare da **uno stato iniziale specifico** a **uno stato obiettivo**.

**Progettazione sistema:** Dividere il sistema in componenti più piccoli, ciascuno dei quali: è **auto-stabilizzante** da solo, e può essere **integrato** con gli altri per ottenere un sistema completo e stabile.

**Layering:** Ogni **strato software** svolge una funzione specifica, e ognuno deve essere **self-stabilizing** indipendentemente. Una volta che **uno strato è stabilizzato**, lo strato superiore che lo usa **si stabilizza automaticamente**. Se  $P \rightarrow Q$  e  $Q \rightarrow R$ , allora  $P \rightarrow R$ .  
**Modularization:** La **modularizzazione** consiste nel suddividere il sistema in **componenti indipendenti**, ognuno progettato per essere **self-stabilizing in modo autonomo**. I moduli vengono poi **combinati** creando complessivamente un sistema **self-stabilizing**.

## Lesson 10 – Consensus

**Consensus:** Raggiungere un accordo su un valore comune tra più nodi di un sistema distribuito, anche se alcuni nodi possono fallire. Il consenso serve a garantire che **tutti i nodi concordino sulla stessa versione dei dati**. Serve anche a scegliere un **leader** tra più nodi.

**Maggioranza:** Un nodo propone un **valore**. Gli altri nodi ricevono la proposta e **votano** per accettarla. Se **più della metà dei nodi** accetta la proposta, il valore viene **scelto definitivamente** come decisione comune del sistema.

**Leader-Based Coordination:** Il leader **propone i valori** su cui gli altri dovranno accordarsi. Gli altri nodi **accettano o rifiutano** le proposte. Se la **maggioranza approva**, il leader **conferma la decisione** e la comunica a tutti.

**Voting and Quorums:** Un numero minimo di nodi devono partecipare e concordare per prendere una decisione. Ogni nodo **vota** su una proposta. Il consenso è raggiunto solo se si ottiene un **quorum**, cioè **una maggioranza (51%) che si sovrappone** a qualsiasi altra maggioranza possibile. Questa **sovraposizione** garantisce che due decisioni diverse non possano coesistere: almeno **un nodo in comune** tra due quorum saprà sempre quale valore è stato scelto.

**Esempio:** Se servono 3 voti su 5 per decidere: un quorum potrebbe essere {A, B, C}, un altro {B, C, D}.

**Timeout:** Se un nodo non risponde entro X secondi, si assume che sia inattivo e si procede.

**Majority Voting:** Basta la **maggioranza dei nodi attivi**, non tutti, per continuare.

**Safety:** Tutti devono (eventualmente) **concordare sullo stesso valore**, e una volta che un valore è stato deciso, **non può essere cambiato**.

**Liveness:** Il sistema **prima o poi deve fare progressi**, cioè prendere una decisione..

Anche se alcuni nodi falliscono, i nodi che funzionano devono comunque riuscire a **decidere un valore**.

**Se scegli di privilegiare la Safety:** Il sistema può bloccarsi (nessuna decisione) pur di non commettere errori.

**Se scegli di privilegiare la Liveness:** Il sistema può “decidere in fretta” ma rischiare decisioni incoerenti.

**Teorema di impossibilità FLP:** In un sistema asincrono, anche con la sola possibilità che **un nodo sia guasto**, è **impossibile garantire simultaneamente** sia la **safety** sia la **liveness**.

Si può avere **sicurezza (nessuna decisione sbagliata)** oppure avere **progresso (il sistema decide sempre qualcosa)**, ma **non entrambe le cose insieme** in ogni caso possibile.

Non ci sono limiti sui tempi di consegna dei messaggi o sulla velocità di esecuzione dei nodi. Un messaggio può arrivare in 1 ms o in 1 ora. Un nodo non può sapere se un altro nodo è semplicemente lento o morto. Se decide troppo presto, rischia che arrivi un messaggio “tardivo” che cambia tutto: **safety violata**. Se invece aspetta per essere sicuro: **liveness violata** (il sistema non progredisce mai). Quindi, **non esiste un algoritmo deterministico** che riesca a garantire entrambi in ogni caso possibile.

## Paxos

**Safety based:** Garantire la safety, ovvero anche se qualcuno entra o esce, il sistema deve **continuare a prendere decisioni corrette**.

**Assunzioni di Paxos:** I processori possono funzionare a velocità diverse, bloccarsi, riavviarsi o smettere di rispondere.

Se un nodo si guasta e poi riparte, può **recuperare il suo stato** da una **memoria stabile HDD**. Quando torna operativo, **non mente e non si comporta in modo malevolo**.

Ogni nodo può comunicare direttamente con ogni altro nodo del sistema, senza restrizioni topologiche. I messaggi **possono arrivare in ritardo e possono perdere** completamente. Quando un messaggio **arriva**, il suo contenuto **non è mai corrotto**.

**Obiettivo:** Far sì che **più nodi** raggiungano un **accordo su un unico valore** anche se alcuni nodi falliscono.

## RUOLI PRINCIPALI

**Proposer:** Propone un valore da accettare.

**Acceptor:** Decide se accettare o rifiutare una proposta, seguendo regole precise.

**Learner:** Una volta che un valore è stato accettato dalla maggioranza, lo apprende e lo considera deciso.

## MESSAGGI PRINCIPALI DI PAXOS

Durante il protocollo, i nodi si scambiano **messaggi** con significati precisi.

**Prepare(n):** “Sto per proporre con numero di proposta n; promettimi di non accettare proposte più vecchie.”

**Promise(n, v):** “Prometto di non accettare proposte con numero < n; ecco il valore e numero dell’ultima proposta accettata.”

**Accept(n, v):** “Accetta il valore v associato al numero di proposta n.”

**Accepted(n, v):** “Ho accettato il valore v per la proposta n.”

Ogni **esecuzione** (o *istanza*) di Paxos serve a scegliere un solo valore. L’intero protocollo si divide in **round**, ognuno condotto da un *Proposer*. Un **round** riuscito ha **due fasi principali**, ognuna con due sottofasi (a e b).

**PHASE 1 — Preparation:** Stabilire un canale di fiducia tra il *Proposer* e una maggioranza di *Acceptors*, in modo che il *Proposer* possa poi proporre un valore “sicuro”.

**Phase 1a — Prepare:** Il **Proposer** sceglie un **numero di proposta univoco n**. Questo numero serve per ordinare le proposte nel tempo. Sceglie almeno una **maggioranza (quorum)** di **Acceptors**. Invia a tutti loro un messaggio **Prepare(n)**.

**Messaggio:** “Sto iniziando una proposta numero 42. Vi chiedo di promettere che non accetterete nessuna proposta con numero inferiore a 42.”

**Phase 1b — Promise:** Quando gli **Acceptor** ricevono un **Prepare(n)**, verificano se n è **maggiore di tutte le proposte precedenti** che conoscono. Se sì, rispondono con **Promise(n)**. Se no, ignorano o rispondono negativamente.

**Messaggio:** La promessa: “Non accetterò proposte con numero < n. Ho già accettato in passato la numero x”.

#### Esempio intuitivo

Immagina che tre giudici (**Acceptors**) votino leggi proposte da deputati (**Proposers**).  
Il deputato A manda: “Vorrei proporre la legge n.42 — promettete di non accettare proposte più vecchie di questa?”  
Ogni giudice risponde: “Promesso — non voterò proposte con numero < 42. Ma ti dico che in passato avevo già votato la n.40 per la legge X.” Così il proponente può capire se c’è già una proposta “in corso”.

**PHASE 2 — Acceptance:** Il **Proposer** procede a proporre (e far accettare) un valore v.

**Phase 2a — Accept Request:** Il **Proposer** analizza le risposte **Promise** ricevute. Se almeno un **Acceptor** aveva già accettato un valore **v\_prev**, allora il **Proposer deve riproporre quel valore**. Se nessuno aveva accettato nulla prima, può proporre un valore nuovo a piacere v. Invia agli **Acceptors** un messaggio **Accept(n, v)**.

**Phase 2b — Accepted:** Gli **Acceptor** quando ricevono un **Accept(n, v)**, controllano se **non hanno già promesso** di accettare solo proposte con numero maggiore di n; se tutto è valido, **accettano** il valore v e lo registrano localmente. Inviano **Accepted(n, v)** ai **Learner** e al **Proposer**.

**Consenso:** Quando una **maggioranza di Acceptors** (es. 3 su 5) invia **Accepted(n, v)** per lo stesso valore v, quel valore è **ufficialmente deciso**. A questo punto i **Learner** lo apprendono, e Paxos termina per quella istanza.

Acceptors: A1, A2, A3, A4, A5  
Maggioranza: 3

1. Proposer vuole il valore X  
P sceglie n = 1
2. Prepare  
 $P \rightarrow \text{Prepare}(n) \text{ a } A1, A2, A3$
3. Promise  
 $A1 = \text{Promise}(1), \text{ nessun valore accettato}$   
 $A2 = \text{Promise}(1), \text{ nessun valore accettato}$   
 $A3 = \text{Promise}(1), \text{ nessun valore accettato}$
4. Scelta del valore  
Nessun Acceptor ha accettato nulla  
P sceglie liberamente X
5. Accept  
 $P \rightarrow \text{Accept}(n, X) \text{ a } A1, A2, A3$
6. Accepted  
 $A1 = \text{Accepted}(1, X)$   
 $A2 = \text{Accepted}(1, X)$   
 $A3 = \text{Accepted}(1, X)$
7. Consenso  
3 Acceptors hanno Accepted(1, X)  
Valore deciso: X

Esempio successivo (dopo che X è già stato deciso)

Acceptors: A1, A2, A3, A4, A5  
Maggioranza: 3  
Valore già deciso in passato X

1. Nuovo proposer vuole il valore Y  
P sceglie n = 2
2. Prepare  
 $P \rightarrow \text{Prepare}(n) \text{ a } A3, A4, A5$
3. Promise  
 $A3 = \text{Promise}(2), \text{ valore accettato X}$   
 $A4 = \text{Promise}(2), \text{ valore accettato X}$   
 $A5 = \text{Promise}(2), \text{ nessun valore accettato}$
4. Scelta del valore  
P sceglie Y
5. Accept  
 $P \rightarrow \text{Accept}(n, Y) \text{ a } A2, A3, A4, A5$
6. Accepted  
 $A2 = \text{Accepted}(2, Y)$   
 $A3 = \text{Accepted}(2, Y)$   
 $A4 = \text{Accepted}(2, Y)$   
 $A5 = \text{Accepted}(2, Y)$
7. Consenso  
3 Acceptors hanno Accepted(2, Y)  
Valore deciso (di nuovo): X

## Lezione 12: Raft

**Funzionamento Raft:** Raft raggiunge il consenso grazie a **un leader eletto**. Ogni nodo può essere in uno di tre stati.

**Follower:** Ascolta e accetta comandi dal leader.

**Candidate:** Si propone per diventare leader.

**Leader:** Coordina il gruppo e replica i comandi sugli altri nodi.

**Ciclo di vita dei nodi:** All’inizio tutti i nodi sono follower. Rimangono in attesa di “heartbeat” dal leader. Il leader, periodicamente, invia un messaggio detto **heartbeat** a tutti i follower per dire: “Ehi, sono ancora vivo, non serve eleggere un nuovo leader!”. Se un follower **non riceve heartbeat entro un certo tempo**, pensa che il leader sia morto. In tal caso il follower diventa **candidate** e avvia una **leader election**. Manda messaggi di richiesta voto agli altri nodi. Se ottiene la maggioranza dei voti, diventa **leader**. Da quel momento, coordina tutto il cluster e gestisce la replica del log.

**Log Replication:** Ogni nodo può ricevere richieste dal client in momenti diversi. I nodi potrebbero eseguire operazioni in ordini diversi, portando a risultati incoerenti.

Es: Nodo A riceve la richiesta “aggiungi 5” prima di “moltiplica per 2”. Nodo B riceve “moltiplica per 2” poi “aggiungi 5”. Il risultato finale è diverso.

Per risolvere questo tutte le operazioni vengono scritte in un **log ordinato** dal leader, e questo log viene **replicato su tutti i follower**. Tutti i nodi applicano le operazioni nella **stessa sequenza**, garantendo uno **stato finale identico**.

**Solo il leader può aggiungere nuove entries** al log. Il leader replica ogni nuova entry a tutti i follower. Un'entry è “**committed**” solo quando è stata replicata su una **maggioranza di nodi** (3 nodi su 5). Tutti i nodi applicano le entry nello stesso ordine alla propria *state machine* (cioè al loro database o stato interno). Il risultato è che tutti i nodi processano la **stessa sequenza di comandi**, ottenendo lo **stesso stato finale**.

Es: N1(leader), N2, N3, N4, N5. Il client invia al leader il comando: “Add balance +100”  
Il leader aggiunge questa entry al proprio log: [1: +100]  
Poi invia il messaggio di replica ai follower.  
Quando almeno 3 nodi (la maggioranza) confermano di averla ricevuta, il leader marca la entry come **committed**.  
A quel punto, tutti (leader + follower) applicano l'operazione alla loro *state machine*.  
Tutti ora sanno che il saldo è +100.

**Fallimento leader:** Se il **leader fallisce**, i follower non ricevono più heartbeat. Scatta il **timeout**, quindi uno dei follower diventa **candidate** e avvia un'**elezione**. Dopo la votazione, viene eletto **un nuovo leader**. Quando un nuovo leader viene eletto, prima di accettare nuove richieste dai client, **controlla il proprio log** rispetto a quello degli altri nodi. Se si accorge che il suo log è **incompleto**, **richiede le entry mancanti** dagli altri nodi, in modo da ricostruire il log completo.

Es: Prima del crash:  
Leader (vecchio): [1, 2, 3, 4, 5] Follower A: [1, 2, 3, 4] Follower B: [1, 2, 3, 4, 5] Follower C: [1, 2, 3, 4, 5]  
Il leader fallisce. Viene eletto **Follower B** come nuovo leader.  
B confronta i log e vede che A è indietro di un'entry. B invia ad A il messaggio AppendEntries con la entry mancante (5).  
Dopo il recupero: Tutti i nodi: [1, 2, 3, 4, 5]

**Batching delle richieste:** Il leader può **raggruppare (batch)** più comandi dei client in una **singola entry del log**.

**Piggybacking con gli Heartbeat:** Raft sfrutta gli heartbeat per trasportare **free** anche le nuove entry del log, in modo da risparmiare messaggi.

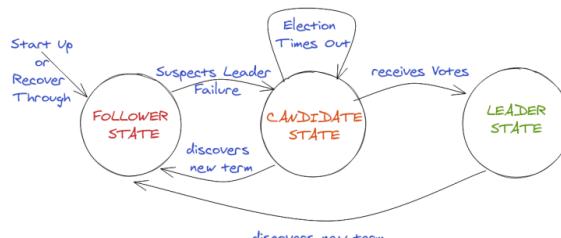
## Funzionamento Raft

### Step 1

**Term number:** Ogni nodo in Raft mantiene un **term**, che è un **orologio logico** che **aumenta** ogni volta che inizia una nuova elezione. Ci può essere **al massimo un leader valido per term**.

**Follower State:** Quando un nodo si avvia o si riprende da un crash, inizia come **Follower**. Il follower aspetta di ricevere messaggi (heartbeat o AppendEntries) dal leader corrente. Se passa troppo tempo senza, il follower sospetta che il leader sia caduto.

**Candidate State:** Quando un nodo diventa candidate: Incrementa il suo currentTerm locale. Si vota da solo (ogni nodo può votare solo una volta per term). Invia un messaggio RequestVote a tutti gli altri nodi nel cluster.



**State diagram**

### Step 2

**RequestVote:** Il RequestVote RPC contiene il **term number** del candidato, il suo **ultimo indice e term del log**, e informazioni sulla sua **idoneità a diventare leader**.

**Voting Process:** Quando un **Follower** riceve un **RequestVote**, **controlla il term number**. Se il **term** del candidato è **più alto** del proprio, aggiorna il proprio **term** (riconosce che è iniziata una nuova elezione), resetta il timer, e **può votare** per il candidato. Se invece il candidato ha un **term più basso o uguale**, il follower **rifiuta il voto** (significa che è un candidato “vecchio”).

Se il candidato ottiene **la maggioranza dei voti**, diventa **Leader** per quel **term**. Se non ottiene abbastanza voti, il candidato **torna follower**.

**Leader State:** Quando un nodo diventa leader, può iniziare a **inviare messaggi AppendEntries** ai follower per replicare le entry del log. Gestisce tutte le richieste dei client: aggiunge nuove entry al log, le replica ai follower, e conferma il **commit** quando la maggioranza le riceve.

**Heartbeats:** Oltre alla replica, il leader manda regolarmente **AppendEntries “vuoti”** (heartbeat), che servono solo per dire “sono vivo”, mantengono l’autorità del leader, e impediscono ai follower di avviare nuove elezioni.

## Lesson 13 – Consistency Models: CAP Theorem

**Consistency:** Tutti i nodi vedono sempre gli **stessi dati** nello **stesso momento**, nella loro **versione più aggiornata**. Dopo un’operazione di scrittura, **ogni lettura successiva**, da qualunque nodo, restituisce **l’ultimo valore scritto**.

**Availability:** Ogni richiesta fatta da un client (lettura/scrittura) riceve sempre **una risposta valida**, anche se alcuni nodi sono guasti. **Il servizio non si blocca mai e fornisce qualche risposta, anche “vecchia”**.

**Partition Tolerance:** Il sistema **continua a funzionare correttamente anche quando la rete si guasta e si divide in più parti non comunicanti**. Ogni parte continua a funzionare in locale, anche se i nodi non riescono a comunicare fra loro.

**CAP Theorem:** In un sistema distribuito è impossibile garantire simultaneamente **Consistency, Availability e Partition Tolerance**. Possono essere garantite **due** di queste **tre proprietà**, ma non **tutte e tre** nello stesso momento.

**Caso normale:** Il sistema garantisce **coerenza dei dati** (Consistency) e **risposte sempre disponibili** (Availability), ma **solo finché non avviene una partizione di rete**.

**Availability + Partition Tolerance:** Il sistema continua a rispondere a tutte le richieste (**Availability**) anche in presenza di partizioni di rete (**Partition Tolerance**), ma non può garantire che tutti i nodi vedano gli stessi dati in tempo reale (**no Consistency**).

Quando la rete si divide, ogni parte continua a funzionare **indipendentemente**, anche se i dati diventano temporaneamente incoerenti. Quando la rete si ripristina, il sistema **riconcilia i dati**.

**Consistency + Partition Tolerance:** Il sistema mantiene dati coerenti (**Consistency**) anche durante le partizioni di rete (**Partition Tolerance**), ma **sacrifica la disponibilità (Availability)**. Se non può garantire consistenza, **rifiuta** o ritarda le richieste. Preferisce **non rispondere** piuttosto che dare una risposta incoerente.

## Dimostrazione CAP Theorem

**Ipotesi iniziali:** Consideriamo un sistema distribuito con dati replicati su più nodi. I nodi comunicano tramite una rete non affidabile, quindi **una partizione di rete può verificarsi**. I client possono inviare richieste a qualsiasi nodo.

**Effetto della partizione:** Quando si verifica una partizione di rete, esistono almeno due nodi che non possono comunicare tra loro, ma continuano a ricevere richieste dai client. La partizione è un fatto esterno al sistema e **non può essere evitata**.

**Decisione inevitabile:** In presenza di una partizione, ogni nodo che riceve una richiesta ha solo due possibilità. **Rispondere immediatamente**, usando lo stato locale.

**Rimandare o rifiutare la risposta**, in attesa di informazioni dagli altri nodi.

**Rispondere implica perdita di Consistency:** Se il nodo risponde immediatamente, garantisce che ogni richiesta ottenga una risposta (**Availability**), ma non può sapere se altri nodi hanno uno stato più aggiornato, quindi la risposta può non riflettere uno stato globale unico, violando la **Consistency**.

**Non rispondere implica perdita di Availability:** Se il nodo aspetta di essere sicuro della correttezza globale, evita di fornire dati potenzialmente incoerenti (**Consistency mantenuta**), ma alcune richieste restano senza risposta finché dura la partizione, quindi il sistema non risponde sempre (**Availability violata**).

**Incompatibilità logica:** Durante una partizione, garantire **Availability** implica rinunciare a **Consistency**, garantire **Consistency** implica rinunciare a **Availability**. Poiché la partizione è assunta possibile, **C e A non possono essere garantite simultaneamente**.

**“2 su 3” fuorviante:** In un sistema davvero distribuito, la tolleranza alle partizioni non è una scelta, è una necessità. Quando **non ci sono partizioni**, puoi avere **C + A + P** tutte insieme. Quando **si verifica una partizione**, devi temporaneamente scegliere se mantenere **C o A**.

## Gradi di Consistency in caso di privilegio Availability

Se privilegiamo la availability si può garantire **consistenza forte**, ma possiamo garantire diversi gradi di consistenza.

**Strong Consistency:** Tutti gli utenti vedono gli stessi dati nello stesso momento.

Subito dopo un aggiornamento, **qualsiasi lettura** (da qualunque nodo) restituisce l'ultimo valore scritto.

**Weak Consistency:** Il sistema **non garantisce che le operazioni di lettura riflettano immediatamente le ultime operazioni di scrittura**. (La coerenza viene garantita dopo una sincronizzazione, ma non in ogni istante).

**Eventual Consistency:** In assenza di nuove operazioni di **scrittura**, tutte le repliche convergono col tempo allo stesso valore. Non è garantita la consistenza immediata ma la convergenza finale.

## Varianti Eventual Consistency

**Sequential Consistency:** Le operazioni sono eseguite in modo tale che **l'ordine delle operazioni di ciascun processo è preservato**, ma l'ordine globale non deve necessariamente coincidere con il tempo reale.

Variabile condivisa: `x = 0`

Perché è Sequentially Consistent

Esiste un ordine globale delle operazioni che spiega il risultato, ad esempio:

Processo P1

1. `write(x = 1)`
2. `write(x = 2)`

Processo P2

1. `read(x)` → ritorna 0
2. `read(x)` → ritorna 2

lua

```
read(x)=0  (P2)
write(x)=1 (P1)
write(x)=2 (P1)
read(x)=2  (P2)
```

- L'ordine delle operazioni di P1 è rispettato (1 prima di 2)
- L'ordine delle operazioni di P2 è rispettato
- Tutte le operazioni possono essere viste come eseguite in un'unica sequenza globale

**Causal Consistency:** Se due operazioni sono **legate da una relazione causale**, tutti i processi le vedranno **nello stesso ordine**. Le operazioni **non correlate** possono invece essere viste in **ordini diversi** su nodi diversi. (Se un'operazione *a* **causa** un'operazione *b*, allora **tutti i nodi devono osservare *a* prima di *b***).

*Es:* Se un utente scrive un post e un altro commenta quel post, tutti vedranno prima il post e poi il commento.

**Read-your-write Consistency:** Dopo aver aggiornato un dato, un processo non leggerà mai una **versione più vecchia** di quel dato da lui aggiornato.

*Es:* Se aggiorni la tua foto profilo, la prossima volta che la visualizzi **vedi subito la nuova**, non quella vecchia.

**Session Consistency:** Durante una **stessa sessione di lavoro**, il sistema garantisce la **read-your-write consistency**. Quando la sessione termina, queste garanzie **non valgono più**.

*Es:* Finché resti loggato su un'app, vedi sempre i tuoi ultimi aggiornamenti; se esci e rientri, potresti vederli solo dopo che il sistema si è completamente sincronizzato.

**Monotonic Read Consistency:** Se un processo ha letto un certo valore di un dato, **non leggerà mai in seguito un valore più vecchio**.

*Es:* Se leggi che il saldo del tuo conto è 100 €, in letture successive **non potrai mai vedere 90 €**, anche se i dati non sono ancora del tutto sincronizzati tra i nodi.

**Monotonic Write Consistency:** Le scritture effettuate dallo **stesso processo** vengono applicate nell'ordine in cui sono state eseguite.

*Es:* Se prima scrivi  $x = 1$  e poi  $x = 2$ , nessun nodo del sistema potrà vedere  $x = 2$  e poi tornare a  $x = 1$ .

Queste proprietà possono essere **combinata** tra loro e con altre. Nella pratica, la combinazione **monotonic reads + read-your-writes** è la **più desiderabile**.

**Heterogeneity:** Non tutti i componenti di un sistema distribuito hanno gli **stessi requisiti di consistenza o disponibilità**. E' meglio **segmentare** il sistema in parti diverse, ognuna con **proprietà di C e A diverse**, in base a ciò di cui ha bisogno.

**Data Partitioning:** Dati diversi possono avere **diversi requisiti di consistenza e disponibilità**.

**Carrello acquisti (shopping cart):** alta disponibilità → l'utente deve poter aggiungere articoli velocemente, anche se il sistema non è perfettamente sincronizzato.

**Informazioni sui prodotti:** devono essere molto accessibili (A), ma leggere variazioni di inventario non sono un problema.

**Pagamento, fatturazione e spedizione:** qui serve **consistenza forte (C)**, perché ogni transazione deve essere corretta e unica.

**Hierarchical Partitioning:** Nei grandi servizi globali, l'architettura è **organizzata a livelli**: i **server locali** garantiscono maggiore consistenza e disponibilità, mentre i **server globali** gestiscono la sincronizzazione e accettano un po' di ritardo o incoerenza.

**No Partizioni:** Anche se tutto funziona perfettamente, c'è comunque un **tempo di propagazione** dei dati tra i nodi. Fornire più **consistenza** significa che serve **più tempo** per aggiornare tutti i nodi, quindi **maggior latenza**. Fornire più **disponibilità** significa scegliere più **velocità (bassa latenza)**, ma alcuni nodi potrebbero rispondere **con dati non ancora aggiornati**.

**PACELC Theorem:** Se c'è una partizione P, devi scegliere tra **Availability (A)** e **Consistency (C)**. Se la rete è normale, devi scegliere tra **Latency (L)** e **Consistency (C)**.

**PA/EL Systems:** Sistemi che **rinunciano alla consistenza** sia durante una partizione (P) sia in condizioni normali (E), per privilegiare **alta disponibilità (A)** e **bassa latenza (L)**.

**PC/EC Systems:** Sistemi che mantengono la consistenza **C** sia in caso di partizioni **P** che in caso normale **E**. Preferiscono **bloccare o rallentare** le operazioni piuttosto che servire dati incoerenti.

**PA/EC Systems:** Questi sistemi mantengono **consistenza C** in condizioni normali **E**, ma la sacrificano per avere **Avalibility A** quando si verifica una **partizione di rete (P)**.

**PC/EL Systems:** Questi sistemi **mantengono la consistenza C** se avviene una partizione P, ma **la rilassano** (accettano dati leggermente vecchi) **per ridurre la latenza L** in condizioni normali E.

## Lezione 13 – Replication

**Replication:** Mantenere copie multiple degli stessi dati o servizi **su più nodi** della rete.

**Replica:** Ogni macchina che contiene una copia del database.

**Proprietà**

**Disponibilità dei dati:** Se un nodo fallisce, **il dato è comunque disponibile** su un'altra replica.

**Scalabilità:** Se arrivano **più utenti**, possiamo aggiungere **nuovi nodi** che ospitano copie del dato.

**Riduzione della latenza:** Replicando i dati in luoghi diversi nel mondo, l'utente riceve la risposta più velocemente dal nodo replica più vicino.

**Sfide**

**Replication Lag:** Tempo per propagare un aggiornamento da una replica all'altra.

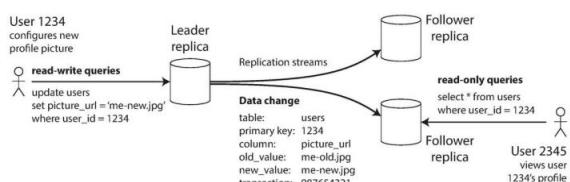
**Consistenza:** È necessario garantire che le copie rimangano **coerenti** tra loro.

**Risoluzione dei conflitti:** Se due repliche modificano la **stessa informazione in modo diverso**, bisogna decidere **qual è la versione valida**.

### Replicazione Single Leader

**Leader:** Unico nodo che accetta **operazioni di scrittura**, modifica i dati nel proprio storage e invia gli aggiornamenti alle repliche follower tramite **replication streams**.

**Followers:** Repliche che **non** possono **scrivere**, **ricevono** gli **aggiornamenti** dal leader e **copiano** gli stessi cambiamenti localmente.



**Synchronous Replication:** Il **leader** invia l'aggiornamento. **Aspetta** che il follower confermi di averlo **ricevuto e salvato**. Solo quando riceve la conferma, il leader risponde al client che la scrittura è andata a buon fine.

**Asynchronous Replication:** Il **leader** invia l'aggiornamento al follower **ma non aspetta** conferma. Risponde subito al client che la scrittura è conclusa. Per qualche istante, i dati **non sono uguali ovunque**.

**Sincronizzazione Pratica:** Tenere 1 follower sincrono per garantire almeno una copia sempre aggiornata, e il resto dei follower asincroni in modo da non rallentare le scritture. **Se il follower sincrono si rompe**, si sceglie uno dei follower asincroni e lo si promuove a **nuovo follower sincrono**.

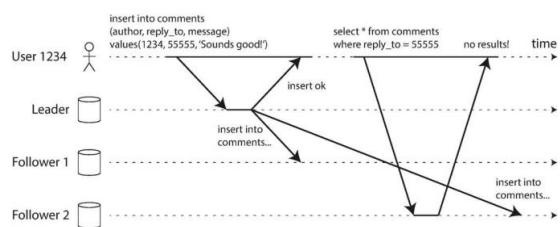
**Replication Lag:** Ritardo di tempo in cui il leader ha i dati nuovi, e i follower dati vecchi, nella sincronizzazione asincrona.

**Problemi con i vari tipi di consistenza**

**Eventual consistency:** Anche se in un certo momento le repliche non sono tutte allineate, **col tempo** i follower **raggiungeranno** lo stato del leader.

**Read-after-write Consistency:** Quando un utente **scrive un dato**, si aspetta di leggere immediatamente ciò che ha appena **scritto**. Può succedere che l'utente scrive un commento, che va al **leader**. Il leader aggiorna i suoi dati. I follower **non hanno ancora**

**aggiornato la loro copia.** Se l'utente ricarica la pagina e la lettura avviene da un follower, **non vede il commento.**

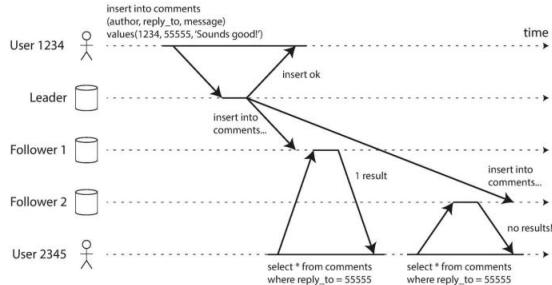


Per implementarla, dopo che un utente fa una scrittura, l'utente deve **leggere solo dal leader**, finchè non si è sicuri che i **follower** si siano **sincronizzati**.

**Monotonic Reads anomaly:** Quando usiamo repliche follower **asincrone**, non tutte le repliche sono aggiornate allo stesso momento. Alcune repliche possono essere **più aggiornate** e altre **più vecchie**. Un utente può vedere i dati **tornare indietro nel tempo**. L'utente fa una prima lettura, legge da una replica **aggiornata**. Subito dopo fa una seconda lettura, ma legge da una replica **in ritardo**. La seconda risposta è **più vecchia** della prima.

L'utente 2345 apre la pagina e vede **un commento nuovo** (perché legge da una replica aggiornata). Dopo qualche secondo ricarica la pagina, ma questa volta la richiesta viene servita da una replica che **non ha ancora ricevuto l'aggiornamento**.

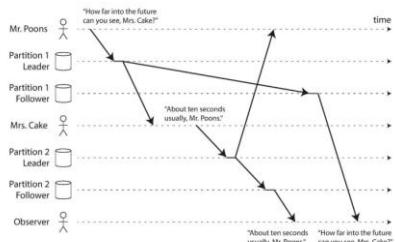
Il commento **sparisce**.



**Soluzione:** Assicurarsi che ogni utente legga sempre dalla stessa replica. Anche se è in ritardo, sarà sempre coerente con se stessa.

**Consistent Prefix Reads Anomaly:** Le **scritture** avvengono in un certo **ordine logico**, ma alcune repliche mostrano i dati in un **ordine sbagliato**, violando la **causalità**.

**Es: Mr. Poons** chiede *"Come stai oggi?"*. **Mrs. Cake** risponde *"Bene, grazie!"*. Ma se le due scritture arrivano alle repliche in momenti diversi, può succedere che l'utente legga prima *"Bene, grazie!"*, e poi *dopo un refresh* *"Come stai oggi?"*. Questo perché le repliche **non ricevono gli aggiornamenti nello stesso ordine**. Replica A riceve prima la domanda, poi la risposta. Replica B riceve prima la risposta, poi la domanda. Se un utente legge da B e poi da A, vede le cose **invertite nel tempo**.



**Problema Single Leader:** Se la rete tra il client e il leader si interrompe, le **scritture non possono essere fatte**. Abbiamo nel leader un singolo punto centrale di fallimento.

## Multi-Leader Replication

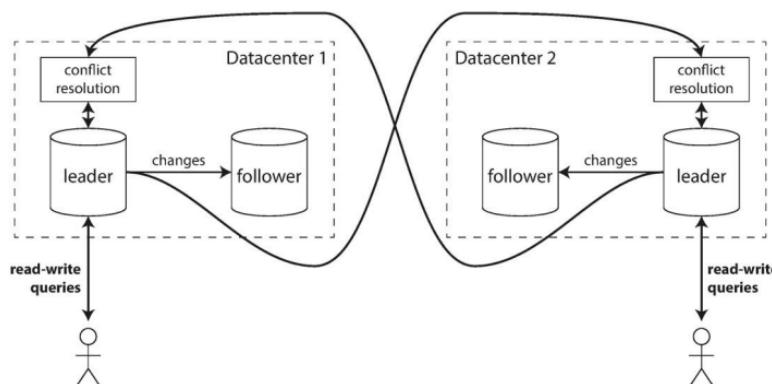
**Multi-Leader Replication:** Più nodi sono **leader** che possono accettare scritture.

Quando un leader riceve una scrittura, la **propaga** agli altri leader. Ogni leader si comporta **anche** come follower degli altri leader, ricevendo gli aggiornamenti e aggiornandosi. Se la connessione verso un leader cade, puoi scrivere su un altro leader.

**Problemi:** Se due leader ricevono **modifiche diverse dello stesso dato** nello stesso momento, devono esistere **regole di riconciliazione**, come vince l'ultimo timestamp.

**Utilità:** Utile quando le app sono **distribuite geograficamente**. Ogni continente può scrivere sul leader più vicino. Quando i server sono nello **stesso datacenter è inutile** aumentando solo la complessità di gestire i conflitti.

**Multi-datacenter Operation:** Scenario in cui il sistema è distribuito **su più datacenter**.



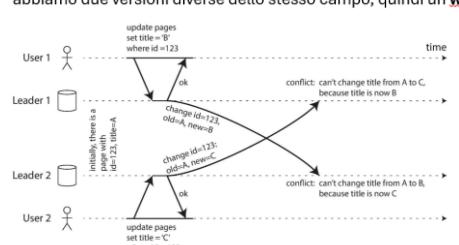
**Single-Leader Configuration:** Il **leader** si trova in **un solo datacenter**. **Tutte le scritture passano da lì**, anche se l'utente si trova lontano. Se un utente in Asia deve scrivere, ma il leader è in USA si ha **alta latenza**. Se il datacenter del leader ha problemi **nessuno può scrivere**.

**Multi-Leader Configuration:** Ogni datacenter ha il **proprio leader**. Ogni leader accetta **scritture localmente**. I leader si scambiano gli **aggiornamenti** tra loro. Ogni leader funziona **anche** come follower degli altri leader. Abbiamo la risoluzione dei due problemi precedenti. Tuttavia, serve una strategia di risoluzione in caso le modifiche tra i due datacenter vadano in conflitto. Dentro ogni datacenter si usa **replicazione leader-follower normale**.

**Offline Operation mode:** Quando un'app deve funzionare senza internet, ogni dispositivo ha una **replica locale leader** del database, che **accetta scritture**, anche offline. Quando il dispositivo torna online, le modifiche locali vengono **sincronizzate** con il server leader. Il server poi le **propaga ad altri dispositivi**. Abbiamo le stesse caratteristiche del multi-leader.

**Gestione dei Conflitti di Scrittura:** Un **conflitto** nasce quando due utenti **modificano lo stesso dato, in due leader diversi, prima** che le modifiche si replichino tra leader.

Esempio: il titolo di un post è inizialmente A. Utente 1 (Europa) cambia A → B. Utente 2 (USA) cambia A → C. Entrambi vedono la loro modifica localmente, subito. Poi, quando i leader si scambiano gli aggiornamenti **asincronamente**, il sistema si accorge che abbiamo due versioni diverse dello stesso campo, quindi un **write conflict**.



**Conflitto asincrono:** Ogni leader accetta la propria scrittura **subito**. Il conflitto viene scoperto **solo dopo**, quando i leader si sincronizzano. È **tropppo tardi** per chiedere agli utenti, quale dei due valori è quello giusto?

**Tentativo di risoluzione sincrona nel multi-leader:** Potremmo provare ad aspettare che ogni scrittura venga replicata a tutti gli altri leader **prima** di confermarla all'utente.

**Evitare i Conflitti:** Tutte le scritture relative a *uno stesso dato* devono passare **sempre** **dallo stesso leader**. Se solo **un leader** modifica quel dato, non ci sono modifiche concorrenti, quindi **non possono nascere conflitti**.

**ID Unico per ogni scrittura:** Ogni scrittura ha un identificatore. Le repliche scelgono la stessa scrittura come "vincente" basandosi su questo ID.

**Precedenza delle repliche:** Se due modifiche entrano in conflitto, "vince" quella proveniente dal leader più prioritario.

**Risoluzione definita dall'app:** Possiamo **scrivere noi stessi le regole** che dicono al sistema cosa fare quando due versioni dello stesso dato si scontrano.

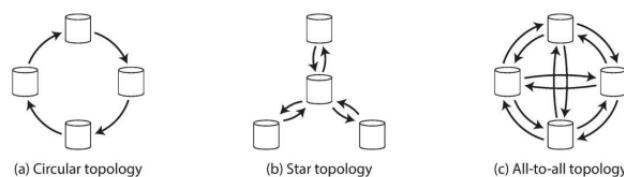
**Risoluzione al momento della scrittura:** Al momento della scrittura, se il dato è in conflitto con quello di un'altra replica leader, l'app deve avere una funzione **handler** che si attiva **automaticamente**, risolvendo il conflitto con una regola preimpostata.

**Risoluzione al momento della lettura:** I conflitti **non vengono risolti immediatamente**, ma **registrati**. Quando un utente fa una lettura, l'app può ricevere **più versioni dello stesso dato**, e deve **decidere cosa mostrare**. L'app può chiedere all'utente cosa scegliere oppure decidere automaticamente.

**Mergeable Persistent Data Structures:** Ogni modifica mantiene una "storia" che può essere **fusa** invece di sovrascritta. Simile al concetto di **version control** (Git).

## Multi-Leader Replication Topologies

**Caso 2 leader:** Se ci sono solo **due leader**, Leader A replica verso Leader B e Leader B replica verso Leader A.



## Quando ci sono più leader (3, 4, 10, ...)

**All-to-All Topology:** Ogni leader **invia direttamente** gli aggiornamenti a **tutti gli altri leader**.

**Circular Topology:** Ogni leader invia gli aggiornamenti **solo al successivo**. Il leader che li riceve li **inoltra** al successivo, e così via.

**Star Topology:** C'è un **leader centrale**. Tutti gli altri leader inviano scritture **al leader centrale**, che poi le inoltra.

## Leaderless Replication

**Funzionamento:** **Non esiste un leader**. Tutti i nodi sono alla pari, ognuno può accettare **lettura e scrittura**.

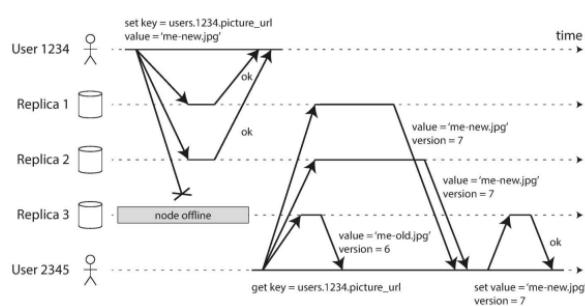
**Gestione dei guasti:** Se un nodo è offline, il client invia la scrittura agli altri nodi **disponibili**. La scrittura è considerata **riuscita** quando **una maggioranza** di nodi la conferma.

**Valori obsoleti:** Quando una replica torna attiva dopo un periodo di disconnessione ha copie dei dati **vecchie**, quindi se un client legge da quel nodo riceve **un valore superato**.

**Read Repair:** Quando un client **legge** un valore, contatta più nodi. Se vede che uno dei nodi ha una versione **più vecchia**, invita quel nodo a **aggiornarsi**.

**Anti-Entropy:** Processo **periodico in background**, che confronta repliche tra loro e sincronizza eventuali differenze.

**Quorum:** Per una **scrittura**, serve **l'approvazione di un certo numero** di repliche. Per una **lettura**, si leggono **più repliche** e si sceglie la versione più **aggiornata**.

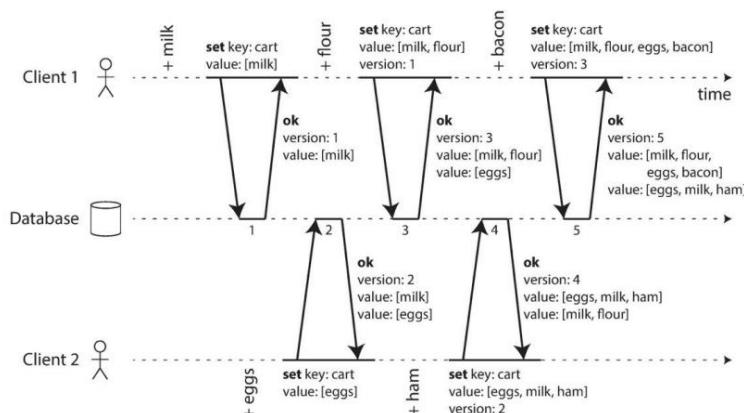


## Monitorare la “vecchiezza” dei dati

**Leaderless Replication:** Non c’è un leader centrale che definisce **l’ordine unico** delle scritture. Ogni replica può ricevere gli aggiornamenti in ordine diverso. Vengono usati red repair ma soprattutto **Anti-entropy periodico**.

**Last Write Wins:** Se due scritture entrano in conflitto, si **tiene solo quella più recente** e si **scarta l’altra**.

**Operazioni concorrenti:** Due operazioni sono **concorrenti** quando avvengono senza che una sia a conoscenza dell’altra, cioè non esiste una relazione di *happens-before* tra di esse.



**Happens-before:** Se un’operazione è eseguita basandosi sull’effetto di un’altra (un client legge una versione di un dato e poi scrive una nuova versione partendo da quella lettura), allora la prima **accade-prima** della seconda e tra le due esiste un legame causale.

**Funzionamento:** Quando il sistema rileva **scritture concorrenti**, non può sceglierne arbitrariamente una ed eliminare l’altra, perché nessuna è più corretta dell’altra e si perderebbero informazioni. In questi casi è necessario **mantenere entrambe le versioni e fonderle** in un’unica versione coerente.

La  **fusione** non può essere decisa dal database in modo generico, perché dipende dal significato dei dati, quindi deve essere spesso gestita a livello applicativo.

**Version numbers:** Numero di versione locale di ogni replica, che incrementa quando tale replica effettua una scrittura.

**Version Vector:** Array di version number relativi ad ogni replica.

$V \text{ include } W$  se per ogni replica  $i$ ,  $V[i] \geq W[i]$ , e per almeno una replica  $j$ ,  $V[j] > W[j]$ . In questo caso la versione  $V$  deriva da  $W$ , ed esiste un **ordine causale**.

Se invece  $V$  è maggiore in alcune posizioni,  $W$  è maggiore in altre, allora **nessuna include l’altra**. Le versioni sono **indipendenti**, quindi **concorrenti**.

**Es Ordine causale**

Supponiamo due repliche: **A** e **B**.

Versione 1 (scritta da A):  $V1 = [1, 0]$

Versione 2 (A scrive di nuovo dopo aver visto V1):  $V2 = [2, 0]$

Confronto:  $2 \geq 1$ ,  $0 \geq 0$ , e almeno una posizione è strettamente maggiore. **V2 include V1**,

$V2$  deriva da  $V1$ , Non c’è concorrenza.

### Es Scrutture concorrenti

Replica A e replica B scrivono **senza comunicare**.

Scrittura su A: VA = [1, 0]. Scrittura su B: VB = [0, 1]

Confronto: VA è maggiore su A, VB è maggiore su B. Nessuna versione include l'altra. Le versioni sono **concorrenti**. Non si può scartarne una, entrambe rappresentano aggiornamenti validi, eliminarne una significherebbe perdere informazione, quindi le fondiamo.

### Es – Merging

Dato condiviso: lista della spesa.

Versione A: [milk, eggs]

Versione B: [milk, bread]

Le due versioni sono concorrenti, quindi: `merge(A, B) = [milk, eggs, bread]`

Il risultato è una nuova versione che: contiene entrambi gli aggiornamenti, avrà un **version vector** che **include entrambi**, [1, 1].

## Lezione 14: Partitioning

**Partitioning:** Dividere il database in più parti più piccole, chiamate **partizioni**.

**Partizione:** Porzione del database che contiene solo una parte dei record complessivi.

Ogni record **sta solo** in una partizione, mai in più partizioni.

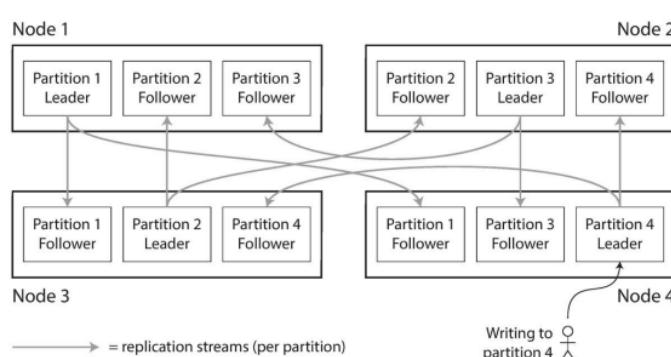
**Es:** Partizione 1 → cognomi A–F Partizione 2 → cognomi G–N Partizione 3 → cognomi O–Z  
Ogni partizione funziona come un **piccolo database autonomo**, con le sue tabelle e con le sue operazioni. È possibile comunque fare operazioni che coinvolgono più partizioni, come una ricerca globale. Le partizioni possono essere **distribuite su nodi server diversi**.

**Why Partitioning:** Se i dati aumentano, diventa difficile gestirli su un solo server.

**Dividendo** i dati, possiamo distribuire il carico di lavoro.

**Partitioning + Replication:** **Partitioning** divide i dati in più parti, mentre **Replication** copia ogni partizione su più nodi. Se un nodo si rompe, si continua a lavorare. Se i dati crescono, si aggiungono nodi e partizioni.

## Leader – Follower Model con Partitioning



**Funzionamento:** Ogni **nodo** può contenere **più partizioni**. Ma per ogni partizione abbiamo un **leader** che **accetta scritture**, e dei **followers** che **ricevono copie aggiornate** dal leader. I followers possono rispondere a **lettura** per distribuire il carico.

Es: Node 1 → Leader per Partition 1, Follower per Partition 2 e 3

Node 2 → Leader per Partition 3, Follower per Partition 1, 2, 4

Node 3 → Leader per Partition 2, Follower per altre

Node 4 → Leader per Partition 4

## Goals and Challenges of Partitioning

**Distribuire in modo uniforme:** Nessun nodo deve avere **troppi dati** rispetto agli altri.

**Distribuire le query:** Nessun nodo deve ricevere **troppi accessi** rispetto agli altri.

**Random Assignment:** Ogni record viene assegnato a un nodo **a caso**. La distribuzione **uniforme è garantita**, quindi nessun nodo ha molto più lavoro degli altri. Tuttavia, non sai dove si trova un dato specifico, quindi per cercare un elemento devi **chiedere** a tutti i nodi in **parallelo**.

**Assegnazione per intervallo di chiavi:** Ogni partizione contiene un **intervallo** continuo di chiavi. Per determinare in quale partizione si trova un dato, si controlla in quale intervallo cade la chiave, in modo da trovare la partizione.

**Es 2: Registrazione Studenti**

Gli studenti hanno ID numerici.

**Partizione Range ID Nodo**

P1 1000 - 1999 Node 1

P2 2000 - 2999 Node 2

P3 3000 - 3999 Node 3

Se lo studente ha ID: 2345 → Cade nel range **2000 - 2999** → quindi è in Partition 2 → Node 2

Alcuni intervalli di chiavi possono contenere **molti più dati** di altri.

Es: Partizione 1 → A-B Partizione 2 → C-D Partizione 12 → T-Z

Ma nella realtà: Molti cognomi iniziano con **S**, quindi "T-Z" contiene **più libri** di "A-B"

La partizione **T-Z** diventa enorme → sovraffollato La partizione **A-B** rimane piccola → risorse sprecate. Questo si chiama skew.

**Adattamento automatico:** Il database monitora il carico e **divide** partizioni troppo grandi e **fonde** partizioni troppo piccole.

**Manuale:** Un amministratore divide e fonde partizioni quando nota squilibri.

**Esempio:** In un E-Commerce, durante i saldi, alcuni prodotti ricevono **tantissime**

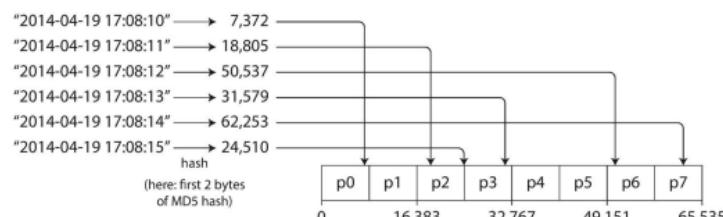
**richieste**. Se la partizione della categoria "Scarpe Nike" riceve molto traffico, diventa una hot partition. Per risolvere, il sistema può **splittere** la partizione in due:

Prima: Partition = SKU1000-SKU2000

Dopo: Partition A = SKU1000-SKU1500 Partition B = SKU1501-SKU2000

## Hash Partitioning

**Funzionamento:** Invece di usare la chiave direttamente per decidere la partizione, calcoliamo un **hash** della chiave:  $\text{partition\_id} = \text{hash}(\text{key}) \bmod N$ , con  $N$  = numero di partizioni. Anche se i dati sono concentrati in un certo range, **l'hash li distribuisce uniformemente**, quindi non si hanno più partizioni con troppi dati. Ma le chiavi che erano vicine ora sono sparse tra partizioni differenti. Le range query sono inefficienti.



**Cassandra Solution:** La chiave primaria può essere composta da **più colonne**. La **prima parte** viene **hashata** e determina **in quale partizione** vanno i dati. Le **altre parti** non vengono hashate e servono per **ordinare i dati** all'interno della partizione.

Questo porta **due vantaggi contemporaneamente**:

Es: Key = (user\_id, update\_timestamp). user\_id → partition key (hashata → distribuisce i dati tra nodi). update\_timestamp → clustering key (ordinata).

Risultato: Ogni utente ha **la sua partizione**. All'interno, i post sono **ordinati per tempo**.

Recuperare tutti gli update di un utente → **efficiente**. Recuperare un intervallo di tempo (es. ultimi 10 minuti) → **efficiente**.

**Gestire Skew:** Ci sono casi in cui **troppe operazioni** finiscono sulla **stessa chiave**, quindi finiscono **nella stessa partizione**, indipendentemente dall'hash.

Es: Un utente **famoso** su un social, milioni di commenti sul suo profilo

Si deve spezzare una chiave molto usata in più sotto-chiavi. Aggiungi un **suffisso random** alla chiave. L'hash sarà diverso per ogni sottochiave ed andrà in partizioni diverse.

Es: Invece di scrivere sempre su "user\_123", scrivi su: user\_123#0, user\_123#1, user\_123#N. Le scritture si distribuiscono su **10 partizioni** diverse.

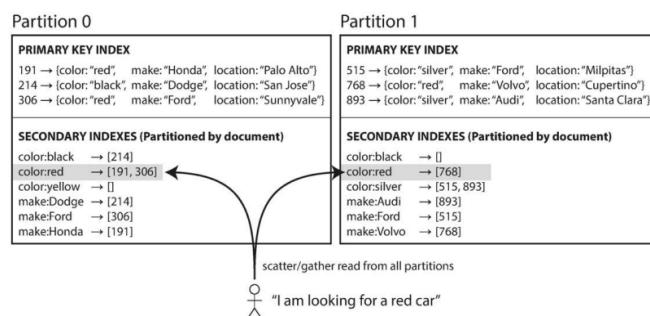
Quando vai a **leggere**, devi però leggere **tutte** le varianti (#0, #1, #2, ...), e **ricombinare i risultati** lato applicazione.

**Indice secondario:** Indice fatto **su una colonna che non è la chiave primaria**.  
**La primary key determina la partizione.** Quindi sappiamo esattamente **in quale nodo cercare**. Un indice secondario invece **non è unico**. Lo stesso valore (es. city = Roma) può apparire **in tante partizioni diverse**. Per usare un secondary index bisogna cercare **in più partizioni**.

### Document-Based Partitioning:

Quando il database è **partizionato per record** (documento), la **primary key** è il document ID. Il range/hash del document ID decide **in quale partizione sta il record**. Se ci sono indici secondari (per esempio: color, make, brand), ogni partizione mantiene il proprio indice secondario locale.

**Effetto pratico:** Per fare una ricerca su un valore secondario, devo interrogare **tutte le partizioni** e poi unire i risultati.



**Term-Based Partitioning:** I record sono partizionati per chiave primaria. Un **termine** è un valore secondario su cui vogliamo fare ricerche, (color = red...). Per cercare per attributi non chiave si usa un indice secondario globale, che è partizionato per termine di ricerca. Questo indice contiene solo i riferimenti ai record.

Quando fai una query come color = red, il sistema sa **esattamente** cerca l'indice nella partizione relativa al termine, legge gli ID dei record dall'indice e poi recupera i documenti dai nodi dove sono fisicamente memorizzati.

Distribuzione dei dati:

```
css
Nodo A → Doc1
Nodo B → Doc2
Nodo C → Doc3
```

Copia codice

Indice globale su color:

```
less
Partizione indice P1 → color:red → [1, 3]
Partizione indice P2 → color:blue → [2]
```

Copia codice

Query:

```
powershell
SELECT * WHERE color = red
```

Copia codice

Risultato:

- vai a P1
- ottieni [1,3]



**Scritture più complesse:** Quando modifichi o aggiungi **un record**, esso può contenere **molti termini** indicizzati. Scrivere **un record** può significare **aggiornare più nodi** a causa degli indici secondari.

### Rebalancing

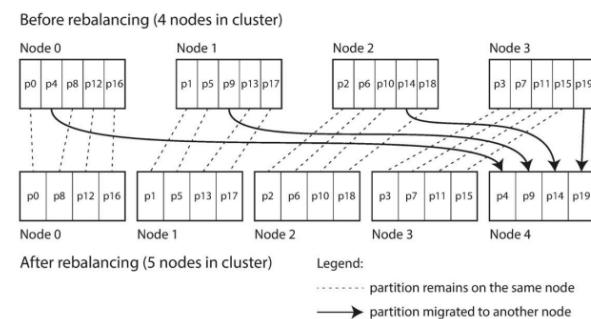
**Rebalancing:** Spostare dati (richieste) da alcuni nodi ad altri, per ripristinare una **distribuzione equilibrata** del carico. Si spostano **alcune partizioni** da un nodo sovraccarico a uno meno carico. Si **aggiorzano** le **regole** che dicono quali nodi servono quali dati. Il sistema deve **continuare a leggere e scrivere** mentre si ribilancia.

**Hash Mod N Approach:** Come detto prima, nodo = hash(key) mod N. Quando il numero di nodi cambia, cambia anche il risultato del modulo, quindi *quasi tutte le chiavi devono essere spostate*, generando tantissimo movimento di dati.

Es: Con **10 nodi**, una chiave poteva andare al nodo  $6 \cdot \text{hash}(\text{key}) \bmod 10 = 6$ .

Se aggiungi un nodo e passi a **11 nodi**:  $\text{hash}(\text{key}) \bmod 11 = 3$ . Quindi la stessa chiave cambia nodo.

**Soluzione:** Invece di avere **N nodi = N partizioni**, si crea **un numero fisso e grande N di partizioni** (es. **1000** partizioni), e poi si **assegnano** queste partizioni ai nodi (10 nodi, ogni nodo gestisce 100 partizioni). Nell'operazione modulo che determina la partizione per una chiave non cambia niente. **Se aggiungi un nodo**, il nuovo nodo **prende alcune partizioni** da altri nodi. Se un nodo muore, le sue partizioni vengono **ridistribuite** agli altri nodi. Le chiavi non devono essere redistribuite.



**Dynamic Partitioning:** Nei sistemi key-range, se una partizione **diventa troppo grande**, viene **divisa** in due. Se una partizione **diventa troppo piccola**, viene **fusa** con la partizione vicina. Ogni partizione è poi **assegnata a un nodo**, e se i nodi sono più o meno carichi, il sistema può ribilanciare le assegnazioni.

**Service Discovery:** Quando il client deve leggere o scrivere una chiave, **non sa** automaticamente su quale nodo si trova la partizione relativa al dato.

**Richiesta ad un nodo qualsiasi:** Il client manda la richiesta a un nodo casuale. Se il nodo **possiede la partizione**, la gestisce. Se non la possiede, **gira la richiesta** al nodo corretto.

**Routing Tier:** Il routing tier **conosce la mappa partizioni**, **nodi** e instrada la richiesta al nodo corretto. Il client manda tutte le richieste al **routing tier**.

**Client Partition-Aware:** I client **conoscono la funzione di partizionamento** e sanno quali nodi possiedono quali partizioni. Mandano la richiesta **direttamente** al nodo corretto.

**Parallel Query Execution:** Ogni nodo processa **una porzione del dataset**. I risultati vengono **combinati** alla fine. Il tempo totale si riduce drasticamente.

Serve un query optimizer sofisticato, un modello di dati ben progettato, un coordinamento accurato dei nodi.

## Lesson 15 – Transactions

**Violazione di atomicità:** Senza transazioni, un'operazione **composta** da più istruzioni sul database **non** viene **eseguita tutta o niente**. Se il sistema si blocca a metà, alcune modifiche restano applicate e altre no, portando il db in uno stato inconsistente.

**Lost Update:** Due operazioni concorrenti leggono lo stesso dato e lo aggiornano.

L'ultimo aggiornamento **sovrascrive** il precedente, che viene perso.

**Dirty Read:** Un'operazione legge dati che sono stati modificati da un'altra operazione **non ancora completata** (e che potrebbe fallire). Le scritture parziali diventano **visibili**.

**Inconsistent Read:** Un'operazione legge dati che vengono scritti da altre operazioni **durante la lettura**, ottenendo una combinazione di valori vecchi e nuovi.

**Non-Repeatable Read:** La stessa query, eseguita due volte dalla stessa operazione, restituisce risultati diversi perché un'altra operazione ha modificato i dati nel frattempo.

**Transazione:** Insieme di più operazioni effettuate sul db, considerate come un'**unica unità logica**. Vengono eseguite dal db come un'unica operazione. Se qualcosa va storto in una operazione, vengono annullate tutte. Una transazione deve essere **ACID**.

**ACID: A – Atomicity, C – Consistency, I – Isolation, D – Durability.**

**ACID Atomicity:** La transazione avviene o **interamente o niente**. Se una transazione fallisce, allora tutti gli **aggiornamenti parziali** vengono **annullati**.

**Consistency ACID:** Dopo ogni transazione, il database deve trovarsi in uno stato che rispetta tutti i **vincoli** di integrità(NOT NULL, FOREIGN KEY), e di dominio (es. età  $\geq 0$ ) definiti dal db, non lasciandolo in uno stato inconsistente.

**Isolation ACID:** Ogni transazione deve essere eseguita come se fosse **l'unica in corso**, senza **interferenze** da altre transazioni **concorrenti**.

Le transazioni devono comportarsi come se fossero eseguite **una dopo l'altra** sequenzialmente, anche se in realtà sono eseguite in parallelo.

**Snapshot Isolation:** Ogni transazione vede una **fotografia coerente** del database presa al momento in cui la transazione inizia. Durante l'esecuzione, la transazione legge sempre i dati da quello snapshot e non vede modifiche fatte da altre transazioni concorrenti.

**Durability ACID:** Una volta completato il commit di una transazione, i dati non vadano più persi, anche in caso di fallimento del db, perché vengono scritti su **memoria non volatile** (SSD, HDD).

**Durabilità reale:** A causa dei guasti che possono capitare a db, macchine, HDD e interi datacenter, la durabilità reale si ottiene combinando **scrittura sincrona** su **SSD/HDD**, **replica** dei dati su più nodi e **backup periodici** anche offline, accettando che non esista una soluzione perfettamente infallibile.

## Single-Object vs Multi-Object Operations

**Single-object operation** Operazione che riguarda **una sola riga del DB**.

**Multi-object operation:** Operazione che coinvolge **più righe** nella stessa tabella o in tabelle diverse.

**Es:** Supponiamo che un sistema di email mantenga sia la tabella delle email sia un contatore unread\_count.

Per mostrare le email non lette si esegue:

**SELECT COUNT(\*)**

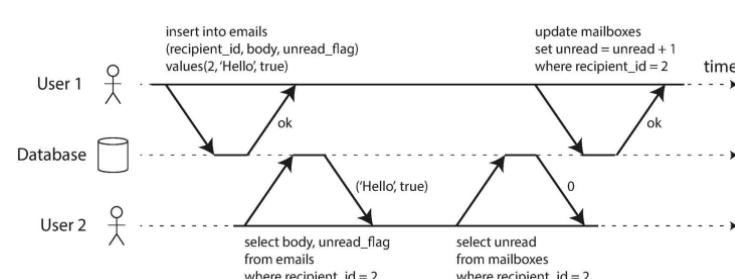
**FROM emails**

**WHERE recipient\_id = 2 AND unread\_flag = true;**

Quando arriva una nuova email, il sistema deve **inserire l'email** con unread\_flag = true e **incrementare unread\_count**, atomicamente. Quando un'email viene letta, deve **decrementare unread\_count**.

Senza transazioni può accadere che l'email venga inserita correttamente come non letta, ma il contatore non venga ancora aggiornato, senza atomicità. Il risultato è che il sistema mostra **1 email non letta**, mentre il contatore indica **0**.

Questo è un classico problema di **multi-object update senza transazione**: serve **atomicità** per aggiornare email e contatore insieme (o nessuno dei due) e **isolamento** per evitare che un client legga uno stato intermedio.



**Transazioni SQL:** Per trattare operazioni multiple come una transazione, si inseriscono le operazioni all'interno del costrutto **BEGIN TRANSACTION / COMMIT**.

**Single-Object Writes:** Quando aggiorni una **singola grande riga**, le proprietà ACID *atomicity* e *isolation* si applicano comunque. Se si salva un oggetto grande a metà, il valore è corrotto. Se un'altra operazione prova a leggere durante la scrittura, legge un dato incoerente.

Se la scrittura non finisce, il log ripristina il valore precedente. Vengono usati lock per permettere l'accesso alla riga ad una sola transazione per volta.

### Operazioni atomiche avanzate

**Increment:** Incrementa un valore come un'unica operazione atomica, senza fare lettura, modifica, scrittura.

**Compare-and-set:** Confronta il valore corrente di una variabile con un **valore atteso** e, solo se coincidono, lo **aggiorna** con un nuovo valore in un **unico passo indivisibile**. Se il valore non coincide, l'operazione fallisce senza modificare nulla.

#### Compare-and-Set (CAS) Operations

```
UPDATE wiki_pages
SET content = 'new content'
WHERE id = 1234
AND content = 'old content';
```

**Gestione degli errori:** Quando una transazione ACID **fallisce**, il sistema esegue un **abort**, cioè **annulla completamente tutti gli effetti parziali** della transazione, riportando il db in uno stato consistente, quindi la transazione può essere **ripetuta (retry)** in modo sicuro.

### Problemi di Isolation

**Concorrenza sicura:** Transazioni diverse possono correre in parallelo **senza problemi** se non modificano gli **stessi dati**. Ci sono problemi quando due transazioni **modificano lo stesso dato**, oppure una **legge un dato** che l'altra sta modificando.

**Serializable:** L'esecuzione parallela ha lo stesso effetto dell'esecuzione di una transazione alla volta. Questo richiede più lock e meno parallelismo.

### Livelli di isolamento più flessibili

**Dirty Reads:** Una transazione **legge** dati scritti da un'altra transazione **non ancora committata**.

Esempio: T1 → aggiorna saldo = 1000 → *non commit* T2 → legge saldo = 1000  
T1 → *rollback* → saldo torna a 500 T2 ha letto un valore **mai realmente esistito**.

**Dirty Writes:** Una transazione **sovrascrive** i dati scritti da un'altra transazione **non ancora committati**.

- La transazione T1 aggiorna il saldo di un conto da 100 a 150, ma non ha ancora fatto commit.
- Prima che T1 termini, la transazione T2 legge lo stesso record e lo aggiorna da 150 a 120, sovrascrivendo il valore scritto da T1.
- Successivamente T1 abortisce.

Risultato: nel database rimane il valore 120, che dipende da una scrittura mai confermata.

**Read Committed:** Una transazione **può leggere solo dati già committati** da altre transazioni. Se un'altra transazione ha fatto delle modifiche ma non ha ancora fatto **commit**, le altre transazioni non possono vederli.

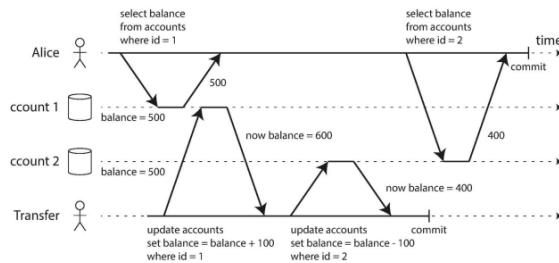
**Funzionamento:** Quando una transazione scrive un nuovo valore, viene **conservato il vecchio valore committato** e si crea il nuovo valore, **visibile solo** alla transazione che scrive. Se un'altra transazione vuole leggere, se la transazione che ha scritto non ha ancora committato, il **lettore** vede il **vecchio valore**. Quando la scrittura fa **commit**, i lettori inizieranno a vedere il **nuovo valore**.

Read Committed NON garantisce che due **lettture** all'interno della **stessa transazione** siano **coerenti** tra loro.

**Read Skew:** Una transazione legge più dati correlati in momenti diversi e, tra una lettura e l'altra, un'altra transazione committa delle modifiche. Il risultato è che la transazione osserva una combinazione di valori vecchi e nuovi che non è mai esistita nel db.

#### Spiegazione dell'immagine

Alice ha **due conti** (account 1 e account 2), entrambi inizialmente con **saldo = 500**, e effettua una transazione di **transfer** che sposta **100** dal conto 2 al conto 1.  
 Alice esegue una prima lettura: `SELECT balance FROM accounts WHERE id = 1` ottiene **500**.  
 Nel frattempo, la transazione **Transfer**: aggiorna **account 1** → saldo diventa **600** aggiorna **account 2** → saldo diventa **400** fa **commit**.  
 Dopo il **commit** del transfer, Alice esegue la seconda lettura: `SELECT balance FROM accounts WHERE id = 2` ottiene **400**.  
**Risultato visto da Alice:** account 1 = **500** (vecchio valore) account 2 = **400** (nuovo valore)  
 Questa coppia di valori **non è mai stata vera nello stesso istante**: o erano **500 / 500** prima del transfer, oppure **600 / 400** dopo il transfer. Questo succede perché ogni `SELECT` vede **solo dati già committati**.

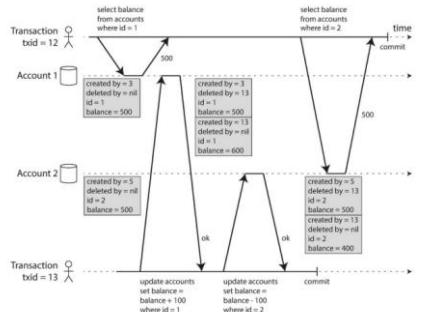


**Snapshot Isolation:** Quando una transazione inizia, riceve una **fotografia** dell'intero database, contenente **tutti i dati committati al momento dell'inizio**. Durante la transazione, la transazione **continua** a vedere **quello snapshot**, anche se altri aggiornano il DB, quindi tutte le letture sono **coerenti tra loro**.

#### Esempio: Alice sotto Snapshot Isolation

T1 (calcolo totale): legge Conto A → 500 (**snapshot**) legge Conto B → 500 (**snapshot**)  
 T2 nel frattempo può cambiare tutto ciò che vuole: A → 400 B → 600 **commit**  
 Ma T1 continua a vedere **A=500, B=500**, perché quello era lo snapshot iniziale. Saldo totale = 1000

**Implementazione:** Come per read committed, il db conserva più versioni della stessa riga. T1 legge riga X, e vede la versione X<sub>1</sub>. T2 modifica riga X, crea una nuova versione X<sub>2</sub>. T1 continua a vedere X<sub>1</sub> (la versione vecchia). T2 vede X<sub>2</sub> (la nuova versione).



**Letture in Snapshot Isolation:** Una transazione legge sempre la versione della riga **visibile nel proprio snapshot**.

**Scritture in Snapshot Isolation:** Quando una transazione aggiorna una riga, **non sovrascrive la riga esistente, ma crea una nuova versione** della riga, e marca la versione vecchia come **superata** (xmax).

**Conflitti di scrittura:** Se due transazioni provano a modificare la **stessa riga**, entrambe possono procedere creando la propria versione, **non bloccandosi** a vicenda per tutta la transazione. **Al commit** la prima che committa **vince**, mentre la seconda viene **abortita**.

**Lost update:** Due transazioni leggono lo stesso dato e lo aggiornano entrambe, ma l'aggiornamento di una viene sovrascritto dall'altra, senza che il sistema rilevi il conflitto, causando la **perdita di un aggiornamento valido**.

**Incrementare un contatore o un saldo:** Due transazioni leggono un valore: 100. La prima vuole fare  $+10 \rightarrow 110$ . La seconda vuole fare  $+20 \rightarrow 120$ . Se la seconda ~~committa~~ per prima e la prima scrive dopo  $\rightarrow$  diventa **110**, perdendo l'update  $+20$ .

**Il lost update** nasce dal ciclo pericoloso **read  $\rightarrow$  modify  $\rightarrow$  write**. La soluzione è **impedire che due aggiornamenti concorrenti scrivano senza coordinamento**.

**Write locks:** Quando una transazione scrive una riga, il database mette un **lock di scrittura**. Le altre transazioni che vogliono scrivere **devono aspettare**.

**Check-and-set:** Quando si aggiorna un valore, il DB controlla che **non sia cambiato rispetto a quando è stato letto**. Se è cambiato, l'update **fallisce**.

**Optimistic Concurrency Control:** Le transazioni **non bloccano subito**. Al commit, il DB verifica se i dati letti sono stati modificati. Se sì, la transazione **abortisce e deve riprovare**.

**MVCC + rilevazione dei conflitti:** Ogni update crea una **nuova versione** della riga. Se una transazione prova a scrivere partendo da una versione **troppo vecchia**, il DB la rifiuta.

**Operazioni di update atomiche:** Usare un'unica operazione di **UPDATE** atomica. Il DB gestisce lock e concorrenza internamente.

**Relational DB:** Gli update sono **atomici per definizione**. I lock interni impediscono lost update.

**Explicit Locking:** Quando servono regole applicative complesse, si usa **explicit locking** per bloccare esplicitamente le righe.

**FOR UPDATE:** Blocca **tutte le righe restituite dalla SELECT**. Nessun'altra transazione può leggerle o modificarle. Solo dopo il COMMIT il lock viene rilasciato.

**Multiple nodes with data copies:** In un sistema replicato, ogni nodo ha la sua copia dei dati, ogni nodo può ricevere richieste di scrittura. Le modifiche possono avvenire **in parallelo** su nodi diversi, sullo stesso dato. I metodi classici di lock e CAS non funzionano globalmente. Es: il nodo A aggiorna  $x = 10$ , il nodo B aggiorna  $x = 20$ , quale valore è giusto?

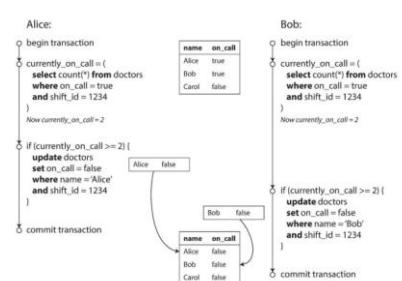
Quando due nodi scrivono sullo stesso dato in parallelo, i sistemi replicati **creano versioni in conflitto** e **mergono le versioni** successivamente.

**Last Write Wins:** In un sistema replicato vince la scrittura con timestamp più alto.

Problemi: Write Skew, Dirty writes e Lost updates.

LWW evita conflitti tecnici, ma **non garantisce correttezza logica**.

**Write skew:** Due transazioni leggono in modo concorrente **oggetti diversi** ma collegati da una **stessa regola logica**, e poi li modificano in modo consistente individualmente, ma **inconsistente globalmente**.



**Violazione della regola:** Nell'ospedale almeno un medico deve essere on call. In modo consistente, due pazienti ottengono un medico libero, leggendo entrambi  $current\_on\_call > 1$ . Localmente era corretto, ma ora **zero medici** risultano on-call.

**Using explicit row locking:** La transazione blocca **tutte le righe che hanno letto**, coinvolte nella stessa regola logica.

**Serial execution:** Il comportamento del DB deve essere equivalente a: Esegui T1 completamente. Poi esegui T2 completamente. Anche se sono in realtà eseguite in parallelo, il risultato deve essere lo stesso che se fossero state eseguite in sequenza.

## Lesson 16 – Distributed File Systems

**Big Data pipeline:** I dati vengono **raccolti dai processi di business**. I dati raccolti vengono: combinati, elaborati e utilizzati per creare **modelli di Machine Learning**, in modo da estrarre conoscenza dai dati grezzi.

Un singolo computer non è in grado di memorizzare e processare efficacemente enormi quantità di dati. Usiamo il **distributed computing**.

### MapReduce

**Map:** I dati vengono suddivisi tra i vari processori. Ogni parte viene processata in parallelo. Si producono risultati intermedi.

**Reduce:** I risultati intermedi, ottenendo un risultato finale significativo.

Ad oggi MapReduce è considerato legacy ed è stato sostituito da altri framework, come Hadoop.

### Il vero problema del Big Data

I Big Data lavorano con **terabyte di dati**. I tempi di elaborazione diventano enormi. Nel frattempo i data scientist modificano variabili e modelli. **C'è bisogno di molta più capacità computazionale**. Per accelerare le data pipeline si usano datacenter basati su **GPU**, che sono 10x rispetto alle CPU.

## Distributed File Systems (DFS)

**Distributed File System:** **File System distribuito** progettato per gestire dati distribuiti su più nodi, e offrire un accesso ai dati **trasparente**, come se fossero su un unico PC. Gli utenti leggono e scrivono i file come se fossero **locali**. La posizione fisica dei dati è irrilevante per gli utenti, poiché i file sono identificati da **nomi logici**.

**Scalabilità:** I dati vengono distribuiti su più nodi e il sistema scala orizzontalmente aggiungendo nuove macchine.

**Affidabilità:** I dati vengono replicati su più nodi. Se una o più macchine falliscono, le informazioni sono presenti nella replica e non vanno perse.

**Prestazioni:** Permette di elaborare i dati in parallelo su più nodi, riducendo i tempi di esecuzione di data processing.

**Data locality:** I calcoli vengono eseguiti direttamente sui nodi che ospitano i dati, riducendo la latenza dovuta alla rete.

## Hadoop

**Hadoop Distributed File System:** File system distribuito usato da Hadoop per memorizzare e replicare **file molto grandi** (TB) su molte macchine diverse.

**Calcolo distribuito:** I job vengono eseguiti direttamente da Hadoop sui nodi che contengono i dati, riducendo i trasferimenti via rete.

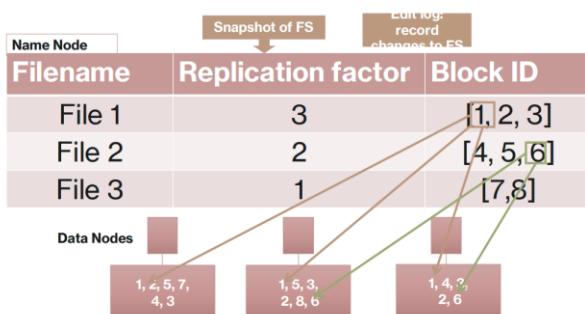
**HDFS Architecture:** Il controllo è centralizzato sul **NameNode**, che coordina tutto il cluster, mentre lo storage effettivo è distribuito tra i **DataNode**.

**NameNode:** Nodo controller che gestisce l'intero cluster di DataNode. Per ogni file mantiene in RAM il **nome del file**, il **fattore di replica** e l'**elenco dei blocchi** che lo compongono, identificati da un **block id**. Ogni blocco viene poi **replicato su più DataNode** in base al replication factor. Il NameNode sa esattamente **quali blocchi stanno su quali nodi**.

Non vengono salvati al suo interno dati effettivi. Il NameNode, per ricostruire lo stato del cluster in caso di riavvio, salva inoltre lo **snapshot del file**.

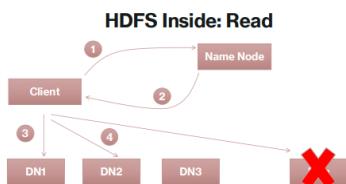
**system**, che rappresenta lo stato completo in un certo momento, e un **edit log**, che registra tutte le modifiche successive.

#### HDFS Inside: Name Node

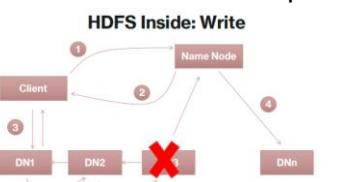


**DataNode: Nodi slave** che memorizzano fisicamente i blocchi dei file e si occupano delle operazioni di lettura, scrittura e replica.

**Lettura dati:** Quando un client vuole leggere un file, **contatta** prima il **NameNode** per sapere su quali DataNode sono memorizzati i blocchi. Viene preferita la replica **più vicina** dal punto di vista di rete (stesso rack). Il trasferimento dei dati avviene direttamente tra **client** e **DataNode**, senza passare dal NameNode. Se durante la lettura un DataNode non risponde, il client si collega automaticamente a **un'altra replica** dello stesso blocco.

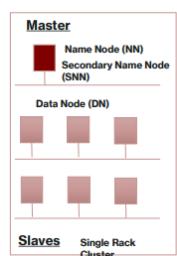


**Scrittura dati:** Quando un client deve scrivere un file, **contatta** prima il **NameNode** per chiedere **dove** può scrivere i dati. A quel punto il client **scrive i blocchi** direttamente sui **DataNode**. Se un nodo fallisce, il NameNode rileva il problema e ordina la **ricostruzione** delle repliche mancanti su altri nodi.



**Dove scrivere le repliche:** Una replica viene scritta sul nodo locale (client), una seconda su un nodo in un **rack diverso**, e una terza su un altro nodo dello stesso rack della seconda. Questo viene scelto per garantire capacità di sopravvivere a guasti di nodi o rack, e in contempo minimizzare la distanza di rete, per minimizzare la latenza di lettura/scrittura dei dati.

**Secondary NameNode:** Nodo che ha il compito di creare **checkpoint**, salvando periodicamente una versione consistente dei metadati.



**Single-rack cluster:** Tutti i DataNode si trovano nello stesso rack, quindi la comunicazione è più semplice ma meno tollerante ai guasti.

**Multi-rack cluster:** I DataNode sono distribuiti su più rack collegati da switch. Evita che il guasto di un intero rack comporti la perdita dei dati.



**Uso blocchi:** Un file può essere molto più grande di un singolo disco. Spezzando un file in blocchi, HDFS può distribuirlo su più macchine e quindi memorizzarlo ed elaborarlo in parallelo. Il blocco ha dimensione fissa (**64 MB**), il che lo rende semplice da spostare e replicare. Blocchi grandi significano meno I/O, meno metadata e quindi migliori prestazioni su grandi file sequenziali. Tuttavia vengono gestiti male i file piccoli, poiché richiedono almeno un blocco.

## Lesson 17 – MapReduce

**Idea:** I dati vengono **divisi** tra più nodi, ogni nodo lavora in **parallelo** e i risultati parziali vengono **combinati** automaticamente.

**MapReduce:** Modello di programmazione in cui lo sviluppatore definisce solo la logica di elaborazione, mentre il sistema gestisce automaticamente parallelismo, distribuzione dei dati e tolleranza ai guasti.

### Three steps

**Map:** Ogni nodo worker applica la funzione Map ai dati locali. Si tratta di filtraggio e trasformazione dei dati. L'output viene scritto in una memoria temporanea.

**Shuffle:** I nodi worker ridistribuiscono i dati in base alle chiavi prodotte dalla fase Map, in modo che tutti i dati associati alla stessa chiave vengano raccolti sullo stesso nodo.

**Reduce:** I nodi worker processano in parallelo i gruppi di dati per ciascuna chiave e producono il risultato finale aggregato.

**Es:** Vogliamo contare quante volte compare ogni parola in una collezione di documenti.

**Input:** Ogni documento è rappresentato come:  $(key_a, value_1) = (ID\_documento, \text{testo})$

Es:  $(1, \text{"big data big problem"})$

**Map:** Legge il testo e, per ogni parola, produce una coppia  $(parola, 1)$ .

**Map**  $(key_a, value_1) \rightarrow list(key_b, value_2)$ , in cui  $key_b$  = parola,  $value_2 = 1$  (una occorrenza)

Output della **Map**:  $(\text{big}, 1) (\text{data}, 1) (\text{big}, 1) (\text{problem}, 1)$

**Shuffle:** Il sistema raggruppa tutte le coppie con la stessa chiave.

$(\text{big}, [1, 1]) (\text{data}, [1]) (\text{problem}, [1])$

**Reduce:** Somma i valori associati a ciascuna parola, contenuti nell'array accanto.

**Reduce**  $(key_b, list(value_2)) \rightarrow list(key_c, value_3)$ , con  $key_c$  = parola e  $value_3 = \#occ$

Output finale:  $(\text{big}, 2) (\text{data}, 1) (\text{problem}, 1)$

**Funzionamento MapReduce:** Partiamo da un insieme di coppie chiave valore. Ogni coppia  $(key_a, value_1)$  viene passata ad ogni nodo.

**Map**  $(key_a, value_1) \rightarrow list(key_b, value_2)$ : Per ogni elemento di input  $(key_a, value_1)$ , Map applica una trasformazione e produce una lista di coppie  $(key_b, value_2)$  intermedie.

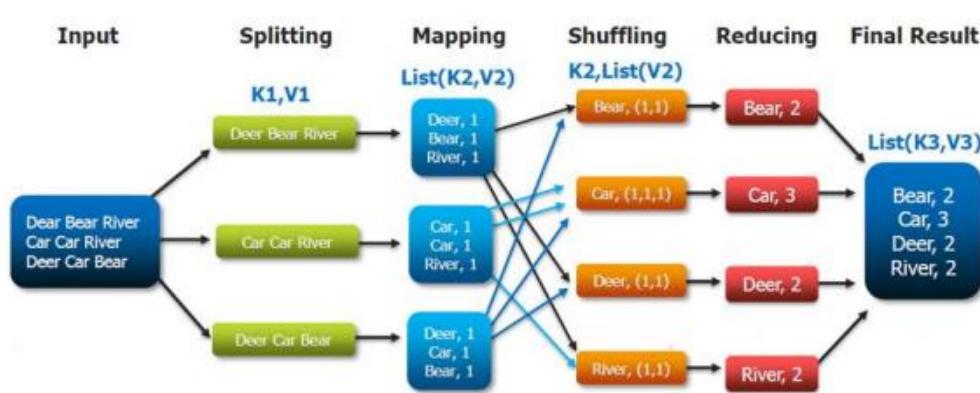
**Shuffle:** Il sistema raggruppa tutte le coppie con la stessa chiave **key\_b**, e le invia allo stesso nodo. Tutte le coppie vengono divise tra i nodi in base a questo criterio, producendo in output **key\_b, list(value\_2)**.

**Reduce**  $(key_b, list(value_2)) \rightarrow list((key_c, value_3))$ : La funzione Reduce riceve una chiave **key\_b** e **tutti i valori value\_2** associati a quella chiave, sotto forma di lista.

Su questa lista applica un'operazione di aggregazione, come somma, media, ecc.

Il risultato è una o più coppie  $(key_c, value_3)$ , che rappresentano l'output finale.

The Overall MapReduce Word Count Process



**Architettura generale** Al di sopra del file system distribuito si trova il **MapReduce Engine**, che si occupa dell'esecuzione dei job MapReduce nel cluster. Il sistema è composto da un **JobTracker**, a cui le applicazioni client sottomettono i job MapReduce, che coordina l'intera esecuzione e assegna il lavoro ai nodi disponibili. Il **JobTracker** distribuisce i task ai **TaskTracker** presenti nei nodi del cluster, cercando di mantenere il calcolo **il più vicino possibile ai dati**. Se il task non può essere eseguito sul nodo che contiene i dati, viene data priorità a nodi appartenenti allo **stesso rack**. Se un **TaskTracker** fallisce o va in timeout, la parte di job assegnata a quel nodo viene **rischedulata** su un altro TaskTracker disponibile. Ogni TaskTracker avvia ogni task in una **JVM separata**, in modo che un crash del job non provochi il fallimento del TaskTracker stesso. Ogni TaskTracker invia periodicamente un **heartbeat** al JobTracker per segnalare il proprio stato.

**Assegnazione dei task ai TaskTracker:** Ogni TaskTracker dispone di un numero fisso di **slot**. Ogni task Map o Reduce occupa **uno slot**. Il JobTracker assegna il lavoro al TaskTracker **più vicino ai dati** che ha uno slot libero.

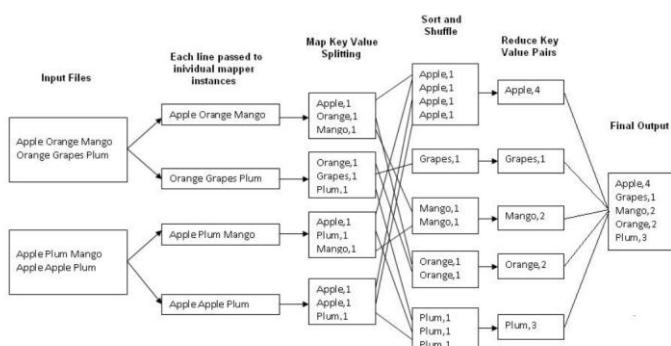
Se un nodo TaskTracker è molto lento, può **ritardare l'intero job MapReduce**, perché tutto il job può finire per attendere il completamento del task più lento.

**Hadoop small cluster:** Include un singolo nodo master e più nodi worker.

Il nodo master ospita i principali servizi di Hadoop: JobTracker, TaskTracker, NameNode e DataNode. Un nodo worker agisce sia come DataNode, per la memorizzazione dei dati, sia come TaskTracker, per l'esecuzione dei task MapReduce.

**Hadoop large clusters:** Abbiamo un nodo NameNode, un nodo Secondary NameNode, un nodo JobTracker e più nodi worker.

## Hello word equivalent in Hadoop: the unavoidable WordCount (1)



### Word Count in Python and Hadoop Run!

- Create a text file `input.txt`

```
Hello Hadoop
Hello Python
Hello MapReduce
Hadoop Python MapReduce
```

- Place it in HDFS

```
hdfs dfs -mkdir -p /user/hadoop/input
hdfs dfs -put input.txt /user/hadoop/input
```

```
hadoop jar
$HADOOP_HOME/share/hadoop/tools/lib/hadoop-
streaming.jar \
-input /user/hadoop/input \
-output /user/hadoop/output \
-mapper mapper.py \
-reducer reducer.py \
-file mapper.py \
-file reducer.py
```

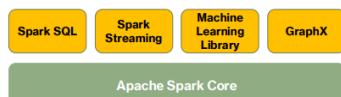
**Svantaggi:** MapReduce è rigido e poco adatto a carichi di lavoro moderni, quindi Hadoop è utilizzato solo a scopo didattico.

**Eager evaluation:** Valutare un'espressione **immediatamente**, non appena viene associata a una variabile. Il valore viene memorizzato anche se non sarà mai utilizzato.

**Lazy evaluation:** Rimandare la valutazione di un'espressione **fino a quando il suo valore è realmente necessario**, ovvero quando un'altra espressione dipende dal suo risultato.

# Apache Spark

**Apache Spark:** Framework per il processamento di dati in modo distribuito che estende l'approccio MapReduce di Hadoop, aggiungendo stream processing, query SQL, algoritmi su grafi, una libreria machine learning e mantenendo i dati in RAM invece che su disco quando possibile.



**Architettura Spark:** Ogni applicazione Spark è composta da un **driver program**, che esegue la funzione main dell'utente, e coordina le operazioni parallele sul cluster.

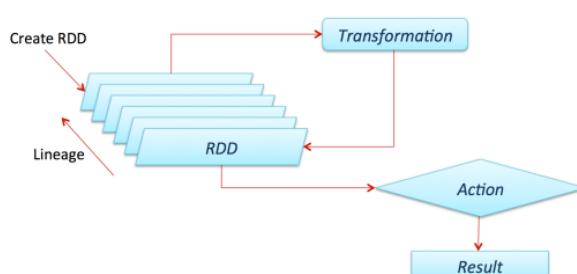
I dati sono mantenuti dal **Resilient Distributed Dataset**, che è una collezione di dati, partizionata tra i nodi del cluster, elaborabile in parallelo.

Ogni programma mantiene delle **shared variables**. Quando Spark esegue una funzione in parallelo su più nodi, invia una copia delle variabili usate dalla funzione a ciascun task, in modo da condividerle tra i vari task.

**Broadcast variables:** Valore memorizzato in RAM su **tutti i nodi** del cluster, evitando copie ripetute.

**Accumulators:** Variabili che possono solo essere **incrementate**, come contatori o somme, usate per statistiche globali.

**Resilient Distributed Datasets:** Collezione distribuita di oggetti, disponibili in sola lettura, suddivisi in **partizioni logiche**, che possono essere elaborate su **nodi diversi** del cluster in **parallelo**. Possono contenere qualsiasi tipo di oggetto Python/Java.



**Creazione degli RDD:** Il primo modo consiste nel **parallelizzare una collezione esistente** all'interno del driver program. Il secondo consiste nel fare riferimento a un **dataset** presente in uno **storage esterno**, come un file system condiviso HDFS.

**Modo 1:** Viene chiamato dal driver program il metodo **SparkContext.parallelize** su una collezione esistente. Gli elementi della collezione vengono **copiati e distribuiti** nel cluster, formando un dataset distribuito che può essere elaborato in parallelo.

**Esempio:**  
Es: `data = [1, 2, 3, 4, 5, 6]`  
`distData = sc.parallelize(data) → Nodo 1: [1, 2] Nodo 2: [3, 4] Nodo 3: [5, 6],`  
L'RDD `distData` viene poi elaborato in parallelo. `distData.reduce(lambda a, b: a + b)`

**#Partizioni:** Di default Spark **imposta automaticamente** il numero di partizioni in base alle risorse del cluster. Manualmente, si consiglia di usare **2–4 partizioni per ogni CPU** disponibile nel cluster.

## Esempio:

```
lines = sc.textFile("data.txt")
lineLengths = lines.map(lambda s: len(s))
totalLength = lineLengths.reduce(lambda a, b: a + b)
```

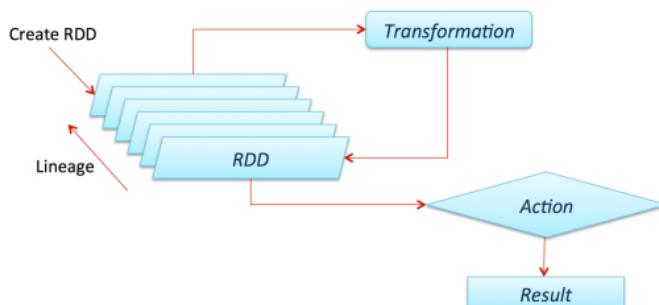
La prima riga definisce un **RDD di base** a partire da un file esterno, assegnandolo al ptr `lines`.

La seconda riga definisce un nuovo RDD, assegnandolo a `lineLengths`, come risultato di una **trasformazione map**, che trasforma ogni linea nella sua lunghezza. Questa operazione è soggetta alla **lazy evaluation**.

La terza riga esegue l'operazione `reduce`, in cui ogni coppia di elementi viene sommata.

Spark qui avvia realmente la computazione, il lavoro viene suddiviso in **task** eseguiti su macchine diverse, ogni nodo esegue la propria parte di map e una riduzione locale, solo il risultato finale viene restituito al **driver program**.

**Persistenza degli RDD:** Se si prevede di riutilizzare lineLengths in seguito, è possibile persistirlo, salvandolo in memoria.



**Operazioni sui RDD:** Le **transformations** creano un **nuovo dataset** a partire da uno esistente. Le **actions** eseguono una computazione su un RDD e **ritornano un valore al driver program**.

**Map:** Transformation che applica una funzione a ciascun elemento del dataset e restituisce un **nuovo RDD** contenente i risultati.

**Reduce:** Action che aggrega tutti gli elementi dell’RDD tramite una funzione e restituisce il **risultato finale al driver program**.

**Lazy evaluation:** Tutte le transformations in Spark sono **lazy**, ovvero non vengono eseguite immediatamente, ma quando un’action ne richiede il risultato.

**Persistenza RDD:** Di default, un RDD viene ricalcolato ogni volta che viene eseguita un’**action**. Tutte le **transformations** che lo generano vengono rieseguite da capo. Se invece un RDD viene **persistito** in RAM, i suoi dati vengono mantenuti in memoria nel cluster, rendendo gli accessi successivi molto più rapidi.

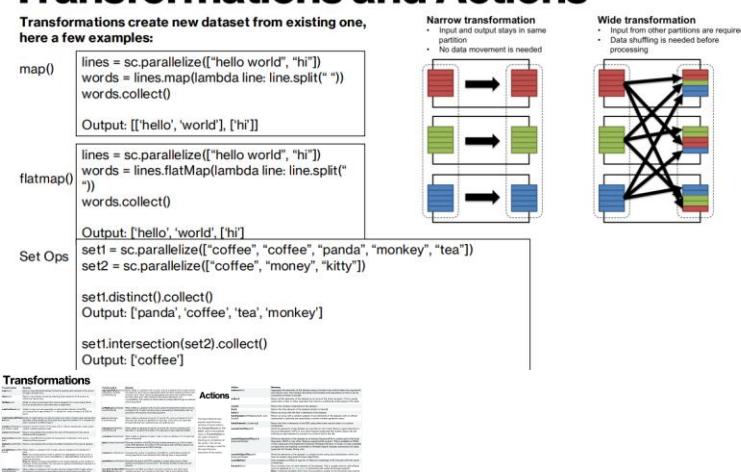
Spark supporta anche la persistenza su **disco** e la **replicazione** degli RDD su più nodi.

**Immutabilità:** Ogni **transformation** non modifica un RDD esistente, ma ne crea sempre uno nuovo. Gli RDD sono immutabili.

**Laziness:** Spark non esegue subito le trasformazioni, ma costruisce un piano di trasformazioni che verrà eseguito solo quando viene richiesta un’action. Un RDD non va visto come un insieme di dati già materializzati, ma come una **descrizione** delle **operazioni necessarie** per calcolare quei dati.

Per il caricamento dei dati negli RDD, questi non vengono letti immediatamente, ma solo quando l’esecuzione è effettivamente richiesta.

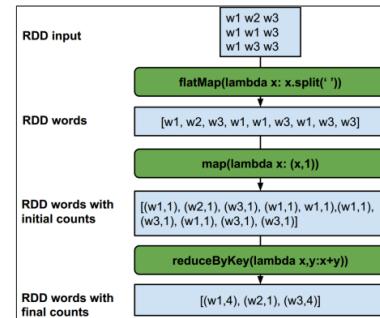
## Transformations and Actions



# Hello word WordCount in Spark

Let us take a look at the code to implement that in Spark

```
1 import sys
2 from pyspark import SparkContext
3 sc = SparkContext(appName="WordCountExample")
4 lines = sc.textFile(sys.argv[1])
5 counts = lines.flatMap(lambda x: x.split(' ')) \
6 .map(lambda x: (x, 1)) \
7 .reduceByKey(lambda x,y:x+y)
8 output = counts.collect()
9 for (word, count) in output:
10     print "%s: %i" % (word, count)
11 sc.stop()
```



## Es WordCount:

**flatMap:** La flatMap() viene applicata all'RDD restituito da sc.textFile, svolgendo due operazioni. Applica la funzione lambda a ogni riga del file, trasformandola in una lista di parole, tolti gli spazi. Appiattisce la struttura risultante. [[w1, w2], [w2, w3]] diventa [w1, w2, w2, w3]. Il risultato è un RDD contenente **tutte le parole**, una per elemento.

**Map:** La map() viene applicata all'RDD prodotto da flatMap. Map() applica la funzione lambda a **ogni elemento dell'RDD**. Per ogni elemento x, viene creata una coppia (x,1), che segna un'occorrenza di tale parola. (parola, 1) per ogni parola.

**reduceByKey:** L'aggregazione viene eseguita con la funzione reduceByKey, che raggruppa prima le coppie in base alla chiave, cioè la parola, e applica poi una funzione di riduzione ai valori associati alla stessa chiave, effettuando la somma delle occorrenze. [(w1,1), (w1,1), (w2,1)] diventa [(w1,2), (w2,1)].

**Collect e stampa:** Dopo la riduzione, il risultato finale viene **raccolto** nel driver program e stampato come elenco di parole con il relativo conteggio.

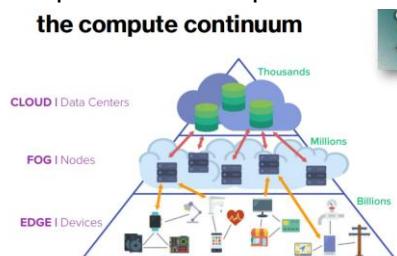
## Lesson 18 – The Actor Model for the Compute Continuum

**Compute continuum:** Infrastruttura di risorse di calcolo, in cui la computazione non avviene solo nel datacenter, ma è distribuita tra dispositivi utente, gateway e data center cloud.

**Livello EDGE:** Dispositivi utente (smartphone, sensori IoT, ecc.). Generano grandi quantità di dati, eseguono calcoli molto semplici, e permettono risposte veloci essendo vicine al dato.

**Livello FOG:** Nodi fog, come gateway, o server distribuiti ma vicini alla fonte del dato. Aggregano i dati provenienti dall'edge, eseguono calcoli un po' più complessi, in modo da ridurre il traffico verso il cloud.

**Livello CLOUD:** Data center cloud. Offrono grande potenza di calcolo, eseguono computazioni complesse e machine learning su grandi quantità di dati.

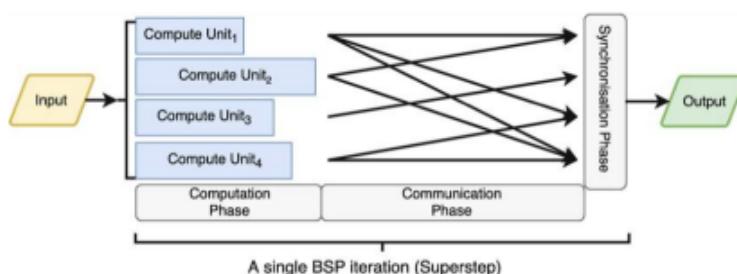


**Uso MapReduce:** MapReduce ha bisogno di banda di rete stabile, disponibilità delle risorse prevedibile, e controllo centralizzato delle risorse. In questo caso è impossibile, perché le risorse sono eterogenee, collegate da larghezza di banda di diverso tipo. I dispositivi non sono controllabili perché gestiti da entità amministrative diverse.

MapReduce può essere visto come un'istanza del modello **Bulk Synchronous Parallel**.

**Bulk Synchronous Parallel:** Sistema parallelo composto da **elementi di calcolo** che eseguono operazioni localmente, una **rete di comunicazione** che permette lo scambio di messaggi tra gli elementi, e un meccanismo di **sincronizzazione** che coordina l'esecuzione. L'esecuzione di un programma BSP è organizzata come una **sequenza di super-step globali**.

## Bulk Synchronous Parallel



**Super-step:** Super-step globale composto da tre sotto-step.

**Computazione concorrente:** Ogni elemento di calcolo esegue operazioni **locali** sui propri dati.

**Comunicazione:** Gli elementi si scambiano messaggi, per comunicare i risultati locali. La comunicazione è asincrona, ma ordinata rispetto ai super-step.

**Barrier synchronization:** Tutti gli elementi si fermano su una **barriera di sincronizzazione**, attendendo che anche gli altri abbiano completato computazione e comunicazione. Dopo questo, può iniziare il super-step successivo.

**Costo:** Il processore più lento + la comunicazione più costosa + la sincronizzazione.

**Limiti:** Il modello BSP assume fasi di calcolo sincrone, richiede sincronizzazioni globali, comportamenti prevedibili. Nel compute continuum queste assunzioni diventano un limite, poiché abbiamo decentralizzazione dell'infrastruttura e delegazione del controllo. Abbiamo bisogno di altri modelli di programmazione.

### Principi per modelli di programmazione decentralizzati

**Adaptability:** Capacità di un sistema di **adattare dinamicamente il proprio comportamento** in risposta a variazioni di **carico, latenza di rete, disponibilità delle risorse** (edge, fog, cloud).

**Openness:** Capacità di un sistema di essere **interoperabile ed estendibile**, evitando lock-in tecnologici, usando **standard aperti** e API ben definite.

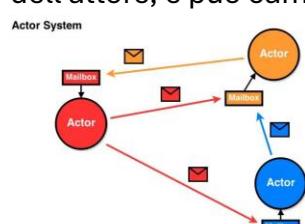
**Modular Philosophies:** Progettazione del sistema come un insieme di **componenti indipendenti e riusabili**, con responsabilità ben definite.

**Resilience:** Capacità di un sistema di continuare a **funzionare correttamente anche in presenza di guasti**. Nodi edge e connessioni di rete possono essere instabili.

## Modello ad attori

**Actor Model:** Modello di programmazione per il **calcolo concorrente e distribuito**.

Gli **attori** sono le **unità di computazione**. Un attore **incapsula** il proprio **stato**, esegue **calcolo** in modo autonomo, e **interagisce** con altri attori solo tramite **messaggi asincroni**. Non esiste condivisione diretta dello stato tra attori. Il **comportamento di un attore** descrive come un attore reagisce ai messaggi. Determina le azioni future dell'attore, e può cambiare nel tempo.



**Azioni di un attore:** In risposta a un messaggio, un attore può contemporaneamente: **Inviare** un numero finito di **messaggi** ad altri attori. **Creare** un numero finito di **nuovi attori**. **Definire il comportamento** da adottare alla ricezione del prossimo **messaggio**. Non esiste alcun **ordine predefinito** tra le azioni elencate, che possono essere eseguite **in parallelo** e in modo indipendente.

Se due messaggi vengono inviati contemporaneamente, non è garantito l'ordine di arrivo.

**Comunicazione asincrona:** Un attore che invia un messaggio non conosce lo stato del destinatario, non attende una risposta immediata e non blocca la propria esecuzione.

**Connessioni tra attori:** Un **attore** può comunicare **solo** con gli attori ai quali è direttamente **connesso**. Un attore può inviare messaggi solo agli attori di cui conosce l'indirizzo, e può ottenere informazioni solo tramite messaggi provenienti da attori connessi. Non esiste accesso globale allo stato di attori non connessi.

**Concorrenza all'interno e tra gli attori:** Ogni attore opera in modo autonomo, mentre molti attori possono essere attivi simultaneamente.

**Creazione dinamica:** Gli attori possono creare dinamicamente nuovi attori durante l'esecuzione.

**Indirizzi degli attori nei messaggi:** Gli indirizzi degli attori possono essere inclusi nei messaggi, permettendo la costruzione dinamica delle relazioni di comunicazione.

**Modularità:** Quando un attore invia un messaggio, il suo compito termina nel momento dell'invio. Da quel momento, la gestione del messaggio diventa **responsabilità del destinatario**. Il mittente non attende risposte, e non conosce né controlla lo stato del ricevente.

Nei modelli tradizionali, l'invio del messaggio è **sincrono**. Il mittente è direttamente coinvolto nella consegna, il messaggio viene trasferito immediatamente in un buffer o in un ambiente condiviso.

**Indeterminacy:** L'ordine con cui i messaggi vengono processati dagli attori **non è deterministico**. Il comportamento complessivo del sistema può variare tra esecuzioni diverse e dipendere dall'ordine effettivo di processamento dei messaggi.

**Garanzie:** I messaggi vengono **consegnati**. Ogni messaggio viene **processato atomicamente** da un singolo attore. Non esiste accesso concorrente allo stato interno di un attore.

**Quasi-commutativity:** Due messaggi sono quasi-commutativi se non dipendono dallo stato l'uno dell'altro, e non modificano la stessa parte di stato dell'attore. In questo caso, elaborare prima un messaggio o l'altro **non cambia il risultato finale**.

Messaggi indipendenti possono essere processati in qualsiasi ordine, o anche in parallelo, senza compromettere la correttezza del risultato.

**Fault-tolerant:** In caso di guasto, un attore può essere **riavviato** o **spostato su un'altra macchina**. Poiché lo stato è encapsulato nell'attore e l'interazione avviene solo tramite messaggi, il fallimento di un attore non compromette l'intero sistema, che può continuare a funzionare.

**Scalabilità:** Gli attori possono essere creati dinamicamente e distribuiti su più macchine. Si può scalare **orizzontalmente** aggiungendo nuove macchine al sistema per gestire un aumento delle richieste.

**Resilienza:** I messaggi sono consegnati in modo affidabile. Ogni messaggio viene elaborato **atomicamente** da un attore. Lo stato interno dell'attore non è accessibile concorrente. Così si previene gli errori di concorrenza.

Questo contribuisce a prevenire errori di concorrenza che potrebbero portare a fallimenti del sistema.

**Limiti:** La progettazione di sistemi basati su attori può risultare **complessa**, richiedendo di ragionare in termini di messaggi asincroni e stati distribuiti. Il testing è particolarmente difficile, poiché il comportamento può cambiare tra esecuzioni diverse.

Si ha un certo **overhead di prestazioni** derivato da gestione e scheduling degli attori, costi dello scambio dei messaggi; serializzazione e deserializzazione dei dati.

Quando gli attori sono eseguiti su macchine diverse, si trovano in data center differenti, o comunicano attraverso reti non affidabili, diventa complesso mantenere una visione coerente dello stato globale.

Es: Sistema di prenotazione biglietti

Attori coinvolti: **UserActor** → rappresenta un utente. **BookingActor** → gestisce una singola prenotazione. **PaymentActor** → gestisce i pagamenti. **NotificationActor** → invia conferme. Ogni attore ha **stato privato** e comunica **solo tramite messaggi**.

Scenari: Due utenti, Alice e Bob, prenotano biglietti **contemporaneamente**.

Fase 1 – Parallelismo tra attori

UserActor(Alice) invia un messaggio Book(ticketA) a BookingActor. UserActor(Bob) invia in parallelo Book(ticketB) a BookingActor. I messaggi arrivano **senza ordine garantito**.

Fase 2 – Sequenzialità interna (no race condition)

Il BookingActor riceve prima Book(ticketA). Aggiorna il proprio stato interno (posti disponibili). Poi riceve Book(ticketB). Aggiorna di nuovo lo stato

Anche se i messaggi sono concorrenti, **sono elaborati uno alla volta**, senza race condition e senza lock.

Fase 3 – Parallelismo distribuito

Dopo ogni prenotazione, il BookingActor crea **due PaymentActor distinti**, e invia loro i messaggi Pay(amount).

I due PaymentActor eseguono il pagamento **in parallelo**, su core diversi o macchine diverse-

Fase 4 – Comunicazione asincrona

Ogni PaymentActor, quando termina, invia PaymentOK al rispettivo NotificationActor.

Il mittente **non aspetta** la risposta. I NotificationActor inviano email di notifica in parallelo.

## Lesson 19 – Distributed Messaging

**Distributed Messaging Systems:** Nei sistemi distribuiti i servizi non hanno memoria condivisa, ma comunicano tramite **messaggi**. La comunicazione è **asincrona**, quindi i servizi non devono stare in invio/ricezione nello stesso momento.

Il **producer** invia il messaggio e continua a lavorare. Il **consumer** lo elabora quando può.

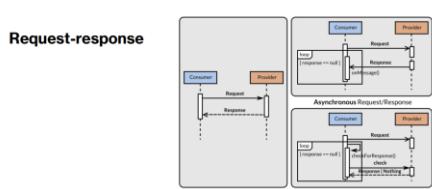
### Message Exchange Patterns

**Request–Response:** Un **Consumer** invia una **Request**. Un **Provider** elabora la richiesta e restituisce una **Response**.

**Synchronous RR:** Il Consumer invia la richiesta, **rimane bloccato** in attesa della risposta, e quando la risposta arriva, l'esecuzione riprende.

**Asynchronous RR:** Il Consumer invia la richiesta, **non resta bloccato**, quando la risposta è pronta, arriva come **messaggio** e viene gestita.

**Polling RR:** Il Consumer invia la richiesta, poi controlla periodicamente se la risposta è disponibile. Finché non c'è risposta, riceve "nothing". Quando arriva, la gestisce.



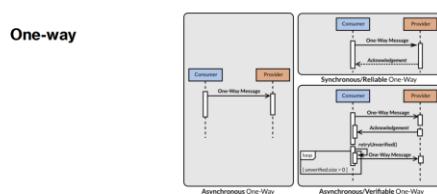
**One-Way:** Il **Consumer** invia un messaggio al **Provider**, non aspetta una risposta, e continua subito la sua esecuzione.

**Asynchronous:** Il Consumer invia il messaggio e prosegue, il Provider lo elabora quando può.

**Synchronous/Reliable:** Il Consumer invia il messaggio, il Provider risponde con un **ack**.

Il Consumer aspetta l'ack di ricezione e solo dopo continua..

**Asynchronous / Verifiable:** Il Consumer invia il messaggio e aspetta un ack, se l'ack non arriva, il messaggio viene marcato come **unverified**, e parte un ciclo di retry che tenta il reinvio.

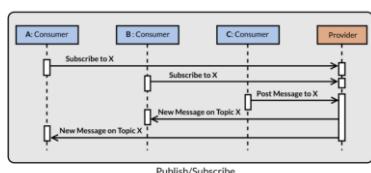


**Publish – Subscribe:** I **Consumer** non comunicano direttamente con il **Provider**, ma si **sottoscrivono** a un canale logico chiamato **topic**, esprimendo il loro interesse a ricevere certi tipi di messaggi. Il provider pubblica un messaggio su un **topic**, e viene passato ad un broker, che lo inoltra a tutti i consumer iscritti.

**Funzionamento:** I Consumer A e B si **sottoscrivono** al **topic X**. Successivamente, il Provider **pubblica un messaggio su X**. Il Provider non sa chi riceverà il messaggio, e non gli interessa saperlo, ma lo affida ad un **broker**. Il messaggio viene automaticamente consegnato a tutti i **Consumer sottoscritti (A e B)** dal broker.

**Asincronia:** Il Provider non aspetta alcuna risposta, e Provider e Consumers non devono **conoscersi** né essere **attivi** nello stesso **momento**. Si possono aggiungere nuovi Consumer o rimuoverli senza modificare il Provider.

#### Publish-subscribe



## Apache Kafka

**Apache Kafka:** Piattaforma **distribuita di event streaming** basata sul paradigma **publish–subscribe**. In Kafka, ogni evento viene scritto in una **struttura append-only**, è **persistito su disco** e mantiene un **ordine totale all'interno di una partizione**.

I dati sono suddivisi in **partizioni** e replicati su più nodi del cluster. I dati possono essere **riletti nel tempo**, permettendo a più consumer di leggere lo stesso stream **in modo indipendente**, ciascuno al proprio ritmo. Il consumo di un messaggio **non ne comporta la scomparsa**.

**Message:** Messaggio pubblicato dal publisher che ha la forma di una **sequenza di byte**. Può includere optionalmente una **key**, anche essa array di byte, utilizzata per determinare la partizione di destinazione tramite hashing. Si sfrutta il fatto che tutti i messaggi con la stessa key vengono scritti nella stessa partizione.

I messaggi non vengono scritti uno alla volta, ma **raggruppati in batches**.

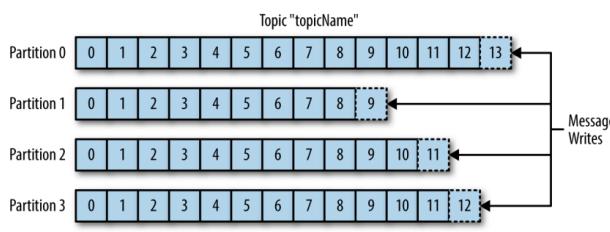
**Batches:** Collezione di messaggi che appartengono allo stesso topic e alla stessa partizione, che vengono **inviai insieme** come un'unica operazione di scrittura. I batch vengono spesso compressi per ridurre i dati occupati su disco e trasferiti in rete.

**Schemas:** Struttura esplicita da dare ai dati (json/xml), in modo che publisher e subscriber li interpretano correttamente.

**Topics:** Un **topic** è una **categoria logica di eventi**, in cui i producer pubblicano messaggi e da cui i consumer leggono.

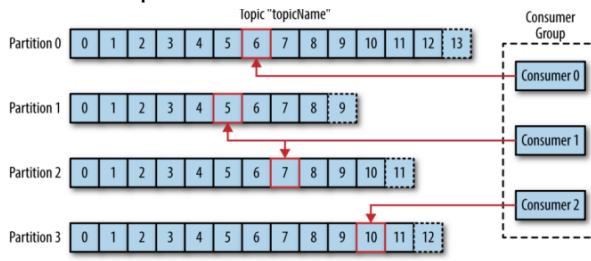
**Partitions:** Ogni topic è suddiviso fisicamente in **partitions**. All'interno di una singola partizione, i messaggi vengono **scritti in append-only** e vengono **letti in ordine**,

dall'inizio alla fine. Quando un topic è composto da più partition, **non esiste un ordine globale** sull'intero topic, ma solo all'interno della singola partition. Ogni partizione può essere ospitata su **un server diverso**, quindi un topic può essere scalato orizzontalmente tra più nodi.



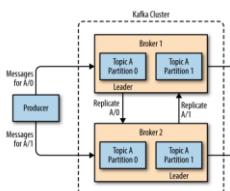
**Producers:** Client che **creano e scrivono nuovi messaggi** in un **topic**. Di default non specificano la **partizione di destinazione**, quindi Kafka li distribuisce tra tutte le partizioni del topic. La partizione può essere controllata tramite key, poiché chiavi con la stessa key hanno stesso hash e stessa partizione.

**Consumers:** Client che **leggono i messaggi**, sottoscrivendosi a dei topic e leggendo i messaggi in ordine all'interno di ogni partizione relativa al topic a cui sono iscritti. Sia l'offset è un **intero crescente** che viene assegnato a ogni messaggio quando viene prodotto, **univoco all'interno della partizione**. Salvando l'offset dell'ultimo messaggio consumato all'interno di una partizione, un consumer può fermarsi e ripartire senza perdere la posizione.



**Consumer groups:** Insieme di consumer che collaborano per leggere un topic. Ogni partizione viene consumata da un **solo consumer alla volta**. Se aumenta il numero di messaggi, si possono aggiungere consumer al gruppo.

**Brokers:** Server Kafka che riceve i messaggi dai **producers** e li invia ai **consumers**. Per ogni messaggio ricevuto da un producer, lo assegna al topic relativo, alla partizione più vuota del topic e gli assegna un **offset**, per poi persistere su **disco**. Serve poi i consumers, restituendo i messaggi ai consumer iscritti al relativo topic. Si hanno cluster di broker, a cui ad ogni nodo vengono assegnate un insieme di partizioni.



**Cluster controller:** Si ha un broker controller all'interno del cluster che viene eletto automaticamente tra i broker attivi e si occupa di assegnare le **partitions** ai broker e monitorare lo stato del cluster, reagendo ai fallimenti.

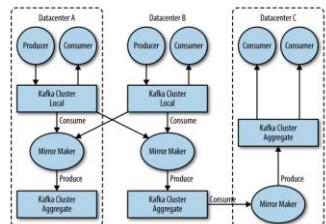
**Partition leadership:** Ogni **partizione** appartiene a **un solo broker alla volta**, leader della partition. I producer e consumer comunicano con il **leader** quando leggono o scrivono dati per quella partition. Una partition può essere **replicata su più broker**, per garantire fault tolerance.

**Retention:** Conservazione dei messaggi per un certo periodo di tempo. I messaggi **non vengono eliminati** subito dopo essere stati **consumati**. Ogni topic viene configurato

con una diversa regola di default, per cui i messaggi vengono mantenuti per un **certo tempo**, o fino al **raggiungimento** di una certa **dimensione**, prima di essere eliminati.

**Log compaction:** Configurazione che se attiva mantiene **solo l'ultimo messaggio per ogni key**, eliminando le versioni precedenti, utile per conoscere l'ultimo stato in caso di messaggi di stato.

**Multiple Clusters:** Utilizzo di **più cluster** in diversi datacenter. E' necessario **replicare i messaggi tra cluster** tramite lo strumento **MirrorMake**. MirrorMake funziona come un **consumer** che legge messaggi da un cluster e come un **producer** che li riscrive su un altro cluster.

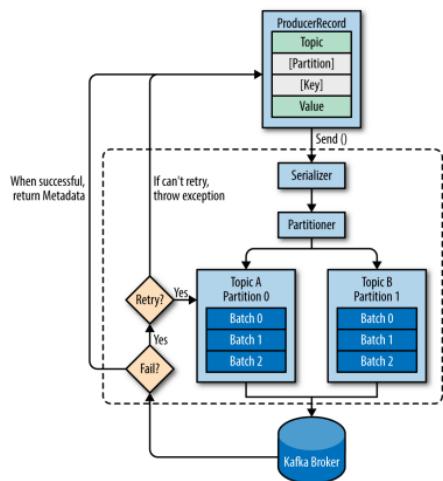


## Kafka producer

**Creazione del messaggio:** Per inviare dati si crea un **ProducerRecord**, che include il **topic** di destinazione, il **value** del messaggio e opzionalmente una **key**. Vengono serializzati key e value e trasmessi al Partitioner, che sceglie la partition del topic in base al carico o all'hash della key.

Il producer sa ora **topic e partition** di destinazione e aggiunge il record a un **batch** insieme ad altri messaggi diretti alla stessa partition. Un **thread separato** si occupa poi di inviare questi batch ai broker Kafka.

Quando il broker riceve i messaggi, se la scrittura va a buon fine, restituisce al producer un **Metadata**, che contiene topic, partition, e offset. Se la scrittura fallisce, il broker restituisce un **errore**. In questo caso, il producer può **ritentare l'invio** per un certo numero di volte prima di segnalare l'errore.



## Modalità di invio dei messaggi

**Fire-and-forget:** Il producer invia il messaggio al broker e **non controlla l'esito dell'operazione**, continuando l'esecuzione. Kafka tenta automaticamente i retry.

**Synchronous send:** Il producer invia il messaggio e il metodo `send()` restituisce un **Future**. Chiamando `get()` sul Future, l'applicazione **si blocca** fino a quando riceve la risposta dal broker.

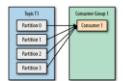
**Asynchronous send:** Il producer invia il messaggio passando una **callback function** al metodo `send()`. L'invio avviene in background e, quando il broker risponde, la callback viene richiamata automaticamente. Approccio ad alte prestazioni più usato.

## Kafka Consumer

**Reading:** Il consumer si sottoscrive a dei **topic** e riceve i messaggi pubblicati su di essi. Quando i **producers scrivono più velocemente** di quanto il consumer riesca a processare, vengono usati i consumer. **Come detto sopra**, quando più consumers appartengono allo **stesso gruppo** e si sottoscrivono allo **stesso topic**, viene assegnato a ciascun consumer un **sottoinsieme diverso** delle partizioni del topic. **Ogni partizione** viene consumata da un **solο consumer** all'interno del gruppo. Per scalare, si aggiungono consumer fino al numero di partizioni, e se necessario si aumenta il numero di partizioni. Oltre questo limite, avremmo consumer fermi. Ogni applicazione ha invece il suo consumer group, perché ognuna deve poter leggere **tutti i messaggi**.

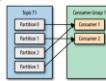
#### Un consumer in un group

Supponiamo di avere un topic T1 con quattro partizioni. Se esiste un solo consumer, C1, nel group G1, allora C1 leggerà tutte e quattro le partizioni. In questo caso non c'è parallelismo nel consumo, ma l'applicazione funziona correttamente.



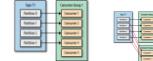
#### Due consumers nello stesso group

Se aggiungiamo un secondo consumer, C2, sempre nel group G1, Kafka ribilancia automaticamente le partizioni. A questo punto:



#### Un consumer per partizione

Se il group G1 arriva ad avere quattro consumer, allora Kafka assegna una partizione a ciascun consumer. Questo è il massimo livello di parallelismo possibile per quel topic, perché il numero di consumer efficaci non può superare il numero di partizioni.



**Partition Rebalance:** Spostamento della **ownership delle partizioni** da un consumer a un altro, in modo di scalare o sostituire un consumer fallito. Durante un rebalance **nessun** consumer del gruppo **può leggere** messaggi.

**Heartbeats:** I consumer inviano periodicamente dei **heartbeat** al **group coordinator** per segnalare che sono ancora attivi. Se un consumer **smette di inviare heartbeat** per un tempo sufficiente, il coordinator lo considera **morto** e avvia un rebalance.

**Chiusura pulita del consumer:** Quando un consumer viene **chiuso correttamente**, notifica il group coordinator che sta lasciando il gruppo. Il coordinator **attiva** immediatamente **il rebalance**, senza dover attendere il timeout degli heartbeat.

## Lesson 20 – Data processing architectures

**Data Processing:** Insieme di operazioni mediante le quali **dati grezzi** vengono raccolti, trasformati e analizzati per ottenere **informazioni utili**. Include la raccolta, pulizia, trasformazione, aggregazione ed elaborazione dei dati, prendendo come input dati grezzi e ritornando come output report e statistiche. Nei sistemi distribuiti include il trasferimento dei dati tra nodi e calcoli e aggregazioni parallele.

### Architectures for Data Processing

**Batch Processing:** Grandi quantità di dati vengono **raccolte** ed elaborate insieme, in **blocco** (batch), a intervalli di tempo, invece che immediatamente quando i dati arrivano.

**Stream Processing:** I dati vengono **elaborati continuamente**, man mano che arrivano, senza aspettare di accumularli in blocchi.

### Type of Data Managed

**Statically Available Data:** Sono dati finiti e stabili nel tempo, completamente disponibili prima dell'inizio dell'elaborazione e che non cambiano durante il processo di elaborazione.

**Dynamic and Ephemeral Data:** Sono **dati dinamici**, che **arrivano in modo continuo nel tempo, possono cambiare rapidamente** e spesso **non sono persistenti**, quindi devono essere elaborati man mano che vengono prodotti.

## Batch Processing

**Batch Processing:** Grandi quantità di dati vengono **raccolte**, memorizzati e solo successivamente elaborati come un unico insieme, in **blocco** (batch), a intervalli di tempo, invece che immediatamente quando i dati arrivano.

Viene usato per eseguire trasformazioni complesse sui dati e per generare report su larga scala, nei casi in cui i dati non sono dinamici.

**Caratteristiche:** Il batch processing si basa sul paradigma **MapReduce**, e viene usato da **Hadoop** e **Spark**, in modo distribuito. Il batch processing è molto scalabile perché sfrutta il parallelismo su più nodi, ma non soddisfa i requisiti di real time. Poiché i dati sono elaborati in blocco, i risultati non sono disponibili immediatamente ma solo al termine del job.

## Stream Processing

**Stream Processing:** I dati vengono **elaborati continuamente**, man mano che arrivano, senza aspettare di accumularli in blocchi.

Viene utilizzato in app che richiedono risposte immediate dai dati, come il trading finanziario o il monitoraggio dei sistemi. Dà risposte in tempo reale, può gestire dataset non limitati. Ha problemi di fault tolerance, perché il fallimento di un nodo può causare la perdita dei dati.

**Consistenza:** E' difficile garantire il livello di correttezza dei risultati prodotti, sopportando arrivo continuo dei dati, elaborati incrementalmente, esecuzione distribuita e possibili guasti.

**Consistency Models:** Definiscono **come e quante volte** un evento **viene processato** all'interno di un sistema di stream processing.

**Exactly-once Processing:** Ogni evento viene processato **una e una sola volta**, senza duplicazioni e senza perdite. Modello ottimo ma complesso da implementare.

**At-least-once Processing:** Ogni evento viene processato **almeno una** volta. Può essere processato più di una volta, introducendo **duplicati**.

**At-most-once Processing:** Ogni evento viene processato al **massimo una volta**.

**State Management:** Insieme dei meccanismi per **mantenere e aggiornare lo stato interno degli operatori**, necessario per elaborazioni che dipendono dalla storia dei dati. Serve per garantire risultati consistenti nel tempo.

**Checkpointing:** Salvataggio periodico dello stato su uno **storage durevole**, in modo da ripartire da uno stato consistente in caso di fallimento.

**State backends:** DB che consente di gestire **grandi volumi di stato in modo efficiente e persistente**, anche in ambienti distribuiti.

**Time Semantics:** Semantic temporale che definisce quale **nozione di tempo** viene usata per interpretare ed elaborare gli eventi in uno stream.

**Event Time:** Momento reale in cui l'**evento è avvenuto** alla sorgente.

**Processing Time:** Momento in cui l'evento **viene elaborato** dal sistema.

**Ingestion Time:** Momento in cui l'evento viene **ricevuto dal sistema**.

**Watermarks:** Meccanismi che permettono di **stimare** fino a che punto il flusso di dati può essere considerato completo, consentendo l'elaborazione corretta delle finestre temporali.

**Transactions:** Un insieme di operazioni su uno stream viene completato interamente oppure annullato.

**Two-phase commit:** Coordina più nodi per garantire che tutti applichino la transazione in modo consistente.

**Scritture idempotenti:** Permettono di ripetere un'operazione senza modificare il risultato finale, anche in caso di retry.

**Fault Tolerance:** Il sistema deve **continuare a funzionare** correttamente anche quando dei **nodi falliscono**.

**Log replay:** Rielaborare gli eventi a partire da un **log persistente**.

**Ripristino dello stato dai checkpoint:** Consente al sistema di recuperare uno stato consistente dopo un guasto, a partire da un checkpoint.

**Buffering:** Trattenere temporaneamente gli eventi per **attendere quelli mancanti**.

**Allowed lateness:** Accettare eventi in **ritardo** entro un **certo limite temporale**.

**Keyed state partitioning:** Assegnare a ciascun nodo la gestione dello stato associato a un insieme di chiavi.

**Windowing:** Tecnica che consente di **suddividere** uno stream di dati potenzialmente infinito in **finestre temporali finite**, ponendo dei confini artificiali, così da poter applicare aggregazioni e operazioni che richiedono un insieme limitato di eventi.

**Time-Based Windowing:** Gli eventi vengono raggruppati in **finestre definite da intervalli di tempo**. Gli eventi che rientrano nello **stesso intervallo** di tempo vengono raccolti e **processati insieme**.

**Count-Based Windowing:** Le finestre vengono create in base a **un numero fisso di eventi**. Una finestra viene **chiusa e processata** quando viene **raggiunto** il numero di eventi **prefissato**.

**Session-Based Windowing:** Raggruppa gli eventi in **sessioni di attività**, delimitate da **periodi di inattività superiori a una soglia predefinita**.

Una sessione **inizia** quando **arrivano eventi** e **termina** quando **non si verificano eventi** per un certo intervallo di tempo.

## Lambda Architecture: Batch and Stream Processing Together

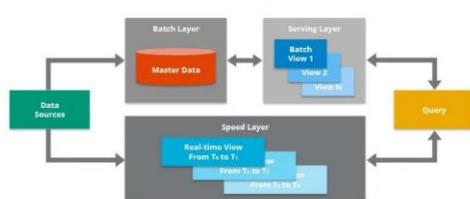
**Lambda Architecture:** Architettura per l'elaborazione dei dati che combina **batch** e **stream processing** per ottenere sia **accuratezza** che **bassa latenza**.

Il **batch layer** elabora grandi quantità di dati storici in modo accurato, producendo risultati corretti e consistenti.

Lo (stream) **speed layer** elabora i dati in tempo reale per fornire risultati immediati, in modo da compensare l'elevata latenza del batch layer, nonostante i risultati possano essere più approssimati.

Il **serving layer** espone i **risultati** delle elaborazioni **agli utenti**, **combinando** i risultati del batch layer e dello speed layer.

Lambda Architecture depicted



**Complessità:** Ha una complessità elevata perché è necessario gestire più layer con logiche diverse. Le elaborazioni possono essere ridondanti.

## Kappa architecture: Only Stream

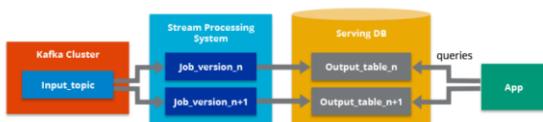
**Kappa Architecture:** Architettura di data processing che utilizza solo lo **stream processing**, eliminando il layer batch, in modo da ridurre la complessità. Essendo in stream si lavora su flussi di dati continui che non hanno una fine naturale.

Tutti i dati sono processati in **stream**.

**Log eventi:** Ogni sorgente di dati scrive lo stato sotto forma **di eventi** in un log. Un unico livello di **stream processing** consuma gli eventi dal log, li elabora in tempo reale e mantiene un eventuale **stato derivato**. I nuovi dati in arrivo vengono aggiunti al log in modo incrementale e processati. I risultati vengono scritti in un **serving DB**, che espone i dati alle app utente. Per effettuare le elaborazioni storiche vengono riprocessati gli eventi dal log tramite un nuovo job.

Lo stato **non** viene considerato **permanente**. Quando è necessario ricalcolare i risultati, il sistema **ricostruisce** lo **stato** riprocesando gli eventi dallo stream, invece di rieseguire job batch su dataset statici.

Viene utilizzato il **checkpointing** per salvare periodicamente lo stato su storage durevole, e gli **state backends** per persistere lo stato localmente nei nodi in stile DB.



## CQRS Architecture

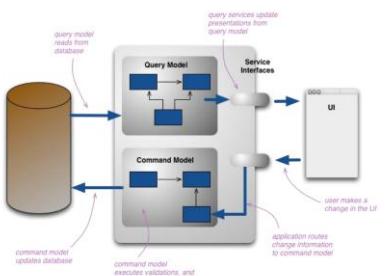
La **CQRS Architecture** è un pattern architetturale in cui **lettura e scrittura dei dati vengono separate** in due responsabilità distinte.

**Command Query Responsibility Segregation:** Le operazioni di **lettura e scrittura** dei dati vengono **separate** in due responsabilità distinte, poiché hanno esigenze diverse in termini di scalabilità e complessità.

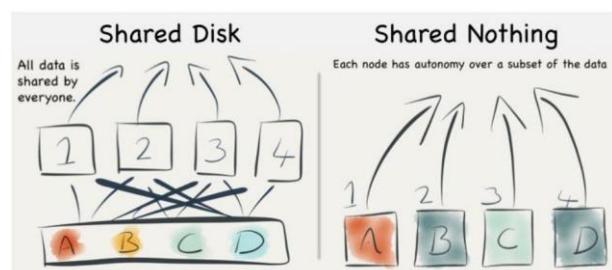
Un'operazione di lettura **non modifica** mai **lo stato** del sistema, mentre un'operazione di scrittura non **restituisce dati complessi**, ma si limita a cambiare lo stato.

**Command Model:** Parte del sistema responsabile delle operazioni di scrittura, che riceve i comandi dall'utente, esegue validazione sui dati e logica applicativa, e **applica modifiche** aggiornando il db o producendo eventi che modificano lo stato del sistema.

**Query Model:** Parte responsabile della lettura, che recupera i dati dal sistema e li prepara per essere mostrati all'utente, restituendo il risultato delle query.



**Funzionamento:** L'utente interagisce con la **UI**. Le azioni dell'utente vengono inviate al **command model**, che valida i comandi ed aggiorna il database. Il **query model** legge i dati dal database e aggiorna le viste esposte alle applicazioni tramite interfacce di servizio.



## Share or Not to Share?

**Share:** Decisione **se e come le risorse vengono condivise** tra i nodi.

**Shared-nothing architecture:** Architettura distribuita in cui ogni nodo è completamente **autosufficiente**, possiede la **propria memoria** e il proprio **storage privato**, **senza condivisione diretta** con gli altri nodi. La **comunicazione** tra nodi

avviene esclusivamente tramite **message passing** sulla rete. Il fallimento di un nodo non compromette il funzionamento degli altri. Il sistema scala aggiungendo nuovi nodi al cluster.

## Data Lakehouse & Data Mesh

**Data Lakehouse:** Architettura che combina i punti di forza dei **data lake** e dei **data warehouse**. Un data lakehouse permette di **archiviare grandi volumi di dati grezzi** (come un data lake) mantenendo però **garanzie di affidabilità, schema e performance** tipiche di un data warehouse.

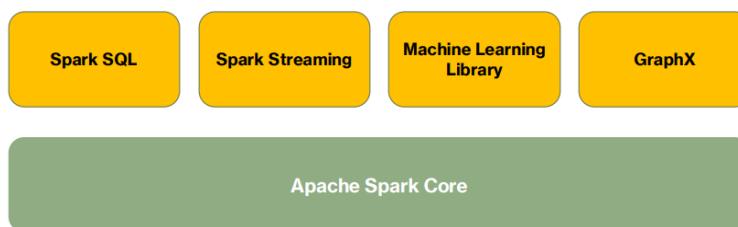
**Data Mesh:** Approccio in cui i dati non sono gestiti da un unico team centrale, ma da più team, ognuno responsabile dei propri dati, come se fossero prodotti. Ogni team (marketing, vendite, ecc.) si occupa dei dati del proprio dominio.

**Federated Learning:** Paradigma di machine learning **decentralizzato** in cui il modello viene addestrato direttamente sui **nodi che possiedono i dati**, senza che questi vengano trasferiti a un server centrale. Ogni nodo calcola aggiornamenti locali del modello, che vengono poi **aggregati** centralmente per ottenere un modello globale.

**Serverless Data Processing:** Modello di elaborazione dei dati in cui la gestione dell'infrastruttura è **completamente astratta**, permettendo agli sviluppatori di concentrarsi esclusivamente sulla logica di processing. Le risorse vengono allocate dinamicamente dal provider cloud e scalano automaticamente in base al carico.

## Lesson 21 – TLAV

### Spark Structure



**Spark Core:** Motore fondamentale su cui si poggiano gli altri moduli Spark, che fornisce le funzionalità essenziali come la **gestione della memoria**, la **schedulazione dei task**, il **fault tolerance** e il **modello di programmazione basato sugli RDD**.

**Spark SQL:** Permette di lavorare con **dati strutturati** offrendo un'interfaccia SQL.

**Spark Streaming:** Consente di elaborare **flussi di dati in tempo quasi reale**, estendendo Spark alle applicazioni di streaming.

**Machine Learning Library:** Fornisce una collezione di **algoritmi di machine learning distribuiti** integrati nativamente con Spark.

**GraphX:** Permette di rappresentare ed elaborare grafi utilizzando operazioni distribuite.

### Spark SQL

**Spark SQL:** Modulo che consente di lavorare di eseguire query SQL su grandi dataset distribuiti gestiti da Spark. Il modulo fornisce a Spark informazioni aggiuntive sulla struttura dei dati e sul tipo di computazione eseguita, andando oltre il modello RDD.

**DataFrame:** Collezione distribuita di dati strutturati, organizzata in **righe e colonne con uno schema**, simile a una tabella SQL, ma eseguita in modo **distribuito** su un cluster Spark. In fase di compilazione non viene controllato se la tabella rispetta lo schema. Le interrogazioni sono ottimizzate automaticamente e sono fornite API.



**Dataset: Collezione distribuita di oggetti fortemente tipizzati.** Spark conosce in anticipo quali campi esistono e che tipo di valore contiene ciascun campo. Se nel codice provi a usare un campo nel modo sbagliato, **l'errore viene bloccato** prima che il programma parta. Grazie alla conoscenza della struttura Spark effettua ottimizzazioni come il riordino delle operazioni, la riduzione dei dati intermedi e la generazione di piani di esecuzione più efficienti.

Esempio di Dataset

```
Da descriviamo le persone come oggetti:
scala
case class Persona(nome: String, eta: Int)

// Dataset
val ds = spark.read.csv("persone.csv").as[Persona]

ds.map(p => p.nome).show() // ✘ NON COMPIA
ds.map(p => p.eta).show()
```

**Creazione DataFrame:** La creazione avviene tramite una **SparkSession**, che crea a partire da **RDD esistenti**, o da altre sorgenti dati supportate (JSON, CSV) un **DataFrame**. Lo schema viene inferito direttamente da Spark.

```
# spark is an existing SparkSession
df = spark.read.json("examples/src/main/resources/people.json")
# Displays the content of the DataFrame to stdout
df.show()
# +---+---+
# | age| name|
# +---+---+
# | null|Michael|
# |  30| Andy|
# |  19| Justin|
# +---+---+
```

**Interrogazione:** Vengono fornite API di interrogazione, come `select(nameColumn)`, `filter()`, `groupBy()`. Possono essere usate query SQL standard, tramite la funzione `sql()`. Il risultato viene fornito sempre sotto forma di DataFrame.

```
# spark, df are from the previous example
# Print the schema in a tree format
df.printSchema()
# root
# |-- age: long (nullable = true)
# |-- name: string (nullable = true)

# Select only the "name" column
df.select("name").show()
# +---+
# | name|
# +---+
# |Michael|
# | Andy|
# | Justin|
# +---+

# Select everybody, but increment the age by 1
df.select(df['name'], df['age'] + 1).show()
# +---+---+
# | name| (age + 1)|
# +---+---+
# |Michael| null|
# | Andy| 31|
# | Justin| 20|
# +---+---+
```

```
# Select people older than 21
df.filter(df['age'] > 21).show()
# +---+
# | age|name|
# +---+
# | 30|Andy|
# +---+

# Count people by age
df.groupBy("age").count().show()
# +---+---+
# | age|count|
# +---+---+
# | 19| 1|
# | null| 1|
# | 30| 1|
# +---+---+
```

```
# Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

sqDF = spark.sql("SELECT * FROM people")
sqDF.show()
# +---+---+
# | age| name|
# +---+---+
# | null|Michael|
# | 30| Andy|
# | 19| Justin|
# +---+---+
```

## Structured Streaming

**Structured Streaming:** Livello che consente di elaborare **dati in streaming** trattandoli come una **tabella che cresce nel tempo**.

**Funzionamento:** Spark **non vede lo stream come eventi singoli**, ma come **una tabella infinita che cresce nel tempo**. I dati arrivano continuamente da una sorgente. Spark legge i dati a intervalli regolari e li trasforma in micro-batch.

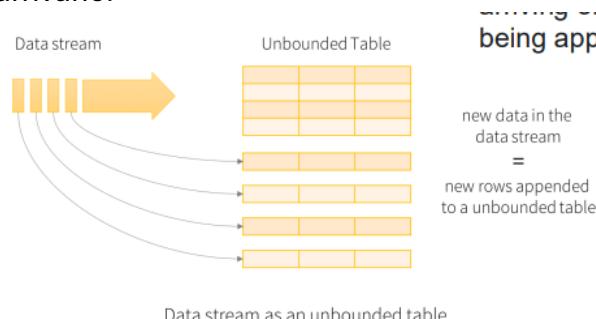
Lo sviluppatore scrive cosa vuole ottenere utilizzando query SQL/DataFrame, o direttamente API fornite da Spark. Spark esegue la query ripetutamente, su piccoli

batch di nuovi dati. Ogni micro-batch legge solo i nuovi record e aggiorna il risultato precedente.

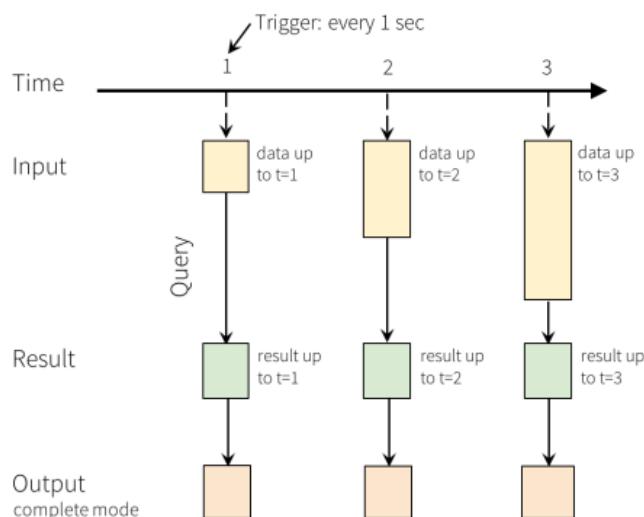
Per operazioni di aggregazione come `groupBy`, `count`, `sum` Spark mantiene uno **stato persistente**.

**Exactly-once semantics:** Ogni evento in ingresso influisce sul risultato finale esattamente una volta. Anche in presenza di errori, non vengono generati **duplicati** e nessun evento rimane **non processato**.

**Input Table:** Ogni dato in arrivo sullo stream corrisponde a **una nuova riga che viene aggiunta** alla **Input Table**. L'esecuzione della query sull'Input Table produce una **Result Table**, che viene **aggiornata progressivamente** man mano che nuovi dati e risultati arrivano.



**Trigger:** Ogni quanto tempo Spark deve elaborare i nuovi dati arrivati. A ogni intervallo nuove righe vengono aggiunte all'Input Table, e di conseguenza la Result Table viene aggiornata in modo incrementale. Ogni volta che la Result Table cambia, Spark scrive **solo le righe di risultato modificate** verso un **sink**, uno storage db/filesystem esterno.



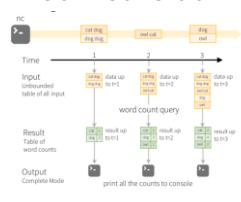
Programming Model for Structured Streaming

**Output Modes:** Definisce **quali parti della Result Table** devono essere emesse a ogni trigger.

**Complete Mode:** L'intera Result Table aggiornata viene scritta nello storage esterno a ogni trigger.

**Append Mode:** Vengono scritte **solo le nuove righe aggiunte alla Result Table dall'ultimo trigger**.

**Update Mode:** Vengono scritte **solo le righe della Result Table che sono state modificate dall'ultimo trigger**.



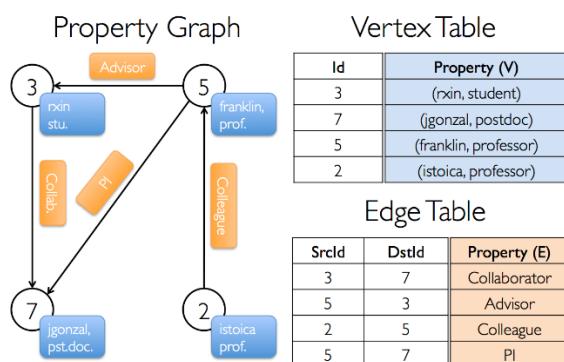
**Data Materialization:** Spark **NON salva** tutta la **tabella di input** dello stream. A ogni micro-batch, legge **solo i nuovi dati**, li usa per **aggiornare il risultato**, e **scarta** i dati di input già elaborati. Spark mantiene **solo lo stato minimo necessario**, come contatori parziali, aggregazioni per chiave e finestre ancora aperte, in modo da aggiornare la **Result Table** senza sprecare troppa memoria.

## Big Data e Grafi

**MapReduce sui Grafi:** MapReduce può essere usato per processare grafi, ma **non è banale esprimere strutture a grafo** all'interno di computazioni MapReduce a causa della loro complessità non lineare. Il codice implementato può essere complesso.

**GraphX:** Livello di Spark per l'esecuzione di computazioni parallele sui grafi.

Un grafo è rappresentato come un insieme distribuito di **vertici** e di **archi**, dove ogni vertice e ogni arco può avere un **attributo etichetta** associato.



**Think Like A Vertex Model:** Progettare l'algoritmo **dal punto di vista del singolo vertice**, invece di ragionare sull'intero grafo in modo globale.

Ogni vertice mantiene il proprio **stato interno**. I vertici **comunicano** tramite **messaggi** scambiati lungo gli archi. **A ogni iterazione**, i vertici aggiornano il proprio stato sulla base dei **messaggi ricevuti**. Il processo continua finché non viene raggiunta una **condizione di convergenza**.

**Actor Model e BSP:** GraphX si basa sull'Actor Model e sul Bulk Synchronous Parallel. Ogni **vertice è visto come un attore**, che comunica con gli altri vertici scambiando messaggi lungo gli archi del grafo. Allo stesso tempo, l'esecuzione segue il modello BSP. La computazione procede in una **serie di superstep**, in cui in ciascuno ogni vertice **riceve i messaggi** dai vertici vicini, **aggiorna il proprio stato** in base ai messaggi ricevuti, **invia** nuovi **messaggi** ai vicini, al **termine** avviene una **sincronizzazione globale** prima di passare allo step successivo.

## Esempio

Il primo passo è creare uno **SparkContext**, che è responsabile della comunicazione con il cluster e della gestione dell'esecuzione distribuita.

```
import org.apache.spark.{SparkConf, SparkContext}

val conf = new SparkConf().setAppName("GraphX Tutorial")
val sc = new SparkContext(conf)
```

Una volta creato lo SparkContext, possiamo **caricare un grafo in GraphX**. In questo esempio usiamo il formato **EdgeList**, un file che contiene coppie di vertici collegati.

**Trasformiamo il file di input in un RDD di archi.** Ogni riga del file rappresenta un edge tra due nodi. Con Graph.fromEdges costruiamo il **grafo distribuito**, specificando anche **come vertici e archi devono essere memorizzati in memoria**.

```

import org.apache.spark.rdd.RDD
import org.apache.spark.graphx.{Graph, Edge}
import org.apache.spark.storage.StorageLevel

val lines: RDD[String] = sc.textFile("path/to/edgelist/file")

val edges: RDD[Edge[Nothing]] = lines.map { line =>
    val parts = line.split(" ")
    Edge(parts(0).toLong, parts(1).toLong)
}

val graph: Graph[Long, Nothing] =
    Graph.fromEdges(
        edges,
        defaultValue = 1L,
        edgeStorageLevel = StorageLevel.MEMORY_ONLY,
        vertexStorageLevel = StorageLevel.MEMORY_ONLY
    )

```

**Assegniamo ora etichette ai vertici e pesi agli archi.**

```

val vertexAttributes = graph.vertices.mapValues(v => s"Label_$v")

val edgeAttributes = graph.edges.map { e =>
    val srcId = e.srcId
    val dstId = e.dstId
    val weight = e.attr.toString
    ((srcId, dstId), s"Weight_$weight")
}

```

Carichiamo in questo altro esempio il grafo direttamente con GraphLoader, eseguiamo l'algoritmo TLAV **PageRank**, che è un **algoritmo iterativo TLAV**, in cui otteniamo per ogni vertice **un punteggio di importanza**.

```

import org.apache.spark.graphx.GraphLoader
import org.apache.spark.graphx.lib.PageRank

val graph = GraphLoader.edgeListFile(sc, "path/to/edgelist/file")

val ranks = PageRank.run(graph, tol = 0.0001).vertices.map {
    case (id, rank) => (id.asInstanceOf[Long], rank)
}

```

Salviamo il risultato su file.

```

ranks
    .map { case (id, rank) => s"$id\t$rank" }
    .saveAsTextFile("path/to/output/file")

```

**Graph:** La classe Graph contiene gli **operatori fondamentali**, implementati in modo **ottimizzato a basso livello**, in modo da garantire massime prestazioni.

**GraphOps:** GraphOps raccoglie operatori di **alto livello**, ottenuti come **composizione** degli operatori base di Graph, fornendo un'API comoda e espressiva.

**Graph Operators:** Operatori che permettono di eseguire **analisi strutturali del grafo**.

**In-degree:** L'**in-degree** di un vertice è il **numero di archi entranti** in quel vertice.

```

val graph: Graph[(String, String), String]
val inDegrees: VertexRDD[Int] = graph.inDegrees

```

**Property Operators:** Modificano **solo le proprietà** di vertici o archi, **senza cambiare la struttura del grafo**, a differenza di map().

**mapVertices:** Trasforma **solo l'attributo dei vertici**. Struttura invariata.

**mapEdges:** Trasforma **solo l'attributo degli archi**. Struttura invariata.

**mapTriplets:** Trasforma l'attributo degli archi. Ha accesso a vertice sorgente, vertice destinazione e arco.